Strength-Based Decomposition of the Property Büchi Automaton for Faster Model Checking

Etienne Renault[1,2], Alexandre Duret-Lutz[1], Fabrice Kordon[2], and Denis Poitrenaud[3]

[1] LRDE, EPITA, Kremlin-Bicêtre, France
[2] LIP6/MoVe, Université Pierre & Marie Curie, Paris, France
[3] LIP6/MoVe and Université Paris Descartes, Paris, France

Abstract. The automata-theoretic approach for model checking of linear-time temporal properties involves the emptiness check of a large Büchi automaton. Specialized emptiness-check algorithms have been proposed for the cases where the property is represented by a weak or terminal automaton.

When the property automaton does not fall into these categories, a general emptiness check is required. This paper focuses on this class of properties. We refine previous approaches by classifying strongly-connected components rather than automata, and suggest a decomposition of the property automaton into three smaller automata capturing the terminal, weak, and the remaining strong behaviors of the property. The three corresponding emptiness checks can be performed independently, using the most appropriate algorithm.

Such a decomposition approach can be used with any automata-based model checker. We illustrate the interest of this new approach using explicit and symbolic LTL model checkers.

1 Introduction

The automata-theoretic approach to linear-time model checking consists in checking the emptiness of the product between two Büchi automata: one automaton that represents the system, and the other that represents the negation of the property to check on this system.

There are many ways to apply this approach. *Explicit model checking* uses a graph-based representation of the automata. Usually the product is constructed on-the-fly as needed by the emptiness-check algorithm, and may be stopped as soon as a counterexample is found [7]. Additionally, partial-order reduction techniques can be used to reduce the state space [15]. *Symbolic model checking* uses a symbolic representation of automata, usually by means of decision diagrams [5]. In this approach the emptiness check is achieved using fixed points.

The run-time of these approaches can be improved by different means. One way is to optimize the property automaton by reducing its number of states or making it more deterministic, hoping for a smaller product with the system. Because the property automaton is small, the time spent optimizing it is negligible compared to the time spent performing the emptiness check of the product. Another possible

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 580–593, 2013.

improvement is to use an emptiness check algorithm tailored to the property automaton used. For instance generalized emptiness checks [19, 9] can be used when the property requires generalized acceptance conditions. Also, simplified procedures can be performed when the strength of the property automaton is weak or terminal [2, 6], improving the worst-case complexity by a constant factor.

For strong property automata (that are neither weak nor terminal), a general Büchi emptiness check algorithms has to be used, even though they could also contain some weak and terminal components. In this paper we focus such properties whose automata mix strong, weak, or terminal components. We show that such automaton can be decomposed into three automata, each of a different strength. These automata can then be emptiness checked independently (and concurrently) using the most appropriate algorithm. Each of these three automata is smaller than the original automaton, moreover, because it is simpler it can usually be even more simplified. This decomposition works regardless of the type model-checking approach and options used (explicit, symbolic, parallel,...).

This paper is organized as follows. In Section 2, we define the type of (generalized) Büchi automata we use, discuss their emptiness checks, and the hierarchy of automaton strengths. Section 3 studies different ways to characterize the strength of a strongly connected component. These strengths are the basis for our decomposition described in Section 4. Finally we present our experimental results in Section 5.

2 Büchi Automata and Their Strengths

Let AP be a finite set of (atomic) propositions, and let $\mathbb{B} = \{\bot, \top\}$ represent Boolean values. We denote $\mathbb{B}(AP)$ the set of all Boolean formulas over AP, i.e., formulas built inductively from the propositions AP, $\mathbb{B}$, and the connectives $\wedge$, $\vee$, and $\neg$. An assignment is a function $\rho : AP \to \mathbb{B}$ that assigns a truth value to each proposition. We denote $\mathbb{B}^{AP}$ the set of all assignments of AP.

The automata-theoretic approach is usually performed using Büchi automata. In this work, we use a slightly more general form of automata called *Transition-based Generalized Büchi Automaton* (TGBA) which allows a more compact representation of properties. Any Büchi automaton can be seen as a TGBA by pushing acceptance sets to outgoing transitions, so the reader working with Büchi automata will have no problem adapting our techniques.

