On Refinement of Büchi Automata for Explicit Model Checking

František Blahoudek[1], Alexandre Duret-Lutz[2], Vojtěch Rujbr[1], and Jan Strejček[1]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xblahoud, xrujbr, strejcek}@fi.muni.cz
[2] LRDE, EPITA, Le Kremlin-Bicêtre, France
adl@lrde.epita.fr

Abstract. In explicit model checking, systems are typically described in an implicit and compact way. Some valid information about the system can be easily derived directly from this description, for example that some atomic propositions cannot be valid at the same time. The paper shows several ways to apply this information to improve the Büchi automaton built from an LTL specification. As a result, we get smaller automata with shorter edge labels that are easier to understand and, more importantly, for which the explicit model checking process performs better.

1 Introduction

LTL model checking can be formulated as the problem of deciding whether a given system has an erroneous behavior specified by an LTL formula φ. In the automata-based approach to model checking, φ is translated into an equivalent Büchi automaton $\mathcal{A}_\varphi$ called *property automaton.* The original problem then reduces to deciding whether there exists a behavior of the system accepted by $\mathcal{A}_\varphi$. In explicit model checking, this is achieved by building a synchronous product of the system and the property automaton, and checking whether the product contains any reachable accepting cycle. This *emptiness check* can be done by several algorithms including the well-known *Nested Depth-First Search (NDFS)* [12] implemented in the model checker Spin [11]. The synchronous product is often constructed *on-the-fly*, i.e., simultaneously with the emptiness check and according to its needs. The product construction and the emptiness check form typically the most expensive part of the whole model checking process as the product to be explored is often very large. The actual difficulty of the check depends not only on the number of states in the product, but also on the number of transitions, the number and positions of accepting states, and other characteristics of the product. As the property automaton $\mathcal{A}_\varphi$ is a component of the product, the difficulty partly depends on the size and other characteristics of $\mathcal{A}_\varphi$.

For several decades, developers of algorithms and tools to translate LTL formulas into Büchi automata have aimed to produce small automata in short time. More recently, there was also a shift into producing automata that are more deterministic, as Sebastiani and Tonetta [16] identified a relation between the performance of the model checking and the determinism of the property automata. As a result, current LTL to Büchi automata translators like Spot [6] and LTL3BA [2] produce relatively small automata that are often deterministic.

One way to create property automata that further accelerate the model checking process is to provide more information for the translation than just the LTL formula. For example, we have recently shown that the position of accepting states in the property automaton can be adjusted according to the expected result of the model checking process: if we expect that the system has no erroneous behavior specified by φ, we can move the accepting states of $\mathcal{A}_\varphi$ further from its initial state to accelerate the model checking [4]. Analogously, relocation of accepting states in the opposite direction can speed up the model checking process if the system contains an error specified by φ.

In this paper, we try to improve the property automata using partial information about the behaviors of the system. More precisely, we use information about combinations of atomic propositions (and their negations) that cannot occur in any state of the system. For example, $x = 5$ and $x > 10$ cannot hold at once. Similarly, a process cannot be in two different locations at the same time. Information about these *incompatible propositions* can often be easily obtained from an implicit description of the system, i.e., without building its state space.

We show that this *a priori knowledge* about incompatible propositions can increase the efficiency of explicit model checking of linear-time properties by *refining* the specification to be checked. In Section 3, we show how to perform this refinement when the specification is given either by an LTL formula (or even a PSL formula) or by a Büchi automaton (or other kind of an ω-automaton). We talk about *formula refinement* or *automaton refinement*, respectively.

By refinement, we get a property automaton that may have fewer edges or even fewer states than the initial property automaton. All these changes often have a positive effect on the rest of the model checking process, as documented by experimental evaluation in Section 4.

As a side effect of the specification refinement, we typically obtain automata with long edge labels. Section 5 shows that complex edge labels have a small, but measurable negative effect on the execution time of Spin. Fortunately, Section 5 also introduces a method that employs the information about incompatible propositions to simplify the labels.

Finally, Section 6 discusses some interesting cases discovered during our intensive experiments.

2 Preliminaries

Let AP be a finite set of atomic propositions. Besides atomic propositions talking about values of program variables (like $x = 5$ or $y < 10$) and their relations (like $x < y$ or $x{\cdot}y = z{+}2$), we also work with atomic propositions of the form $p@loc$ saying that process p is in location loc.

Let $\mathbb{B} = \{\top, \bot\}$ represent Boolean values. An assignment is a function $\ell : AP \to \mathbb{B}$ that valuates each proposition. $\mathbb{B}^{AP}$ is the set of all assignments.

We assume familiarity with Linear-time Temporal Logic (LTL) [15]. Our examples use mainly the temporal operators $\mathsf{F}\varphi$ (meaning that φ *eventually* holds) and $\mathsf{G}\varphi$ (saying that φ *always* holds), but the results are valid for property formulas of any linear-time logic including the linear fragment of PSL [1].

A *Büchi automaton* (BA or simply an *automaton*) is a tuple $\mathcal{A} = (Q, q_0, \delta, F)$, where 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 a transition relation labeling each transition by an assignment, and $F \subseteq Q$ is a set of accepting states. Every triple $(r_1, \ell, r_2) \in \delta$ is called a *transition* from r_1 to r_2 under ℓ. As an implementation optimization, and to simplify illustrations, we often use *edges* labeled by Boolean formulas to group transitions with same sources and destinations: an edge $(r_1, a \vee \neg b, r_2)$ represents all transitions from r_1 to r_2 labeled with assignments ℓ such that $\ell(a) = \top$ or $\ell(b) = \bot$. To shorten the notation of edge labels, we write $\bar{a}$ instead of $\neg a$ and we omit $\wedge$ in conjunctions of atomic propositions (e.g., $a\bar{b}$ stands for $a \wedge \neg b$). An infinite sequence $\pi = (r_1, \ell_1, r_2)(r_2, \ell_2, r_3)\ldots \in \delta^\omega$ where $r_1 = q_0$ is a *run* of $\mathcal{A}$ over the word $\ell_1\ell_2\ldots$. The run is *accepting* if some accepting state appears infinitely often in π. A word is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ over that word. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A})$ of all words accepted by $\mathcal{A}$.

