Automata Cascades for Model Checking

Luca Geatti

University of Udine, Italy

Abstract

The cascade product is a method for combining two automata in which the first automaton reads symbols from an alphabet Σ, while the second automaton reads symbols from the Cartesian product of Σ and the set of states of the first automaton. This capability of the second automaton to process not only input symbols but also the current state of the first automaton has made the cascade product a fundamental tool in theoretical computer science, with the most notable result being the Krohn–Rhodes Cascade Decomposition Theorem.

In this paper, we propose the use of the cascade product in the context of formal verification, specifically for safety model checking. We show that specifying the undesired behaviors of a system through a cascade of automata enables an approach to model checking that is *incremental with respect to the specification*, where verification proceeds component by component, potentially reusing information obtained from the verification of previous steps. We also demonstrate how to represent a cascade of automata *symbolically*, thereby making it possible to perform symbolic verification of such properties. A proof-of-concept on a set of simple models, evaluated experimentally, indicates that this approach is promising.

Keywords

Model Checking, Safety, Cascade Product, Automata

1. Introduction

The concept of the cascade product $\mathcal{A}_1 \circ \mathcal{A}_2$ of two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is a fundamental notion in theoretical computer science [1, 2]. In this framework, the first automaton operates over an alphabet Σ, while the second automaton processes symbols drawn from the Cartesian product of Σ and the state set of the first automaton. The distinguishing feature of the cascade product–extending the classical notion of the direct product–is that the second automaton transitions from state s to state s' upon reading the pair (σ, q) if and only if the input symbol is σ and the first automaton is in state q. Although initially introduced as a product between semiautomata–that is, automata devoid of initial and final states–the cascade product of automata has been the subject of recent investigation, particularly in relation to the class of recognizable languages [3].

A fundamental result concerning cascades of automata is the Krohn–Rhodes Cascade Decomposition Theorem [4]. Although originally formulated in the language of semigroups, its automata-theoretic variant [5, 1] asserts that every semiautomaton can be decomposed, up to homomorphism, into a cascade of permutation-reset automata. Furthermore, if the semiautomaton is counter-free[1], then the decomposition yields a cascade consisting solely of *two-state* reset automata, rendering permutations unnecessary.

In this paper, we investigate the problem of safety model checking [7] when the undesidered behaviors are expressed through a cascade of automata and examine the advantages this approach offers. We contend that, akin to the case of other applications of the cascade product, treating each component of the cascade individually can yield significant benefits.

First (Section 3), we demonstrate that expressing the undesired behaviors of a system through a cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \cdots \circ \mathcal{A}_n$ enables safety model checking to be performed *incrementally with respect to*

7th International Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis (OVERLAY 2025), October 26, 2025, Bologna, Italy

luca.geatti@uniud.it (L. Geatti)

https://users.dimi.uniud.it/~luca.geatti/ (L. Geatti)

0000-0002-7125-787X (L. Geatti)

[1] It is worth pointing out that the class of counter-free (semi)automata corresponds to the class of languages definable in Linear Temporal Logic (LTL,[6]), and to the class of star-free regular languages.

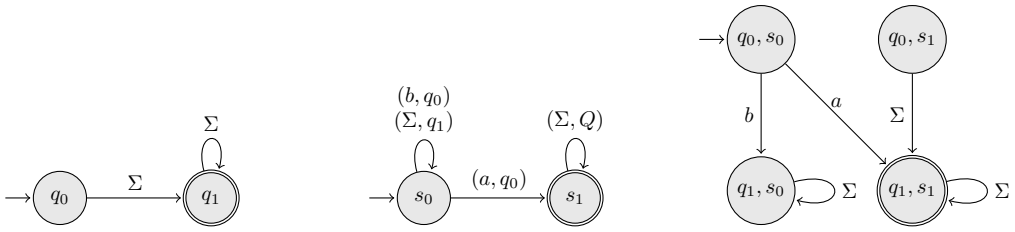

Figure 1: On the left, the automaton $\mathcal{A}_1$ over $\Sigma = \{a, b\}$. In the center, the automaton $\mathcal{A}_2$ over $\Sigma \times Q$, with $Q \coloneqq \{q_0, q_1\}$. On the right, the cascade product $\mathcal{A}_1 \circ \mathcal{A}_2$ over Σ that recognizes the languages $a \cdot \Sigma^*$.