Definition 1. *A* ***TGBA*** *is a 5-tuple* $A = \langle AP, Q, q^0, \delta, F\rangle$ *where:*

- AP *is a finite set of atomic propositions,*
- Q *is a finite set of states,*
- $q^0 \in Q$ *is the initial state,*
- $\delta \subseteq Q \times \mathbb{B}^{AP} \times Q$ *is the transition relation, labeling each transition by an assignment of the atomic propositions,*
- $F \subseteq 2^{\delta}$ *is a set of acceptance sets of transitions.*

A *run* of A is an infinite sequence of transitions $\pi = (s_1, \ell_1, d_1) \ldots (s_i, \ell_i, d_i) \ldots$ with $s_1 = q^0$ and $\forall i \geq 1$, $d_i = s_{i+1}$. Such a run is *accepting* iff it visits all acceptance sets infinitly often, i.e, $\forall f \in F$, $\forall i \geq 1$, $\exists j \geq i$, $(s_j, \ell_j, d_j) \in f$.

An infinite word $w = \rho_1\rho_2\cdots$ over $\mathbb{B}^{AP}$ (i.e., $\rho_i \in \mathbb{B}^{AP}$), is accepted by A iff there exists an accepting run $\pi = (s_1, \ell_1, d_1)\ldots(s_i, \ell_i, d_i)\ldots$ such that $\forall i, \rho_i = \ell_i$. The language $\mathscr{L}(A)$ is the set of infinite words accepted by A.

The automata-theoretic approach to model checking amounts to check the emptiness of the language of a TGBA that represents the product of a system (a TGBA where $F = \emptyset$) with the negation of the property to verify (another TGBA).

A *path* of length $n \geq 1$ between two states $q, q' \in Q$ is a finite sequence of transitions $\rho = (s_1, \ell_1, d_1)\ldots(s_n, \ell_n, d_n)$ with $s_1 = q$, $d_i = q'$, and $\forall i \in \{1, \ldots, n-1\}$, $d_i = s_{i+1}$. Let $S \subseteq Q$ such that $\{s_1, s_2, \ldots, s_n, d_n\} \subseteq S$, we denote the existence of such a path by $q \overset{S}{\rightsquigarrow} q'$. If $q = q'$ we say that such a path is a *cycle*. A *cycle* is *accepting* iff it visits all acceptance sets, i.e., $\forall f \in F$, $\exists i \in \{1, \ldots, n\}$, $(s_i, \ell_i, d_i) \in f$. A cycle is *elementary* iff it does not visit any state twice (i.e., $\forall 1 \leq i < j \leq n$, $s_i \neq s_j$).

If a TGBA has an (infinite) accepting run, then the run necessarily visits one of the states infinitely often, which means that the automaton has an accepting cycle that is reachable from q^0. One way to perform the emptiness check of a TGBA explicitly is therefore to search for such cycles using nested DFS (Depth First Search). Although there exists a nested DFS algorithm that works on TGBA [25], most of the usual nested DFS algorithms [23] require a degeneralized Büchi automaton with a single acceptance set (the degeneralization of a TGBA with n acceptance sets may multiply its number of states by n). In these algorithms, a first DFS is used to detect the start of potential cycles, and another (or several in the generalized case) DFS is started to detect an accepting cycle.

A second emptiness-check approach is to compute the accepting strongly-connected components of the TGBA.

Definition 2. *A* ***Strongly-Connected Component*** *(SCC) of a TGBA is a maximal set of states C such that there is a path between any two distinct states of C (i.e., $\forall s, s' \in C$, $(s \neq s') \Rightarrow (s \overset{C}{\rightsquigarrow} s')$).*

C is ***accepting*** *iff it contains an accepting cycle.*

C is ***complete*** *iff $\forall s \in C$, $\forall f \in \mathbb{B}^{AP}$, $\exists (q, \ell, q') \in \delta$ such that $s = q$, $f = \ell$, and $q' \in C$.*

While SCC-based emptiness checks [8, 16] are still based on a DFS exploration of the automaton, they do not require another nested DFS, and their complexity does not depend on the number of acceptance sets.

Symbolic emptiness checks [19, 14] are also based on the computation of SCCs in the symbolic representation of the automaton. This is done using fixed points on symbolic set of states, and amounts to performing a BFS-based emptiness check.

Whether based on nested DFS or SCC, explicit or symbolic, these emptiness-check procedures can be simplified according to the *strength* of the automaton representing the property to check [2, 13, 6, 23, 1].

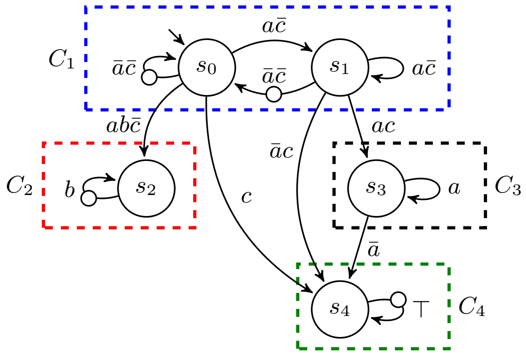

Fig. 1. TGBA for $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$

Before defining the strength of the property automaton, let us first characterize the strength of an SCC.

