Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formal Verification of an Executable LTL Model Checker with Partial Order Reduction

Julian Brunner · Peter Lammich

the date of receipt and acceptance should be inserted later

Abstract We present a formally verified and executable on-the-fly LTL model checker that uses ample set partial order reduction. The verification is done using the proof assistant Isabelle/HOL and covers everything from the abstract correctness proof down to the generated SML code.

Building on Doron Peled's paper "Combining Partial Order Reductions with On-the-Fly Model-Checking", we formally prove abstract correctness of ample set partial order reduction. This theorem is independent of the actual reduction algorithm. We then verify a reduction algorithm for a simple but expressive fragment of Promela. We use static partial order reduction, which allows separating the partial order reduction and the model checking algorithms regarding both the correctness proof and the implementation. Thus, the Cava model checker that we verified in previous work can be used as a back end with only minimal changes. Finally, we generate executable SML code using a stepwise refinement approach. We test our model checker on some examples, observing the effectiveness of the partial order reduction algorithm.

1 Introduction

Partial order reduction [28] is an important optimization for model checkers, enabling them to deal better with models involving concurrency. It allows the model checker to consider only a subset of all possible interleavings of concurrently executing operations by identifying equivalences between them. Unfortunately, partial order reduction is notoriously complex and can easily affect the correctness of the model checker. For instance, [28] describes a partial order reduction algorithm and claims that it can simply be used with on-the-fly nested depth-first search. It was found out later that this compromises correctness due to the reduction possibly differing between the inner and the outer search [9]. Moreover, while formalizing the algorithm in [28], we discovered that its correctness proof uses an invalid lemma.

Implementation correctness is usually assessed via testing in the context of model checking algorithms. However, testing is necessarily incomplete and may lead to incorrect

Research supported by DFG grant Cava (Computer Aided Verification of Automata)

Technische Universität München

implementations due to missed corner cases. Furthermore, when using models of realistic size, determining the correct outcome for a given test input requires the use of a model checker.

Thus, although in widespread use, neither the correctness of partial order reduction algorithms, nor the correctness of their implementations can be taken for granted. This is especially problematic since the trust in the correctness of a single model checker is used to justify the confidence in the correctness of the many models that it checks. In order to meet the very strict correctness requirements of model checking algorithms, we implement and formally verify a partial order reduction algorithm.

In previous work [6], we have presented the CAVA model checker, a fully verified and executable LTL model checker à la SPIN. The verification was done with the proof assistant Isabelle/HOL [27] and covers everything from the correctness of the algorithms down to the implementation. Due to its LCF-like architecture, Isabelle/HOL is more trustworthy than a large unverified implementation like SPIN. This paper now adds the following contributions:

1. Formalization of a fragment of the modeling language PROMELA
2. Formalization of the static analysis required for partial order reduction
3. Formal abstract correctness proof for ample set partial order reduction
4. Verified implementation and integration into the CAVA model checker
5. Development of reusable libraries for automata and trace theory

This results in what we believe to be the first formally verified and executable implementation of partial order reduction, addressing both of the issues mentioned earlier. The verification is carried out completely in Isabelle/HOL, such that the correctness of the model checker only depends on the correctness of Isabelle/HOL. This integration avoids logical gaps that may arise when manually composing the results of different verification tools. Most importantly, we now have a formally verified reference implementation that can deal with many models that would be infeasible without partial order reduction. This improves its usefulness for testing other model checkers. To the best of our knowledge, there has been only one other attempt at formalizing partial order reduction [5]. However, it does not cover the reduction algorithm and is restricted to a specific fairness assumption.

The rest of the paper is organized as follows. In section 2, we cover theoretical aspects of partial order reduction and elaborate on our choice of algorithm. In section 3, we report on our Isabelle/HOL formalization. In section 4, we test the reduction effectiveness of our implementation. Finally, in section 6, we give conclusions and future research directions.

This paper is an extended version of [4]. It includes more details about the formalization of the abstract correctness proof of partial order reduction in section 3.6. There is also the new section 3.9 describing the architecture of the CAVA model checker. Finally, we added section 5. It describes a counterexample for the invalid lemma used in the correctness proof of dynamic partial order reduction with on-the-fly model checking.

2 Theory

Figure 1 illustrates the basics of partial order reduction. In regular model checking, the system automaton 'S' is derived from the system and used as input for the model checker together with the formula 'φ'. The model checker then determines if the system automaton satisfies the property expressed by the formula ($\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi$). When using partial order reduction, a reduction algorithm obtains a reduced system automaton 'R' from the system instead, which fulfills certain *reduction conditions*. These conditions imply stuttering equivalence between the language of the system automaton and that of the reduced system automaton

($\mathcal{L}\ S \approx \mathcal{L}\ R$). Since properties expressed by next-free LTL formulae are stuttering-invariant [29], using the reduced system automaton instead of the system automaton when model checking yields the same result ($\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi \iff \mathcal{L}\ R \subseteq \mathcal{L}\ \varphi$).

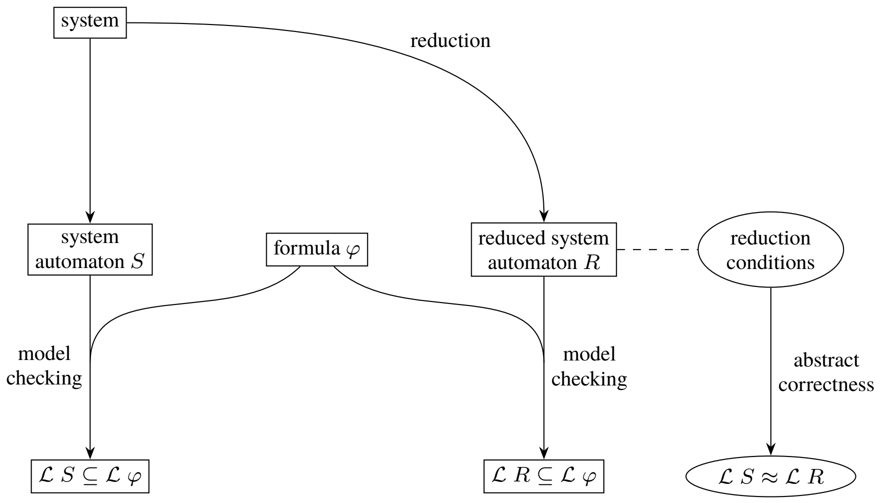
system
reduction
system automaton S
formula φ
reduced system automaton R
reduction conditions
model checking
model checking
abstract correctness
$\mathcal{L}\ S \subseteq \mathcal{L}\ \varphi$
$\mathcal{L}\ R \subseteq \mathcal{L}\ \varphi$
$\mathcal{L}\ S \approx \mathcal{L}\ R$

Fig. 1: Partial Order Reduction Overview. A reduction algorithm obtains the reduced system automaton 'R', which is then used as an input of the model checker instead of the system automaton 'S'. The reduction algorithm guarantees that the reduced system automaton fulfills certain reduction conditions, from which one can prove stuttering equivalence between the two languages. This implies that the result of the model checker is not affected by the reduction.

