Computing and Informatics, Vol. 29, 2010, 1337–1348

OPTIMIZING AN LTS-SIMULATION ALGORITHM

Lukáš Holík, Jiří Šimáček

Faculty of Information Technology
Brno University of Technology
Božetěchova 2
612 66 Brno, Czech Republic
e-mail: {holik, isimacek}@fit.vutbr.cz

Revised manuscript received 11 May 2010

Abstract. When comparing the fastest algorithm for computing the largest simulation preorder over Kripke structures with the one for labeled transition systems (LTS), there is a noticeable time and space complexity blow-up proportional to the size of the alphabet of an LTS. In this paper, we present optimizations that suppress this increase of complexity and may turn a large alphabet of an LTS to an advantage. Our experimental results show significant speed-ups and memory savings. Moreover, the optimized algorithm allows one to improve asymptotic complexity of procedures for computing simulations over tree automata using recently proposed algorithms based on computing simulation over certain special LTS derived from a tree automaton.

1 INTRODUCTION

A practical limitation of automated methods dealing with LTSs – such as LTL model checking, regular model checking, etc. – is often the size of generated LTSs. One of the well established approaches to overcome this problem is the reduction of an LTS using a suitable equivalence relation according to which the states of the LTS are collapsed. A good candidate for such a relation is simulation equivalence. It strongly preserves logics like $ACTL^*$, $ECTL^*$, and LTL [6, 7, 8], and with respect to its reduction power and computation cost, it offers a desirable compromise among the other common candidates, such as bisimulation equivalence [10, 12] and language equivalence. The currently fastest LTS-simulation algorithm (denoted as LRT – labeled RT) has been published in [1]. It is a straightforward modification of the fastest algorithm (denoted as RT, standing for Ranzato-Tapparo) for computing

simulation over Kripke structures [11], which improves the algorithm from [8]. The time complexity of RT amounts to $\mathcal{O}(|P_{Sim}||\delta|)$, space complexity to $\mathcal{O}(|P_{Sim}||S|)$. In the case of LRT, we obtain time complexity $\mathcal{O}(|P_{Sim}||\delta| + |\Sigma||P_{Sim}||S|)$ and space complexity $\mathcal{O}(|\Sigma||P_{Sim}||S|)$. Here, S is the set of states, δ is the transition relation, Σ is the alphabet and P_{Sim} is the partition of S according to the simulation equivalence. The space complexity blow-up of LRT is caused by indexing the data structures of RT by the symbols of the alphabet.

In this paper, we propose an optimized version of LRT (denoted OLRT) that lowers the above-described blow-up. We exploit the fact that not all states of an LTS have incoming and outgoing transitions labeled by all symbols of an alphabet, which allows us to reduce the memory footprint of the data structures used during the computation. Our experiments show that the optimizations we propose lead to significant savings of space as well as of time in many practical cases. Moreover, we have achieved a promising reduction of the asymptotic complexity of algorithms for computing tree-automata simulations from [1] using OLRT.

2 PRELIMINARIES

Given a binary relation ρ over a set X, we use $\rho(x)$ to denote the set $\{y \mid (x, y) \in \rho\}$. Then, for a set $Y \subseteq X$, $\rho(Y) = \bigcup\{\rho(y) \mid y \in Y\}$. A *partition-relation pair* over X is a pair $\langle P, Rel\rangle$ where $P \subseteq 2^X$ is a partition of X (we call elements of P *blocks*) and $Rel \subseteq P \times P$. A partition-relation pair $\langle P, Rel\rangle$ *induces* the relation $\rho = \bigcup_{(B,C)\in Rel} B \times C$. We say that $\langle P, Rel\rangle$ is the *coarsest partition-relation pair* inducing ρ if any two $x, y \in X$ are in the same block of P if and only if $\rho(x) = \rho(y)$ and $\rho^{-1}(x) = \rho^{-1}(y)$. Note that in the case when ρ is a preorder and $\langle P, Rel\rangle$ is coarsest, then P is the set of equivalence classes of $\rho \cap \rho^{-1}$ and Rel is a partial order.

A *labeled transition system (LTS)* is a tuple $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$, where S is a finite set of states, Σ is a finite set of labels, and for each $a \in \Sigma$, $\delta_a \subseteq S \times S$ is an a-labeled transition relation. We use δ to denote $\bigcup_{a\in\Sigma} \delta_a$. A *simulation* over T is a binary relation ρ on S such that if $(u, v) \in \rho$, then for all $a \in \Sigma$ and $u' \in \delta_a(u)$, there exists $v' \in \delta_a(v)$ such that $(u', v') \in \rho$. It can be shown that for a given LTS T and an *initial preorder* $I \subseteq S \times S$, there is a unique maximal simulation Sim_I on T that is a subset of I, and that Sim_I is a preorder (see [1]).

3 THE ORIGINAL LRT ALGORITHM

In this section, we describe the original version of the algorithm presented in [1], which we denote as LRT (see Algortihm 1).

The algorithm gradually refines a partition-relation pair $\langle P, Rel\rangle$, which is initialized as the coarsest partition-relation pair inducing an initial preorder I. After its termination, $\langle P, Rel\rangle$ is the coarsest partition-relation pair inducing Sim_I. The basic invariant of the algorithm is that the relation induced by $\langle P, Rel\rangle$ is always a superset of Sim_I.