Definition 3. *The strength of an SCC is:*
Non Accepting. *if it does not contain any accepting cycle,*
Inherently Terminal. *if it contains only accepting cycles and is complete,*
Inherently Weak. *if it contains only accepting cycles and it is not inherently terminal,*
Strong. *if it is accepting and contains some non-accepting cycle.*
These four strengths define a partition of the SCCs of an automaton.

There are two kinds of non accepting SCCs. If an SCC can only reach other non-accepting SCCs, it is **useless** and may be removed from the automaton without changing its language. This simplification is traditionally performed right after the translation of the property into an automaton. If the non accepting SCC can reach an accepting one, it is **transient**. In the rest this paper we assume that useless SCCs have been removed, i.e., all non-accepting SCCs are transient.

Figure 1 shows an example TGBA with a single acceptance set represented with white dots on transitions. Transitions are labeled by Boolean formulas instead of assignments (for instance a transition labeled by a is shorthand for two transitions labeled by ab and $a\bar{b}$). The dashed boxes highlight the five SCCs of the automaton. C_1 is a strong SCC (the cycle between s_0 and s_1 is accepting, while the self-loop on s_1 is a non-accepting cycle), C_2 is an inherently weak SCC, C_3 is transient, and C_4 is inherently terminal.

Definition 4. *An automaton is* ***inherently terminal*** *iff all its accepting SCCs are inherently terminal. An automaton is* ***inherently weak*** *iff all its accepting SCCs are inherently terminal or inherently weak. Any automaton is* ***general****. These three classes form a hierarchy where inherently terminal automata are inherently weak, which in turn are general.*

Note that the above constrains concern only accepting SCCs, but these automata may also contain non-accepting (transient) SCC.

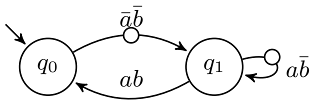

Fig. 2. An inherently weak automaton which is not weak

The notion of **inherently weak** automaton [3] generalizes the more common notion of **weak** automaton [2, 6]. If we define a weak SCC to be an accepting SCC whose transitions belong to all acceptance sets, then a weak automaton is an automaton that contains only weak, terminal, or non-accepting SCCs. A weak automaton is inherently weak, and an inherently weak automaton can be easily converted into a weak automaton [3]. For example the automaton from Fig. 2 can be easily converted into a weak automaton, by adding the transition from q_1 to q_0 into the ○ acceptance set.

Similarly, our definition of **inherently terminal** is a generalization of the notion of **terminal** automaton [2, 6]. If we define a terminal SCC to be weak and complete, then a terminal automaton should have only terminal, or non-accepting SCCs. A terminal automaton is inherently terminal, and an inherently terminal automaton can be obviously converted into a terminal automaton.

The emptiness-check algorithms previously discussed will obviously work with general automata. More efficient algorithms can be used for inferior strengths. For inherently weak automata, the explicit emptiness check reduces to the detection of a cycle in a inherently weak or terminal SCC. This can be performed using a single DFS [6]. Symbolic emptiness checks of inherently weak automata can be simplified similarly [2]. Furthermore, when the system to verify does not have any deadlock (each state has at least one successor) and the property automaton is terminal, then the emptiness check of the product becomes a reachability problem. Here again, both explicit and symbolic emptiness checks can take advantage of this simplification [2, 6].

Considering this strength hierarchy can also help when implementing techniques such as partial order reduction [6] or distributed model checking [1]. In most of the approaches suggested so far the improvements have only concerned (inherently) weak or terminal automata: if an automaton contains at least one strong SCC, a general emptiness check is required, even if it also contains SCCs of inferior strengths. However Edelkamp et al. [13] have suggested to consider the strengths of the SCCs to limit the scope of the nested DFS to the strong SCCs.

The technique we present in section 4 improves the emptiness check of properties that mix accepting SCCs of different strengths. A necessary step towards this goal is to be able to determine the strength of SCCs.

3 Determining SCC Strength

The SCCs of an automaton, and their acceptance, can be obtained by applying the algorithms of Couvreur [8] or Geldenhuys and Valmari [16].