This is a very abstract description of partial order reduction. In actual implementations, the reduced system automaton may be represented implicitly, and the reduction algorithm may be merged with the model checking algorithm. However, this view allows us to identify the three major tasks involved in developing a verified implementation of partial order reduction:

1. Reduction algorithm correctness: The automaton produced by the reduction algorithm fulfills the reduction conditions.
2. Abstract correctness: If an automaton fulfills the reduction conditions, its language is stuttering equivalent to that of the system automaton.
3. Implementation and verification of the reduction algorithm.

Unlike our formalization, [5] only covers the second task. This means there is no input language, no static analysis, no reduction algorithm, no implementation, and no executable model checker. Furthermore, it only covers the case where a certain fairness assumption is met, which simplifies the abstract correctness proof. In absence of other formalizations, we believe that our work is a significant contribution over the existing body of research.

2.1 Reduction Conditions

Both the reduction algorithm and the abstract correctness are built around the reduction conditions, making them the main object of interest when dealing with partial order reduction. We chose to implement an algorithm based on the ample set method and chose the reduction conditions accordingly. Let '$\text{en}\ q$' be the set of enabled actions at state 'q' of the system automaton (*enabled set*). Let '$\text{ren}\ q$' be the set of enabled actions at state 'q' of the reduced system automaton (*ample set*) Let '$\text{ex}\ a\ q$' be the successor of state 'q' after executing action 'a' ('ex' is called *execution function*). This way, the pair '(en, ex)' represents the system automaton, while '(ren, ex)' represents the reduced system automaton. The set of finite paths executable at state 'q' of the system automaton '$\text{paths}\ q$' is defined in terms of 'en' and 'ex'. For a more detailed description of the system definitions, see section 3.4. With these prerequisites, we define the following reduction conditions:

subset	$\forall q.\ \text{ren}\ q \subseteq \text{en}\ q$
nonempty	$\forall q.\ \text{ren}\ q \subset \text{en}\ q \implies \text{ren}\ q \neq \{\}$
independent	$\exists$ independence relation $I.\ \forall q\ w.\ \text{ren}\ q \subset \text{en}\ q \implies$ $w \in \text{paths}\ q \implies \text{ren}\ q \cap \text{set}\ w = \{\} \implies I\ (\text{ren}\ q)\ (\text{set}\ w)$
wellfounded	$\exists$ well-founded relation $R.\ \forall q\ a.\ \text{ren}\ q \subset \text{en}\ q \implies$ $a \in \text{ren}\ q \implies R\ (\text{ex}\ a\ q)\ q$
invisible	$\forall q.\ \text{ren}\ q \subset \text{en}\ q \implies \text{ren}\ q \subseteq \text{invisible}$

Condition **subset** states that the reduced system automaton is a subautomaton of the system automaton and is usually not stated explicitly in the literature. Condition **nonempty** states that the reduction algorithm must not omit all of the actions at any state. Condition **independent** requires that all the actions that are executed after reaching some state but before an action from the ample set at this state are *independent* of all the actions in this ample set. Condition **wellfounded** requires that every cycle in the system automaton contains at least one state where no reduction is performed. Condition **invisible** states that when a proper reduction takes place, the ample set cannot contain any actions that are *visible* to the formula. Conditions **nonempty**, **independent**, and **wellfounded** correspond to conditions C0, C1, and C2 in [5, pages 268, 269]. Condition **invisible** corresponds to condition C3$'$ in [28, page 50]. Note that even though the reduction conditions are similar, our formalization is not based on [5].

2.2 Reduction Algorithm

These conditions are very abstract, so there are still many choices to be made with respect to the actual reduction algorithm. We originally planned to verify dynamic partial order reduction with on-the-fly model checking [28], but soon encountered difficulties. Dynamic partial order reduction detects cycles during the emptiness check in order to ensure condition **wellfounded**. This tight integration with the emptiness check has led to bugs in the past [9]. When used with on-the-fly model checking, this integration also extends to the product construction, effectively turning the whole system into one monolithic algorithm. It also introduces a mismatch since an algorithm that conceptually works on a system automaton is now used with a product automaton, requiring complicated reasoning. And indeed, during our effort of formalizing the proof given in [28], we discovered a counterexample for one of the lemmata used in this proof. This counterexample is based on the fact that, when exploring the product automaton, different instances of the system automaton appearing in the product automaton may be reduced differently. A detailed description can be found in section 5. Note that this, while refuting the lemma, does not necessarily invalidate the correctness theorem, only this particular proof thereof. However, despite investing a significant amount of time, we were unable to find an alternative proof as it seems that the reasoning required is more complex than anticipated in the original paper.

We chose to implement a static partial order reduction [10] algorithm instead, which avoids these problems of the dynamic approach. It ensures condition **wellfounded** by performing some static analysis initially, identifying a set of *sticky* edges which break every cycle in the control flow graph. Static partial order reduction is much more modular, making it possible to verify the reduction algorithm independently of the product construction and the emptiness check. This way, we were able to simply add the reduction algorithm as a preprocessing step to the existing CAVA model checker, enabling reuse of existing optimizations.

The reduction algorithm itself is similar to the one used in SPIN [8]. The basic idea is to take the set of enabled actions of each process at some state as a candidate for an ample set. For each candidate, an over-approximation of the reduction conditions is tested. If no candidate satisfies the conditions, the state is fully expanded, that is, no reduction is performed.

For instance, our approximation checks that, in order to be used as an ample set, the actions of a process must be independent of all actions of other processes. Moreover, it is checked that no additional action of this process can be enabled as a consequence of executing actions of other processes. Thus, only independent actions of other processes can be executed before an action of the ample set, which implies condition **independent**.

3 Formalization

Our formalization contains all three of the tasks outlined in section 2. We integrated our implementation into the CAVA model checker, which was published previously [6, 7]. Since then, various features have been added to this model checker. For instance, it now supports using PROMELA as an input language [25]. Furthermore, the library for automata has been updated [20, 14] and a new framework for depth-first search algorithms has been formalized [18]. Also, an alternative algorithm for deciding language emptiness of Büchi automata based on Gabow's strongly-connected components algorithm has been implemented [16]. In order to make all of these changes possible, the architecture of the CAVA model checker was improved to be more modular and extensible (see section 3.9). However, the focus of this paper is on the implementation and verification of the partial order reduction algorithm.

In this section, we give some technical background regarding the tools that were used as well as a high-level overview of the formalization. We also describe certain noteworthy aspects of the formalization in isolated detail. The full formalization is available at `https://cava.in.tum.de/CAVA_POR`.

3.1 Isabelle/HOL

Isabelle/HOL [27, 26] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalizations done in Isabelle/HOL are trustworthy for two reasons. Firstly, Isabelle's LCF architecture guarantees that all proofs are checked using a very small logical core which is rarely modified but tested extensively over time. This reduces the trusted code base to a minimum. Secondly, bugs in the core rarely lead to accidentally proving false propositions. Bugs that have large effects are easily caught, while the limited applicability of bugs with small effects is unlikely to coincide with a logical mistake in the large-scale structure of the proof.

Isabelle/HOL notation resembles standard mathematical notation with just a few differences. For instance, as in functional programming, functions are usually curried in HOL. This means that instead of '$f :: A \times B \to C$' with application syntax '$f(x, y)$', we have '$f :: A \to B \to C$' with application syntax '$f\ x\ y$'.