Algorithm 1: (O)LRT Algorithm

Input: an LTS $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$, partition-relation pair $\langle P_I, Rel_I \rangle$
Output: partition-relation pair $\langle P, Rel \rangle$

```
   /* initialization */
1  ⟨P, Rel⟩ ← ⟨P_I, Rel_I⟩                          /* ← ⟨P_{I∩Out}, Rel_{I∩Out}⟩ */
2  forall the B ∈ P and a ∈ Σ do                    /* a ∈ in(B) */
3  |  forall the v ∈ S do Count_a(v, B) = |δ_a(v) ∩ ⋃Rel(B)| ;  /* v ∈ δ_a^{-1}(S) */
4  |  Remove_a(B) ← S \ δ_a^{-1}(⋃Rel(B))          /* ← δ_a^{-1}(S)\δ_a^{-1}(⋃Rel(B)) */
   /* computation */
5  while exists B ∈ P and a ∈ Σ such that Remove_a(B) ≠ ∅ do
6  |  Remove ← Remove_a(B);
7  |  Remove_a(B) ← ∅;
8  |  ⟨P_prev, Rel_prev⟩ ← ⟨P, Rel⟩;
9  |  P ← Split(P, Remove);
10 |  Rel ← {(C, D) ∈ P × P | (C_prev, D_prev) ∈ Rel_prev};
11 |  forall the C ∈ P and b ∈ Σ do                 /* b ∈ in(C) */
12 |  |  Remove_b(C) ← Remove_b(C_prev);
13 |  |  forall the v ∈ S do Count_b(v, C) ← Count_b(v, C_prev) ;
   |  |  /* v ∈ δ_b^{-1}(S) */
14 |  forall the C ∈ P such that C ∩ δ_a^{-1}(B) ≠ ∅ do
15 |  |  forall the D ∈ P such that D ⊆ Remove do
16 |  |  |  if (C, D) ∈ Rel then
17 |  |  |  |  Rel ← Rel \ {(C, D)};
18 |  |  |  |  forall the b ∈ Σ and v ∈ δ_b^{-1}(D) do /* b ∈ in(D) ∩ in(C) */
19 |  |  |  |  |  Count_b(v, C) ← Count_b(v, C) − 1;
20 |  |  |  |  |  if Count_b(v, C) = 0 then
21 |  |  |  |  |  |  Remove_b(C) ← Remove_b(C) ∪ {v}
```

The while-loop refines the partition P and then prunes the relation Rel in each iteration of the while-loop. The role of the $Remove$ sets can be explained as follows: During the initialization, every $Remove_a(B)$ is filled by states v such that $\delta_a(v) \cap \bigcup Rel(B) = \emptyset$ (there is no a-transition leading from v "above" B wrt. Rel). During the computation phase, v is added into $Remove_a(B)$ after $\delta_a(v) \cap \bigcup Rel(B)$ becomes empty (because of pruning Rel on line 17). Emptiness of $\delta_a(v) \cap Rel(B)$ is tested on line 20 using counters $Count_a(v, B)$, which record the cardinality of $\delta_a(v) \cap Rel(B)$. From the definition of simulation, and because the relation induced by $\langle P, Rel \rangle$ is always a superset of Sim_I, $\delta_a(v) \cap \bigcup Rel(B) = \emptyset$ implies that for all $u \in \delta_a^{-1}(B)$, $(u, v) \notin Sim_I$ (v cannot simulate any $u \in \delta_a^{-1}(B)$). To reflect this, the relation Rel is pruned each time $Remove_a(B)$ is processed. The code on lines 8–13 prepares the partition-relation pair and all the data structures. First, $Split(P, Remove_a(B))$

divides every block B' into $B' \cap \mathit{Remove}_a(B)$ (which cannot simulate states from $\delta_a^{-1}(B)$ as they have empty intersection with $\delta_a^{-1}(\mathit{Rel}(B))$), and $B' \setminus \mathit{Remove}_a(B)$. More specifically, for a set $\mathit{Remove} \subseteq S$, $\mathit{Split}(P, \mathit{Remove})$ returns a finer partition $P' = \{B \setminus \mathit{Remove} \mid B \in P\} \cup \{B \cap \mathit{Remove} \mid B \in P\}$. After refining P by the *Split* operation, the newly created blocks of P inherit the data structures (counters *Count* and *Remove* sets) from their "parents" (for a block $B \in P$, its parent is the block $B_{\mathsf{prev}} \in P_{\mathsf{prev}}$ such that $B \subseteq B_{\mathsf{prev}}$). *Rel* is then updated on line 17 by removing the pairs (C, D) such that $C \cap \delta_a^{-1}(B) \neq \emptyset$ and $D \subseteq \mathit{Remove}_a(B)$. The change of *Rel* causes that for some states $u \in S$ and symbols $b \in \Sigma$, $\delta_a(u) \cap \bigcup \mathit{Rel}(C)$ becomes empty. To propagate the change of the relation along the transition relation, u will be moved into $\mathit{Remove}_b(C)$ on line 20, which will cause new changes of the relation in the following iterations of the while-loop. If there is no nonempty *Remove* set, then $\langle P, \mathit{Rel}\rangle$ is the coarsest partition-relation pair inducing Sim_I and the algorithm terminates. Correctness of LRT is stated by Theorem 1.

