

An Efficient Simulation Algorithm on Kripke Structures

Francesco Ranzato

Dipartimento di Matematica, University of Padova, Italy

Abstract A number of algorithms for computing the simulation preorder (and equivalence) on Kripke structures are available. Let Σ denote the state space, \rightarrow the transition relation and P_{sim} the partition of Σ induced by simulation equivalence. While some algorithms are designed to reach the best space bounds, whose dominating additive term is $|P_{\text{sim}}|^2$, other algorithms are devised to attain the best time complexity $O(|P_{\text{sim}}||\rightarrow|)$. We present a novel simulation algorithm which is both space and time efficient: it runs in $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}| + |\Sigma| \log |\Sigma|)$ space and $O(|P_{\text{sim}}||\rightarrow| \log |\Sigma|)$ time. Our simulation algorithm thus reaches the best space bounds while closely approaching the best time complexity.

1 Introduction

The simulation preorder is a fundamental behavioral relation widely used in process algebra for establishing system correctness and in model checking as a suitable abstraction for reducing the size of state spaces. In process algebra, specification and implementation of concurrent systems are both given as processes of a common algebra and the simulation preorder in this algebra is then used as an appropriate relation to determine whether the implementation conforms to the specification [10, 19]. On the other hand, a well-known remedy to the state explosion problem in model checking [7] consists in quotienting the state space of the model w.r.t. some behavioral relation and the simulation preorder plays here an important role since it can provide a significantly better state space reduction than bisimilarity while retaining the ability of strongly preserving expressive branching-time temporal languages such as ACTL* [2, 8]. Moreover, simulation relations are also used in database systems where graphs representing some database can be (also locally) minimized w.r.t. simulation equivalence (see e.g. [13, 26]).

The problem of efficiently computing the simulation preorder (and consequently simulation equivalence, i.e. its symmetric reduction) on finite systems has been thoroughly investigated and generated a number of simulation algorithms [3, 4, 5, 6, 9, 12, 17, 18, 20, 22, 25, 31, 32]. Both time and space complexities play a significant role in simulation algorithms, since in several applications, especially in model checking, memory requirements may become a serious bottleneck as the input transition system grows.

Algorithm	Space complexity	Time complexity
Space-Céc [6]	$O(P_{\text{sim}} ^2 + \rightarrow \log \rightarrow)$	$O(P_{\text{sim}} ^2 \rightarrow)$
Time-Céc [6]	$O(P_{\text{sim}} \Sigma \log \Sigma + \rightarrow \log \rightarrow)$	$O(P_{\text{sim}} \rightarrow)$
GPP [17]	$O(P_{\text{sim}} ^2 \log P_{\text{sim}} + \Sigma \log \Sigma)$	$O(P_{\text{sim}} ^2 \rightarrow)$
Mar [25]	$O((\Sigma + P_{\text{sim}} ^2) \log P_{\text{sim}})$	$O(\rightarrow + P_{\text{sim}} \Sigma + P_{\text{sim}} ^3)$
RT [31]	$O(P_{\text{sim}} \Sigma \log \Sigma)$	$O(P_{\text{sim}} \rightarrow)$
ESim (this paper)	$O(P_{\text{sim}} ^2 \log P_{\text{sim}} + \Sigma \log \Sigma)$	$O(P_{\text{sim}} \rightarrow \log \Sigma)$

Table 1 Complexities of simulation algorithms.