3.2 Refinement Framework

We want our model checker and the partial order reduction algorithm contained therein to be executable. 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 implementation details obfuscate the main ideas of the proof. A standard approach to this problem is stepwise refinement [1], which modularizes the correctness proof: One starts with an abstract version of the algorithm and then refines it in correctness-preserving steps to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program, replace abstract mathematical specifications by concrete algorithms, 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. This approach separates the correctness proof of the algorithm, which focuses on the main algorithmic ideas, from the correctness proof of the implementation, where the proof of each refinement step focuses on a specific implementation detail, not caring about the overall correctness property.

In Isabelle/HOL, stepwise refinement is supported by the Refinement Framework [19, 12, 13, 15] and the Isabelle Collection Framework [17, 11]. The former framework implements a refinement calculus [1] based on a nondeterminism monad [30], and the latter provides a library 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.

3.3 Basics

The most basic concept needed for nearly all parts of the formalization is that of *sequences*. With HOL being very similar to functional programming languages like SML or Haskell, the standard library already includes extensive support for *finite sequences* via the type 'α list $=$ Nil | Cons α (α list)'. For *infinite sequences*, the type 'α word' is used, which is simply a type synonym for '$\mathbb{N} \to \alpha$'.

We also use the library Coinductive [21] which formalizes lazy lists using codatatypes [2]. It provides the type 'α llist', which models both finite and infinite sequences. This is useful for selecting subsequences of infinite lists that can be either finite or infinite. Reasoning about selections and indices of lazy lists required us to significantly extend the library Coinductive.

Another important component needed for partial order reduction is stuttering equivalence and the proof that next-free LTL formulae can only express stuttering-invariant properties. The library Stuttering Equivalence [23] is used for both.

3.4 Systems

Model checkers usually represent systems using the type '$(\mathrm{state} \times \mathrm{state})\ \mathrm{set}$'. Reasoning about partial order reduction requires transitions to be labeled with actions, suggesting the type '$(\mathrm{state} \times \mathrm{action} \times \mathrm{state})\ \mathrm{set}$'. However, this type allows multiple successor states to be reached given a state and an action, making the type a bad fit for the deterministic action model of partial order reduction. This leads to unnecessary wellformedness conditions, inaccessible successor states, and overspecified path predicates. We thus chose the following representation of the system automaton which was already referred to in section 2.1:

$$\text{en} :: \text{state} \rightarrow \text{action set} \tag{1a}$$
$$\text{ex} :: \text{action} \rightarrow \text{state} \rightarrow \text{state} \tag{1b}$$
$$\text{init} :: \text{state set} \tag{1c}$$

Here, ‘en’ is the set of enabled actions at a state (*enabled set*), ‘ex’ is the function that, given an action, maps each state to its successor state (*execution function*), and ‘init’ is the *set of initial states*.

This representation allows paths to be introduced in a straightforward way via the inductively defined set ‘$\text{paths} :: \text{state} \rightarrow \text{action list set}$’:

$$[] \in \text{paths}\ p \tag{2a}$$
$$a \in \text{en}\ p \implies w \in \text{paths}\ (\text{ex}\ a\ p) \implies a \mathbin{\#} w \in \text{paths}\ p \tag{2b}$$

Inductive definitions specify the smallest sets that satisfy the given rules. Equivalently, they specify the sets containing those elements whose membership can be derived using the given rules. These rules can be declared as safe introduction rules, so that whenever Isabelle/HOL encounters proof obligations of the form ‘$[] \in \text{paths}\ p$’ or ‘$a \mathbin{\#} w \in \text{paths}\ p$’, it can automatically split them into simpler goals or discharge them completely.

We prove an additional rule for the append operator on lists:

$$u \in \text{paths}\ p \implies v \in \text{paths}\ (\text{fold ex}\ u\ p) \implies u \mathbin{@} v \in \text{paths}\ p \tag{3}$$

Note how ‘fold’ lifts the execution function ‘$\text{ex} :: \text{action} \rightarrow \text{state} \rightarrow \text{state}$’ from single actions to sequences of actions ‘$\text{fold ex} :: \text{action list} \rightarrow \text{state} \rightarrow \text{state}$’. Also note how this rule generalizes rule 2b.

Together, rules 2a, 2b, and 3 form a set of introduction rules that break down most goals automatically. For instance, the goal ‘$u \mathbin{@} a \mathbin{\#} v \in \text{paths}\ p$’ gets transformed into three subgoals:

$$u \in \text{paths}\ p \tag{4a}$$
$$a \in \text{en}\ (\text{fold ex}\ u\ p) \tag{4b}$$
$$v \in \text{paths}\ (\text{ex}\ a\ (\text{fold ex}\ u\ p)) \tag{4c}$$

Compared to the formalization using the type ‘$(\text{state} \times \text{action} \times \text{state})\ \text{set}$’, this automates proofs significantly. In some cases, proofs comprised of 50 to 100 lines become one-liners. We have proven many more rules about this system formalization, making it a useful addition to the Cava automata library.

3.5 Trace Theory

In order to formalize partial order reduction, we need the concept of *independent* actions, which can be executed in any order without changing the result or enabling or disabling each other. Trace theory [22] lifts this notion of commutable items to that of *equivalent* ($\equiv_I$) sequences, which is needed in the abstract correctness proof.

Finite sequences are equivalent if they differ by a finite number of commutations of independent actions. Using equivalence on finite sequences, it is possible to define equivalence on infinite sequences via a series of definitions [28, page 41]. Unfortunately, these definitions are not easily generalized to lazy lists, so we decided to work with separate types and definitions for finite and infinite sequences.

Formalizing the necessary parts of trace theory took significant effort due to the large number of theorems. There are also some theorems that look simple but are difficult to prove, for instance:

$$w_1 \equiv_I w_2 \iff u \mathbin{@} w_1 \mathbin{@} v \equiv_I u \mathbin{@} w_2 \mathbin{@} v \tag{5}$$

The left to right direction can be proven via rule induction on the transitive structure of '$\equiv_I$'. Doing the same for the right to left direction results in an unprovable induction step. It was necessary to prove the following lemmata:

$$w_1 \equiv_I w_2 \implies \text{remove1 } c\, w_1 \equiv_I \text{remove1 } c\, w_2 \tag{6a}$$

$$u \mathbin{@} w_1 \equiv_I u \mathbin{@} w_2 \implies w_1 \equiv_I w_2 \tag{6b}$$

$$w_1 \equiv_I w_2 \implies \text{rev } w_1 \equiv_I \text{rev } w_2 \tag{6c}$$

Here, '$\text{remove1 } c\ w$' removes the first occurrence of 'c' from the sequence 'w', and '$\text{rev } w$' reverses the sequence 'w'. Lemma 6a uses 'remove1' to avoid the fact that rule induction does not work with modified assumptions. We use lemma 6a to prove lemma 6b via reverse induction on the sequence 'u'. Lemma 6c is proven via rule induction and with lemma 6b, it completes the proof of theorem 5.

We also had to define some concepts specific to partial order reduction. For instance, the predicate specifying that the first occurrence of a symbol in a sequence is independent of all symbols before it. In the end, the formalization of the relevant aspects of trace theory required about as much proof text as the formalization of the abstract correctness proof itself.