Kripke structures are a low-level formalism representing finite state systems. A Kripke structure is a tuple $\mathcal{S} = (S, s_0, R, L)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $R \subseteq S \times S$ is a transition relation, $L : S \to \mathbb{B}^{AP}$ is a labeling function. A *product* of a Kripke structure $\mathcal{S} = (S, s_0, R, L)$ and an automaton $\mathcal{A} = (Q, q_0, \delta, F)$ is the automaton $\mathcal{S} \otimes \mathcal{A}$ defined as $(S \times Q, (s_0, q_0), \delta', S \times F)$, where $\delta' = \{((s_1, q_1), l, (s_2, q_2)) \mid (s_1, s_2) \in R,\ (q_1, l, q_2) \in \delta,\ L(s_1) = l\}$.

3 Specification Refinement

Assume that we have a Kripke structure $\mathcal{S}$ and an LTL formula φ that describes the infinite erroneous behaviors we do not want to see in $\mathcal{S}$. Let $AP(\varphi)$ denote the set of atomic propositions in φ. A typical explicit model checker translates φ as a Büchi automaton $\mathcal{A}_\varphi$, and then constructs a product $\mathcal{S} \otimes \mathcal{A}_\varphi$ while simultaneously checking whether the language of this product is empty or not. As the product accepts all behaviors of the system also accepted by the automaton $\mathcal{A}_\varphi$, the system contains an error if and only if $L(\mathcal{S} \otimes \mathcal{A}_\varphi) \neq \emptyset$.

In practice, the system $\mathcal{S}$ is often described in some high-level formalism, which can be a programming language or a dedicated modeling language like Promela [11, Ch. 3]. This high-level description is translated into (the relevant part of) the corresponding Kripke structure during construction of the product.

The high-level description already provides some relevant information about the system. In particular, one can detect that some combinations of propositions in $AP(\varphi)$ and their negations are never valid at the same time. For instance, $x > 10$, $y < 5$, and $x < y$ cannot hold together. This information follows directly from the atomic propositions themselves. However, a static analysis of the system can identify more impossible combinations. For instance, the analysis can find out that if a process p is in a location loc, then local variable $p{:}x$ has value 0, and thus atomic propositions $p@loc$ and $p{:}x > 0$ never hold together. In the following, we assume that we are given a *constraint* κ, which is a Boolean formula over $AP(\varphi)$ satisfied by all combinations of atomic propositions except the invalid combinations. For example, the constraints corresponding to the two instances mentioned above are $\neg((x > 10) \wedge (y < 5) \wedge (x < y))$ and $\neg((p@loc) \wedge (p{:}x > 0))$.

One can frequently detect sets of atomic propositions that are mutually exclusive. For example, atomic propositions saying that a process p is in various locations (e.g., $p@loc1$, $p@loc2$, and $p@loc3$) are always mutually exclusive. Similarly, atomic propositions talking about values of the same variable (e.g., $x > 10$ and $x < 5$) are often contradictory. For a set $\mathcal{E}$ of mutually exclusive propositions (also called *exclusive set*), we define the constraint as:

$$excl(\mathcal{E}) = \bigwedge_{\substack{u,v \in \mathcal{E} \\ u \neq v}} \neg(u \wedge v)$$

While such a constraint may seems obvious to the reader, tools that translate LTL formulas into Büchi automata do not analyze the semantics of atomic propositions and thus they do not know that $x > 10$ and $x < 5$ are incompatible.

3.1 Formula Refinement

The *refinement* of an LTL formula φ with respect to a constraint κ is the formula $r_\kappa(\varphi)$ defined by

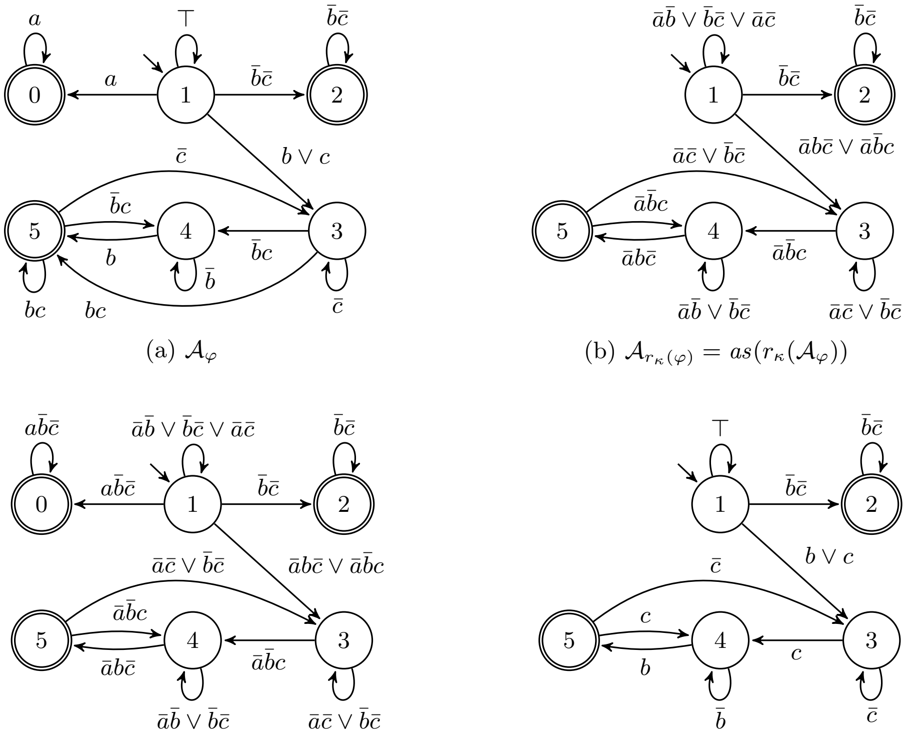
$$r_\kappa(\varphi) = \varphi \wedge \mathsf{G}\kappa.$$

where the knowledge about the constraint is made explicit.

This extra information allows tools that translate LTL formulas into automata to produce smaller automata. For instance the Büchi automaton of Figure 1(a) was generated by Spot [6] from the formula $\mathsf{F}(\mathsf{G}a \vee (\mathsf{GF}b \leftrightarrow \mathsf{GF}c))$. If the formula is refined with a constraint built for the exclusive set $\{a, b, c\}$, then the translator produces the smaller automaton from Figure 1(b): the edge between states 3 and 5 labeled by bc is known to be never satisfiable, and the state 0 is found to be superfluous (its incoming edges would be labeled by $a\bar{b}\bar{c}$, so this part of the automaton is covered by state 2 already).