We now consider three approaches to classify accepting SCCs. The **inherent approach**, that sticks to definition 3. A **structural heuristic**, based on the graph's structure. And a **syntactic heuristic**, which can only be applied when translation algorithm labels a state s of the automaton A by the LTL formula recognized from this state (this is the case in our implementation). The latter two heuristics may misclassify an SCC in a higher class, requiring a more general emptiness check algorithm.

We evaluate these three approaches on a benchmark of 10 000 random LTL formulas, translated into TGBA using Couvreur's algorithm [8] and where useless SCCs have been pruned. Couvreur's translation naturally outputs an inherently weak (resp. terminal) TGBA for any syntactic-persistence (resp. syntactic-guarantee) formula, in the syntactic classification of Černá and Pelánek [6]. For example, when translating the LTL formula $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$, this translation produces the automaton from Fig. 1 in which states s_0, s_1, s_2, s_3, and s_4 respectively correspond to the LTL formulas $(\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c$, $\mathsf{F}\,\bar{a} \wedge ((\mathsf{G}a \rightarrow \mathsf{G}b)\,\mathsf{W}\,c)$, $\mathsf{G}b$ (a syntactic-persistence formula), $\mathsf{F}\,\bar{a}$ and $\top$ (two syntactic-guarantee formulas).

We now describe how we characterize weak and terminal SCCs in the aforementioned three approaches.

If an accepting SCC contains any non-accepting cycle, then it necessarily contains a non-accepting elementary cycle. Therefore whether an accepting SCC is inherently weak can be determined by enumerating all its elementary cycles. As soon as one non-accepting cycle is found, the algorithm can claim the SCC to be non-inherently weak. This cycle enumeration can be costly since it may theoretically have to explore an exponential number of elementary cycles [20]. As an alternative, a structural heuristic, is to check whether all transitions in the accepting SCC belong to all acceptance sets (the SCC is weak), this information can be collected while we determine the accepting SCCs of the automaton. On our benchmark this approach correctly classifies $99,85\%$ of the weak SCCs. Another heuristic is to consider the LTL formulas labeling the states of the accepting SCC: if one of them is a syntactic-persistence then the SCC is either inherently weak or terminal. On our benchmark this test catches only $87,77\%$ of the weak SCCs.

Terminal SCCs can be similarly detected in three ways. The inherent approach is to check that (1) the disjunction of the labels of the outgoing transitions (that remain in the SCC) of each state is $\top$, and (2) there is no non-accepting elementary cycles. A structural heuristic would be to replace (2) by a check that all transitions belong to all acceptance sets. Finally, a syntactic heuristic would be to check that one state in the accepting SCC is labeled by a syntactic-guarantee formula. On our benchmarks these three approaches all catch 100% of the terminal SCCs.

The structural heuristics presented above correspond to the definition of the weak and terminal used by Bloem et al. [2] to characterize the strength of the entire automaton. Looking into the $0,15\%$ of SCCs that this structural heuristic fails to detect as inherently weak reveals that these SCCs are the results from the translation of pathological formulas: formulas whose syntactic class is above their actual strength. For instance $\varphi = \mathsf{G}(c \vee (\mathsf{X}\, c \wedge (\bar{c}\,\mathsf{U}\, b)))$ is a syntactic-recurrence formula equivalent to the safety formula $\mathsf{G}(c \vee (\bar{c} \wedge \mathsf{X}(c \wedge b)) \vee (b \wedge \mathsf{X}\, c))$, yet our translation of φ will produce an inherently weak automaton that is not weak.

In our experiments the structural approach was 3 times slower than the syntactic one, and 10 times faster than the inherent one. Since it caught 99, 85% of the weak SCCs, we adopted the structural approach in our upcoming experimentation. Regardless of these comparisons, all these approaches are instantaneous in practice.

Additional post-processing, as suggested by Somenzi and Bloem [24], would likely improve the "weakness" of the property automata.

4 Decomposing the Property Automaton According to Its SCCs Strengths

In this section, we focus on general property automata that cannot be handled by a specialized emptiness check (e.g. for inherently weak automata) because the property automaton contains SCCs of different strengths. The automaton from Fig. 1 is such an automaton. We show how they can be decomposed into three property automata representing their strong, weak, and terminal behaviors, that can be used concurrently.

We denote T, W, and S, the set of all transitions belonging respectively to some terminal, weak, or strong SCC. For a set of transitions X, we denote $\mathrm{Pre}(X)$ the set of states that can reach some transition in X. We assume that $q^0 \in \mathrm{Pre}(X)$ even if X is empty or unreachable.

