A Fully Verified Executable LTL Model Checker*

Javier Esparza[1], Peter Lammich[1], René Neumann[1], Tobias Nipkow[1],
Alexander Schimpf[2], Jan-Georg Smaus[3]

[1] Technische Universität München, {esparza,lammich,neumannr,nipkow}@in.tum.de
[2] Universität Freiburg, schimpfa@informatik.uni-freiburg.de
[3] IRIT, Université de Toulouse, smaus@irit.fr

Abstract. We present an LTL model checker whose code has been completely verified using the Isabelle theorem prover. The checker consists of over 4000 lines of ML code. The code is produced using recent Isabelle technology called the Refinement Framework, which allows us to split its correctness proof into (1) the proof of an abstract version of the checker, consisting of a few hundred lines of "formalized pseudocode", and (2) a verified refinement step in which mathematical sets and other abstract structures are replaced by implementations of efficient structures like red-black trees and functional arrays. This leads to a checker that, while still slower than unverified checkers, can already be used as a trusted reference implementation against which advanced implementations can be tested. We report on the structure of the checker, the development process, and some experiments on standard benchmarks.

1 Introduction

Developers of verification tools are often asked if they have verified their own tool. The question is justified: verification tools are trust-multipliers—they increase our confidence in the correctness of many other systems—and so their bugs may have a particularly dangerous multiplicative effect. However, with the current state of verification technology, proving software correct is dramatically slower than testing it. The strong advances in verification technology of the last two decades would have not been possible if verifiers had only deployed verified tools.

In this paper we propose a pragmatic solution to this dilemma, precisely because of the advances in verification: verified *reference implementations* of standard verification services. Verifiers working on sophisticated techniques to increase efficiency can test their tools against the reference implementation, and so gain confidence in the correctness of their systems. We present a reference implementation for an LTL model checker for finite-state systems à la SPIN [10]. The model checker follows the well-known automata-theoretic approach. Given a finite-state program P and a formula ϕ, two Büchi automata are constructed that recognize the executions of P, and all potential executions of P that violate ϕ, respectively. The latter is constructed using the algorithm of Gerth et al. [7], which—while not the most efficient—is particularly suitable for formal proof

* Research supported by DFG grant CAVA, *Computer Aided Verification of Automata*

because of its inductive structure. Then the product of the two automata is computed and tested on-the-fly for emptiness. For the emptiness check we use an improved version of the nested depth-first search algorithm [6,22].

A reference implementation of an LTL model checker must be *fully* verified (i. e. it must be proved that the program satisfies a *complete* functional specification). At the same time, it must be reasonably efficient—as a rule of thumb, the verifier should be able to run a test on a medium-size benchmark in seconds or at most a few minutes. Simultaneously meeting these two requirements poses big challenges. An imperative programming style allows one to use efficient update-in-place and random-access data structures, like hash tables; however, producing fully verified imperative code is hard. In contrast, a functional style, due to its extensive use of recursion and recursively defined data structures like lists and trees, allows for standard proofs by induction; however, efficiency can be seriously compromised. In this paper we choose a functional style (we produce ML code), and overcome the efficiency problem by means of a development process based on refinement. We use the Isabelle Collection and Refinement Frameworks [13,16] presented in Section 5. The Refinement Framework allows us to first prove correct an inefficient but simple formalization and then refine it to an efficient version in a stepwise manner. The Collection Framework provides a pre-proved library of efficient implementations of abstract types like sets by red-black trees or functional versions of arrays, which we use as replacement of hash tables.

To prove full functional correctness of executable code, we define the programs in the logic HOL of the interactive theorem prover Isabelle [19]. More precisely, the programs are defined in a subset of HOL that corresponds to a functional programming language. After proving the programs correct, ML (or OCaml, Haskell or Scala) code can be generated *automatically* from those definitions [8]; like in PVS and Coq, the code generator (which translates equations in HOL into equations in some functional programming language) is part of the trusted kernel of Isabelle. Isabelle proofs are trustworthy because Isabelle follows the LCF architecture and all proofs must go through a small kernel of inference rules.

We conduct some experiments on standard benchmarks to check the efficiency of our code. To gain a first impression, for products with 10^6–10^7 states, our implementation explores 10^4–10^5 states per second. While this is still below the efficiency of the fastest LTL model checkers, it fulfills the goal stated above: products with 10^6–10^7 states allow to explore many corner cases of the test space.

All supporting material, including the ML code, can be found online at `http://cava.in.tum.de/CAV13`.

1.1 Related work

To the best of our knowledge, we present the first fully verified executable LTL model checker. In previous work [21], we had presented a formalization of the LTL-to-Büchi translation by Gerth et al. [7] in Isabelle. However, instead of sets, lists were used everywhere for executability reasons. In the present work, we have re-formalized it using the Refinement Framework, in order to separate the abstract specification (in terms of sets) and the concrete implementation

(refining sets to red-black trees, ordered lists etc., as appropriate). This increased modularity and performance. Moreover we have added a second phase that translates the resulting generalized Büchi automata into ordinary ones.

The first verification of a model checker we are aware of is by Sprenger for the modal μ-calculus in Coq [23]. No performance figures or larger examples are reported.