3.6 Abstract Correctness

In this section, we discuss the part of the formalization dealing with the abstract correctness proof of ample set partial order reduction. Assume that 'S' is a system automaton and 'R' is a reduced system automaton such that the reduction conditions introduced in section 2.1 hold. Then, the abstract correctness theorem states that the languages of 'S' and 'R' are stuttering equivalent:

$$\mathcal{L}\ S \approx \mathcal{L}\ R \tag{7}$$

The proof of this theorem required about 1000 lines of formal proof text including dozens of lemmata. Its structure is similar to that of the informal proof [28].

However, we present the formalization of a lemma [28, Theorem 3.11] in detail and highlight the differences between the formal and the informal proof. Informally, the lemma states that, given an infinite sequence in the system automaton, it is possible to find a corresponding sequence in the reduced system automaton. We would like to convey an idea of what the formal proof looks like without going into every detail of it.

To do so, we need some definitions and some notation. We use '$\frown$' to denote concatenation of a finite sequence and an infinite sequence. The constant 'Ind' lifts the independence relation 'I' to sets. The operators '$=_F$' and '$=_I$' denote equivalence between finite and infinite sequences, respectively.

First, we construct an arbitrarily long but finite sequence in the reduced system automaton by transcribing longer and longer prefixes of the infinite sequence 'v' in the system automaton. In order to do so, we inductively define a predicate that describes a valid state during this construction process where a prefix of the sequence in the system automaton has already been processed. We use the command '**inductive**' to define the constant 'reduced_run' as the least predicate that satisfies the rules init, absorb, and extend.

inductive reduced_run :: "state ⇒ action list ⇒ action word ⇒ action list ⇒
  action list ⇒ action list ⇒ action list ⇒ action word ⇒ bool" **where**
  init: "reduced_run q [] v [] [] [] [] v" |
  absorb: "reduced_run $q\ v_1$ $([a] \frown v_2)$ $l\ w\ w_1\ w_2\ u$ $\Longrightarrow$ $a \in$ set l $\Longrightarrow$
    reduced_run q $(v_1$ @ $[a])$ v_2 (remove1 $a\ l$) $w\ w_1\ w_2\ u$" |
  extend: "reduced_run $q\ v_1$ $([a] \frown v_2)$ $l\ w\ w_1\ w_2\ u$ $\Longrightarrow$ $a \notin$ set l $\Longrightarrow$
    b @ $[a] \in R$.paths (fold ex $w\ q$) $\Longrightarrow$ Ind $\{a\}$ (set b) $\Longrightarrow$ set $b \subseteq$ invisible $\Longrightarrow$
    $b =_F b_1$ @ b_2 $\Longrightarrow$ $[a] \frown b_1 \frown u' =_I u$ $\Longrightarrow$ Ind (set b_2) (range u') $\Longrightarrow$
    reduced_run q $(v_1$ @ $[a])$ v_2 $(l$ @ $b_1)$ $(w$ @ b @ $[a])$ $(w_1$ @ b_1 @ $[a])$ $(w_2$ @ $b_2)$ u'"

The predicate 'reduced_run' uses the following parameters:

q	initial state for 'v'
v_1	prefix of 'v' that has been processed so far
v_2	suffix of 'v' that has not yet been processed
l	actions that have been appended to 'w_1' but did not yet appear in 'v_1'
w	sequence in the reduced system automaton constructed so far
w_1	possibly visible part of 'w'
w_2	invisible part of 'w'
u	possible continuation of 'w'

This predicate specifies that the state of the construction where both the sequence in the system automaton and the one in the reduced system automaton are empty is valid (init). It also specifies how one can extend a valid construction state by adding a step in the system automaton and a sequence of corresponding steps in the reduced system automaton (absorb and extend).

Next, we present some theorems that can be proven about this predicate using Isabelle syntax, which should be fairly intuitive. After the theorem name, assumptions are stated using the '**assumes**' directive and the conclusion is stated using the '**shows**' directive. The notation '**obtains** x **where** P' denotes that the theorem proves an existential statement of the form '$\exists\, x.\ P$'. We first prove that certain invariants hold at each point of the construction:

lemma reduced_run_invariants:
  assumes "reduced_run $q\ v_1\ v_2\ l\ w\ w_1\ w_2\ u$"
  shows "$w \in R$.paths q" "$v_2 =_I l \frown u$" "v_1 @ $l =_F w_1$" "$w =_F w_1$ @ w_2"
    "filter visible w_1 = filter visible w" "set $w_2 \subseteq$ invisible"
    "Ind (set w_2) (range u)" "length $v_1 \leq$ length w_1" "length $v_1 \leq$ length w"

We also prove that the construction can always be extended:

lemma reduced_run_step :
  assumes "$q \in$ reachable" "$v_1 \frown [a] \frown v_2 \in$ runs q"
  assumes "reduced_run $q\ v_1$ $([a] \frown v_2)$ $l\ w\ w_1\ w_2\ u$"
  obtains $l'\ w'\ w_1'\ w_2'\ u'$

```
    where "reduced_run $q$ $(v_1$ @ $[a])$ $v_2$ $l'$ $(w$ @ $w')$ $(w_1$ @ $w_1')$ $(w_2$ @ $w_2')$ $u'$"
```

Proving reduced_run_invariants and reduced_run_step required a lot of effort as the informal proof only provided a rough sketch of the arguments underlying these proofs.

With these lemmata proven, we can now show our lemma [28, Theorem 3.11]:

```
lemma reduction_word:
  assumes "$q \in$ reachable" "$v \in$ runs $q$"
  obtains $u$ $w$
  where "$w \in R$.runs $q$" "$v =_I u$" "$u \preceq_I w$"
    "lfilter visible (inf_llist $u$) = lfilter visible (inf_llist $w$)"
proof –
  def $P \equiv$ "$\lambda$ $k$ $w$ $w_1$. $\exists$ $l$ $w_2$ $u$. reduced_run $q$ (prefix $k$ $v$) (suffix $k$ $v$) $l$ $w$ $w_1$ $w_2$ $u$"
  obtain $w$ $w_1$ where "$\forall$ $k$. $P$ $k$ $(w$ $k)$ $(w_1$ $k)$" "chain $w$" "chain $w_1$" ...
  proof (rule chain_construct_2'[of $P$])
    show "$P$ 0 [] []" ...
  next
    fix $k$ $w$ $w_1$
    assume "$P$ $k$ $w$ $w_1$"
    show "$\exists$ $w'$ $w_1'$. $P$ $(Suc\, k)$ $w'$ $w_1'$ $\wedge$ $w \leq w'$ $\wedge$ $w_1 \leq w_1'$" ...
    show "$k \leq$ length $w$" "$k \leq$ length $w_1$" ...
  qed rule
  show ?thesis
  proof
    show "limit $w \in R$.runs $q$" ...
    show "$v =_I$ limit $w_1$" ...
    show "limit $w_1$ $\preceq_I$ limit $w$" ...
    show "lfilter visible (inf_llist (limit $w_1$)) = lfilter visible (inf_llist (limit $w$))"
      ...
  qed
qed
```

Here, 'lfilter' is the filter function on lazy lists and 'inf_llist' converts an infinite sequence to a lazy list. Proofs are enclosed in '**proof** ... **qed**' blocks, with '**next**' separating subgoals. Inside these blocks, arbitrary propositions can be proven. Existential statements use the form '**obtain** ... **where**'. Local definitions can also be made via '**def**'. The directives '**fix**', '**assume**', and '**show**' are used to work with universally quantified variables, assumptions, and goals, respectively. Once all goals have been discharged via '**show**', the proof block can be closed via '**qed**'. Note that we used '...' to signify that a proof was omitted at a certain position.