Definition 5. *Let $A = \langle AP, Q, q^0, \delta, \{f_1, \ldots, f_n\}\rangle$ be a TGBA. We define three derived automata $A_T = \langle AP, Q_T, q^0, \delta_T, F_T\rangle$, $A_W = \langle AP, Q_W, q^0, \delta_T, F_W\rangle$, $A_S = \langle AP, Q_S, q^0, \delta_T, F_S\rangle$ that represent respectively the terminal, weak and strong behaviors of A, with:*

$$
\begin{array}{lll}
Q_T = \mathrm{Pre}(T) & F_T = \{T\} & \delta_T = \{(q, l, q') \in \delta \mid q, q' \in Q_T\} \\
Q_W = \mathrm{Pre}(W) & F_W = \{W\} & \delta_W = \{(q, l, q') \in \delta \mid q, q' \in Q_W\} \\
Q_S = \mathrm{Pre}(S) & F_S = \{f_1 \cap S, \ldots, f_n \cap S\} & \delta_S = \{(q, l, q') \in \delta \mid q, q' \in Q_S\}
\end{array}
$$

Fig. 3 shows the result of the decomposition of the TGBA of Fig. 1. The SCCs that are highlighted with boxes represent the terminal, weak, and strong SCCs that have been preserved. The rest of these automata is made of the prefixes leading to these accepting SCCs.

Property 1. The strengths of A_T, A_W, A_S are respectively terminal, weak, and strong (unless they have no transition).

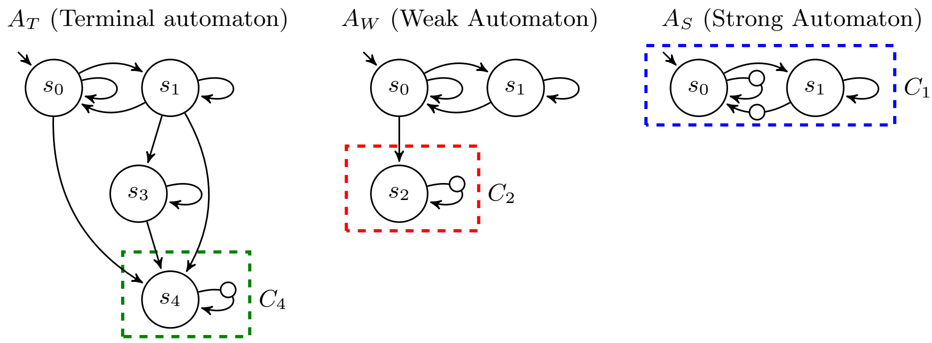

Fig. 3. Decomposition of the automaton from Fig. 1 into three automata (labels have been ommited for clarity)

Theorem 1. $\mathscr{L}(A) = \mathscr{L}(A_T) \cup \mathscr{L}(A_W) \cup \mathscr{L}(A_S)$.

Intuition of the proof: ($\subseteq$) A word accepted by A is recognized by a run that will eventually be captured by an accepting SCC of A. Since every accepting SCC belongs to one of the three automata, this SCC is necessarily reproduced in an accepting form in one of the three derived automata, and it is necessarily reachable from the initial state. ($\supseteq$) Because the three automata are restrictions of A, a word accepted by any of these is straightforwardly accepted by A.

Using this decomposition, we can perform three model-checking procedures in parallel, choosing the emptiness algorithm most suited to the strength of each derived automaton. This way, the more complex algorithm will have to deal with a smaller automaton (by construction), and the three procedures may abort as soon as one of them finds a counterexample.

The weak and terminal automata A_W and A_T, require very simple emptiness check algorithms [6, 2] because the acceptance conditions are easier to check. They also make it easier to apply other reduction such as partial order reductions [18], and they tend to produce smaller counterexamples [13].

For the strong derived automaton A_S, a general emptiness check is required. Implementations using an emptiness-check that can only deal with a single acceptance set (i.e., Büchi-style) need to degeneralize only this derived automaton.

This decomposition scheme can be further improved by minimizing each derived automaton. For instance, weak and terminal automata can be reduced very efficiently with techniques such as WDBA minimization [10]. Also simulations reductions [24] will be more efficient on automata with less acceptance conditions. As these techniques will not augment the strength of an automaton, they can be used without restriction.

In addition to reducing the number of states and acceptance sets in the automaton, the decomposition might also produce automata that observe fewer atomic propositions. Emptiness check techniques that are sensitive to the number of observed propositions [e.g., 22] will therefore benefit from the decomposition.

As a final note, this decomposition approach is suitable for any type of model checker (explicit, symbolic, parallel, ...) as long as it uses an automaton to represent the property.