the specification, i.e., starting from the first component and proceeding step by step down the cascade. We show that if a system model M is safe with respect to $A_1 \circ \cdots \circ A_i$ (for some $i \in \{1, \ldots, n\}$), then it is also safe with respect to the entire cascade. This approach has the advantage that verifying a portion of the cascade is generally more efficient than verifying the entire cascade. Furthermore, we show that each verification step can be formulated as an invariant checking problem [8], and that information about the reachable states at step i can be leveraged to accelerate step $i + 1$.

Second (Section 4), we present a *symbolic* modeling for automata cascades, in which the cascade is represented not through memory locations and pointers, but rather via Boolean formulas [9], making it possible to use efficient, symbolic invariant checking algorithms, like IC3 [8, 10]. Interestingly, we demonstrate that the concept of an automata cascade corresponds to a hierarchy of modules in the SMV language [11, 12].

In Section 5, we present a proof-of-concept in which model checking of cascades of automata is applied to simple system models. The results obtained are promising and suggest the potential of this novel approach.

2. Background

An *automaton* $\mathcal{A}$ is a tuple $(\Sigma, Q, \delta, q_0, F)$ such that: (i) Σ is a (finite) alphabet; (ii) Q is a finite set of states; (iii) $\delta : Q \times \Sigma \to Q$ is a transition function; (iv) $q_0 \in Q$ is the initial state; and (v) $F \subseteq Q$ is the set of final states. We say that $\mathcal{A}$ is *two-state* iff $|Q| = 2$.

Given an automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ and a (finite) word $\sigma \coloneqq \langle \sigma_0, \ldots, \sigma_n \rangle \in \Sigma^*$, we say that σ is *accepted* by $\mathcal{A}$ iff $\mathcal{A}$ reaches a final state reading σ. We define the *language of* $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, as the set of words accepted by $\mathcal{A}$.

The cascade product of automata is defined as follows.

Definition 1 (Cascade product of automata [2, 3])**.** *Let Σ be an alphabet and let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ and $\mathcal{A}' = (\Sigma \times Q, Q', \delta', q_0', F')$ be two automata over the alphabets Σ and $\Sigma \times Q$, respectively. We define the* cascade product between $\mathcal{A}$ and $\mathcal{A}'$*, denoted with $\mathcal{A} \circ \mathcal{A}'$, as the automaton $(\Sigma, Q \times Q', \delta'', (q_0, q_0'), F \times F')$ such that, for all $(q, q') \in Q \times Q'$ and for all $a \in \Sigma$:*

$$\delta''((q, q'), a) = (\delta(q, a), \delta'(q', (a, q)))$$

Fig. 1 shows the cascade product of two automata defining the language $a \cdot \Sigma^*$. We will often simply use *"cascade"* for *"cascade product"*. We define the *height* of the cascade $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$ as n.

Two classes of automata are particularly important in the context of cascades, namely *reset* and *permutation-reset* automata, which are defined in terms of the form of their transitions, as follows.

Definition 2 (Reset and permutation functions)**.** *Let $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ be an automaton. For each $a \in \Sigma$, we define the* function induced by a in $\mathcal{A}$*, denoted by τ_a, as the transformation $\tau_a : Q \to Q$ such that, for all $q \in Q$, it holds $\tau_a(q) = q'$ iff $\delta(q, a) = q'$. We say that τ_a is a* reset function *iff there exists $q' \in Q$ such that $\tau_a(q) = q'$, for all $q \in Q$. In this case, we say that τ_a is a* reset on q'*. If $\tau_a : Q \to Q$ is a bijection, then it is called a* permutation*.*

The following classes of automata are defined with respect to the functions induced by the symbols in their alphabets [2, 3]. We say that an automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ is:

- a *permutation-reset automaton* iff, for each $a \in \Sigma$, τ_a is either a permutation or a reset.
- a *reset automaton* iff, for each $a \in \Sigma$, τ_a is either the *identity* function or a reset function.