In the proof, we have to show that there exists an infinite sequence 'w' with the required properties in the reduced system automaton. While this step is almost completely skipped in the informal proof, the formal one forces us to consider it rigorously. Lemma reduced_run_step guarantees that for any number of steps that were already taken, another step can be taken, extending the sequence in the process. Intuitively, such a theorem can be applied "infinitely often" to obtain an infinite sequence, but this is not logically sound. Performing a step like this in a formal proof requires precise reasoning and in our case the use of Hilbert's epsilon operator in lemma chain_construct_2'. This lemma turns the ability to perform an arbitrary number of steps into an infinite chain of finite sequences where each sequence is a prefix of the one following it. This property is stated via the constant 'chain'. The constant 'limit' is then used to derive the uniquely determined infinite sequence from this chain.

We believe that these difficulties do not point to a shortcoming of formal logic or the particular system we are using. Instead, we think that situations like this one point to areas where it became customary to use sloppy reasoning in informal proofs, possibly leading to mistakes or overlooked side conditions. For instance, it is often not made clear in which way variables depend on each other or what guarantees that an infinite sequence can actually be constructed from an infinite set of finite sequences. Formal proofs point out required side conditions like the fact that the infinite concatenation of these finite sequences needs to be infinite. It also brought attention to the fact that many concepts need to be defined on both finite and infinite sequences and that they need to correspond to each other in a specific way.

The formal proof constitutes both a certificate of the theorem's correctness as well as a detailed documentation of the reasoning used to prove it. As mentioned in sections 3.3, 3.4, and 3.5, a large amount of foundational work was required in order to formally prove the abstract correctness theorem.

3.7 The SM Language

In order to implement an executable reduction algorithm, we require a concrete modeling language. We use a simple fragment of PROMELA that is expressive enough to model interesting examples. We call this fragment the *SM language* (simple modeling language).

A program in this language consists of a set of processes, each of which is described using a guarded command language. Each process has a set of local variables and communication between processes is modeled via global variables. A configuration of the system consists of a valuation of the global variables and a list of process configurations, where a process configuration consists of a command and a valuation of the local variables.

Most features of PROMELA are either contained in the SM language or can be expressed directly using global variables. The behavior of channels, small integers, arrays, dynamic processes, and process priorities has to be emulated via constructions using global variables.

We specify a structural operational semantics that establishes a control flow graph where the nodes are commands and the edges are labeled with *local actions*. A local action can be a guarded assignment, a test, or the skip action. Each local action is assigned an enabledness check and an effect function on the local and global variables.

The system semantics describes a step relation between configurations by nondeterministically picking a process from a configuration, following an edge in the control flow graph from the process' command that is labeled with an enabled local action, and applying the effect of the local action to the local and global state. To ensure that all runs of the system are infinite, we apply a stuttering extension, that is, if there is no process with an enabled action, the system may take a step that does not change the configuration.

Since we want to use the SM language in an LTL model checker, we need to define *atomic propositions* and their connection to the system states. In our case, atomic propositions are simply expressions in the SM language that contain only global variables. Then, we define the *interpretation function* to map each state to the set of expressions that evaluate to a non-zero value in this state. Finally, we define the language of a program as the set of infinite sequences of sets of atomic propositions that correspond to infinite runs of the program:

$$\mathcal{L} :: \text{program} \rightarrow \text{exp set word set} \tag{8}$$

We define a *global action* to consist of a process ID and a control flow graph edge. The process ID is the position of the associated process in the list of all processes. A global action is enabled if the associated process exists, the control flow graph edge is consistent with the current command of the associated process, and the corresponding local action is enabled. Execution of a global action transforms the state of the associated process and the global variables according to the corresponding local action.

3.8 Reduction Algorithm

We make some approximations similar to those made in Spin in order to define an efficiently executable function which selects an ample set for a configuration, thereby implementing the reduction algorithm. We call an action *statically enabled* if it occurs on a control flow graph edge consistent with the current command of its process in the configuration. This overapproximates the set of enabled actions by ignoring the enabledness conditions.

Similar to Spin, candidates for ample sets are the sets of enabled actions of each process. We make a crude approximation and allow a nonempty set of enabled actions of a process as an ample set, if (1) there is no statically enabled action of the process that reads or writes global variables, and (2) none of the enabled actions corresponds to a sticky edge in the control flow graph. Here, (1) is a way of guaranteeing condition **independent** (see section 2.1), and (2) is the condition imposed by static partial order reduction (see section 2.2).

We implemented and verified an algorithm based on depth-first search which computes the set of sticky edges before the model checking phase. This algorithm starts with the set of edges labeled with actions containing global variables and extends it to a feedback arc set on the control flow graphs of the processes. For this task, we used the DFS Framework [18], which simplifies the implementation and verification of efficient DFS-based algorithms.

We define the reduced system automaton based on this ample function and prove that all of the reduction conditions from section 2.1 are fulfilled. This allows us to invoke the abstract correctness theorem to obtain stuttering equivalence between the language of the system automaton and that of the reduced system automaton. Together with the assumption that the formula is next-free, this implies that using the reduced system automaton for model checking instead of the system automaton does not change the result.

3.9 Architecture of the Cava Model Checker

Since the previous publication on the Cava model checker [6], many improvements have been made. Following an overhaul of the automata library [20, 14], the architecture of the model checker has been generalized to a point where it can be considered a generic framework for assembling formally verified LTL model checkers. It is this architecture that we want to describe in this section. Figure 2 shows the data flow of the Cava model checker. We use *generalized Büchi graphs* to represent the product automaton, which are generalized Büchi automata over the unit alphabet. As the alphabet is not needed to check for emptiness, this

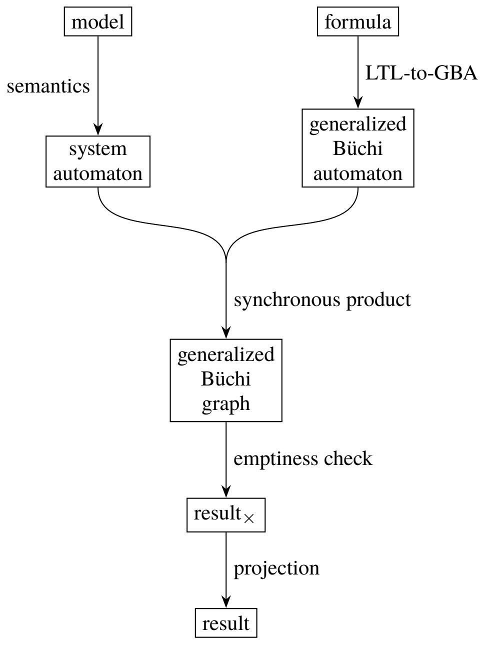

Fig. 2: Architecture of the Cava Model Checker. The boxes show the data, and the arrows show the operations of the model checker. The input is a model and a formula. The semantics of the modeling language interprets the model as a system automaton, and the LTL-to-GBA conversion creates a generalized Büchi automaton from the formula. Then, the synchronous product of the two is formed, and checked for emptiness. The result is either emptiness or an accepting run of the product automaton. In the latter case, the run is projected back to the system automaton.

avoids forcing implementations to include alphabet information in their product automaton implementation.