5 Assessment

We compare the new decomposition approach against the classical one in four setups:

SE. This explicit setup uses Schwoon and Esparza's improved NDFS algorithm [23], to our knowledge, the best NDFS to date. This emptiness checks requires a degeneralization.

ELL. A refinement of the previous setup restricting the nested DFS to the strong components, as suggested by Edelkamp et al. [13].

Cou. This explicit setup uses Couvreur's SCC-based algorithm [8] and supports TGBA directly.

OWCTY. This symbolic setup uses an implementation of the classical OWCTY algorithm with multiple acceptance sets [19].

When the decomposition approach is used, the above emptiness checks are applied only on the strong automaton $K \otimes A_S$. For the products with weak and terminal automata, we use explicit or symbolic dedicated algorithms as described by Černá and Pelánek [6].

In all approaches, LTL formulas representing properties are first simplified, translated into TGBA, and these automata are postprocessed (using aforementioned techniques) in Spot [11]. In the decomposition scheme, the three resulting automata are postprocessed again.

The models we use come from the BEEM benchmark [21]. In explicit setup, we generate the system automaton K using a version of DiVinE 2.4 patched by the LTSmin team[1]. For the symbolic setup, we use a symbolic representation provided by `its-ltl`[2] [12].

Because the LTL formulas supplied by the BEEM benchmark are few and are usually safety automata (their negation translates into a terminal automaton), we opted to generate random LTL formulas for each model.

We ran our different approaches on 13 models, for which we selected formulas such (1) the property automaton contains different SCC strengths, (2) the product with the system has more than 2000 states, (3) for each model 100 formulas yield an empty product, and 100 formulas yield a non-empty one.[3] The second point is to avoid cases where the formula is trivial to verify.

1 `http://fmt.cs.utwente.nl/tools/ltsmin/#divine`

2 `http://ddd.lip6.fr/`

3 This has been done by generating random formulas and running an emptiness check over the product automaton until 100 empty products and 100 non empty products were found. For a more detailed description of our setup, including selected models and formulas, see `http://move.lip6.fr/~Etienne.Renault/benchs/TACAS-2013/benchs.html`

Table 1. Sizes of the automata A_S, A_W, A_T relative to A, with or without the post-processing applied after decomposition, averaged on all our formulas

	no postproc.		postproc.	
	states	trans.	states	trans.
A_S	50.66%	37.87%	46.57%	34.85%
A_W	68.71%	51.47%	62.95%	44.77%
A_T	75.27%	63.68%	64.70%	49.28%

These tool chains were executed on a cluster of Intel Xeon E5645@2.40GHz, running Linux. The memory was confined to 4GB, and the run time to 1 hour.

Table 1 shows the reduction effect of the decomposition and additional post-processing on the sizes of the property automata. It can be noted that it is the strong automaton that obtains the greatest reduction, a good news, since this is the hardest to check.

Table 2 shows how many pairs of (model,formula) were successfully processed by each setup within the run-time and memory confinement. We separated empty products (verified formulas) from non-empty products (violated formulas) because the emptiness check may abort as soon as a counter example is found in the latter. It can be observed that using the decomposition always helps.

Table 2. Number of formulas processed by the classical (class.) and decomposition (dec.) approach, using different emptiness checks, out of a total of 2600 formulas

	empty		non-empty		total	
	class.	dec.	class.	dec.	class.	dec.
SE	1258	1297	1300	1300	2558	2597
ELL	1250	1297	1300	1300	2550	2597
Cou	1257	1299	1300	1300	2557	2599
OWCTY	1293	1299	1285	1299	2578	2598

Table 3 is an excerpt of our complete benchmark showing only a selection of the models whose verification required a significant run time (still, the observed trends are similar in other models). In order to compare the different algorithms, we restricted these measurements to formulas that could be processed by all setups.

For the “classical” explicit approaches, we measure the average number of visited states (counted once) and explored transitions (counted at most twice depending on the algorithm) during the emptiness check of $K \otimes A$ (the product of the system with A).

For the “decomposition-based” explicit approaches, three algorithms have been launched in parallel (on three different hosts) to check the emptiness of $K \otimes A_T$, $K \otimes A_W$, and $K \otimes A_S$. When $\mathscr{L}(K \otimes A) = \emptyset$, we have to wait for the three emptiness checks, and we report the performances of the last to terminate. When $\mathscr{L}(K \otimes A) \neq \emptyset$, we report the performance of the first emptiness check that finds a counterexample.