As an example, the two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Fig. 1 (left and center) are both reset automata.

Model checking [13, 14, 15] is an automatic and exhaustive verification technique used to determine whether all executions of a system model–typically represented as a transition system–satisfy a given logical specification, which is usually formulated in temporal logics or via automata.

A transition system M is a tuple (Σ, S, T, I) such that: (i) Σ is a finite alphabet; (ii) S is a set of states; (iii) $T \subseteq S \times \Sigma \times S$ is the transition relation; (iv) $I \subseteq S$ is the set of initial states. We define the *language of* M, denoted by $\mathcal{L}(M)$, as the set of all its finite computations, *i.e.*, finite sequences $\langle (s_0, \sigma_0), (s_1, \sigma_1), \ldots \rangle \in (S \times \Sigma)^+$ that start in initial state $s_0 \in I$ and respect the transition relation T. We say that M is *symbolically represented* when M is described with Boolean formulas (or SMT formulas, when S is infinite). In this case, $M = (\Sigma, X, T(X, \Sigma, X'), I(X))$, where: (i) X is a set of *state variables* and $X' \coloneqq \{x' \mid x \in X\}$; (ii) $T(X, \Sigma, X')$ is the formula encoding the transition relation of M; and (iii) $I(X)$ is the formula encoding the initial states of M. In this case, we say that M is a *symbolic transition system.*

Below, we present the classical definition of *safety model checking* of a transition system $M = (\Sigma, S, T, I)$. Here, the specification consists of a language $\mathcal{L} \subseteq (S \times \Sigma)^+$ of finite words, each representing an undesidered behavior that the system M should avoid.

Definition 3 (Safety Model Checking)**.** *Let $M = (\Sigma, S, T, I)$ be a transition system and let $\mathcal{L} \subseteq (S \times \Sigma)^+$. The model checking problem of M with respect to $\mathcal{L}$ is the problem of establishing whether $\mathcal{L}(M) \cap \mathcal{L} = \varnothing$.*

If $\mathcal{L}(M) \cap \mathcal{L} \neq \varnothing$, then the system M is deemed unsafe with respect to $\mathcal{L}$, as it admits at least one computation that is undesired, *i.e.*, belonging to $\mathcal{L}$. It is important to emphasize that this does not correspond to model checking over finite traces (*e.g.*, as studied in [16]). Instead, it exactly aligns with the Safety Model Checking problem (also referred to as the Invariant Checking problem) studied in [7, 8, 10]. At the core of this problem lies the reachability question: determining whether a bad state in the product of M and an automaton representing $\mathcal{L}$ is reachable from the initial state. This reachability problem is very well-studied in the model checking community and it can be efficiently solved by advanced algorithms, such as IC3 [8, 10].

The above definition is often referred to as explicit-state model checking, meaning that all states of M are represeted as memory locations and transitions are implemented as pointers. With the term *symbolic model checking*, we refer to the same problem in the case M is a symbolic transition system.

3. Exploiting automata cascades in model checking

In this section, we illustrate how specifying the property by means of an automata cascade $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$ can be advantageous for safety model checking. In particular, we highlight two main benefits: (i) this formulation enables an *incremental* approach with respect to the specification, wherein the model checking of $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_{i+1}$ is invoked only if the verification of $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i$ finds a counterexample, *i.e.*, if $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i) \neq \varnothing$; moreover, (ii) the set of reachable states computed at the i-th step can be leveraged as assumptions in the model checking of the (i+1)-th step, thereby potentially reducing the overall verification effort.