Theorem 1 ([1])**.** With an LTS $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$ and the coarsest partition-relation pair $\langle P_I, \mathit{Rel}_I\rangle$ inducing a preorder $I \subseteq S \times S$ on the input, LRT terminates with the coarsest partition-relation pair $\langle P, \mathit{Rel}\rangle$ inducing Sim_I.

4 OPTIMIZATIONS OF LRT

The optimization we are now going to propose reduces the number of counters and the number and the size of *Remove* sets. The changes required by OLRT are indicated in Algorithm 1 on the right hand sides of the concerned lines.

We will need the following notation. For a state $v \in S$, $\mathrm{in}(v) = \{a \in \Sigma \mid \delta_a^{-1}(v) \neq \emptyset\}$ is the set of *input symbols* and $\mathrm{out}(v) = \{a \in \Sigma \mid \delta_a(v) \neq \emptyset\}$ is the set of *output symbols* of v. The *output preorder* is the relation $\mathit{Out} = \bigcap_{a \in \Sigma} \delta_a^{-1}(S) \times \delta_a^{-1}(S)$ (this is, $(u, v) \in \mathit{Out}$ if and only if $\mathrm{out}(u) \subseteq \mathrm{out}(v)$).

To make our optimization possible, we have to initialize $\langle P, \mathit{Rel}\rangle$ by the finer partition-relation pair $\langle P_{I \cap \mathit{Out}}, \mathit{Rel}_{I \cap \mathit{Out}}\rangle$ (instead of $\langle P_I, \mathit{Rel}_I\rangle$), which is the coarsest partition-relation pair inducing the relation $I \cap \mathit{Out}$. As both I and *Out* are preorders, $I \cap \mathit{Out}$ is a preorder too. As $\mathit{Sim}_I \subseteq I$ and $\mathit{Sim}_I \subseteq \mathit{Out}$ (any simulation on T is a subset of *Out*), Sim_I equals the maximal simulation included in $I \cap \mathit{Out}$. Thus, this step itself does not influence the output of the algorithm.

Assuming that $\langle P, \mathit{Rel}\rangle$ is initialized to $\langle P_{I \cap \mathit{Out}}, \mathit{Rel}_{I \cap \mathit{Out}}\rangle$, we can observe that for any $B \in P$ and $a \in \Sigma$ chosen on line 5, the following two claims hold:

Claim 1. If $a \notin \mathrm{in}(B)$, then skipping this iteration of the while-loop does not affect the output of the algorithm.

Proof. In an iteration of the while-loop processing $\mathit{Remove}_a(B)$ with $a \notin \mathrm{in}(B)$, as there is no $C \in P$ with $\delta_a(C) \cap \mathit{Rel}(B) \neq \emptyset$, the for-loop on line 16 stops immediately. No pair (C, D) will be removed from *Rel* on line 17, no counter will be decremented, and no state will be added into a *Remove* set. The only thing that can happen is that $\mathit{Split}(P, \mathit{Remove})$ refines P. However, in this case, this refinement of P would be

done anyway in other iterations of the while-loop when processing sets $Remove_b(C)$ with $b \in \text{in}(C)$. To see this, note that correctness of the algorithm does not depend on the order in which nonempty *Remove* sets are processed. Therefore, we can postpone processing all the nonempty $Remove_a(B)$ sets with $a \notin \text{in}(B)$ to the end of the computation. Recall that processing no of these *Remove* sets can cause that an empty *Remove* set becomes nonempty. Thus, the algorithm terminates after processing the last of the postponed $Remove_a(B)$ sets. If processing some of these $Remove_a(B)$ with $a \notin \text{in}(B)$ refines P, P will contain blocks C, D such that both (C, D) and (D, C) are in *Rel* (recall that when processing $Remove_a(B)$, no pair of blocks can be removed from *Rel* on line 17). This means that the final $\langle P, Rel\rangle$ will not be coarsest, which is a contradiction with Theorem 1. Thus, processing the postponed $Remove_a(B)$ sets can influence nor *Rel* neither P, and therefore they do not have to be processed at all. □

Claim 2. It does not matter whether we assign $Remove_a(B)$ or $Remove_a(B) \setminus (S \setminus \delta_a^{-1}(S))$ to *Remove* on line 6.