State of the Art. Consider a finite transition system where Σ denotes the state space, \rightarrow the transition relation and P_{sim} the partition of Σ induced by simulation equivalence. The best simulation algorithms are those by, in chronological order, Gentilini, Piazza and Policriti (GPP) [16, 17] (subsequently corrected in [20]), Ranzato and Tapparo (RT) [29, 31], Markovski (Mar) [25], Cécé (Space-Céc and Time-Céc) [6]. The simulation algorithms GPP and RT are designed for Kripke structures, i.e. the standard models used in model checking, while Space-Céc, Time-Céc and Mar are for more general labeled transition systems, i.e. the system models used in process algebra. Space and time complexities of these algorithms are summarized in Table 1. We remark that all the space bounds in Table 1 are bit space complexities, i.e., the word size is a single bit. Thus, a pointer to some state in Σ is stored in $\log |\Sigma|$ space, while a pointer to some partition block in P_{sim} takes $\log |P_{\text{sim}}|$ space. Let us also remark that all the articles [16, 17, 20] state that the bit space complexity of GPP is in $O(|P_{\text{sim}}|^2 + |\Sigma| \log |P_{\text{sim}}|)$. However, as observed also in [6], this is not precise. In fact, the algorithm GPP (as described in [17, Section 4, p. 98]) assumes that the states belonging to some block are stored as a doubly linked list, and this therefore entails a bit space complexity in $O(|\Sigma| \log |\Sigma|)$. Furthermore, GPP uses Henzinger, Henzinger and Kopke [22] simulation algorithm (HKK) as a subroutine, whose bit space complexity is in $O(|\Sigma|^2 \log |\Sigma|)$, which is called on a Kripke structure where states are blocks of the current partition. Consequently, the bit space complexity of GPP must include an additive term $|P_{\text{sim}}|^2 \log |P_{\text{sim}}|$ and therefore results to be $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}| + |\Sigma| \log |\Sigma|)$. It is worth observing that a space complexity in $O(|P_{\text{sim}}|^2 + |\Sigma| \log |P_{\text{sim}}|)$ can be considered optimal for a simulation algorithm, since this is of the same order as the size of the output, which needs $|P_{\text{sim}}|^2$ space for storing the simulation preorder as a partial order on simulation equivalence classes and $|\Sigma| \log |P_{\text{sim}}|$ space for storing the simulation equivalence class for any state. Hence, the bit space complexities of GPP and Space-Céc can be considered quasi-optimal. In general, the space complexities of GPP and Space-Céc are not comparable since $|\Sigma| \log |\Sigma| \leq |\rightarrow| \log |\rightarrow|$ while $|P_{\text{sim}}|^2 \leq |P_{\text{sim}}|^2 \log |P_{\text{sim}}|$. As far as time complexity is concerned, the simulation algorithms RT and Time-Céc both feature the best time bound $O(|P_{\text{sim}}| |\rightarrow|)$.

Contributions. We present here a novel space and time Efficient Simulation algorithm for Kripke structures, called ESim, which features a time complexity in $O(|P_{\text{sim}}| |\rightarrow| \log |\Sigma|)$ and a bit space complexity in $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}| + |\Sigma| \log |\Sigma|)$. Thus, ESim reaches the best space bound of GPP and significantly improves the GPP time bound $O(|P_{\text{sim}}|^2 |\rightarrow|)$ by replacing a multiplicative factor $|P_{\text{sim}}|$ with $\log |\Sigma|$. Furthermore, ESim significantly improves the RT space bound

$O(|P_{\text{sim}}||\Sigma| \log |\Sigma|)$ and closely approaches the best time bound $O(|P_{\text{sim}}||\rightarrow|)$ of RT and Time-Céc by adding a multiplicative logarithmic factor $\log |\Sigma|$.

ESim is a so-called partition refinement algorithm. This means that ESim maintains and iteratively refines a partition-relation pair $\langle P, \sqsubseteq \rangle$, where P is a partition of the state space Σ that overapproximates the final simulation partition P_{sim} , while \sqsubseteq is a binary relation over the partition P which overapproximates the final simulation preorder relation on P_{sim} . ESim relies on the following three main points, which in particular allow to attain the above space and time complexity bounds.

- (1) Two distinct notions of partition and relation stability for a partition-relation pair are introduced. Accordingly, at a logical level, ESim is designed as a partition refinement algorithm which iteratively performs two clearly distinct refinement steps: the refinement of the current partition P which splits some blocks of P and the refinement of the relation \sqsubseteq which removes some pairs of blocks of P from \sqsubseteq .
- (2) ESim exploits a logical characterization of partition refiners, i.e. blocks of P that allow to split the current partition P , which admits an efficient implementation.
- (3) ESim only relies on data structures, like lists and matrices, that are indexed on blocks of the current partition P . In particular, no explicit (i.e., indexed on single states in Σ) data structure is used. The hard task here is to devise efficient ways to keep updated these partition-based data structures along the iterations of ESim. We show that this can be done efficiently, in particular by resorting to Hopcroft’s “process the smaller half” principle [24] when updating a crucial data structure after a partition split: if a partition block B is split into two blocks $B_1, B_2 \subseteq B$ and $|B_1| \leq |B_2|$ then this critical update can be performed by scanning only the states belonging to the smaller block B_1 .