An incremental procedure for model checking cascades of automata. Let $M = (\Sigma, S, T, I)$ be a transition system and let $\mathcal{C} = \mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$ be an automata cascade over the alphabet $\Sigma' \coloneqq S \times \Sigma$ used for specifying the language $\mathcal{L}$ of bad behaviors (*cf.*, Definition 3).

The classical approach to safety model checking [7] would construct the (direct) product of M and $\mathcal{C}$ ($M \times \mathcal{C}$), and then verify whether there exists a path from an initial to a final state within this product.

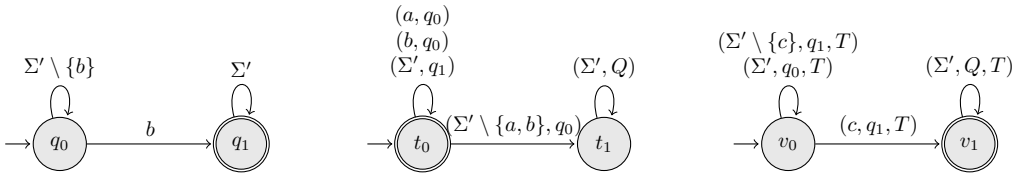

Figure 2: The cascade for the language $a^*b\Sigma'^*c\Sigma'^*$ made of: (a) the reset automaton $\mathcal{A}_1$ with set of states Q over the alphabet Σ'; (b) the reset automaton $\mathcal{A}_2$ with set of states T over the alphabet $\Sigma' \times Q$; (c) the reset automaton $\mathcal{A}_3$ over the alphabet $\Sigma' \times Q \times T$.

However, when the automaton specifying the property has a large number of states, this check may become impractical. This issue persists even in the case of cascades, since the state space of the cascade product $\mathcal{C}$ is at least of size 2^n. Nevertheless, in the context of automata cascades, we can exploit the following proposition.

Proposition 1. *If $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i) = \varnothing$, for some $i \in \{1, \ldots, n\}$, then $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{C}) = \varnothing$.*

The above proposition asserts that if there exists an index i such that the model checking of M with respect to $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i$ fails to detect a violation, then no violation exists in M with respect to the full specification $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$. This incremental approach with respect to the specification may allow the verification process to be terminated early, before analyzing the entire cascade (in Section 5, we present case studies in which this indeed occurs).

This suggests the following incremental approach for model checking M with respect to $\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n$:

1. Initialize $i = 1$.
2. Check whether $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i) = \varnothing$.
   - If so, terminate: there is no bad behavior M with respect to $\mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n)$.
   - Otherwise, increment i and repeat Step 2.

Although this approach is also possible in the case where the specification is expressed through a classical direct product of automata, a property written in the form $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ can only capture a set of constraints combined conjunctively. This typically results in either a trivial specification or an overly complex individual automaton $\mathcal{A}_i$. In contrast, by employing a cascade product, one can express intricate properties (not merely a conjunction of separate constraints) while keeping the individual components remarkably simple–as simple as two-state automata (see Section 2).

As an example, consider the cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$ depicted in Fig. 2 for the language $a^*b\Sigma'^*c\Sigma'^*$ (for sake of simplicity, here we call a, b, c, …the elements of Σ'). It is worth noting that such a property cannot be defined using the classical direct product $\mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$, except in the trivial case where one of the automata $\mathcal{A}_i$ encodes the entire property while the others recognize the universal language. The incremental approach we propose intuitively performs the following checks. It first verifies whether there exists a path in M that contains at least one occurrence of b (this amounts to checking whether $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1) = \varnothing$). If this is not the case, the procedure terminates; otherwise, it proceeds to check whether there exists a computation of M that features a b preceded exclusively by symbols a, *i.e.*, it checks whether $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \mathcal{A}_2) = \varnothing$. If this not the case, the procedure terminates; otherwise, it finally verifies whether there exist violations with respect to the entire specification by checking $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3) = \varnothing$.