In order to get a flexible framework, we specify the data and components on an abstract level. Automata are specified as relations over nodes and a function assigning sets of atomic propositions to nodes. The specification is parameterized over the types of nodes and atomic propositions. We do not specify an abstract model, but start with the system automaton.

The abstract components are nondeterministic specifications of the expected behavior. For example, we specify the emptiness check 'echk_abs' to return 'UNSAT' together with some accepting run '$r_\times$' of the generalized Büchi graph 'G' if there is one, and 'SAT' otherwise:

$$\begin{aligned} &\textbf{case}\ \text{echk\_abs}\ G\ \textbf{of} \\ &\quad \text{SAT} \Rightarrow G.\text{runs} = \{\}\ | \\ &\quad \text{UNSAT}\ r_\times \Rightarrow r_\times \in G.\text{runs} \end{aligned} \tag{9}$$

We assemble the abstract components to form the abstract model checker 'cava_abs' and show its correctness:

$$\begin{aligned} &\textbf{case}\ \text{cava\_abs}\ S\ \varphi\ \textbf{of} \\ &\quad \text{SAT} \Rightarrow \text{intp}^{\backprime}\ S.\text{runs} \subseteq \mathcal{L}\ \varphi\ | \\ &\quad \text{UNSAT}\ r \Rightarrow r \in S.\text{runs} \wedge \text{intp}\ r \notin \mathcal{L}\ \varphi \end{aligned} \tag{10}$$

For a system automaton 'S' and an LTL formula 'φ', the model checker returns 'SAT' if all runs of the system satisfy the formula, and '$\mathrm{UNSAT}\ r$' if 'r' is a run that does not satisfy the formula. Note that LTL formulae are independent of the system: They specify valid sequences of sets of atomic propositions, while a run of the system is a sequence of system states. The function 'intp' maps a run to the sequence of sets of atomic propositions that hold at the states of the run, and '$\mathrm{intp}`$' is its extension to sets of runs. Note that '$\mathrm{intp}`\ S.\mathrm{runs} = \mathcal{L}\ S$'.

Next, we assume that we have implementations of the components that match the implementations of the data. Formally, we fix *refinement relations* to relate the implementation of data to its abstract representation, and assume that the components' implementations *refine* the abstract components' specifications. This means that for related arguments, the implementation and the abstract component return related results. For example, we assume refinement relations '$\mathrm{gbg\_rel}$' and '$\mathrm{resx\_rel}$' for generalized Büchi graphs and results of the emptiness check, and an implementation '$\mathrm{echk\_impl}$', such that:

$$(\mathrm{echk\_impl}, \mathrm{echk\_abs}) \in \mathrm{gbg\_rel} \to \mathrm{resx\_rel} \tag{11}$$

Then, we assemble the assumed implementations of the components to a model checker implementation '$\mathrm{cava\_impl}$', and show that it refines the abstract model checker:

$$(\mathrm{cava\_impl}, \mathrm{cava\_abs}) \in \mathrm{sa\_rel} \to \mathrm{res\_rel} \tag{12}$$

Here, '$\mathrm{sa\_rel}$' relates the system automata implementation to the abstract system automata, and '$\mathrm{res\_rel}$' relates projected results.

Instantiating the above with actual implementations of the components and the data yields an executable model checker and its correctness theorem. This approach has several advantages. Firstly, the components of the model checker can be developed separately, as long as they match the implementations of the passed data. Secondly, the CAVA Automata Library [14] provides efficient implementations of the required automata types, which can be conveniently used by the components. Finally, components can easily be added or exchanged. For example, the current CAVA model checker supports two emptiness check algorithms, one based on nested DFS, and the other based on SCCs, which can be selected by a configuration option. Adding another algorithm amounts to proving that the algorithm refines the abstract emptiness check, and then adding a new configuration option to CAVA. In particular, it does *not* require changing other components or the overall correctness proof.

Finally, using this refinement-based approach allows for the seamless integration of many implementation techniques required to design an efficient model checker:

– When instantiating CAVA with a modeling language, the modeling language has a semantics which maps a model to an abstract system automaton. Moreover, we compile a model to a system automaton implementation, usually a successor function, which maps configurations to lists of successor configurations. Showing compiler correctness amounts to showing that the abstract system automaton is related to the concrete one. Then, the CAVA correctness theorem (theorem 12) implies that the result of the implementation is related to the abstract system automaton, that is, the semantics of the model.
– The states of the product automaton are constructed lazily. This saves memory if a counterexample is found before the whole state space is explored, as the unexplored parts of the state space do not occupy memory. This only affects the product construction component and the generalized Büchi graph that is implemented by its successor function.
– The alphabets of the automata are sets of atomic propositions. However, these sets have exponential size in general, so storing them explicitly is not efficient. Instead, the automaton constructed from the LTL formula stores two sets 'P' and 'N' of atomic

propositions, representing all sets 'A' with '$P \subseteq A \wedge A \cap N = \{\}$'. This representation is naturally generated by many algorithms that convert LTL formulae to automata.
The automaton constructed from the model uses system states to represent the set of all atomic propositions that hold in a state. On product construction, it has to be decided whether the intersection of two sets of atomic propositions, one represented by '(P, N)' and the other represented by a system state 'S' is empty. This can simply be done by evaluating the atomic propositions in 'P' and 'N' on 'S'.

– The result of the emptiness check may contain a counterexample, which is an infinite run of the generalized Büchi graph. Clearly, a direct representation of infinite runs is not possible. However, a common representation is to use a *lasso*, that is, a finite path to an accepting state, and a finite, non-empty loop on this state, which has to contain states from all acceptance classes of the generalized Büchi graph. For a non-empty graph, there is always an accepting path that can be described by a lasso, which can be computed naturally by the emptiness check algorithms.

3.10 Integration of Partial Order Reduction

With all these prerequisites out of the way, we can now integrate partial order reduction into the Cava model checker. We refine the ample function, the execution function, and the interpretation function to efficiently executable implementations. This includes compilation of the model to a more efficient representation. Then, we replace the implementation of the successor function with the ample function. Instantiating the generic infrastructure of the Cava model checker then yields an executable LTL model checker 'cava_por' which uses the reduced system automaton. Combining its correctness theorem with that of abstract partial order reduction and the theorem about stuttering invariance of LTL properties then yields the main theorem of our development:

$$\textbf{case}\ \text{cava\_por}\ S\ \varphi\ \textbf{of}\ \text{SAT} \Rightarrow \mathcal{L}\ S \subseteq \mathcal{L}\ \varphi \mid \text{UNSAT} \Rightarrow \mathcal{L}\ S \nsubseteq \mathcal{L}\ \varphi \qquad (13)$$

This theorem states that the function 'cava_por' decides whether or not the sequences of atomic propositions admitted by runs of the program satisfy the LTL formula. The meaning of this statement only depends on the abstract semantics of the SM language ($\mathcal{L}\ S$) and the abstract semantics of LTL formulae ($\mathcal{L}\ \varphi$). All other parts of the formalization, including partial order reduction, LTL model checking, and implementations, are covered by this machine-checked correctness theorem. Note that the model checker can actually provide a counterexample in case the program does not satisfy the formula. However, we only show the simplified view here as it is easier to understand.