Proof. Observe that v with $a \notin \text{out}(v)$ (i.e., $v \in S \setminus \delta_a^{-1}(S)$) cannot be added into $Remove_a(B)$ on line 20, as this would mean that v has an a-transition leading to D. Therefore, v can get into $Remove_a(B)$ only during initialization on line 4 together with all states from $S \setminus \delta_a^{-1}(S)$. After $Remove_a(B)$ is processed (and emptied) for the first time, no state from $S \setminus \delta_a^{-1}(S)$ can appear there again. Thus, $Remove_a(B)$ contains states from $S \setminus \delta_a^{-1}(S)$ only when it is processed for the first time and then it contains all of them. It can be shown that for any partition Q of a set X and any $Y \subseteq X$, if $Split(Q, Y) = Q$, then also for any $Z \subseteq X$ with $Y \subseteq Z$, $Split(Q, Z) = Split(Q, Z \setminus Y)$. As P refines $P_{I \cap Out}$, $Split(P, S \setminus \delta_a^{-1}(S)) = P$. Therefore, as $S \setminus \delta_a^{-1}(S) \subseteq Remove_a(B)$, $Split(P, Remove_a(B)) = Split(P, Remove_a(B) \setminus (S \setminus \delta_a^{-1}(S)))$. We have shown that removing $S \setminus \delta_a^{-1}(S)$ from *Remove* does not influence the result of the *Split* operation in this iteration of the while-loop (note that this implies that all blocks from the new partition are included in or have empty intersection with $S \setminus \delta_a^{-1}(S)$). It remains to show that it also does not influence updating *Rel* on line 17. Removing $S \setminus \delta_a^{-1}(S)$ from *Remove* could only cause that the blocks D such that $D \subseteq S \setminus \delta_a^{-1}(S)$ that were chosen on line 15 with the original value of *Remove* will not be chosen with the restricted *Remove*. Thus, some of the pairs (C, D) removed from *Rel* with the original version of *Remove* could stay in *Rel* with the restricted version of *Remove*. However, such a pair (C, D) cannot exist because with the original value of *Remove*, if (C, D) is removed from *Rel*, then $a \in \text{out}(C)$ (as $\delta(C) \cap B \neq \emptyset$) and therefore also $a \in \text{out}(D)$ (as *Rel* was initialized to $Rel_{I \cap Out}$ on line 1 and $(C, D) \in Rel$). Thus, $D \cap (S \setminus \delta_a^{-1}(S)) = \emptyset$, which means that (C, D) is removed from *Rel* even with the restricted *Remove*. Therefore, it does not matter whether $S \setminus \delta_a^{-1}(S)$ is a subset of or it has an empty intersection with *Remove*. □

As justified above, we can optimize LRT as follows. Sets $Remove_a(B)$ are computed only if $a \in \text{in}(B)$ and in that case we only add states $q \in \delta_a^{-1}(S)$ to

$Remove_a(B)$. As a result, we can reduce the number of required counters by maintaining $Count_a(v, B)$ if and only if $a \in \text{in}(B)$ and $a \in \text{out}(v)$.

5 IMPLEMENTATION AND COMPLEXITY OF OLRT

We first briefly describe the essential data structures (there are some additional data structures required by our optimizations) and then we sketch the complexity analysis. For the full details, see the technical report [9].

Data Structures. The input LTS is represented as a list of records about its states. The record about each state $v \in S$ contains a list of nonempty $\delta_a^{-1}(v)$ sets[1], each of them encoded as a list of its members. The partition P is encoded as a doubly-linked list (DLL) of blocks. Each block is represented as a DLL of (pointers to) states of the block. Each block B contains for each $a \in \Sigma$ a list of (pointers on) states from $Remove_a(B)$. Each time when any set $Remove_a(B)$ becomes nonempty, block B is moved to the beginning of the list of blocks. Choosing the block B on line 5 then means just scanning the head of the list of blocks. The relation Rel is encoded as a resizable Boolean matrix.

Each block $B \in P$ and each state $v \in S$ contains an Σ-indexed array containing a record $B.a$ and $v.a$, respectively. The record $B.a$ stores the information whether $a \in \text{in}(B)$ (we need the test on $a \in \text{in}(B)$ to take a constant time). If $a \in \text{in}(B)$, then $B.a$ also contains a reference to the set $Remove_a(B)$, represented as a list of states (with a constant time addition), and a reference to an array of counters $B.a.Count$ containing the counter $Count_a(v, B)$ for each $v \in \delta_a^{-1}(S)$. Note that for two different symbols $a, b \in \Sigma$ and some $v \in S$, the counter $Count_a(v, B)$ has different index in the array $B.a.Count$ than the counter $Count_b(v, B)$ in $B.b.Count$ (as the sets $\delta_a^{-1}(S)$ and $\delta_b^{-1}(S)$ are different). Therefore, for each $v \in S$ and $a \in \Sigma$, $v.a$ contains an index v_a under which for each $B \in P$, the counter $Count_a(v, B)$ can be found in the array $B.a.Count$. Using the Σ-indexed arrays attached to symbols and blocks, every counter can be found/updated in a constant time. For every $v \in S, a \in \Sigma$, $v.a$ also stores a pointer to the list containing $\delta_a^{-1}(v)$ or $null$ if $\delta_a^{-1}(v)$ is empty. This allows the constant time testing whether $a \in \text{in}(v)$ and the constant time searching for the $\delta_a^{-1}(v)$ list.

Complexity analysis (Sketch). We first point out how our optimizations influence complexity of the most costly part of the code which is the main while loop. The analysis of lines 14–16 of LRT is based on the observation that for any two $B', D' \in P_{Sim_I}$ and any $a \in \Sigma$, it can happen at most once that a and some B with $B' \subseteq B$ are chosen on line 14 and at the same time $D' \subseteq Remove_a(B)$. In one single iteration of the while-loop, blocks C are listed by traversing all $\delta^{-1}(v), v \in B$ (the

[1] We use a list rather than an array having an entry for each $a \in \Sigma$ in order to avoid a need to iterate over alphabet symbols for which there is no transition.