Table 3. Evaluation of the decomposition technique when model-checking different models in four possible setups. All values are averaged over all cases considered for one model. Time is in seconds, memory is in MB.

			classical				decomposition			
	model	algorithm	states	transitions	time	mem	states	transitions	time	mem
$\mathscr{L}(K \otimes A) = \emptyset$	at.4	SE	11778840	55765492	112.21	3034	7620732	30150665	63.35	2691
	84 cases	ELL	11778840	55748407	117.22	3050	7620732	30150665	63.07	2688
		Cou	11692421	54326243	95.95	2913	7542343	28859760	58.88	2657
		OWCTY			149.91	3227			75.68	2841
	bopdp.3	SE	2672100	14245549	20.59	1790	1430033	5249648	9.65	1460
	99 cases	ELL	2672100	13637796	21.27	1811	1440798	5250679	9.51	1443
		Cou	2515568	10389823	17.93	1717	1414104	4037319	7.96	1362
		OWCTY			241.26	3313			166.98	3151
	elevator2.3	SE	17583328	208607370	273.95	3622	12709300	106105555	161.48	3418
	64 cases	ELL	17583328	200251800	287.67	3639	12709300	106105555	161.02	3419
		Cou	17144611	171043227	186.22	3464	12479194	99666774	141.25	3348
		OWCTY			14.59	1607			6.48	1534
	elevator.4	SE	2928295	15794777	26.73	1969	1543723	4728263	10.58	1505
	94 cases	ELL	2928295	14908666	26.65	1984	1543723	4728263	10.54	1498
		Cou	2849219	12734156	20.14	1831	1547016	4430731	9.70	1463
		OWCTY			638.29	3812			245.55	3718
	prodcell.3	SE	3488725	25182172	34.28	1952	1358518	5065228	9.64	1400
	100 cases	ELL	3488725	23975933	35.11	1954	1358849	5065967	9.58	1397
		Cou	3194579	19584772	26.40	1797	1323029	4391328	8.38	1357
		OWCTY			145.60	3003			50.04	2731
$\mathscr{L}(K \otimes A) \neq \emptyset$	at.4	SE	362202	2384803	4.61	842	138	181	0.00	795
	93 cases	ELL	362202	2100874	4.38	861	146	186	0.00	798
		Cou	362196	2095924	3.63	837	172	217	0.00	799
		OWCTY			343.86	3501			80.95	2623
	bopdp.3	SE	32131	90859	0.19	765	1145	2333	0.01	803
	99 cases	ELL	31989	90668	0.20	762	1134	2310	0.01	802
		Cou	32120	80027	0.17	780	1152	2331	0.01	800
		OWCTY			292.19	3275			69.46	2594
	elevator2.3	SE	998871	14729965	15.29	1023	7967	50455	0.07	721
	100 cases	ELL	998725	13980443	16.54	1031	7978	50466	0.07	720
		Cou	984226	9916942	10.29	986	7975	50464	0.07	720
		OWCTY			30.53	2079			6.68	1172
	elevator.4	SE	37389	141012	0.28	745	54	58	0.00	719
	87 cases	ELL	37336	137843	0.29	751	44	47	0.00	718
		Cou	37386	118119	0.21	732	41	43	0.00	723
		OWCTY			491.27	3747			174.15	3087
	prodcell.3	SE	52458	313946	0.46	753	497	876	0.00	758
	97 cases	ELL	52375	271454	0.44	779	495	862	0.00	759
		Cou	48589	199349	0.32	744	491	857	0.00	757
		OWCTY			196.47	3209			57.83	2469

For all approaches (explicit and symbolic), we report peak memory usage and run time following the same rules as above.

A first observation is that while the run time is always improved by the decomposition, the memory gain is no always so obvious.

For non-empty products, the table shows that counterexamples are found much more rapidly. However when comparing the results of explicit approaches for non-empty products, we should keep in mind that there is a part of luck involved: depending on the order in which transitions of the property automaton are ordered, an emptiness check may find a counterexample faster. The results for empty products are easier to appreciate: since the entire product has to be explored transition order has no importance.

Table 3 can also be used as yet another comparison of emptiness check algorithms. We can notice that our benchmark favors explicit approaches over symbolic ones. This is a consequence of our selection of models and may certainly not be used to denigrate symbolic approaches. Still, if we order the emptiness check algorithm in the classical approach according to their average run time, we can observe that adding the decomposition does not change the order of these algorithms.

The ELL algorithm explores less transitions than SE because it restricts its nested DFS to the strong SCCs of the property, however this smaller exploration does not always reflect on the run-time because of the small overhead required to apply this restriction.