3.2 Automaton Refinement

Alternatively, the refinement can be performed on the property automaton $\mathcal{A}$. This allows the specification of erroneous behaviors to be supplied directly as an automaton. Given an automaton $\mathcal{A}$ and a constraint κ, we obtain the refined automaton $r_\kappa(\mathcal{A})$ by replacing any edge (r_1, ℓ, r_2) of $\mathcal{A}$ by $(r_1, \ell \wedge \kappa, r_2)$ and removing the edge whenever the new label reduces to false. Figure 1(c) shows the result of applying this to the automaton of Figure 1(a). Note that as the edge labels are Boolean functions, they accept many representations: we display them


a
$\top$
$\bar{b}\bar{c}$
a
$\bar{b}\bar{c}$
$\bar{c}$
$b \vee c$
$\bar{b}c$
b
$\bar{b}c$
$\bar{b}$
bc
bc
$\bar{c}$
$\bar{a}\bar{b} \vee \bar{b}\bar{c} \vee \bar{a}\bar{c}$
$\bar{b}\bar{c}$
$\bar{b}\bar{c}$
$\bar{a}b\bar{c} \vee \bar{a}\bar{b}c$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$\bar{a}\bar{b}c$
$\bar{a}b\bar{c}$
$\bar{a}\bar{b}c$
$\bar{a}\bar{b} \vee \bar{b}\bar{c}$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$a\bar{b}\bar{c}$
$\bar{a}\bar{b} \vee \bar{b}\bar{c} \vee \bar{a}\bar{c}$
$\bar{b}\bar{c}$
$a\bar{b}\bar{c}$
$\bar{b}\bar{c}$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$\bar{a}b\bar{c} \vee \bar{a}\bar{b}c$
$\bar{a}\bar{b}c$
$\bar{a}b\bar{c}$
$\bar{a}\bar{b}c$
$\bar{a}\bar{b} \vee \bar{b}\bar{c}$
$\bar{a}\bar{c} \vee \bar{b}\bar{c}$
$\top$
$\bar{b}\bar{c}$
$\bar{b}\bar{c}$
$\bar{c}$
$b \vee c$
c
b
c
$\bar{b}$
$\bar{c}$
(a) $\mathcal{A}_\varphi$ (b) $\mathcal{A}_{r_\kappa(\varphi)} = as(r_\kappa(\mathcal{A}_\varphi))$

(c) $r_\kappa(\mathcal{A}_\varphi)$ (d) $ls(\mathcal{A}_{r_\kappa(\varphi)}) = ls(as(r_\kappa(\mathcal{A}_\varphi)))$

Fig. 1: Automata for $\varphi = \mathsf{F}(\mathsf{G}a \vee (\mathsf{GF}b \leftrightarrow \mathsf{GF}c))$ and $\kappa = excl(\{a, b, c\})$.

as some irredundant sums-of-products[3] by convention. In this case, state 0 is not removed, but it can be removed if we run some simplification algorithms (such as simulation-based reductions [3]), which are often employed in LTL to automata translators. The result of this simplification pass is then again in Figure 1(b).

If $as(\mathcal{A})$ denotes the operation that simplifies an automaton $\mathcal{A}$ using the same simplification algorithms that are used by a tool translating φ into $\mathcal{A}_\varphi$, one would expect that $\mathcal{A}_{r_\kappa(\varphi)} = as(r_\kappa(\mathcal{A}_\varphi))$ always holds (as in the example of Figure 1(b)). This is not true in practice for two reasons:

– Some translators have LTL rewriting rules that may react strangely to the refined formula, sometimes to the point of producing larger automata.
– Some translators include automata simplification algorithms (such a WDBA-minimization [5, 6]) that can only be applied when the formula is known, so they cannot be run on arbitrary automata.

[3] A sum-of-product is irredundant if all its products are prime implicants, and no product can be removed without changing the function [13].

Table 1: Considered LTL-to-BA translators, for reference.

translator	version	command
Spin [9, 11]	6.3.2	`spin`
LTL2BA [10]	1.1	`ltl2ba`
LTL3BA [2]	1.1.2	`ltl3ba`
LTL3BA-det		`ltl3ba -M0`
Spot [6]	1.99b	`ltl2tgba -s`
Spot-det		`ltl2tgba -s --deterministic`

Nonetheless, both formula refinement or automaton refinement have three noticeable effects on the model checking process:

– First, the removal of unsatisfiable transitions saves the model checker from having to repeatedly evaluate the labels of these transitions during the product construction, only to finally ignore them.
– Second, the automaton constructed with formula or automaton refinement is often smaller than the original automaton (for example, removing some transitions can make two states equivalent and such states can be merged). This can have a very positive effect on the model checking process.
– Last, the longer labels produced by this refinement may take longer to evaluate depending on how the model checker is implemented. This is the only negative effect, and we fix it in Section 5.

4 Experimental Evaluation

First we describe the general setting of our experiments. Then we show the impact of formula refinement and automaton refinement. Finally, we compare the two refinement approaches.

Benchmark. Our benchmark is made of 3316 verification tasks (i.e., a model plus a specification) where some propositions are referring to different locations of a single process so that we can construct exclusive sets. These tasks employ 101 instances of 16 parametrized models from Beem [14]; 50 tasks use specifications from Beem, the others combine Beem models with random LTL formulas.

Tools. In our experiments, we use four LTL-to-BA translators presented in Table 1. Two of the translators, namely LTL3BA and Spot, are used with two settings: the default ones and the settings with the suffix "-det" that aim to produce more deterministic automata. All translators are restricted by 20 minute timeout. For formula refinement and automata refinement, we use tools `ltlfilt` and `autfilt` from Spot 1.99.1. For emptiness checks, we use the same version of Spin with the maximum search depth set to 100 000 000, memory limit 20 GiB, option `-DNOSTUTTER` (see Section 6.3 for the explanation), and partial-order reduction enabled for tasks with *next*-free formulas. Emptiness check is always restricted by 30 minute timeout.