Reusing the set of reachable states. Each of the steps i described above effectively amounts to a *reachability check*: we construct the (direct) product $M \times (\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i)$, and verify whether at least one final state is reachable from an initial state. If this is the case, then $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i) \neq \varnothing$, and *vice versa*. Typically, the reachability check between nodes q and q' is performed by computing an over-approximation of the states reachable from q, and iteratively refining this set until either q' is excluded or a path from q to q' is discovered.

Now, suppose that the i-th step detects a violation, *i.e.*, $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i) \neq \varnothing$. During this check, we can store an over-approximation of the states reachable (from the initial state) that we have computed, and use it as a hypothesis to accelerate the (i+1)-th step. Indeed, as established by the following proposition, if a state is unreachable in $M \times (\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i)$, then it remains unreachable in $M \times (\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_{i+1})$. Let $R_i \coloneqq \{q \in Q_{\mathcal{A}_1 \circ \cdots \circ \mathcal{A}_n} \mid q = (s^0, s^1, \ldots, s^n), (s^0, \ldots, s^i) \text{ is reachable in } \mathcal{A}_1 \circ \cdots \circ \mathcal{A}_i\}$.

Proposition 2. *For all $i \in \{1, \ldots, n\}$, it holds that $R_{i+1} \subseteq R_i$.*

Thus, if O_i is an over-approximation of the states reachable at step i, the reachability check at step i+1 does *not* need to examine states outside of O_i. In general, this has the potential to significantly reduce the search space, thereby yielding a substantial speed-up in the overall verification process.

4. A symbolic approach for automata cascades

In this section, we demonstrate how cascades of automata can be conveniently represented in a *symbolic* manner, thereby enabling the application of efficient symbolic algorithms for safety model checking, such as IC3 [8, 10], to implement the individual steps of the incremental approach outlined in the preceding section. To this end, we employ the SMV modeling language [11, 17].

First, we use a separate SMV module for each component of the cascade. In fact, the notion of a cascade naturally corresponds to the concept of a "hierarchy of modules" in SMV[2]: the first module (corresponding to the first component of the cascade) takes as input parameters only the symbols in Σ', whereas the second module takes as input not only Σ' but also the state variables of the first module (in this case, q), and so on and so forth.

Second, we model the states (initial and final) and the transition function of each component by means of Boolean state variables and formulas. Here, two considerations arise from the fact that, by Krohn-Rhodes theorem [4], every star-free property (*i.e.*, a property definable in LTL) corresponds to a cascade of two-state reset automata. The first is that, when starting from such properties (as in Fig. 2), it suffices to use a single state variable, say x, since each component has only two states. The second is that, since every transition in each component is either a reset or the identity (*cf.*, Definition 2), the Boolean formula encoding the transition relation always takes the same form: the value of the state variable x at the next time step is set to 1 (resp., 0) if a reset toward the state encoded by 1 (resp., 0) is read, and remains unchanged otherwise (because it is an identity). The previous two points imply that, in the case of star-free properties, all SMV modules can be structured in one of the following forms:

```
MODULE A(Σ′)
VAR
  x : boolean;
ASSIGN
  init(x) := FALSE;
  next(x) := case
    a  : TRUE; -- if 'a' is a reset on state x=1
    TRUE : x;  -- identities
  esac;
INVARSPEC
  ...
```

```
MODULE A(Σ′)
VAR
  x : boolean;
ASSIGN
  init(x) := FALSE;
  next(x) := case
    a  : FALSE; -- if 'a' is a reset on state x=0
    TRUE : x;   -- identities
  esac;
INVARSPEC
  ...
```

Third, we specify (the complement of) the set of final states via the INVARSPEC keyword followed by the negation of the Boolean formula encoding the final states of the component under consideration. In this way, when performing model checking of M with respect to, say, the first component $\mathcal{A}_1$, if the invariant in INVARSPEC is violated, it means that at least one final state is reachable and, according to the incremental approach described in the previous section, verification can proceed to the next component. Otherwise, verification terminates, since no final state is reachable in the first component.