Finally, Isabelle/HOL can generate Standard ML code from the definition of the function 'cava_por'. This code then constitutes a formally verified and executable LTL model checker. A snapshot of this formalization can be found at `https://cava.in.tum.de/CAVA_POR`. We are currently working on integrating the partial order reduction formalization into an up-to-date AFP entry of the Cava model checker, which can be found at `https://www.isa-afp.org`.

We conclude with some statistics about the formalization, which took about 15 person-months and resulted in about 13k lines of theory text being added to the model checker. This includes both definitions and proofs and splits up into 6k lines for abstract partial order reduction and 7k lines for the SM language and the associated program analysis. The size of the whole codebase of the model checker and its libraries is about 140k lines of theory text.

4 Evaluation

We perform some basic sanity checks using systems that admit no reduction and complete sequentialization. As a practical example, we implement a distributed mutual exclusion algorithm called Mulog [24] using the supported Promela fragment. The tested property specifies that at most one process can be in the critical section at any point in time. We perform model checking using both the Cava and the Spin model checkers, both with and without partial order reduction. Figure 3 shows the reduction effectiveness for this algorithm.

n	**Spin**	**Spin***	**Factor Spin**	**Cava**	**Cava***	**Factor Cava**
1	27	27	1	52	52	1
2	2,674	2,004	1.33	5,538	4,284	1.29
3	2,376,180	1,171,578	2.03	5,205,376	2,779,218	1.87

Fig. 3: Reduction Effectiveness for Mulog. Shown are the number of states that were explored during model checking using both the Spin and the Cava model checkers for a given number of processes 'n'. The starred variants indicate where partial order reduction was used. The table also shows the reduction factor that was achieved by each model checker.

Both the Cava and the Spin model checker show a significant reduction in the number of states. The reduction factors are comparable (roughly 1.3 for '$n = 2$' and roughly 2 for '$n = 3$'). The Spin model checker explores fewer states in total (roughly factor 2) and has shorter execution times (roughly factor 400) than the Cava model checker.

We would like to emphasize that in this paper, it is not our goal to compete with Spin. Instead, our focus is on providing a verified and executable reference implementation of partial order reduction. The Spin model checker employs various other optimizations and compilation to C code, while the Cava model checker interprets the semantics of the modeling language. Thus, little insight can be gained by directly comparing execution time and memory consumption. Incorporating these optimizations is orthogonal to partial order reduction and we consider this subject of further research. Due to the modular architecture of the Cava model checker, doing so will not make this contribution obsolete. At this point, it will also be possible to perform a more comprehensive evaluation with multiple example algorithms.

5 Dynamic Partial Order Reduction with On-The-Fly Model Checking

This section presents a counterexample for the invalid lemma mentioned in section 2.2. In [28], the partial order reduction algorithm designed for off-line model checking is modified and used with on-the-fly model checking. When doing off-line partial order reduction, the reduced system automaton is explored via depth-first search and its product with the formula automaton '$\mathcal{B}$' is checked for emptiness. On-the-fly partial order reduction consists of defining the reduced product automaton '$\mathcal{A}'$' directly and checking its emptiness while exploring it via nested depth-first search.

The correctness proof introduces an intermediate automaton 'G'' that is structurally similar to '$\mathcal{A}'$' but fulfills the conditions of the off-line correctness theorems. Together with the formula automaton '$\mathcal{B}$', the following claim is then used to prove correctness:

$$\mathcal{L}\ \mathcal{A}' = \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} \tag{14}$$

However, in this section, we will present an example for which this claim does not hold. This example is adapted from [3, section 8.4], where we originally discovered the problem.

Figures 4 and 5 show the system automaton 'G' and the formula automaton '$\mathcal{B}$', respectively.

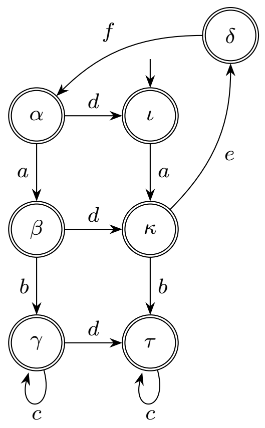

Fig. 4: System Automaton 'G'.

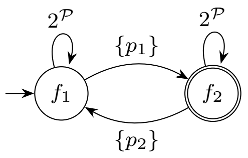

Fig. 5: Formula Automaton '$\mathcal{B}$'.

We define the interpretation function 'intp' as follows:

$$\text{intp } q = \begin{cases} \{p_1\} & \text{if } q = \delta \\ \{p_2\} & \text{if } q = \gamma \\ \{p_2\} & \text{if } q = \tau \\ \{\} & \text{otherwise} \end{cases} \tag{15}$$

With this, we have 'visible $= \{b, e, f\}$'.

We use the independence relation '$I = \{(a, d), (d, a), (b, d), (d, b), (c, d), (d, c)\}$' and define the ample function as follows:

$$\text{ample } (x, y) = \begin{cases} \{d\} & \text{if } x = \alpha \text{ and } (\iota, y) \text{ is not open} \\ \{a\} & \text{if } x = \alpha \text{ and } (\beta, y) \text{ is not open} \\ \text{en } x & \text{otherwise} \end{cases} \tag{16}$$

If more than one condition is met, the topmost valid equation is used. This function fulfills all the necessary conditions.

Figure 6 shows the reduced product automaton '$\mathcal{A}'$' generated by the ample function given above. In this case, the intermediate automaton 'G''' looks exactly like '$\mathcal{A}'$', except that all states are accepting.

In order to create a counterexample, we define the word 'w':

$$w = \{\} \cdot \{\} \cdot \{p_1\} \cdot \{\} \cdot \{\} \cdot \{p_2\}^{\omega} \tag{17}$$

We have '$w \in \mathcal{L}\ G' \wedge w \in \mathcal{L}\ \mathcal{B} \wedge w \notin \mathcal{L}\ \mathcal{A}'$' and thus obtain:

$$\mathcal{L}\ \mathcal{A}' \neq \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} \tag{18}$$

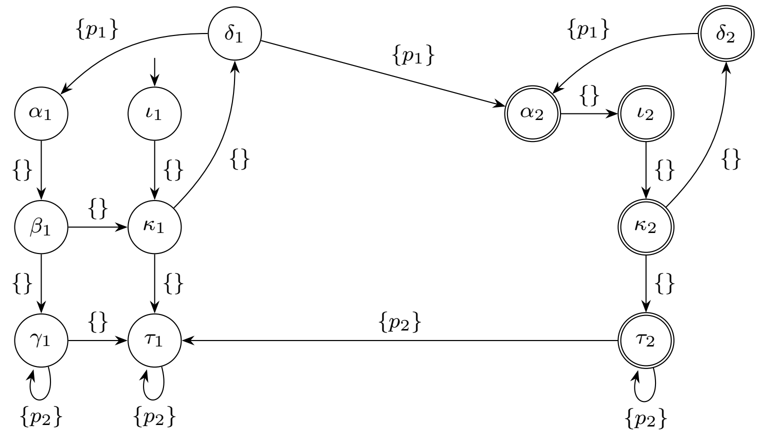

Fig. 6: Reduced product automaton '$\mathcal{A}'$'. We use shorthand state notation. For instance, the state '(α, f_1)' is simply written as 'α_1'.