This is the full version of the conference paper [28] that contains all the proofs and the algorithmic procedures.

2 Background

Notation. If $R \subseteq \Sigma \times \Sigma$ is any relation on a set Σ and $X \subseteq \Sigma$ then $R(X) \triangleq \{x' \in \Sigma \mid \exists x \in X. (x, x') \in R\}$. Recall that R is a preorder relation when it is reflexive and transitive. If f is a function defined on $\wp(\Sigma)$ and $x \in \Sigma$ then we often write $f(x)$ to mean $f(\{x\})$. A partition P of a finite set Σ is a set of nonempty subsets of Σ , called blocks, that are pairwise disjoint and whose union gives Σ . $\text{Part}(\Sigma)$ denotes the set of partitions of a finite set Σ . If $P \in \text{Part}(\Sigma)$, $s \in \Sigma$ and $S \subseteq \Sigma$ then $P(s)$ denotes the block of P that contains s while $P(S) \triangleq \cup_{s \in S} P(s)$. $\text{Part}(\Sigma)$ is endowed with the standard partial order \preceq : $P_1 \preceq P_2$, i.e. P_2 is coarser than P_1 , iff for any $s \in \Sigma$, $P_1(s) \subseteq P_2(s)$. By $P_1 \prec P_2$ we denote the fact that $P_1 \preceq P_2$ and $P_1 \neq P_2$. If $P_1 \preceq P_2$ and $B \in P_1$ then $P_2(B)$ is a block of P_2 which is also denoted by $\text{parent}_{P_2}(B)$. For a given nonempty subset $S \subseteq \Sigma$ called splitter, we denote by $\text{Split}(P, S)$ the partition obtained from P by replacing each block $B \in P$ with the nonempty sets $B \cap S$ and $B \setminus S$, where we also allow no splitting, namely $\text{Split}(P, S) = P$ (this happens exactly when $P(S) = S$). If $\langle Q, \leq \rangle$ is a poset and $X \subseteq Q$ then $\text{max}(X) \triangleq \{x \in X \mid \forall y \in Q. x \leq y \Rightarrow x = y\}$ denotes the set of maximal elements of X in Q .

Simulation Preorder and Equivalence. A transition system (Σ, \rightarrow) consists of a set Σ of states and of a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. Given a set AP of atoms (of some specification

language), a Kripke structure (KS) $(\Sigma, \rightarrow, \ell)$ over AP consists of a transition system (Σ, \rightarrow) together with a state labeling function $\ell : \Sigma \rightarrow \wp(AP)$. The state partition induced by ℓ is denoted by $P_\ell \triangleq \{\{s' \in \Sigma \mid \ell(s) = \ell(s')\} \mid s \in \Sigma\}$. The predecessor/successor transformers $\text{pre}, \text{post} : \wp(\Sigma) \rightarrow \wp(\Sigma)$ are defined in a standard way: $\text{pre}(T) \triangleq \{s \in \Sigma \mid \exists t \in T. s \rightarrow t\}$ and $\text{post}(S) \triangleq \{t \in \Sigma \mid \exists s \in S. s \rightarrow t\}$. If $S_1, S_2 \subseteq \Sigma$ then $S_1 \rightarrow^{\exists} S_2$ iff there exist $s_1 \in S_1$ and $s_2 \in S_2$ such that $s_1 \rightarrow s_2$.

A relation $R \subseteq \Sigma \times \Sigma$ is a simulation on a Kripke structure $\mathcal{K} = (\Sigma, \rightarrow, \ell)$ if for any $s, s' \in \Sigma$, if $s' \in R(s)$ then:

- $\ell(s) = \ell(s')$;
- for any $t \in \Sigma$ such that $s \rightarrow t$, there exists $t' \in \Sigma$ such that $s' \rightarrow t'$ and $t' \in R(t)$.