There is a growing body of verified basic software like a C compiler [17] or the seL4 operating system kernel [11]. With Leroy's compiler we share the functional programming approach. In contrast, seL4 is written in C for performance reasons. But verifying its 10,000 lines required 10–20 person years (depending on what you count). We believe that verifying a model checker at the C level would require many times the effort of our verification, and that verifying functional correctness of significant parts of SPIN is not a practical proposition today. Even seL4, although very performant, was designed with verification in mind.

Of course the idea of development by refinement is an old one (see Section 5 for references). A popular incarnation is the B-Method [1] for which a number of support tools exist. The main difference is that B aims at imperative programs while we aim at functional ones.

2 Isabelle/HOL

Isabelle/HOL [19] is an interactive theorem prover based on Higher-Order Logic (HOL). You can think of HOL as a combination of a functional programming language with logic. Although Isabelle/HOL largely follows ordinary mathematical notation, there are some operators and conventions that should be explained. Like in functional programming, functions are mostly curried, i. e. of type $\tau_1 \to \tau_2 \to \tau$ instead of $\tau_1 \times \tau_2 \to \tau$. This means that function application is usually written $f\ a\ b$ instead of $f(a, b)$. Lambda terms are written in the standard syntax $\lambda x.\ t$ (the function that maps x to t) but can also have multiple arguments $\lambda x\ y.\ t$, paired arguments $\lambda(x,y).\ t$, or dummy arguments $\lambda\text{-}.\ t$. Names may contain hyphens, as in *nested-dfs*; do not confuse them with subtraction.

Type variables are written *'a*, *'b* etc. Compound types are written in postfix syntax: τ *set* is the type of sets of elements of type τ, similarly for τ *list*. Lists come with the standard functions *length*, *set* (converts a list into a set), *distinct* (tests if all elements are distinct), and *xs!i* (returns the *i*th element of list *xs*). The function *insert* inserts an element into a set.

A record with fields $l_1, \ldots, l_n$ that have the values $v_1, \ldots, v_n$ is written $(\!|l_1 = v_1,\ \ldots,\ l_n = v_n|\!)$. The field l of a given record r is selected just by function application: $l\ r$.

In some places in the paper we have simplified formulas or code marginally to avoid distraction by syntactic or technical details, but in general we have stayed faithful to the sources.

3 A First View of the Model Checker

In this section we give an overview of the checker and its correctness proof for non-specialists in interactive theorem proving. The checker consists of about 4900 lines of ML. The input consists of a system model and an LTL formula. The modeling language, which we call *Boolean programs*, is a simple guarded command language with Booleans as datatype. The atomic propositions of the formula are of the form `x`, stating that variable `x` is currently true.

Boolean programs are compiled using a function that translates them into an interpreted assembly program, also defined within Isabelle, with a simple notion of configuration. We then call *cava-code*, the main function of the model checker; applied to a compiled Boolean program *bpc* (more precisely, *bpc* is always a pair consisting of a compiled program and an initial configuration) and a formula *phi*, *cava-code* returns either *NO-LASSO* or *LASSO y to-y cyc*. *NO-LASSO* means that the product automaton contains no accepting lasso, i. e. that every execution of the program satisfies the property. *LASSO y to-y cyc* describes an accepting lasso, which corresponds to a counter-example, i. e. an execution of the program violating *phi*: *y* is an accepting state, *to-y* is a path leading to it (given as a list of states), and *cyc* is a cycle from *y* to *y*. The generated ML code looks as follows:

```
fun cava-code bpc phi =
  let val y = LTL-to-BA-code (LTLcNeg phi);
      val x = (graph-Delta-code bpc y, graph-F-code y);
  in nested-dfs-code x start-node end;
```

The main subfunctions already appear in the text of *cava-code*:

- *LTL-to-BA-code phi* is based on the tableau construction by Gerth et al. [7]. The function takes an LTL formula as input and returns the initial state, transition function, and acceptance condition of a Büchi automaton for the formula *phi*. The construction of [7] proceeds by recursion on the structure of the formula, which makes it particularly suitable for verification.
- *graph-Delta-code bpc y* returns the transition function of the product of a Büchi automaton recognizing the runs of *bpc* and the Büchi automaton *y*. *graph-F-code y* returns the accepting states of the product. (As the Büchi automaton of *bpc* has only final states, it is not required as a parameter of *graph-F-code*.)
- *nested-dfs-code x start-node* implements the algorithm of [22] for emptiness of Büchi automata. This algorithm is an improvement of the nested depth-first search algorithm of [9], which in turn improves on the original nested depth-first search algorithm of [6].

Function *LTLcNeg phi* returns the negation of *phi*. So the program negates the formula, computes a Büchi automaton for it, intersects it with a Büchi automaton for the Boolean program, and checks for emptiness.

The model checker and its correctness proof are developed in three steps, using Isabelle's Refinement Framework [15,16], which is described in Section 5. Here, we give a brief overview. Each function *foo-code* in the final ML code is the result

of a three-step process. We first formalize an abstract function *foo*, together with its specification. Abstract functions are allowed to use nondeterministic choice, abstract sets as data structures, etc.; they can be seen as formalized pseudocode. The abstract functions comprise about 250 lines of Isabelle code, which can be found in `http://cava.in.tum.de/files/CAV13/abstract-functions.pdf`. The first proof step consists of showing that they satisfy their specifications.