Ds can be enumerated during the *Split* operation). Within the whole computation, for any $B' \in P_{Sim_I}$, transitions leading to B' are traversed on line 14 at most P_{Sim_I} times, so the complexity of lines 14–16 of LRT is $\mathcal{O}(\sum_{a\in\Sigma}\sum_{D\in P_{Sim}}\sum_{v\in S}|\delta_a^{-1}(v)|) = \mathcal{O}(|P_{Sim_I}||\delta|)$. In the case of OLRT, the number and the content of remove sets is restricted in such a way that for a nonempty set $Remove_a(B)$, it holds that $a \in \mathrm{in}(B)$ and $Remove_a(B) \subseteq \delta_a^{-1}(S)$. Hence, for a fixed a, a-transition leading to a block $B' \in P_{Sim_I}$ can be traversed only $|\{D' \in P_{Sim_I} \mid a \in \mathrm{out}(D')\}|$ times and the complexity of lines 14–16 decreases to $\mathcal{O}(\sum_{D\in P_{Sim_I}}\sum_{a\in\mathrm{in}(D)}|\delta_a|)$.

The analysis of lines 17–20 of LRT is based on the fact that once (C, D) appears on line 17, no (C', D') with $C' \subseteq C, D' \subseteq D$ can appear there again. For a fixed (C, D), the time spent on lines 17–20 is in $\mathcal{O}(\sum_{v\in B}|\delta^{-1}(v)|)$ and only those blocks C, D can meet on line 17 such that $C \times D \subseteq I$. Thus, the overall time spent by LRT on lines 17–20 is in $\mathcal{O}(\sum_{B\in P_{Sim_I}}\sum_{v\in I(B)}|\delta^{-1}(v)|)$. In OLRT, blocks C, D can meet on line 17 only if $C \times D \subseteq I \cap Out$, and the complexity of lines 17–20 in OLRT decreases to $\mathcal{O}(\sum_{B\in P_{Sim_I}}\sum_{v\in(I\cap Out)(B)}|\delta^{-1}(v)|)$.

In addition, OLRT refines $\langle P_I, Rel_I\rangle$ to $\langle P_{I\cap Out}, Rel_{I\cap Out}\rangle$ on line 1. This can be done by successive splitting according to the sets $\delta_a^{-1}(S), a \in \Sigma$ and after each split, breaking the relation between blocks included in $\delta_a^{-1}(S)$ and the ones outside. This procedure takes time $\mathcal{O}(|\Sigma||P_{I\cap Out}|^2)$.

Apart from some other smaller differences, the implementation and the complexity analysis of OLRT are analogous to the implementation and the analysis of LRT [1]. The overall time complexity of OLRT is $\mathcal{O}\big(|\Sigma||P_{I\cap Out}|^2+|\Sigma||S|+|P_{Sim_I}|^2+\sum_{B\in P_{Sim_I}}(\sum_{a\in\mathrm{in}(B)}(|\delta_a^{-1}(S)|+|\delta_a|)+\sum_{v\in(I\cap Out)(B)}|\delta^{-1}(v)|)\big)$.

The space complexity of OLRT is determined by the number of counters, the contents of the *Remove* sets, the size of the matrix encoding of *Rel*, and the space needed for storing the $B.a$ and $v.a$ records (for every block B, state v and symbol a). Overall, it gives $\mathcal{O}(|P_{Sim_I}|^2 + |\Sigma||S| + \sum_{B\in P_{Sim_I}}\sum_{a\in\mathrm{in}(B)}|\delta_a^{-1}(S)|)$.

Observe that the improvement of both time and space complexity of LRT is most significant for systems with large alphabets and a high diversity of sets of input and output symbols of states. Certain regular diversity of sets of input and output symbols is an inherent property of LTSs that arise when we compute simulations over tree automata. We address the impact of employing OLRT within the procedures for computing tree automata simulation in the next section.

6 TREE AUTOMATA SIMULATIONS

In [1], authors propose methods for computing tree automata simulations via translating problems of computing simulations over tree-automata to problems of computing simulations over certain LTSs. In this section, we show how replacing LRT by OLRT within these translation-based procedures decreases the overall complexity of computing tree-automata simulations.

A (finite, bottom-up) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked

alphabet with a ranking function $r : \Sigma \to \mathbb{N}$, and $\Delta \subseteq Q^* \times \Sigma \times Q$ is a set of transition rules such that if $(q_1 \dots q_n, f, q) \in \Delta$, then $r(f) = n$. Finally, we denote by r_{m} the smallest $n \in \mathbb{N}$ such that $n \geq m$ for each $m \in \mathbb{N}$ such that there is some $(q_1 \dots q_m, f, q) \in \Delta$. We omit the definition of the semantics of TA as we will not need it, and we only refer to [5, 1].

For the rest of this section, we fix a TA $A = (Q, \Sigma, \Delta, F)$. A *downward simulation* D is a binary relation on Q such that if $(q, r) \in D$, then for all $(q_1 \dots q_n, f, q) \in \Delta$, there exists $(r_1 \dots r_n, f, r) \in \Delta$ such that $(q_i, r_i) \in D$ for each $i : 1 \leq i \leq n$. Given a downward simulation D which is a preorder called an *inducing simulation*, an *upward simulation U induced by D* is a binary relation on Q such that if $(q, r) \in U$, then (i) for all $(q_1 \dots q_n, f, q') \in \Delta$ with $q_i = q, 1 \leq i \leq n$, there exists $(r_1 \dots r_n, f, r') \in \Delta$ with $r_i = r$, $(q', r') \in U$, and $(q_j, r_j) \in D$ for each $j : 1 \leq j \neq i \leq n$; (ii) $q \in F \implies r \in F$. From now on, let D denote the maximal downward simulation on A and U the maximal upward simulation on A induced by D.