This contradicts the claim 14 made earlier. However, the proof only needs a weaker version of the claim:

$$\mathcal{L}\ \mathcal{A}' = \{\} \iff \mathcal{L}\ G' \cap \mathcal{L}\ \mathcal{B} = \{\} \tag{19}$$

Our example does not contradict this statement and we do in fact believe that it holds true. Unfortunately, we have not been able to find a proof for this statement.

6 Conclusion

Formal verification is sometimes downplayed as "careful documentation of proven theorems" or "filling in obvious details in proofs". In practice, formal proofs usually involve extensive modeling as well as abstraction, generalization, and simplification. What may seem like trivial completion of the informal proof often involves bridging large gaps and proving omitted corner cases. In this project, it even helped us discover an issue with the correctness proof given in [28]. This demonstrates both the need for and the usefulness of formal verification.

More importantly, we developed a formally verified and executable LTL model checker with partial order reduction. As the verification is machine-checked and covers everything from the abstract algorithm to the generated SML code, this is a very strong correctness guarantee. Our model checker is fast enough to serve as a reference implementation for other model checkers on models of realistic size. This constitutes a much-needed source of trust given the widespread use of partial order reduction together with its history of issues. The formalization can further serve as a detailed description of the theory of partial order reduction and its correctness proof, which is useful since nontrivial gaps were bridged in the proof. We also developed a significant amount of foundational theories that can be reused in other projects. Finally, our work demonstrates that large systems can now be verified using proof assistants via modularization and reuse of existing theories.

Future work consists of extending the SM language to make it more practical, with the ultimate goal of supporting most or all of the features of Promela. It is also possible to find smaller sets of sticky actions by incorporating heuristics about variable increments/decrements [10]. Another way to improve reduction consists of using additional static analysis to find larger independence relations. Finally, there is still room for improvement concerning the implementation, especially via the use of imperative data structures [13].

References

[1] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus. A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[2] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. “Truly Modular (Co)datatypes for Isabelle/HOL”. In: *ITP*. Vol. 8558. LNCS. Springer, 2014, pp. 93–110.

[3] Julian Brunner. “Implementation and Verification of Partial Order Reduction for On-The-Fly Model Checking”. MA thesis. Technische Universität München, July 15, 2014. 83 pp. URL: `http://www21.in.tum.de/~brunnerj/documents/ivporotfmc.pdf`.

[4] Julian Brunner and Peter Lammich. “Formal Verification of an Executable LTL Model Checker with Partial Order Reduction”. In: *NFM*. Springer, 2016, pp. 307–321.

[5] Ching-Tsun Chou and Doron Peled. “Formal Verification of a Partial-Order Reduction Technique for Model Checking”. In: *TACAS*. Vol. 1055. LNCS. Springer, 1996, pp. 241–257.

[6] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: *CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 463–478.

[7] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. “A Fully Verified Executable LTL Model Checker”. In: *Archive of Formal Proofs* (May 2014). Formal proof development. URL: `http://afp.sf.net/entries/CAVA_LTL_Modelchecker.shtml`.

[8] Gerard J. Holzmann. *The SPIN Model Checker. Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003.

[9] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. “On Nested Depth First Search”. In: *SPIN Workshop*. Vol. 32. 1996, pp. 81–89.

[10] Robert Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. “Static Partial Order Reduction”. In: *TACAS*. Vol. 1384. LNCS. Springer, 1998, pp. 345–357.

[11] Peter Lammich. “Collections Framework”. In: *Archive of Formal Proofs* (Nov. 2009). Formal proof development. URL: `http://afp.sf.net/entries/Collections.shtml`.

[12] Peter Lammich. “Refinement for Monadic Programs”. In: *Archive of Formal Proofs* (Jan. 2012). Formal proof development. URL: `http://afp.sf.net/entries/Refine_Monadic.shtml`.

[13] Peter Lammich. “Refinement to Imperative/HOL”. In: *ITP*. Vol. 9236. LNCS. Springer, 2015, pp. 253–269.

[14] Peter Lammich. “The CAVA Automata Library”. In: *Archive of Formal Proofs* (May 2014). Formal proof development. URL: `http://afp.sf.net/entries/CAVA_Automata.shtml`.

[15] Peter Lammich. “The Imperative Refinement Framework”. In: *Archive of Formal Proofs* (Aug. 2016). `http://isa-afp.org/entries/Refine_Imperative_HOL.shtml`, Formal proof development. ISSN: 2150-914x.

[16] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Component Algorithm”. In: *ITP*. Vol. 8558. LNCS. Springer, 2014, pp. 325–340.

[17] Peter Lammich and Andreas Lochbihler. “The Isabelle Collections Framework”. In: *ITP*. Vol. 6172. LNCS. Springer, 2010, pp. 339–354.

[18] Peter Lammich and René Neumann. “A Framework for Verifying Depth-First Search Algorithms”. In: *CPP*. ACM, Jan. 13, 2015, pp. 137–146.

[19] Peter Lammich and Thomas Tuerk. “Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm”. In: *ITP*. Vol. 7406. LNCS. Springer, 2012, pp. 166–182.

[20] Peter Lammisch. “The CAVA Automata Library”. In: *Isabelle Workshop 2014*. May 2014.

[21] Andreas Lochbihler. “Coinductive”. In: *Archive of Formal Proofs* (Feb. 2010). Formal proof development. URL: `http://afp.sf.net/entries/Coinductive.shtml`.

[22] Antoni Mazurkiewicz. “Trace Theory”. In: *Advances in Petri Nets, Part II*. Vol. 255. LNCS. Springer, 1987, pp. 278–324.

[23] Stephan Merz. “Stuttering Equivalence”. In: *Archive of Formal Proofs* (May 2012). Formal proof development. URL: `http://afp.sf.net/entries/Stuttering_Equivalence.shtml`.

[24] Mohamed Naimi, Michel Trehel, and André Arnold. “A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal”. In: *Journal of Parallel and Distributed Computing* 34.1 (1996), pp. 1–13.

[25] René Neumann. “Using Promela in a Fully Verified Executable LTL Model Checker”. In: *VSTTE*. LNCS. Springer, 2014, pp. 105–114.

[26] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[27] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. *Isabelle*. 2014. URL: `http://isabelle.in.tum.de`.

[28] Doron Peled. “Combining Partial Order Reductions with On-the-Fly Model-Checking”. In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64.

[29] Doron Peled and Thomas Wilke. “Stutter-Invariant Temporal Properties are Expressible Without the Next-Time Operator”. In: *Information Processing Letters* 63.5 (1997), pp. 243–246.

[30] Philip Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science* 2 (04 Dec. 1992), pp. 461–493.