Given two states $s, t \in \Sigma$, t simulates s , denoted by $s \leq t$, if there exists a simulation relation R on \mathcal{K} such that $t \in R(s)$. Also, s and t are simulation equivalent (or similar) when $s \leq t$ and $t \leq s$. It turns out that the largest simulation on a given KS exists, is a preorder relation called simulation preorder and is denoted by R_{sim} . Thus, for any $s, t \in \Sigma$, we have that $s \leq t$ iff $(s, t) \in R_{\text{sim}}$. The simulation partition $P_{\text{sim}} \in \text{Part}(\Sigma)$ is the symmetric reduction of R_{sim} , namely, for any $s, t \in \Sigma$, $P_{\text{sim}}(s) = P_{\text{sim}}(t)$ iff $s \leq t$ and $t \leq s$.

3 Logical Simulation Algorithm

3.1 Partition-Relation Pairs

A partition-relation pair $\mathcal{P} = \langle P, \trianglelefteq \rangle$, PR for short, is a state partition $P \in \text{Part}(\Sigma)$ together with a binary relation $\trianglelefteq \subseteq P \times P$ between blocks of P . We write $B \triangleleft C$ when $B \trianglelefteq C$ and $B \neq C$, and $(B', C') \triangleleft (B, C)$ when $B' \trianglelefteq B$ and $C' \trianglelefteq C$. When \trianglelefteq is a preorder/partial order relation then \mathcal{P} is called, respectively, a preorder/partial order PR.

PRs allow to represent symbolically, i.e. through state partitions, a relation between states. A relation $R \subseteq \Sigma \times \Sigma$ induces a partition-relation pair $\text{PR}(R) = \langle P, \trianglelefteq \rangle$ defined as follows:

- for any s , $P(s) \triangleq \{t \in \Sigma \mid R(s) = R(t)\}$;
- for any s, t , $P(s) \trianglelefteq P(t)$ iff $t \in R(s)$.

It is easy to note that if R is a preorder then $\text{PR}(R)$ is a partial order PR. On the other hand, a PR $\mathcal{P} = \langle P, \trianglelefteq \rangle$ induces the following relation $\text{Rel}(\mathcal{P}) \subseteq \Sigma \times \Sigma$ on states in Σ :

$$(s, t) \in \text{Rel}(\mathcal{P}) \Leftrightarrow P(s) \trianglelefteq P(t).$$

Here, if \mathcal{P} is a preorder PR then $\text{Rel}(\mathcal{P})$ is clearly a preorder.

A PR $\mathcal{P} = \langle P, \trianglelefteq \rangle$ is defined to be a simulation PR on a KS \mathcal{K} when $\text{Rel}(\mathcal{P})$ is a simulation on \mathcal{K} , namely when \mathcal{P} represents a simulation relation between states. Hence, if \mathcal{P} is a simulation PR and $P(s) = P(t)$ then s and t are simulation equivalent, while if $P(s) \trianglelefteq P(t)$ then t simulates s .

Given a PR $\mathcal{P} = \langle P, \trianglelefteq \rangle$, the map $\mu_{\mathcal{P}} : \wp(\Sigma) \rightarrow \wp(\Sigma)$ is defined as follows:

$$\text{for any } X \in \wp(\Sigma), \mu_{\mathcal{P}}(X) \triangleq \text{Rel}(\mathcal{P})(X) = \cup\{C \in P \mid \exists s \in X. P(s) \trianglelefteq C\}.$$

Therefore, if \mathcal{P} is a simulation PR then $\mu_{\mathcal{P}}(X)$ gives the set of all the states s' that simulate some state $s \in X$. Note that, for any $s \in \Sigma$, $\mu_{\mathcal{P}}(s) = \mu_{\mathcal{P}}(P(s)) = \cup\{C \in P \mid P(s) \trianglelefteq C\}$. For preorder PRs, this map $\mu_{\mathcal{P}}$ allows us to characterize the property of being a simulation PR as follows.