Table 2: Statistics of fails and successfully solved verification tasks with and without formula refinement.

	original tasks $(\mathcal{S}, \varphi)$			refined tasks $(\mathcal{S}, r_\kappa(\varphi))$			
translator	translation timeouts	Spin fails	tasks solved	translation timeouts	Spin fails	tasks solved	both tasks solved
Spin	801	232	2283	926	201	2189	2183
LTL2BA	5	341	2970	2	302	3012	2929
LTL3BA	0	80	3236	0	55	3261	3227
LTL3BA-det	0	34	3282	0	27	3289	3279
Spot	2	27	3287	0	19	3297	3286
Spot-det	2	26	3288	0	19	3297	3287

Hardware. All computations are performed on an HP DL980 G7 server with 8 eight-core processors Intel Xeon X7560 2.26GHz and 448 GiB DDR3 RAM. The server is shared with other users and its variable workload has led to a high dispersion of measured running times. Hence, instead of running times, we use the number of transitions visited by Spin, which is stable across multiple executions and should be proportional to the running time.

Additional data and detailed information about this benchmark are available at: `http://fi.muni.cz/~xstrejc/publications/spin2015/`

4.1 Impact of Formula Refinement

For each verification task $(\mathcal{S}, \varphi)$ and each translator of Table 1, we translate φ to automaton $\mathcal{A}_\varphi$ and run Spin on $\mathcal{S}$ and $\mathcal{A}_\varphi$. Then we refine the formula to $r_\kappa(\varphi)$ and repeat the process. Table 2 shows the numbers of translation timeouts, Spin fails (this number covers the cases when Spin timeouts or runs out of memory or reaches the maximum search depth), and successfully solved verification problems. The data indicates that formula refinement has a mostly positive effect on the model checking process: for all but one translator, the refinement increases the number of successfully solved tasks (we discuss the case of Spin translator in more details in Section 6.2). Nevertheless, the number of tasks solved both with and without formula refinement is always smaller that the number of solved original tasks, which means that the effect of formula refinement is negative in some cases. In the rest of this section, for each translator we consider only the tasks counted in the last column of the table, i.e., tasks solved both with and without formula refinement.

We now look at the effect of formula refinement on the sizes of property automata. Table 3 shows that the property automaton for a refined formula has very frequently fewer states than the automaton for the original formula. However, we cannot easily tell whether states are removed simply because they are inaccessible after refinement (i.e., the constraint κ removed all the transitions leading to a state) or if the refinement enabled additional simplifications as in

Table 3: Effect of formula refinement on property automata. For each translator and each verification task, we compare the size of $\mathcal{A}_\varphi$ with the size of $\mathcal{A}_{r_\kappa(\varphi)}$ and report on the number of cases where the refinement resulted in additional states (+states) or fewer states (−states). In case of equality, we look at the number of edges or transitions.

effect	Spin	LTL2BA	LTL3BA	LTL3BA-det	Spot	Spot-det
+states	514	41	15	148	13	17
−states	168	1482	1679	1723	1722	1720
=states,+edges	37	17	0	0	9	10
=states,−edges	43	337	293	326	345	344
=states,=edges,+trans.	153	211	283	173	280	280
=states,=edges,−trans.	1226	785	899	848	849	848
no size change	42	56	58	61	68	68

Figure 1. In the former case, the refinement would have a little impact on the size of the product: it is only saving useless attempts to synchronize transitions that can never be synchronized while building this product.

Finally, we turn our attention to the actual effect of formula refinement on performance of the emptiness check implemented in Spin. For each translator and each verification task, let t_1 be the number of transitions visited by Spin for the original task and t_2 be the same number for the refined task. Scatter plots of Figure 2 show each pair (t_1, t_2) as a dot at this coordinates. The color[4] of each dot says whether the property automaton for the refined formula has more or less states than the automaton for the original formula. The data is shown separately for each translator. We also distinguish the tasks with some erroneous behavior from those without error. As many dots in the scatter plots are overlapping, we present the data also via *improvement ratios* t_2/t_1. Values of t_2/t_1 smaller than 1 correspond to cases where formula refinement actually helped Spin, while values larger than 1 correspond to cases where the refinement caused Spin to work more. Figure 3 gives an idea of the distribution of these improvement ratios in our benchmark. On this figure, all improvement ratios for a given tool are sorted from lowest to highest, and then they are plotted using their rank as x coordinate, and using a logarithmic scale for the ratio. One can immediately see on these curves that there is a large plateau around $y = 1$ corresponding to the cases where there is no substantial improvement. In the tasks without error, there are usually many cases with ratio below 0.95 (definite improvement), and very few cases above 1.05 (cases where refinement hurt more than it helped). A special class of cases that are improved are those that are found equivalent to false after refinement: those usually have a very high improvement ratio, as the exploration of the product is now limited to a single transition (after which Spin immediately realizes that the empty never claim cannot be satisfied). Note that in tasks with error, the refined formula

[4] We suggest viewing these figures in color using the electronic version of this article.

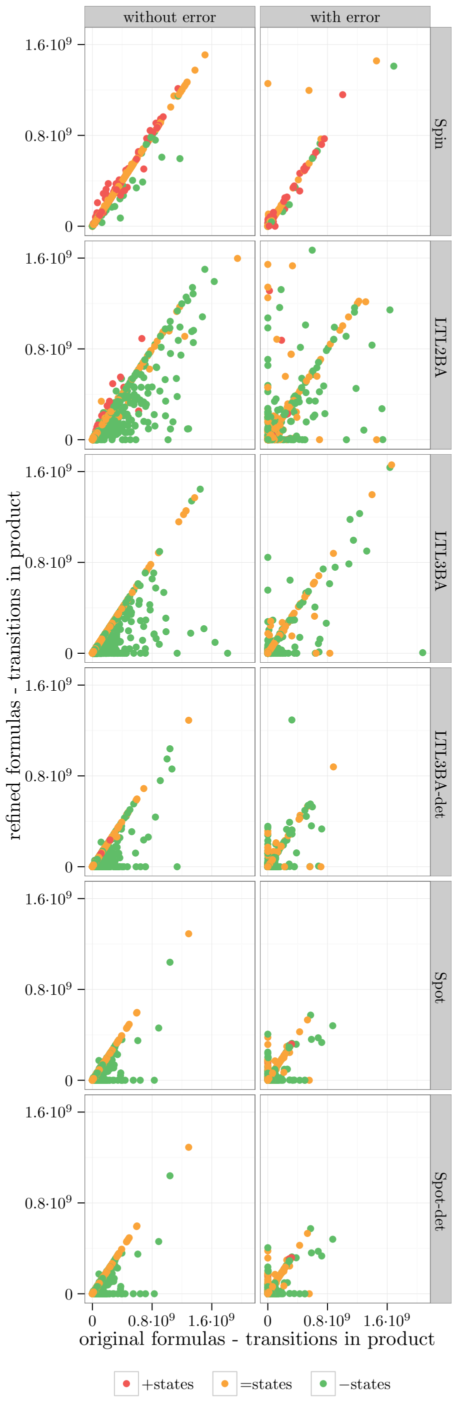


Fig. 2: Comparison of the numbers of product transitions visited by Spin on the original tasks (t_1) and their *formula*-refined versions (t_2).