To define the translations from downward and upward simulation problems, we need the following notions. Given a transition $t = (q_1 \dots q_n, f, q) \in \Delta$, $q_1 \dots q_n$ is its *left-hand side* and $t(i) \in (Q \cup \{\Box\})^* \times \Sigma \times Q$ is an *environment* – the tuple which arises from t by replacing state q_i, $1 \leq i \leq n$, at the i^{th} position of the left-hand side of t by the so called hole $\Box \notin Q$. We use *Lhs* of to denote the set of all left-hand sides of A and *Env* to denote the set of all environments of A.

We translate the downward simulation problem on A to the simulation problem on the LTS $A^\bullet = (Q^\bullet, \Sigma^\bullet, \{\delta^\bullet_a \mid a \in \Sigma^\bullet\})$ where $Q^\bullet = \{q^\bullet \mid q \in Q\} \cup \{l^\bullet \mid l \in \mathit{Lhs}\}$, $\Sigma^\bullet = \Sigma \cup \{1, \dots, r_{\mathsf{m}}\}\}$, and for each $(q_1 \dots q_n, f, q) \in \Delta$, $(q^\bullet, q_1 \dots q_n^\bullet) \in \delta^\bullet_f$ and $(q_1 \dots q_n^\bullet, q_i^\bullet) \in \delta^\bullet_i$ for each $i : 1 \leq i \leq n$. The initial relation is simply $I^\bullet = Q^\bullet \times Q^\bullet$. The upward simulation problem is then translated into a simulation problem on LTS $A^\odot = (Q^\odot, \Sigma^\odot, \{\delta^\odot_a \mid a \in \Sigma^\odot\})$, where $Q^\odot = \{q^\odot \mid q \in Q\} \cup \{e^\odot \mid e \in \mathit{Env}\}$, $\Sigma^\odot = \Sigma^\bullet$, and for each $t = (q_1 \dots q_n, f, q) \in \Delta$, for each $1 \leq i \leq n$, $(q_i^\odot, t(i)^\odot) \in \delta^\odot_i$ and $(t(i)^\odot, q^\odot) \in \delta^\odot_a$. The initial relation $I^\odot \subseteq Q^\odot \times Q^\odot$ contains all the pairs $(q^\odot, r^\odot)$ such that $q, r \in Q$ and $r \in F \implies q \in F$, and $((q_1 \dots q_n, f, q)(i)^\odot, (r_1 \dots r_n, f, r)(i)^\odot)$ such that $(q_j, r_j) \in D$ for all $j : 1 \leq i \neq j \leq n$. Let $\mathit{Sim}^\bullet$ be the maximal simulation on $A^\bullet$ included in $I^\bullet$ and let $\mathit{Sim}^\odot$ be the maximal simulation on $A^\odot$ included in $I^\odot$. The following theorem shows correctness of the translations.

Theorem 2 ([1])**.** For all $q, r \in Q$, we have $(q^\bullet, r^\bullet) \in \mathit{Sim}^\bullet$ if and only if $(q, r) \in D$ and $(q^\odot, r^\odot) \in \mathit{Sim}^\odot$ if and only if $(q, r) \in U$.

The states of the LTSs ($A^\bullet$ as well as $A^\odot$) can be classified into several classes according to the sets of input/output symbols. Particularly, $Q^\bullet$ can be classified into the classes $\{q^\bullet \mid q \in Q\}$ and for each $n : 1 \leq n \leq r_{\mathsf{m}}$, $\{q_1 \dots q_n^\bullet \mid q_1 \dots q_n \in \mathit{Lhs}\}$, and $Q^\odot$ can be classified into $\{q^\odot \mid q \in Q\}$ and for each $a \in \Sigma$ and $i : 1 \leq i \leq r(a)$, $\{t(i)^\odot \mid t = (q_1 \dots q_n, a, q) \in \Delta\}$. This turns to a significant advantage when computing simulations on $A^\bullet$ or on $A^\odot$ using OLRT instead of LRT. Moreover, we now propose another small optimization, which is a specialized procedure for

computing $\langle P_{I\cap Out} Rel_{I\cap Out}\rangle$ for the both of $A^{\odot}$, $A^{\bullet}$. It is based on the simple observation that we need only a constant time (not a time proportional to the size of the alphabet) to determine whether two left-hand sides or two environments are related by the particular *Out* (more specifically, $(e_1^{\odot}, e_2^{\odot}) \in Out$ if and only if the inner symbols of e_1 and e_2 are the same, and $(q_1 \dots q_n^{\bullet}, r_1 \dots r_m^{\bullet}) \in Out$ if and only if $n \leq m$).

Complexity of the optimized algorithm. We only point out the main differences between application of LRT [1] and OLRT on the LTSs that arise from the translations described above. For implementation details and full complexity analysis of the OLRT versions, see the technical report [9].