6 Conclusion

In the automata-theoretic approach to model checking for linear-time properties, specialized emptiness checks algorithms have been proposed for the cases where the property automaton is represented by a weak or terminal automaton. For strong automata, a general emptiness check is required.

In this paper we focused on properties whose automata (strong or weak) mix SCCs of different strengths, and for which we propose a decomposition approach based on these strengths.

Our experimentation of various ways to implement the characterization of SCC strengths has shown that trying to detect inherently weak SCCs (by enumerating all its elementary cycles) was not worth it: detecting weak SCCs is faster and easier to implement, and will miss very few inherently weak SCCs. However this study was performed on automata produced by Spot whose translation algorithm produce automata in the form preferred by the structural heuristic.

In the decomposition approach, instead of translating the property into one Büchi automaton A, we build three automata A_S, A_W, A_T of different strengths. These three automata are smaller than the original one, so checking them in parallel is necessarily faster. They also have a simpler structure, with less transitions in the acceptance sets of A_S and only one acceptance set for A_W and A_T, so they can be simplified more easily than A, improving the run time even more. Last but not least, more efficient algorithms are used for the emptiness check of $K \otimes A_W$ and $K \otimes A_T$.

Although we have experimented this approach with LTL formula, it will obviously work with any logic that can be translated into Büchi automata: for instance our implementation actually supports PSL. Similarly, we have experimented with some custom explicit and symbolic model checkers, but the same approach would be easily applied to any model checker based on the automata-theoretic approach. For instance we can decompose a property into three never claims to feed to the Spin model checker [17] and benefit from its partial-order reduction; or this approach could be integrated in VIS [4] and benefit from its SAT-based emptiness checks.

References

[1] Barnat, J., Brim, L., Ročkai, P.: On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties. Science of Computer Programming 77(12), 1272–1288 (2012)

[2] Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 222–235. Springer, Heidelberg (1999)

[3] Boigelot, B., Jodogne, S., Wolper, P.: On the use of weak automata for deciding linear arithmetic with integer and real variables. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 611–625. Springer, Heidelberg (2001)

[4] Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: VIS: A System For Verification and Synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 428–432. Springer, Heidelberg (1996)

[5] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 1–33. IEEE Computer Society Press, Washington, D.C (1990)

[6] Černá, I., Pelánek, R.: Relating Hierarchy of Temporal Properties to Model Checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003)

[7] Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithm for the verification of temporal properties. Formal Methods in System Design 1, 275–288 (1992)

[8] Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)

[9] Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-Fly Emptiness Checks for Generalized Büchi Automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)

[10] Dax, C., Eisinger, J., Klaedtke, F.: Mechanizing the Powerset Construction for Restricted Classes of ω-Automata. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 223–236. Springer, Heidelberg (2007)

[11] Duret-Lutz, A.: LTL translation improvements in Spot. In: Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2011). Electronic Workshops in Computing. British Computer Society, Tunis (2011), `http://ewic.bcs.org/category/15853`

[12] Duret-Lutz, A., Klai, K., Poitrenaud, D., Thierry-Mieg, Y.: Self-Loop Aggregation Product — A New Hybrid Approach to On-the-Fly LTL Model Checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 336–350. Springer, Heidelberg (2011)

[13] Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2-3), 247–267 (2004)
[14] Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is There a Best Symbolic Cycle-Detection Algorithm? In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 420–434. Springer, Heidelberg (2001)
[15] Geldenhuys, J., Hansen, H., Valmari, A.: Exploring the Scope for Partial Order Reduction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 39–53. Springer, Heidelberg (2009)
[16] Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoretical Computer Science 345(1), 60–82 (2005); Conference paper selected for journal publication
[17] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
[18] Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) Proceedings of the 2nd Spin Workshop. DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. American Mathematical Society (May 1996)
[19] Kesten, Y., Pnueli, A., Raviv, L.-O.: Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)
[20] Loizou, G., Thanisch, P.: Enumerating the cycles of a digraph: A new preprocessing strategy. Information Sciences 27(3), 163–182 (1982)
[21] Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
[22] Peled, D., Valmari, A., Kokkarinen, I.: Relaxed visibility enhances partial order reduction. Formal Methods in System Design 19(3), 275–289 (2001)
[23] Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
[24] Somenzi, F., Bloem, R.: Efficient Büchi automata for LTL Formulæ. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 247–263. Springer, Heidelberg (2000)
[25] Tauriainen, H.: Nested emptiness search for generalized Büchi automata. In: Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004), pp. 165–174. IEEE Computer Society (June 2004)