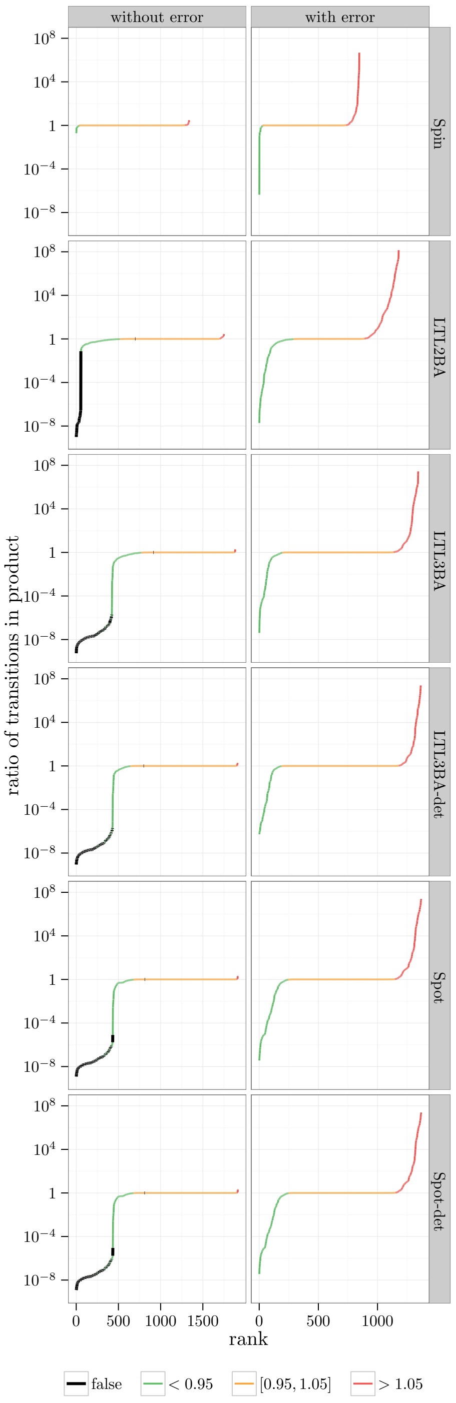


Fig. 3: Distribution of the improvement ratios (t_2/t_1) using logarithmic scales. Cases that have been reduced to false are highlighted in bold.

Table 4: Distribution of the improvement ratios for formula refinement. The counts of false cases are not included in the <0.95 classes.

	without error					with error			
	false	<0.95	[0.95,1.05]	>1.05	All	<0.95	[0.95,1.05]	>1.05	All
Spin	0	30	1257	50	1337	27	708	111	846
LTL2BA	61	462	1179	48	1750	288	602	289	1179
LTL3BA	374	401	1101	7	1883	194	942	208	1344
detLTL3BA	382	264	1255	12	1913	186	993	187	1366
Spot	384	300	1213	20	1917	244	902	223	1369
detSpot	385	297	1218	18	1918	248	903	218	1369

cannot be equivalent to false as all states of an erroneous behavior comply with the constraint. Relatively high numbers of these "false" cases imply that the formula refinement technique is an effective sanity check detecting specifications unsatisfiable under given constraints.[5] Table 4 gives counts of improvement ratios in these classes.

Figures 2 and 3 and Table 4 show that for tasks without error, formula refinement has negative effect only very rarely and such effect is relatively small. The positive effect is more frequent and substantial in many cases. The table implies that LTL3BA and Spot can profit more from the refinement as they identify radically more false cases and they have significantly less cases with negative effect than the other translators (some of the negative cases are discussed in Section 6). This observation can be explained by advanced simplification techniques implemented in LTL3BA and Spot.

In the tasks with erroneous behaviors, we observe that the number of improved cases is almost balanced by the number of degraded cases (except for Spin). This can be explained by the fact that refining an LTL formula my change the shape of the output automaton, and thus change its transition order. Therefore the model checker may have more or less luck in finding an erroneous run. When such a run is found, Spin ends the computation without exploring the rest of the product.

4.2 Impact of Automaton Refinement

As mentioned before, automaton refinement itself only cuts off some parts of the automaton that are not used in the product. It has a bigger effect only when simplification algorithms are executed after the refinement. In our experiments, we combined automaton refinement with the automata simplifications implemented in Spot.

To measure the effect of automaton refinement, we prepared the benchmark as follows. We took the 3316 verification tasks used before. For every task, we

[5] The high number of "false" cases is due to the use of random formulas. In real tasks, such a *false* case would likely indicate a bug in the specification.

Table 5: Statistics of fails and successfully solved verification tasks with and without automata refinement.

original tasks $(\mathcal{S}, \mathcal{A})$		refined tasks $(\mathcal{S}, as(r_\kappa(\mathcal{A})))$			
Spin fails	tasks solved	simplification of $r_\kappa(\mathcal{A})$ timeouts	Spin fails	tasks solved	both tasks solved
291	9061	12	99	9241	9038

Table 6: Effect of automaton refinement on property automata.

effect	
+states	0
−states	4955
=states,+edges	0
=states,−edges	1013
=states,=edges,+trans.	0
=states,=edges,−trans.	2400
no size change	670

Table 7: Distribution of the improvement ratios for automaton refinement.

	without error	with error
false	906	0
< 0.95	853	735
$[0.95, 1.05]$	3251	2743
> 1.05	5	545
All	5015	4023

translated the formula with all considered translators and simplified the produced automata using Spot. The simplification is here applied to make the comparison of model checking with and without automaton refinement fair: without this step we could not really distinguish the effect of automata refinement (followed by simplification) from the effect of simplification itself. If the automaton translation and simplification successfully finishes, we get a pair of a model and a simplified automaton. In the rest of this section, we call such pairs *verification tasks*. After removing duplicates, we have 9352 verification tasks.