In a second step, we formalize a *foo-code* function based on *foo*. Here, operations on sets are replaced by corresponding operations on red-black trees or arrays. For instance, an instruction like "**let** $X' = X \cup \{x\}$" is replaced by an insert operation on, say, red-black trees. The second proof step consists of proving that *foo-code* is a refinement of *foo*. Loosely speaking, this means that the result of the (deterministic) *foo-code* is one of the results of the (usually nondeterministic) *foo*. The second step does not significantly increase the code length.

Finally, *foo-code* is automatically transformed into ML code (the ML function keeps the same name). The generated 4900 lines of ML contain the model-checker and all its prerequisites, like the code for red-black trees and other data structures.

For example, the main theorem proving the correctness of *cava*, the abstract function of *cava-code*, looks as follows:

theorem *cava-correct:*
cava bpc $\phi \leq$ **spec** *res. res = NO-LASSO*
$\leftrightarrow (\forall w.$ *BP-accept bpc* $w \longrightarrow w \models \phi)$

It says that the *res*ult of *cava bpc* ϕ (where *bpc* is a compiled Boolean program, see above) satisfies the given **spec**ification: The result is *NO-LASSO* iff $\forall w.$ *BP-accept bpc* $w \longrightarrow w \models \phi$. See Section 5 for details on $\leq$ and **spec**. Here, w is an infinite sequence of (i. e. a function from $\mathbb{N}$ to) sets of Boolean variables of the program. The formula *BP-accept bpc* w is true iff there is a run of the program *bpc* such that at time point n exactly the variables $w(n)$ are true. Hence $\forall w \ldots$ states that every program run satisfies ϕ. The following lemma proves the refinement step:

lemma *cava-code-refine:*
return (*cava-code bpc* ϕ) $\leq$ *cava bpc* ϕ

Once *cava-correct* and *cava-code-refine* have been proven, we can (almost automatically) prove the correctness of *cava-code*:

lemma *cava-code-correct:*
cava-code bpc ϕ = *NO-LASSO* $\leftrightarrow (\forall w.$ *BP-accept bpc* $w \longrightarrow w \models \phi)$

4 A Closer Look at the Model Checker

This section describes and assembles the model checker components on the abstract level. At the end we summarize the size of the complete development.

4.1 Modeling Language

Our Boolean programs are similar to Dijkstra's guarded command programs, with all variables ranging over Booleans. There is `SKIP`, simultaneous assignment $v_1\texttt{,}\ldots\texttt{,}v_n$ `:=` $b_1\texttt{,}\ldots\texttt{,}b_n$ (where the b_i are Boolean expressions), sequential composition c_1`;` c_2, conditional statements `IF [(`b_1`,`c_1`),...,(`b_n`,`c_n`)] FI` (please excuse the syntax), and loops `WHILE` b `DO` c. We use (terminating) recursion in HOL to define programs that depend on some parameter. For example, the program for the n dining philosophers is defined via a function $\mathit{dining}(n)$ that returns a list of pairs of Boolean expressions and commands: $\mathit{dining}(0) = []$ and $\mathit{dining}(n+1) = \ldots\ \mathit{dining}(n)\ldots$. The overall program is simply `WHILE TT DO IF` $\mathit{dining}(n)$ `FI` (where `TT` is the constant true).

Since our focus is the model checker, we have refrained from extending our modeling language further, say with arrays or bounded integers. This would be straightforward in a higher-order interactive theorem prover, as the formalizations of much more complicated languages like C [17] and Java [12] have shown.

The semantics of our modeling language is formalized in HOL by a translation into a simple interpreted assembly language. The reason is speed: executing commands on the source code level is slow. Because the execution has to be interleaved with the state space exploration this would slow down the model checker considerably. The semantics of an assembly language program is given by a function *nexts* that computes the list of possible next configurations from a given configuration. Function *nexts* always returns a nonempty list (in the worst case by cycling), which means that every program has a run and all runs are infinite. Based on *nexts*, *BP-accept* is defined just as sketched at the end of Section 3 above.

4.2 LTL-to-Büchi Translator

The LTL-to-Büchi translator has two parts. The first part implements the algorithm of Gerth et al. [7] to translate an LTL formula into a generalized Büchi-automaton (LGBA, named *LGBArel* in the theories due to its transition *relation*). Recall that the acceptance condition of a generalized Büchi automaton consists of a set $\{F_0, \ldots, F_{m-1}\}$ of sets of accepting states. A run is accepting if it visits each F_i infinitely often. (Ordinary) Büchi automata are the special case $m = 1$. The function *LTL-to-LGBArel* (not shown) implements the tableau construction by Gerth et al. [7]. The correctness proof shows that the resulting LGBA recognizes the language of computations satisfied by the formula.

lemma *LTL-to-LGBArel-sound:*
LTL-to-LGBArel $\phi \leq$ **spec** A_L. $\forall w$. *LGBArel-accept* A_L $w \leftrightarrow w \models \phi$

Since the nested depth-first search algorithm only works for Büchi automata, not for generalized ones, the second part transforms LGBAs into equivalent Büchi automata. The construction for this is simple and well-known (see e. g. [4]), but we use this function to illustrate some points of our approach and (in the next section) of our use of the Refinement Framework.