[2] See, for instance, the statement *"A module can contain instances of other modules, allowing a structural hierarchy to be built."* in the manual of the nuXmv model checker [12, Section 2.3.11])

```
MODULE A_1(Σ′)
VAR
  q : boolean;
ASSIGN
  init(q) := FALSE;
  next(q) := case
    b  : TRUE; -- resets
    TRUE : q;  -- identities
  esac;
INVARSPEC
  !q;
```

Listing 1: SMV module for $\mathcal{A}_1$

```
MODULE A_2(Σ′,autom1)
VAR
  t : boolean;
ASSIGN
  init(t) := FALSE;
  next(t) := case
    !(a|b) & !autom1.q : TRUE;
    TRUE : t;
  esac;
INVARSPEC
  t;
```

Listing 2: SMV module for $\mathcal{A}_2$

```
MODULE A_3(Σ′,autom1,autom2)
VAR
  v : boolean;
ASSIGN
  init(v) := FALSE;
  next(v) := case
    c & autom1.q : TRUE;
    TRUE : v;
  esac;
INVARSPEC
  !v;
```

Listing 3: SMV module for $\mathcal{A}_3$

Figure 3: SMV code for the symbolic representation of the cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$ in Fig. 2.

Fig. 3 illustrates the symbolic representation of the cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$ shown in Fig. 2. In particular, it is worth observing how the input of each module reflects the definition of a cascade of automata (Definition 1), and how the transition relation always has the form indicated above, given that all automata $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are reset automata.

To the best of our knowledge, this is the first symbolic approach to representing cascades of automata and leveraging them for model checking. We highlight two important aspects: (i) the fact that each step amounts to a reachability check allows the use of highly efficient symbolic algorithms such as IC3; (ii) the fact that the transition relation induced by cascades of reset automata always results in a Boolean formula of a particular shape suggests the potential for future specialized verification techniques tailored to this structure, capable of exploiting this information to explore the state space more efficiently.

5. Proof-of-Concept

This section provides a proof-of-concept that illustrates the potential of the symbolic approach based on cascades of automata for safety model checking. Specifically, by considering the cascade depicted in Fig. 3 (which corresponds to the LTL formula $a \,\mathsf{U}\, (b \wedge \mathsf{XF}c)$) and employing the nuXmv model checker [17], we show how the incremental approach outlined in Section 3, using IC3 at each step, proves to be significantly more advantageous than verifying the entire cascade at once. Moreover, it emerged that in certain cases, our proposed approach is orders of magnitude more effective than verifying directly the equivalent LTL formula–in this case, $a \,\mathsf{U}\, (b \wedge \mathsf{XF}c)$.

5.1. Discussion of the experimental evaluation

We tested the incremental approach, implemented through the symbolic method, for model checking the property specified by the cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$ (Fig. 2), which is represented symbolically in Fig. 3. We evaluated this property on three models, shown in Fig. 4.

In the first model (M_1), there is no computation that contains at least one occurrence of b. Thus, there exists no model in M_1 of the property specified by the cascade. When performing model checking of M_1 against the entire specification (*i.e.*, the full cascade $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$), the time required by IC3 (as implemented in nuXmv) was 8.64 seconds. However, we observe that the absence of counterexamples is already captured by the first component of the cascade, namely $\mathcal{A}_1$, which never reaches its final state. Therefore, we also performed model checking of M_1 against $\mathcal{A}_1$ alone, as would be done by the incremental approach described in Section 3, and found that the verification time was only 0.14 seconds. Since, as noted, there are no counterexamples even for A_1, the incremental approach allows us to avoid proceeding to the verification of both $\mathcal{A}_1 \circ \mathcal{A}_2$ and $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$.

```
MODULE model
VAR
  a : boolean;
  b : boolean;
  c : boolean;
  d : 0..1000;
INIT
  a & !b & d=0;
TRANS
  (next(a) <-> (d < 789 | (d
    mod 231)=0)) &
  ((d > 555 & b) -> !next(b)) &
  ((d > 555 & !b) -> next(b)) &
  ((d < 876 & !b) -> !next(b))&
  (next(c) <-> d=554) &
  (next(d) = ((d+1) mod 987));
```