For each task, we run Spin with the original automaton. Then we refine and simplify the automaton and run Spin again. While automaton refinement is very cheap, its simplification can be quite expensive. So we apply a 20 minute timeout. Table 5 provides numbers of Spin fails on original tasks, timeouts of refined automata simplifications, and Spin failures on refined tasks. In the following, we work only with tasks solved both with and without automaton refinement.

As in the previous section, Table 6 presents the effect of automaton refinement and simplification on the sizes of property automata. The refined and simplified automata are smaller in the vast majority of cases and never bigger.

The effect of automaton refinement and simplification on performance of emptiness check in Spin is presented in Figures 4 and 5, and Table 7 in the same way as previously. On tasks without error, the effect is similar to formula refinement: it is often positive and almost never negative. On tasks with error, the positive effect is more frequent than the negative one.

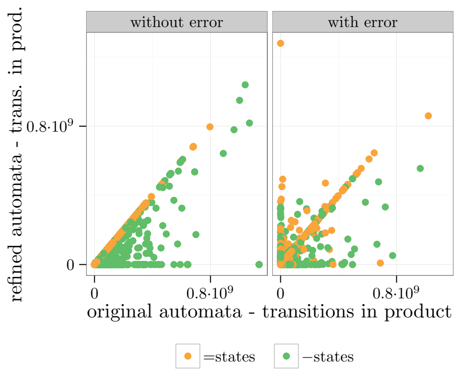


Fig. 4: Comparison of the numbers of product transitions visited by Spin on the original tasks (t_1) and their *automata*-refined versions (t_2).

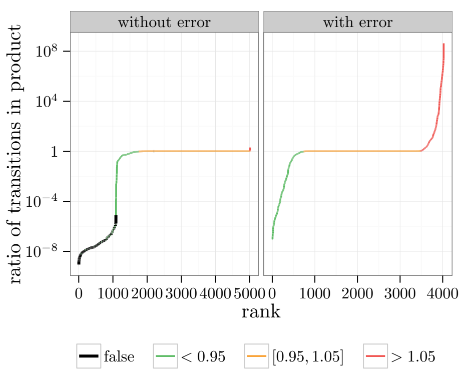


Fig. 5: Distribution of the improvement ratios (t_2/t_1) using logarithmic scales. Cases that have been reduced to false are highlighted in bold.

Table 8: Statistics of fails and successfully solved verification tasks with formula refinement and automaton refinement.

tasks with formula refinement			tasks with automaton refinement			
automaton construction timeouts	Spin fails	tasks solved	automaton construction timeouts	Spin fails	tasks solved	both tasks solved
0	19	3297	35	25	3256	3256

4.3 Comparison of Formula and Automaton Refinement

Here we compare the formula refinement and automaton refinement using Spot for the formula translation. For each of the 3316 considered tasks, we refine the formula, translate it by Spot, and run Spin. Then we take the task again, translate the original formula by Spot, refine and simplify the automaton, and run Spin. Table 8 provides statistics about automata construction timeouts (this comprises Spot timeouts and, in the case of automaton refinement, also simplification of refined automata timeouts), Spin timeouts, and solved tasks. Both approaches detected 380 identical cases where the refined specification reduces to false. In the following, we present the data from the $3256 - 380 = 2876$ tasks solved by both approaches and not trivially equivalent to false.

Table 9, Figures 6 and 7, and Table 10 are analogous to the tables and figures in the previous sections (the position of original tasks in the previous sections is taken by tasks with formula refinement). Table 9 says that automaton refinement often produces property automata with more states than formula refinement. However, Figure 6 and Table 10 show that the overall effect of automata and

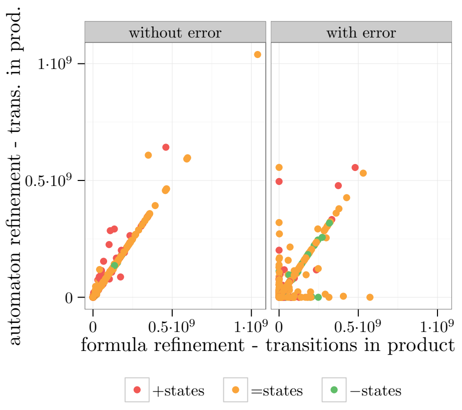


Fig. 6: Comparison of the numbers of product transitions visited by Spin in *formula*-refined tasks (t_1) and their *automata*-refined versions (t_2).

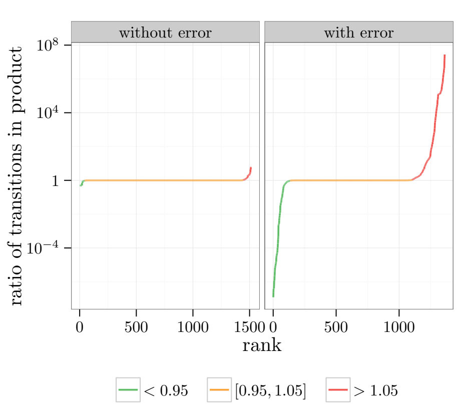


Fig. 7: Distribution of the improvement ratios (t_2/t_1) using logarithmic scales. Cases that have been reduced to false are highlighted in bold.

formula refinement on performance of Spin is fully comparable, slightly in favor of formula refinement.

5 Label Simplification

As mentioned in Section 3, a side-effect of specification refinement is that edges get more complex labels. This is visible when comparing the automaton of Figure 1(b) to the one of Figure 1(a). For example the self-loop on state 3 is labeled by $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ instead of the original $\bar{c}$. In our experiment, the overall average length of an edge label (counted as the number of occurrences of atomic propositions

Table 9: Comparison of automata produced by formula refinement and automaton refinement (+states counts tasks where $as(r_\kappa(\mathcal{A}_\varphi))$ has more states than $\mathcal{A}_{r_\kappa(\varphi)}$ and so on).

effect	
+states	315
−states	82
=states,+edges	52
=states,−edges	51
=states,=edges,+trans.	26
=states,=edges,−trans.	428
no size change	1922

Table 10: Distribution of the improvement ratios for automaton refinement over formula refinement.