We briefly recall the LGBA-to-Büchi construction. For each state q of the LGBA we have states (q, k) in the Büchi automaton, where $0 \leq k \leq m-1$ and m is the number of acceptance sets. If $q \to q'$ is a transition of the LGBA and q is labeled with a, then we add transitions $(q,k) \xrightarrow{a} (q',k)$ for every k to the Büchi automaton, but if q belongs to the i-th acceptance set then instead of $(q,i) \xrightarrow{a} (q',i)$ we add $(q,i) \xrightarrow{a} (q', i+1 \mod m)$. An additional technical point is that the LGBA produced by [7] carries labels on the states, and not on the transitions. Therefore, a label on a state of an LGBA must be translated into a label on all outgoing transitions of the corresponding Büchi automaton. Our abstract function for this is *LGBA-to-BA*, shown below. The function takes an LGBA as an argument and returns a Büchi automaton with transition function *trans*, initial states *initial*, and a predicate *accept* defining the accepting states. The predicate $L\ A\ q\ a$ is true if the state q of A is labeled by a.

```
LGBArel-to-BA A = do {
  Flist ← spec xs. F_GBA(A) = set xs ∧ distinct xs;
  return
    (| BA-Δ =
       (λ(q, k) a. if L A q a then
                     {q' | (q,q') ∈ Δ_GBA(A)}
                      × {if k < length Flist ∧ q ∈ Flist!k
                         then (k+1) mod (length Flist)
                         else k}
                   else {},
     BA-I = I_GBA(A) × {0},
     BA-F = (λ(q, k). (k = 0) ∧ (length Flist = 0 ∨ q ∈ Flist!0)) |) }
```

The acceptance family of A is a set of sets of states. However, for indexing, we actually need a *list* of sets. The second line of the above definition assigns this list to the name *Flist*. We prove language equivalence of the LGBA and its corresponding Büchi automaton:

```
lemma LGBArel-to-BA-sound: LGBA A_L ⟶
  LGBArel-to-Buchi A_L ≤
    spec A_B. ∀w. LGBArel-accept A_L w ↔ BA-accept A_B w
```

Assumption *LGBArel* A_L restricts the LGBA to fulfill some consistency properties. Function *LTL-to-BA* is the composition of *LTL-to-LGBArel* and *LGBArel-to-BA*. Combining their correctness lemmas yields the overall correctness lemma:

```
lemma LTL-to-BA-sound:
  LTL-to-BA φ ≤ spec A_B. ∀w. BA-accept A_B w ↔ w ⊨ φ
```

4.3 Language Emptiness Check

The model checker checks emptiness of the language of the product automaton using the algorithm of [22]. It is implemented in the function *nested-dfs-code*, whose corresponding abstract function is *nested-dfs*. It is defined on any graph

consisting of an initial vertex x, a successor function *succs*, and a distinguished subset F of vertices. In case of the product automaton, these correspond to the initial state, the transition function, and the set of accepting states of the automaton, respectively. The correctness theorem is phrased in terms of the transition relation $\rightarrow = \{(a,b) \mid b \in succs\ a\}$:

lemma *nested-dfs-NO-LASSO-iff:*
nested-dfs (*succs, F*) $x \leq$ **spec** *res. res = NO-LASSO* $\leftrightarrow$
$\neg(\exists\ v.\ x \rightarrow^* v \wedge v \in F \wedge v \rightarrow^+ v)$

It expresses that the function returns *NO-LASSO* iff there is no final state v reachable from x and reachable (in at least one step) from itself.

4.4 The Model Checker

Now we combine the individual components of the model checker to obtain the main function *cava*. We do not show this in the paper (see `http://cava.in.tum.de/CAV13`) but sketch (we do not explain all the details) how the individual correctness lemmas are combined into the main theorem *cava-correct* above.

In the first step, the input formula ϕ is translated into a Büchi automaton by *LTL-to-BA*. Lemma *LTL-to-BA-sound* states the correctness of this step. Then function *SA-BA-product* (not shown because straightforward) computes the product of the model A_S (which is an automaton-view of the program *bpc*) and the result A_B of the formula translation. The following lemma tells us that on the language level this corresponds to intersection:

lemma *SA-BA-product-correct:*
SA $A_S \wedge$ *finite*(*BA-Q* A_B)
$\longrightarrow \mathcal{L}_{BA}$ (*SA-BA-product* A_S A_B) $= \mathcal{L}_{BA}\ A_S \cap \mathcal{L}_{BA}\ A_B$

Note that $\mathcal{L}_{BA}\ A$ is simply the set of all w such that *BA-accept* A w. Now we characterize non-emptiness of the product automaton by the existence of a lasso, i. e. a path from a start state to an accepting state q_f together with a non-empty loop from q_f to q_f. The following lemma states the more interesting of the two implications:

lemma *Buchi-accept-lasso:*
$\mathcal{L}_{BA}\ A \neq \emptyset$
$\longrightarrow \exists q_i\ q_f\ r_1\ r_2.\ q_i \in$ *BA-I* $A \wedge$ *BA-F* $A\ q_f$
$\wedge$ *is-finite-run* $A\ r_1 \wedge$ *head* $r_1 = q_i \wedge$ *last* $r_1 = q_f$
$\wedge$ *is-finite-run* $A\ r_2 \wedge$ *head* $r_2 = q_f \wedge$ *last* $r_2 = q_f \wedge$ *length* $r_2 > 1$