To be able to express the complexity of running OLRT on $A^{\bullet}$ and $A^{\odot}$, we extend D to the set *Lhs* such that $((q_1 \dots q_n), (r_1 \dots r_n)) \in D$ if and only if $(q_i, r_i) \in D$ for each $i : 1 \leq i \leq n$, and we extend U to the set *Env* such that $((q_1 \dots q_n, f, q)(i), (r_1 \dots r_n, f, r)(i)) \in U \iff m = n \wedge i = j \wedge (q, r) \in U \wedge (\forall k \in \{1, ..., n\}.\ k \neq i \Longrightarrow (q_k, r_k) \in D)$. For a preorder ρ over a set X, we use X/ρ to denote the partition of X according to the equivalence $\rho \cap \rho^{-1}$.

The procedures for computing $Sim^{\bullet}$ and $Sim^{\odot}$ consist of (i) translating A to the particular LTS ($A^{\bullet}$ or $A^{\odot}$) and computing the partition-relation pair inducing the initial preorder ($I^{\bullet}$ or $I^{\odot}$), and (ii) running a simulation algorithm (LRT or OLRT) on it. Here, we analyze the impact of replacing LRT by OLRT on the complexity of step (ii), which is the step with dominating complexity (as shown in [1] and also by our experiments; step (ii) is much more computationally demanding than step (i)).

As shown in the technical report [9], OLRT takes on $A^{\bullet}$ and $I^{\bullet}$ space $\mathcal{O}(Space_D)$ where $Space_D = (r_{\mathsf{m}} + |\Sigma|)|Lhs \cup Q| + |Lhs \cup Q/D|^2 + |\Sigma||Lhs/D||Q| + r_{\mathsf{m}}|Q/D||Lhs|$ and time $\mathcal{O}(Space_D + |\Sigma||Q/D|^2 + |Lhs/D||\Delta|)$. On $A^{\odot}$ and $I^{\odot}$, OLRT runs in time $O(Space_U)$ where $Space_U = (r_{\mathsf{m}} + |\Sigma|)|Env| + |Env/U|^2 + |Env/U||Q| + |Q/U||Env|$ and time $\mathcal{O}(Space_U + |\Sigma||Q/U|^2 + |Env/U||\delta|)$.

We compare the above results with [1], where LRT is used. LRT on $A^{\bullet}$ and $I^{\bullet}$ takes $\mathcal{O}(Space_D^{old})$ space where $Space_D^{old} = (|\Sigma| + r_{\mathsf{m}})|Q \cup Lhs||Q \cup Lhs/D|$, and $\mathcal{O}(Space_D^{old} + |\Delta||Q \cup Lhs/D|)$ time. In the case of $A^{\odot}$ and $I^{\odot}$, we obtain space complexity $\mathcal{O}(Space_U^{old})$ where $Space_U^{old} = |\Sigma||Env||Env/U|$ and time complexity $\mathcal{O}(Space_U^{old} + r_{\mathsf{m}}|\Delta||Env/U|)$.

The biggest difference is in the space complexity (decreasing the factors $Space_D^{old}$ and $Space_U^{old}$). However, the time complexity is better too, and our experiments show a significant improvement in space as well as in time.

7 EXPERIMENTS

We implemented the original and the improved version of the algorithm in a uniform way in OCaml and experimentally compared their performance.

The simulation algorithms were benchmarked using LTSs obtained from the runs of the abstract regular model checking (ARMC) (see [3, 4]) on several classic

	LTS			LRT		OLRT	
source	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	256	16	416	0.12	9.6	0.02	1.9
random	4 096	16	3 280	13.82	714.2	2.02	78.2
random	16 384	16	26 208	o.o.m.		268.85	4 514.9
random	4 096	32	6 560	62.09	1 844.2	4.36	121.4
random	4 096	64	13 120	158.38	3 763.2	6.59	211.2
pc	1 251	43	49 076	7.52	418.1	2.63	119.0
rw	4 694	11	20 452	81.28	3 471.8	19.25	989.3
lr	6 160	35	90 808	390.91	12 640.8	45.69	1 533.6

Table 1. LTS simulation results

examples – producer-consumer (pc), readers-writers (rw), and list reversal (lr) – and using a set of tree automata obtained from the run of the abstract regular tree model checking (ARTMC) (see [2]) on several operations, such as list reversal, red-black tree balancing, etc. We also used several randomly generated LTSs and tree automata.

	TA				LTS			LRT		OLRT	
source	$\|Q\|$	$\|\Sigma\|$	r_{m}	$\|\Delta\|$	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	16	16	2	245	184	18	570	0.06	6.2	0.02	1.4
random	32	16	2	935	655	18	2 165	0.87	74.4	0.21	14.4
random	64	16	2	3 725	2 502	18	8 568	26.63	1 417.9	3.50	195.4
random	32	32	2	1 164	719	34	2 511	2.67	166.6	0.23	16.8
random	32	64	2	2 026	925	66	3 780	12.17	623.5	0.56	25.4
ARTMC[2]	47	132	2	837	241	134	1 223	0.84	70.6	0.05	6.2
ARTMC	variable[3]							517.98	116.2	80.84	22.1

Table 2. Downward simulation results

We performed the experiments on AMD Opteron 8389 2.90 GHz PC with 128 GiB of memory (however we set the memory limit to approximately 20 GiB for each process). The system was running Linux and OCaml 3.10.2.