	without error	with error
< 0.95	44	133
$[0.95, 1.05]$	1399	970
> 1.05	71	259
All	1514	1362

```
if (!(((!((((int)((P1 *)Pptr(f_pid(1)))->_p)==27))&&
       !((((int)((P1 *)Pptr(f_pid(1)))->_p)==5)))||
       (!((((int)((P1 *)Pptr(f_pid(1)))->_p)==27))&&
       !((((int)((P1 *)Pptr(f_pid(1)))->_p)==9)))))) ...
```

```
if (!( !((((int)((P1 *)Pptr(f_pid(1)))->_p)==27)))) ...
```

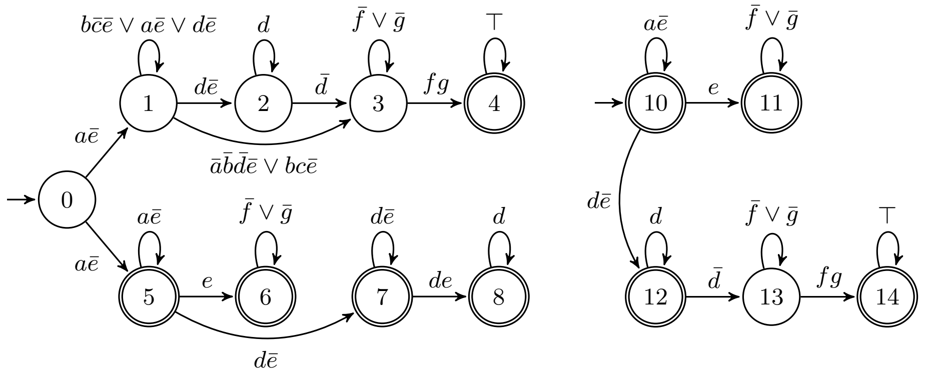
Fig. 8: Code listings of a `pan.m` file. The upper part resulted from the edge labeled by $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ and the last line is from label $\bar{c}$.

in the label) in the automata $\mathcal{A}_{r_\kappa(\varphi)}$ for refined formulas is 6.58, while the average label length in the corresponding automata $\mathcal{A}_\varphi$ for unrefined formulas is only 4.20. When executing Spin, the labels are compiled into C code to match system transitions during the construction of the synchronized product. For example, Figure 8 depicts the C code corresponding to the labels $\bar{a}\bar{c} \vee \bar{b}\bar{c}$ and $\bar{c}$. Clearly, longer labels can slow down the verification process without influencing any Spin statistics like visited transitions and stored states. However, the expected slowdown should be only small as checking the labels is much cheaper than computing successors for states of the system or storing the states.

To eliminate the slowdown, we simplify the labels in a step that can be though of as the converse of refinement: instead of using a given constraint to make labels more precise, we use it to make them *less* precise and shorter, but equivalent to the original labels under the given constraint. For instance, $b\bar{c}$ can be shortened as b if we know that b and c cannot be both true in the model. This simplification can be implemented by performing Boolean function simplification with *don't care* information: we do not care if the simplified label additionally covers some variable assignments that can never happen in the system. Concretely, we have implemented the simplification in Spot using the Minato-Morreale algorithm [13]. The algorithms inputs two Boolean functions $\lfloor f \rfloor$ and $\lceil f \rceil$ and produces an irredundant sum-of-product that covers at least all the assignments satisfying $\lfloor f \rfloor$, and that is not satisfiable by at least all the assignments not satisfying $\lceil f \rceil$. To simplify a label ℓ using a constraint κ, we call the algorithm with $\lfloor f \rfloor = \ell \wedge \kappa$ and $\lceil f \rceil = \ell \vee \neg\kappa$. Figure 1(d) shows the result of applying this label simplification (denoted as function *ls*) to Figure 1(b).

Note: The definition of $\lceil f \rceil$ is bogus in the proceedings of SPIN'15. This version of the paper is fixed.

We applied the label simplification to automata obtained by formula refinement and the average label length drops to 3.19, which is even lower that the mentioned value for automata without refinement. We selected several cases with high reduction of label length and run Spin several times with automata before and after label simplification on a weaker, but isolated machine to get reliable running times. In these tests, Spin runs up to 3.5% slower with automata before label simplification.



(a) useful part of $\mathcal{A}_\varphi$ (b) $\mathcal{A}_{r_\kappa(\varphi)}$

Fig. 9: An uncommon case where $\mathcal{A}_{r_\kappa(\varphi)}$ is much smaller than $\mathcal{A}_\varphi$, and yet Spin performs better with $\mathcal{A}_\varphi$.

6 Interesting Cases

In this section we investigate several interesting cases where using refinement caused worse performance.

6.1 The Case of Strongly Connected Components

Figure 9 shows an interesting case that we discovered among the few tasks without error where the refined formula translated by Spot degrades the performance of Spin. In this case, Spin performs better with the automaton $\mathcal{A}_\varphi$ of Figure 9(a) than with the smaller automaton $\mathcal{A}_{r_\kappa(\varphi)}$ of Figure 9(b). Please note that the automaton presented in Figure 9(a) is a pruned version of the real automaton, in which we removed all transitions that do not appear in the product with the model. For instance, in this pruned automaton it is obvious that the state 7 can be merged with state 8, but the presence of other edges in the original automaton prevented this simplification.

The reason Spin works better with the larger of these two automata is related to the emptiness check used. The emptiness check procedure used in Spin by default is based on two nested depth-first searches [12]: the main DFS, which we shall call *blue*, explores the product (on-the-fly) and every time it would backtrack from an accepting state s (i.e., all successors of s have been explored by the blue DFS) it starts a second, *red* DFS from s. If the red DFS reaches any state on the blue DFS search stack, then a reachable and accepting cycle is found (since s is reachable from all states on the blue DFS search stack) and the algorithm reports it as a counterexample. Otherwise, the red DFS terminates and the blue DFS can continue. The two DFS always ignore states that have been completely explored by an instance of the red DFS, so a state is never visited more than twice.

In the automaton of Figure 9(b), whenever the blue DFS backtracks a state of the product that is synchronized with state 12, it has to start a red DFS that will explore again states synchronized with 13 and previously explored by the blue DFS (states synchronized with 12 and 14 will be ignored as they have already been seen by a previous red DFS). This re-exploration of states synchronized with 13 is something that (i) did not happen in the original automaton because there is no accepting state above the corresponding state 3, and (ii) is useless because there is no way to get back to state 12 after moving to state 13.