Combined with lemma *nested-dfs-NO-LASSO-iff* above (where $x \rightarrow^* v$ corresponds to *is-finite-run* $A\ r_1 \wedge$ *head* $r_1 = x \wedge$ *last* $r_1 = v$) this tells us that *nested-dfs* returns *NO-LASSO* iff the language is empty. Now it is just a set theoretic step that takes us to lemma *cava-correct* because *cava* builds the product of the model and the negated formula, which is empty iff every run of the program satisfies the formula.

4.5 Size of the Development

The following table summarizes the size of the development in lines of Isabelle "code", i. e. definitions and proofs, where typically 90% are proofs. We have distinguished the verification on the abstract level from the refinement steps:

	Abstract verification	Refinement
LTL-to-Büchi	3200	1500
Product construction	800	100
Emptiness check	2500	1600
Top level	500	400

In addition, there are approximately 5000 lines of supporting material that do not fit into the above classification. In total, this comes to roughly 16,000 lines. Moreover we rely on the separately developed and independent Collections Framework (30,000 lines, in total) and Refinement Framework (10,000 lines), described in Section 5.

5 Refinement Framework

When developing formally verified algorithms, there is a trade-off between the efficiency of the algorithm and the efficiency of the proof: For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as proving implementation details blows up the proof and obfuscates the main ideas of the proof. A standard approach to this problem is stepwise refinement [2,3], where this problem is solved by modularization of the correctness proof: One starts with an abstract version of the algorithm and then refines it (in possibly many steps) to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program and replace abstract datatypes by their implementations. For example, selection of an arbitrary element from a set may be refined to getting the head of a list. The main point is, that correctness properties can be transferred over refinements, such that correctness of the concrete program easily follows from correctness of the abstract algorithm and correctness of the refinement steps. The abstract algorithm is not cluttered with implementation details, such that its correctness proof can focus on the main algorithmic ideas. Moreover, the refinement proofs only focus on the local changes in a particular refinement step, not caring about the overall correctness property.

In Isabelle/HOL, refinement is supported by the Refinement Framework [15,16] and the Isabelle Collection Framework [14,13]. The former framework implements a refinement calculus [3] based on a nondeterminism monad [24], and the latter one provides a large collection of verified efficient data structures. Both frameworks come with tool support to simplify their usage for algorithm development and to automate canonical tasks such as verification condition generation.

In the nondeterminism monad, each program yields a *result* that is either a set of possible values or the special result **fail**. A result r *refines* another result r', written $r \leq r'$, if $r' = \textbf{fail}$ or if $r \neq \textbf{fail} \neq r'$ and every value of r is also

```
dfs E vd v0 = do {
 rec_T (λD (V,v).
   if v = vd then return True
   else if v ∈ V then return False
   else do {
     let V = insert v V;
     foreach_C {v' | (v,v') ∈ E} (λx. x=False)
         (λv' -. D (V,v')) False
   }
 ) ({},v0) }
```

Algorithm 1: Simple Depth-First-Search Algorithm

```
function dfs(E : set of pairs of 'a, vd : 'a, v0 : 'a)
    return dfs-body (E, vd, v0, ∅, v0)

function dfs-body(E, vd, v0, V : set of 'a, v : 'a)
    if v = vd then return true
    else if v ∈ V then return false
    else
        V := V ∪ {v}
        ret := false
        for all x ∈ {v' | (v, v') ∈ E} do
            if ret = true then break
            else ret := dfs-body (E, vd, v0, V, x)
        return ret
```

Algorithm 2: Simple DFS Algorithm (imperative equivalent)

a value of r'. The result **return** x contains the single value x, and the result **spec** $x.\ \Phi$ contains all values x that satisfy the predicate Φ. Thus, correctness of a program f w. r. t. precondition P and postcondition Q can be specified as: $P\ x \longrightarrow f\ x \leq$ **spec** $r.\ Q$. Intuitively, this reads as: If the argument x satisfies precondition P, then all possible result values of f satisfy postcondition Q.

As an example we present a depth-first search algorithm, which is a simplified version of the nested DFS algorithm used in the model checker. Algorithm 1 uses the syntax of Isabelle/HOL, and Algorithm 2 displays its equivalent in imperative pseudocode. In Isabelle/HOL, a Haskell-like **do**-notation is used. The $\mathbf{rec_T}$ combinator is recursion, where the recursive call is bound to the first parameter D. The $\mathbf{foreach_C}$ combinator iterates over all elements of the set, and additionally has a continuation condition, i. e. the iteration is terminated if the continuation condition does not hold any more. Here, we use the continuation condition to break the loop if the recursive call returns true.