The performance of the algorithms is compared in Table 1 (general LTSs), Table 2 (LTSs generated while computing the downward simulation), and Table 3 (LTSs generated while computing the upward simulation), which contain the running times ([s]) and the amount of memory ([MiB]) required to finish the computation.

As seen from the results of our experiments, our optimized implementation performs substantially better than the original. On average, it improves the running time and space requirements by about one order of magnitude. As expected, we can see the biggest improvements especially in the cases where we tested the impact of the growing size of the alphabet.

[2] One of the automata selected from the ARTMC set.

[3] A set containing 10 305 tree automata of variable size (up to 50 states and up to 1 000 transitions per automaton). The results show the total amount of time required for the computation and the peak size of allocated memory.

	TA				LTS			LRT		OLRT	
source	$\|Q\|$	$\|\Sigma\|$	r_{m}	$\|\Delta\|$	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	16	16	2	245	472	17	952	1.03	96.5	0.09	4.8
random	32	16	2	935	1 791	17	3 700	18.73	1 253.8	1.37	54.7
random	64	16	2	3 725	7 126	17	14 824	405.89	14 173.9	22.83	752.6
random	32	32	2	1 164	2 204	33	4 548	64.10	3 786.7	2.36	193.4
random	32	64	2	2 026	3 787	65	7 874	o.o.m.		6.72	245.8
ARTMC[2]	47	132	2	837	1 095	133	3 344	66.46	4 183.2	0.69	68.2
ARTMC	variable[3]							12 669.94	4 412.6	400.62	106.6

Table 3. Upward simulation results

8 CONCLUSION

We proposed an optimized algorithm for computing simulations over LTSs, which improves the asymptotic complexity in both space and time of the best algorithm (LRT) known to date (see [1]) and which also performs significantly better in practice. We also show how employing OLRT instead of LRT reduces the complexity of the procedures for computing tree-automata simulations from [1]. As our future work, we want to develop further optimizations, which would allow to handle even bigger LTSs and tree automata. One of the possibilities is to replace existing data structures by a symbolic representation, for example, by using BDDs.

Acknowledgement

This work was supported in part by the Czech Science Foundation (projects P103/10/0306, 102/09/H042, 201/09/P531), the BUT FIT grant FIT-10-1, the Czech COST project OC10009 associated with the ESF COST action IC0901, the Czech Ministry of Education by the project MSM 0021630528, and the ESF project Games for Design and Verification.

REFERENCES

[1] Abdulla, P. A.—Bouajjani, A.—Holík, L.—Kaati, L.—Vojnar, T.: Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In Proc. of TACAS '08, LNCS 4963, Springer 2008.

[2] Bouajjani, A.—Habermehl, P.—Holík, L.—Touili, T.—Vojnar, T.: Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In Proc. of CIAA '08, LNCS 5148, Springer 2008.

[3] Bouajjani, A.—Habermehl, P.—Moro, P.—Vojnar, T.: Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In Proc. of TACAS '05, LNCS 3440, Springer 2005.

[4] Bouajjani, A.—Habermehl, P.—Vojnar, T.: Abstract Regular Model Checking. In Proc. of CAV '04, LNCS 3114, Springer 2004.

[5] Comon, H.—Dauchet, M.—Gilleron, R.—Löding, C. L.—Jacquemard, F.—Lugiez, D.—Tison,S.—Tommasi, M.: Tree Automata Techniques and Applications. http://www.grappa.univ-lille3.fr/tata, 2007, release October 12, 2007.

[6] Dams, D.—Grumberg, O.—Gerth, R.: Generation of Reduced Models for Checking Fragments of CTL. In Proc. of CAV '93, 1993.

[7] Grumberg, O.—Long, D. E.: Model Checking and Modular Verifiation. ACM Transactions on Programming Languages and Systems, Vol. 16, 1994.

[8] Henzinger, M. R.—Henzinger, T. A.—Kopke, P. W.: Computing Simulations on Finite and Infinite Graphs. In Proc. of FOCS '95, IEEE Computer Society 1995.

[9] Holík , L.—Šimáček, J.: Optimizing an LTS-Simulation Algorithm. Technical Report FIT-TR-2009-03, Brno University of Technology 2009, available on http://www.fit.vutbr.cz/~holik/pub/FIT-TR-2009-03.pdf.

[10] Piage, R.—Tarjan, R.: Three Partition Refinement Algorithms. SIAM Journal on Computing, Vol. 16, 1987.

[11] Ranzato, F.—Tapparo, F.: A New Efficient Simulation Equivalence Algorithm. In Proc. of LICS '07, 2007.

[12] Sawa, Z.—Jančar, P.: Behavioural Equivalences on Finite-State Systems are PTIMEhard. Computing and Informatics, Vol. 24, 2005.

Lukáš Holík is a doctoral student and a member of VeriFit research group at Faculty of Information Technology, Brno University of Technology. Home page: http://www.fit.vutbr.cz/~holik.

Jiří Šimáček is a doctoral student of Faculty of Information Technology at Brno University of Technology and a doctoral student of Université Joseph Fourier in Grenoble. He is a member of VeriFit research group and a member of VERIMAG.