Listing 4: SMV module for M_1

```
MODULE model
VAR
  a : boolean;
  b : boolean;
  c : boolean;
  d : 0..1000;
INIT
  a & !b & d=0;
TRANS
  (next(a) <-> (d < 789))   &
  ((d != 998) -> !next(b))  &
  (next(c) <-> d=999)       &
  (next(d) = ((d+1) mod 1001));
```

Listing 5: SMV module for M_2

```
MODULE model
VAR
  a : boolean;
  b : boolean;
  c : boolean;
  d : 0..5000;
INIT
  a & !b & d=0;
TRANS
  (next(a) <-> (d < 998))   &
  ((d != 998) -> !next(b))  &
  (next(c) <-> d=4999)      &
  (next(d) = ((d+1) mod 5000));
```

Listing 6: SMV module for M_3

Figure 4: The three models of our proof-of-concept.

In model M_2, every path eventually contains an occurrence of b, but also has an intermediate point where a does not hold, rendering M_2 free of counterexamples (*i.e.*, models of the cascade). In this case as well, verification of the entire cascade requires 24.21 seconds. Applying the incremental approach, we start by verifying automaton $\mathcal{A}_1$: the model checking of $\mathcal{A}_1$ against M_2 finds a counterexample in 6.13 seconds. According to the incremental approach, we must then proceed to verify the second automaton. We simulated the reuse of the reachable states of $\mathcal{A}_1$ discovered in the first step by adding to $\mathcal{A}_1$ an invariant of the form $M.d < 998 \rightarrow \neg q$. With this additional information, the verification of M_2 against $\mathcal{A}_1 \circ \mathcal{A}_2$ takes only 0.08 seconds, and since no counterexamples are found, we can terminate without verifying $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$.

Model M_3, on the other hand, contains at least one trace in $\mathcal{L}(\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3)$. The verification time for the entire cascade is 530.22 seconds. By contrast, using the incremental approach: (i) model checking of $\mathcal{A}_1$ alone takes 11.69 seconds; (ii) verification of $\mathcal{A}_1 \circ \mathcal{A}_2 \circ \mathcal{A}_3$, simulating the reuse of the reachable states of $\mathcal{A}_1$ by adding the invariant $M.d < 998 \rightarrow \neg q$, requires only 1.28 seconds. For this case, we also verified the property expressed directly by the LTL formula $a \,\mathsf{U}\, (b \wedge \mathsf{X}\mathsf{F}c)$, which required 408.44 seconds.

At https://users.dimi.uniud.it/~luca.geatti/data/overlay-25/SMV.zip, we provide the SMV source code for all experiments, along with detailed instructions for executing them.

This proof-of-concept highlights the potential of the symbolic approach based on cascades of automata for safety model checking. It demonstrates not only the advantages of the incremental approach–which reuses the reachable states at each iteration compared to verifying the entire cascade at once–but also its superiority over directly verifying the property expressed in LTL.

6. Conclusions and Future Work

In this paper, we have shown how using cascades of automata to specify the undesired properties of a system can be beneficial in the context of safety model checking. We proposed an incremental approach that verifies components in an incremental fashion–potentially reusing information obtained from previous steps–and demonstrated how a symbolic approach to verifying cascades is also feasible.

Undoubtedly, specifying properties directly through cascades may not always be convenient, especially when compared to more declarative formalisms such as LTL. For this reason, it is crucial to develop effective methods for translating temporal logic formulas into (symbolic) cascades of automata. Moreover, an extended experimental evaluation, building on the proof-of-concept presented in Section 5, is essential to thoroughly assess the effectiveness of this approach.