We now prove the following lemma, stating that if the algorithm returns true, then the node *vd* is reachable from *v0*:

lemma *dfs-sound:*
finite $\{v.\ (v0,v) \in E^*\} \longrightarrow$ *dfs E vd v0* $\leq$ **spec** $r.\ r \longrightarrow (v0,vd) \in E^*$

The proof of this lemma in Isabelle/HOL reads as follows:

unfolding *dfs-def*
apply (*refine-rcg refine-vcg impI*
RECT-rule[**where**
$\Phi = \lambda(V,v).\ (v0,v) \in E^* \wedge V \subseteq \{v.\ (v0,v) \in E^*\}$ **and**
$V =$ *finite-psupset* $(\{v.\ (v0,v) \in E^*\}) \times_{\mathsf{lex}} \{\}$]
FOREACHc-rule[**where** $I = \lambda\text{-}\ r.\ r \longrightarrow (v0,\ vd) \in E^*$])
... [*3 lines of straightforward Isabelle script*]

In the first line, we unfold the definition of *dfs*. In the **apply**-command starting in the second line, we invoke the verification condition generator. The crucial part here is to specify the right invariants: For the recursion, we need a precondition Φ and a variant V. The precondition states that the current node v is reachable and that the set of visited nodes is reachable. The variant states that, in each recursive call, the set of visited nodes gets closer to the finite set of reachable nodes. This is required to show termination. For the foreach-loop, we need an invariant I. It states that, if we break the loop, the target node vd is reachable. The remaining lines of the proof show that the precondition implies the postcondition, that the variant is valid, and that the invariant is preserved. As this only involves argumentation about sets, which enjoy good tool support in Isabelle/HOL, this can be done in a few straightforward lines of Isabelle script.

Once we have defined the abstract algorithm and proved that it satisfies its specification, we refine it to an executable version. This includes *data refinement*, e. g. implementing sets by red-black trees, and nondeterminism reduction, e. g. implementing the foreach-loop by in-order iteration over the red-black tree.

A *refinement relation* is a single-valued relation between concrete and abstract values (e. g. between red-black trees and sets). Single-valuedness means that a concrete value must not be related to more than one abstract value.

The *concretization function* $\Downarrow$ lifts a refinement relation R to results. For programs f and f', the statement $(x,x') \in Ri \longrightarrow f\ x \leq \Downarrow Ro\ (f'\ x')$ means that program f refines program f', where the argument is refined according to relation Ri, and the result is refined according to the relation Ro.

The refinement of *dfs* is straightforward, and the Refinement Framework can generate an executable version together with the refinement proof automatically.

As an example for a more complex refinement, reconsider *LGBA-to-Buchi* from Section 4.2, which translates node-labeled generalized Büchi automata to edge-labeled Büchi automata. Its refined version is Algorithm 3. Here, the input automaton is represented by a tuple, where the transitions D are represented by nested red-black trees, the sets of initial states I and final states F are represented by lists of distinct elements, and the representation of the labeling function L is not changed. The result automaton is represented by a tuple consisting of

- a successor function, which maps a state and a label to a list of distinct successor states,
- a list of distinct initial states, and
- the characteristic function of the sets of final states.

```
LGBArel-to-Buchi-impl ((-, -, D, I, F), L) = do {
  let Flist = lsi-to-list F;
  return
    (λ(q, k) l.
      if L q l then
        (let k' = (if (k < length Flist ∧ lsi-memb q (Flist!k))
                   then (k+1) mod (length Flist) else k) in
         let succs = rs-lts-succ-it D q () (λ-. True) lsi-ins (lsi-empty ()) in
           lsi-product succs (list-to-lsi [k']))
       else lsi-empty (),
      lsi-product I (list-to-lsi [0]),
      (λ(q, k). (k = 0) ∧ (length Flist = 0 ∨ lsi-memb q (Flist!0))))
}
```

Algorithm 3: Implementation of the LGBA-to-Büchi Translation

For this refinement, the **spec**-statement, which was used to nondeterministically select a list representation of the set of final states, has been replaced by a let statement, which deterministically uses the list *lsi-to-list F*. In the **return**-statement, we have replaced the abstract operations on sets (e. g. $\times$) by their concrete counterparts (e. g. *lsi-product*). Note that functions from the Isabelle Collection Framework follow a standard naming scheme: The prefix *lsi* denotes operations on sets represented by lists of distinct elements. Analogously, the prefix *rs* denotes operations on sets represented by red-black trees.

The following lemma relates the abstract and the concrete algorithm:

lemma *LGBArel-to-BA-impl-refine:* $(A_L, A'_L) \in$ *LGBArel-impl-rel* $\longrightarrow$
LGBArel-to-BA-impl $A_L \leq \Downarrow$*BA-impl-rel* (*LGBArel-to-BA* A'_L)

Here, *LGBArel-impl-rel* relates the input automaton A'_L to its representation A_L. Similarly, *BA-impl-rel* relates the result automaton to its representation. The proof of the above lemma is quite straightforward. The main proof effort goes into showing that the successor states of q (abstractly: $\{q' \mid (q,q') \in \Delta\_GBA(A)\}$) are correctly implemented by the iterator *rs-lts-succ-it*, which iterates over the successor states and collects them in a list.