The NDFS algorithm could be patched to avoid this problem by simply constraining the red DFS to explore only the states of the product whose projection on the property automaton belongs to the same strongly connected component as its starting accepting state. This optimization was already suggested by Edelkamp et al. [7, 8] with the additional trick that if the current SCC is known to be weak (i.e., its states are all accepting and or all non-accepting), then running a red DFS is not needed at all, as the blue DFS is guaranteed to find any accepting cycle by itself. In the scenarios described by Figures 9(a) and 9(b), all the SCCs have a single state, so the product automaton will be weak and the red DFS should not be needed. Computing the strongly connected components of the property automaton can be done in time that is linear to the size of that automaton (typically a small value) before the actual emptiness check starts, so this is a very cheap way to improve the model checking time.

6.2 Problems with LTL simplifications

A special class of interesting cases consists of formulas where formula refinement leads to bigger automata. Such cases are surprisingly often connected with issues in the earliest phases of LTL to automata translation, namely in formula parsing or simplification. For example, LTL3BA implements several specific formula reduction rules applied after all standard formula reductions. If such a rule is applied, the reduced formula is checked again for possible application of some reduction rule, but only on its top level. Hence, some reductions are not applied when the input formula is refined with a constraint. This is considered as a bug and it will be fixed in the next release of LTL3BA.

LTL2BA has even more problems with formula simplifications as it is sensitive to superfluous parentheses. For instance, the command `ltl2ba -f '<>([]<>X p)'` generates an automaton with 2 states, while the equivalent `ltl2ba -f '<>[]<>X p'` produces an automaton with 4 states. This is because the presence of parentheses causes another pass of formula reduction to occur.

Table 3 indicates that Spin's translator benefits less than the other translators from addition of constraints. Part of the problem, it seems, is due to a change that was introduced in Spin 6 to allow LTL formulas embedding atomic propositions with arbitrary Promela conditions. As a consequence of this change, many parenthetical blocks are now considered as atomic propositions by Spin's translator, and simplifications are therefore missed. For instance, the formula

$(a \mathsf{R} b) \wedge \mathsf{G}(\neg(a \wedge b))$ is translated as if $\neg(a \wedge b)$ was an independent atomic proposition. While Spin 5 translates this formula into an automaton with one state and one edge, Spin 6 outputs an automaton with two states and three edges, where the edge connecting the states has unsatisfiable label $\neg(a \wedge b) \wedge a \wedge b$.

6.3 Problem with Spin

During our experiments, we discovered a handful of cases where equivalent never claims would cause Spin to produce different results: e.g., a counterexample for automata built by some tools, and no counterexamples for (equivalent) automata built by other tools. Sometime the automata would differ only by the order in which the transitions are listed. In turned out that this bug[6] was due to a rare combination of events in the red DFS in the presence of a deadlock in the system. While it will be fixed in Spin 6.4.4, the fix came too late for us: our benchmark takes more than a week of computation. All the presented results are computed by compiling the Spin verifier with `-DNOSTUTTER`, which effectively means that we ignore deadlock scenario, and we are safe from this bug.

7 Conclusions

We have reported on the effect of using information about impossible combinations of propositions in the model to improve model checking. We proposed two techniques: *refinement* is the process of making this information explicit in the property automaton, while *label simplification* is the process of making this information implicit. Our experiments show that these two operations, that can be combined, have a positive effect on the model checking process. By refinement we are able to obtain automata that are usually smaller, and then by *label simplification* we shorten the labels of the automata to speedup the process of transition matching during model checking.

The refinement can also be used as a sanity check: when a refinement leads to a property automaton with no accepting state, it usually represent a bug in the specification.

In the experiments, we only considered incompatibilities between atomic propositions that denote a process being in different locations. More sources of incompatibilities could be considered, such as atomic propositions that refer to different variable values.

We could also extend the principle to more than just incompatible propositions: for instance from the model we could extract information about the validity of atomic propositions in the initial state, the order of locations in a process, or learn the fact that some variable will always be updated in a monotonous way (e.g., can only be increased). All these informations can be used to produce stricter property automata that disallow these impossible behaviors, and we think these automata should offer more opportunity for simplifications, and should also contribute to better sanity checks.

[6] `http://spinroot.com/fluxbb/viewtopic.php?pid=3316`

We demonstrated the usefulness of refinement to model checking, but we believe it should also be useful in other contexts like probabilistic model checking or controller synthesis.

Acknowledgments. The authors would like to thank Tomáš Babiak and Jiří Barnat for discussions and tool support. František Blahoudek and Jan Strejček have been supported by The Czech Science Foundation grant GBP202/12/G061.

References

1. Accellera. Property specification language reference manual v1.1, 2004. URL `http://www.eda.org/vfv/`.
2. T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS'12*, vol. 7214 of *LNCS*, pp. 95–109. Springer, 2012.
3. T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In *SPIN'13*, vol. 7976 of *LNCS*, pp. 81–98. Springer, 2013.
4. F. Blahoudek, A. Duret-Lutz, M. Křetínský, and J. Strejček. Is there a best Büchi automaton for explicit model checking? In *SPIN'14*, pp. 68–76. ACM, 2014.
5. C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In *ATVA'07*, vol. 4762 of *LNCS*. Springer, 2007.
6. A. Duret-Lutz. LTL translation improvements in Spot 1.0. *Int. J. on Critical Computer-Based Systems*, 5(1/2):31–54, 2014.
7. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN'01*, vol. 2057 of *LNCS*, pp. 57–79. Springer, 2001.
8. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2–3):247–267, 2004.
9. K. Etessami and G. J. Holzmann. Optimizing Büchi Automata. In *CONCUR'00*, vol. 1877 of *LNCS*, pp. 153–167. Springer, 2000.
10. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, vol. 2102 of *LNCS*, pp. 53–65. Springer, 2001.
11. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
12. G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In *SPIN'96*, vol. 32 of *DIMACS*. American Mathematical Society, 1996.
13. S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *SASIMI'92*, pp. 64–73, 1992.
14. R. Pelánek. BEEM: benchmarks for explicit model checkers. In *SPIN'07*, vol. 4595 of *LNCS*, pp. 263–267. Springer, 2007.
15. A. Pnueli. The temporal logic of programs. In *FOCS'77*, pp. 46–57. IEEE, 1977.
16. R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *CHARME'03*, vol. 2860 of *LNCS*, pp. 126–140. Springer, 2003.