Last but not least, an interesting direction for future work is to extend the cascade product to *nondeterministic* automata. This could simplify the specification of a property, but it would clearly make the notion of reset automata, and thus the related considerations on the structure of formulas within the corresponding SMV modules, no longer meaningful.

Acknowledgments

Luca Geatti acknowledges the support from the project "ENTAIL - intEgrazioNe tra runTime verification e mAchlne Learning" - funded by the European Union – NextGenerationEU, under the PNRR- M4C2I1.5, Program "iNEST - interconnected nord-est innovation ecosystem" - Creazione e rafforzamento di "Ecosistemi dell'Innovazione per la sostenibilità" – ECS_00000043, CUP G23C22001130006 - R.S. Geatti.

Declaration on Generative AI

During the preparation of this work, the author used *ChatGPT* in order to: *Text Translation and Improved writing style.* After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] O. Maler, On the Krohn-Rhodes Cascaded Decomposition Theorem, in: Z. Manna, D. A. Peled (Eds.), Time for Verification, Essays in Memory of Amir Pnueli, volume 6200 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 260–278. URL: https://doi.org/10.1007/978-3-642-13754-9_12. doi:10.1007/978-3-642-13754-9_12.

[2] O. Maler, A. Pnueli, Tight Bounds on the Complexity of Cascaded Decomposition of Automata, in: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II, IEEE Computer Society, 1990, pp. 672–682. URL: https://doi.org/10.1109/FSCS.1990.89589. doi:10.1109/FSCS.1990.89589.

[3] R. Borelli, L. Geatti, M. Montali, A. Montanari, On cascades of reset automata, in: O. Beyersdorff, M. Pilipczuk, E. Pimentel, K. T. Nguyen (Eds.), 42nd International Symposium on Theoretical Aspects of Computer Science, STACS 2025, March 4-7, 2025, Jena, Germany, volume 327 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 20:1–20:22. URL: https://doi.org/10.4230/LIPIcs.STACS.2025.20. doi:10.4230/LIPICS.STACS.2025.20.

[4] K. Krohn, J. Rhodes, Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines, Transactions of the American Mathematical Society 116 (1965) 450–464.

[5] A. Ginzburg, Algebraic theory of automata, Academic Press, 2014.

[6] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), IEEE, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[7] O. Kupferman, M. Y. Vardi, Model checking of safety properties, Formal Methods in System Design 19 (2001) 291–314. doi:10.1023/A:1011254632723.

[8] A. R. Bradley, SAT-based model checking without unrolling, in: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, 2011, pp. 70–87.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L.-J. Hwang, Symbolic model checking: 1020 states and beyond, Information and computation 98 (1992) 142–170.

[10] A. R. Bradley, Understanding IC3, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2012, pp. 1–14.

[11] K. L. McMillan, The SMV language, Cadence Berkeley Labs (1999) 1–49.

[12] R. Cavada, A. Cimatti, A. Mariotti, A. Micheli, M. Pensallorto, S. Tonetta, et al., nuXmv User Manual, FBK-IRST, 2014. URL: https://nuxmv.fbk.eu/downloads/nuxmv-user-manual.pdf, version 1.0.

[13] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (1986) 244–263. URL: https://doi.org/10.1145/5397.5399. doi:10.1145/5397.5399.

[14] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, Handbook of model checking, volume 10, Springer, 2018. doi:10.1007/978-3-319-10575-8.

[15] E. M. Clarke, E. A. Emerson, J. Sifakis, Model checking: algorithmic verification and debugging, Communications of the ACM 52 (2009) 74–84.

[16] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, A. M. Wells, Model checking strategies from synthesis over finite traces, in: É. André, J. Sun (Eds.), Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Singapore, October 24-27, 2023, Proceedings, Part I, volume 14215 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 227–247. URL: https://doi.org/10.1007/978-3-031-45329-8_11. doi:10.1007/978-3-031-45329-8_11.

[17] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuXmv Symbolic Model Checker, in: CAV, 2014, pp. 334–342.