The algorithm *LGBArel-to-BA-impl* is already deterministic. However, it is still defined in the nondeterminism monad, which is not executable. Thus, a further refinement step removes the nondeterminism monad. This step is fully automatic: The Refinement Framework defines a constant *LGBArel-to-BA-code* and proves the lemma **return** (*LGBArel-to-BA-code* A_L) $\leq$ *LGBArel-to-BA-impl* A_L. Finally, the code-generator exports ML-code for *LGBArel-to-BA-code*.

Using transitivity of $\leq$ and monotonicity of the concretization function, we could combine the above result with Lemma *LGBArel-to-BA-sound* from Section 4.2, and obtain:

lemma *LGBArel-to-BA-code-sound:* $(A_L, A'_L) \in$ *LGBArel-impl-rel* $\longrightarrow$
$\exists A_B.$ (*LGBArel-to-BA-code* A_L,A_B) $\in$ *BA-impl-rel*
$\wedge$ ($\forall w.$ *LGBArel-accept* A'_L $w \leftrightarrow$ *BA-accept* A_B w)

Note, however, that we need to prove such lemmas only for the interface functions of our tool, not for internal functions like *LGBArel-to-BA*.

6 Some experiments

As mentioned in the introduction, our project must fulfill two conflicting requirements: a mechanized proof of functional correctness, and adequate performance for a reference implementation. In this section we provide some evidence for the latter.

The natural tool for a comparison with our checker is SPIN [10], while keeping in mind that SPIN is implemented in C, while our checker is implemented in ML. We use SPIN version 6.2.3 with turned-off optimizations (`-o1 -o2 -o3 / -DNOREDUCE`), and MLton version 20100608 as the compiler for our checker. We take three standard well-known and easily scalable benchmarks: the Dining Philosophers, a Readers-Writers system guaranteeing concurrent read but exclusive write, and the Leader-Filters example of [5].

The comparison with SPIN requires some care because the compilation of a program into a Kripke structure can lead to substantial differences in the number of reachable states. For instance, consider a program $P_1 \parallel \ldots \parallel P_n$, where $P_i =$ **while true do** $b := 0$ (in a generic program notation). Each parallel component can be represented by an automaton with one single state and a self-loop labeled by $b := 0$. If initially $b = 0$, then the complete system has one reachable state. However, a compilation process might also lead to an automaton that cycles between two states, moving from the first to the second state by means of a transition labeled by $b := 0$, and back to the initial state by a silent transition. This harmless change has a large impact in the state space: the new version has 2^n reachable states, where n is the number of components, leading to much larger verification times. We have observed this effect in our experiments: SPIN's mature compilation process generates fewer states than our tool, where this aspect has not yet been optimized (cf. Table 1).[4]

For this reason, we compare not only the time required to explore the state space of the product automaton, but also the *state exploration speed*, i. e. the number of states explored per time unit.

If a property does not hold, then the verificiation time and the number of states explored may depend on arbitrary choices in how the depth-first search is conducted. So we only consider properties that hold, for which every tool explores the complete state space. Table 1 shows verification times for the trivial property $\mathbf{G}\mathit{true}$. All times are in milliseconds. Since experiments with other properties yield similar results and no new insight, the results are omitted. We observe that our checker is between 7 and 26 times slower than SPIN.

We now consider the exploration speed. Since it depends on the number of states (if the number is large then state descriptors are also large, and more costly

[4] A similar effect is observed in the number of states of the Büchi automaton for a formula: While both SPIN and our checker are based on the algorithm of [7], it is known that the number of states is very sensitive to simplification heuristics.

Phils					RW					LF				
	SPIN		Cava			SPIN		Cava			SPIN		Cava	
#	Time	States	Time	States	#	Time	States	Time	States	#	Time	States	Time	States
10	70	6	839	50	10	20	1	134	11	3	10	4	104	8
11	190	16	2773	132	11	50	2	335	25	4	220	64	3611	134
12	500	39	8857	343	12	100	4	862	53	5	4720	1006	122620	2271
13	1350	95	27957	890	13	230	8	2283	115	6	91200	15305	OoM	

Table 1: Construction time (ms) for the state space (in thousands of states)

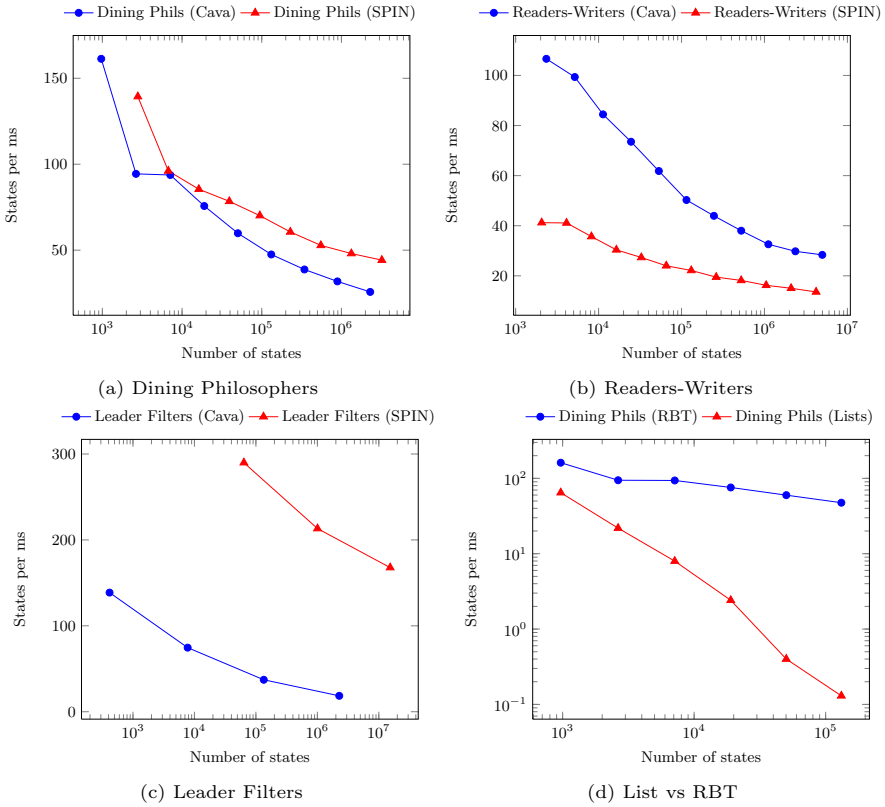
to process), we plot it against the size of the state space. Overall, our checker generates about 10^4–10^5 states per second (this was also the speed reported in [10], published in 2003), and is consistent with the fact that after one decade SPIN is about one order of magnitude faster. Figures 1a–1c show the results for each of the three benchmarks, with exploration speed in states per *millisecond*. Our checker is about 3 times faster than SPIN on *Readers-Writers*, about eight times slower on *Leader Filters*, and about as fast as SPIN on *Dining Philosophers*.

Summarizing, while our checker is slower than SPIN, we think it is fast enough for the purpose of a reference implementation. Most of the functionality of an LTL model checker can be tested on examples with 10^4–10^7 states, for which our checker is about one order of magnitude slower than SPIN, which will be somewhat reduced once our compilation process is optimized.

We can now also illustrate the importance of the Refinement Framework. Without it, we would at most have been able to prove correctness of a model checker with sets implemented as lists. Using the Refinement Framework we can easily generate code for this "slow" checker, and compare its speed with the one of the optimized version where sets are implemented as red-black trees. The result for the dining philosophers is shown in Figure 1d, which uses a double logarithmic scale. For systems with 10^5 states the slow checker is already almost three orders of magnitude slower, which makes it fully inadequate as a reference implementation.

7 Conclusion

Model checkers are a paradigm case of systems for which both correctness and efficiency are absolutely crucial. We have presented the—to the best of our knowledge—first model checker whose code has been fully verified using a theorem prover and is efficient enough to constitute a reference implementation for testing purposes. A key element was our use of a refinement process: specify and verify the model checker on an abstract mathematical level (about 250 lines of code, not counting comments etc.), then improve efficiency of the algorithms and data structures by stepwise refinement, and finally let the theorem prover generate ML code (4900 lines). Our experiments indicate that our checker is only one order of magnitude slower than SPIN in the range of systems with 10^6–10^7 states. We think this result is very satisfactory, since SPIN is a highly optimized checker, programmed in C; moreover, our results indicate that the distance to



(a) Dining Philosophers

(b) Readers-Writers

(c) Leader Filters

(d) List vs RBT

Fig. 1: State-exploration speed

SPIN can be shortened by means of optimizations in the compiler that generates the state space from the high-level system model.

An alternative approach to obtain verified results is to use checkers that provide certificates of their answer (e. g. a Hoare proof) that can be independently checked by a trusted certifier [18,20]. The advantage of this approach is a much smaller formalization effort, and its disadvantage the potentially very large size of the certificates (worst case: the same order of magnitude as the state space).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
3. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)

5. Choy, M., Singh, A.K.: Adaptive solutions to the mutual exclusion problem. Distributed Computing 8(1), 1–17 (1994)
6. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design 1(2/3), 275–288 (1992)
7. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proc. Int. Symp. Protocol Specification, Testing, and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1996)
8. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS. LNCS, vol. 6009, pp. 103–117. Springer (2010)
9. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) Proc. of the 2nd SPIN Workshop. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. American Mathematical Society (1997)
10. Holzmann, G.J.: The Spin Model Checker — Primer and Reference Manual. Addison-Wesley (2003)
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proc. ACM Symp. Operating Systems Principles. pp. 207–220. ACM (2009)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Syst. 28(4), 619–695 (2006)
13. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP. LNCS, vol. 6172, pp. 339–354. Springer (2010)
14. Lammich, P.: Collections framework. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Collections.shtml` (Dec 2009), formal proof development
15. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Refine_Monadic.shtml` (2012), formal proof development
16. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
17. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning 43, 363–446 (2009)
18. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV. LNCS, vol. 2102, pp. 2–13. Springer (2001)
19. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
20. Peled, D., Pnueli, A., Zuck, L.D.: From falsification to verification. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS. LNCS, vol. 2245, pp. 292–304. Springer (2001)
21. Schimpf, A., Merz, S., Smaus, J.G.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs. LNCS, vol. 5674, pp. 424–439. Springer (2009)
22. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L. (eds.) TACAS. LNCS, vol. 3440, pp. 174–190. Springer (2005)
23. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) TACAS. LNCS, vol. 1384, pp. 167–183. Springer (1998)
24. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2, 461–478 (1992)