Theorem 3.1. Let $\mathcal{P} = \langle P, \sqsubseteq \rangle$ be a preorder PR. Then, \mathcal{P} is a simulation iff

- (i) if $B \sqsubseteq C$, $b \in B$ and $c \in C$ then $\ell(b) = \ell(c)$;
- (ii) if $B \rightarrow^{\exists} C$ and $B \sqsubseteq D$ then $D \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$;
- (iii) for any $C \in P$, $P = \text{Split}(P, \text{pre}(\mu_{\mathcal{P}}(C)))$.

Proof. (\Rightarrow) Condition (i) clearly holds. Assume that $B \rightarrow^{\exists} C$ and $B \sqsubseteq D$. Hence, there exist $b \in B$ and $c \in C$ such that $b \rightarrow c$. Consider any state $d \in D$. Since \mathcal{P} is a simulation and $P(b) \sqsubseteq P(d)$, there exist some state e such that $d \rightarrow e$ and $C = P(c) \sqsubseteq P(e)$. Hence, $D \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$. Finally, if $C \in P$ and $x \in \text{pre}(\mu_{\mathcal{P}}(C))$ then there exists some block $D \sqsupseteq C$ and state $d \in D$ such that $x \rightarrow d$. If $y \in P(x)$ then since $P(x) = P(y)$, by reflexivity of \sqsubseteq , we have that $P(x) \sqsubseteq P(y)$, so that, since \mathcal{P} is a simulation, there exists some state e such that $y \rightarrow e$ and $P(d) \sqsubseteq P(e)$. Since \sqsubseteq is transitive, we have that $C \sqsubseteq P(e)$. Hence, $y \in \text{pre}(\mu_{\mathcal{P}}(C))$. We have thus shown that $P(x) \subseteq \text{pre}(\mu_{\mathcal{P}}(C))$, so that $P = \text{Split}(P, \text{pre}(\mu_{\mathcal{P}}(C)))$.

(\Leftarrow) Let us show that $\text{Rel}(\mathcal{P})$ is a simulation, i.e., if $P(s) \sqsubseteq P(s')$ then: (a) $\ell(s) = \ell(s')$; (b) if $s \rightarrow t$ then there exists t' such that $s' \rightarrow t'$ and $P(t) \sqsubseteq P(t')$. Condition (a) holds by hypothesis (i). If $s \rightarrow t$ then $P(s) \rightarrow^{\exists} P(t)$ so that, by condition (ii), we have that $P(s') \rightarrow^{\exists} \mu_{\mathcal{P}}(P(t))$, namely there exists $s'' \in P(s')$ such that $s'' \in \text{pre}(\mu_{\mathcal{P}}(P(t)))$. By condition (iii), $P(s'') \subseteq \text{pre}(\mu_{\mathcal{P}}(P(t)))$, i.e., $s' \in \text{pre}(\mu_{\mathcal{P}}(P(t)))$. Hence, there exists t' such that $s' \rightarrow t'$ and $P(t) \sqsubseteq P(t')$. \square

3.2 Partition and Relation Refiners

By Theorem 3.1, assuming that condition (i) holds, there are two possible reasons for a PR $\mathcal{P} = \langle P, \sqsubseteq \rangle$ for not being a simulation:

- (A) There exist $B, C, D \in P$ such that $B \rightarrow^{\exists} C$, $B \sqsubseteq D$, but $D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)$; in this case we say that the block C is a *relation refiner* for \mathcal{P} since the relation \sqsubseteq can be refined by removing the pair (B, D) .
- (B) There exist $B, C \in P$ such that $B \cap \text{pre}(\mu_{\mathcal{P}}(C)) \neq \emptyset$ and $B \setminus \text{pre}(\mu_{\mathcal{P}}(C)) \neq \emptyset$; in this case we say that the block C is a *partition refiner* for \mathcal{P} since the partition P can be refined by splitting the block B .

We therefore define:

$$\begin{aligned} \text{RRefiner}(\mathcal{P}) &\triangleq \{C \in P \mid C \text{ is a relation refiner for } \mathcal{P}\}; \\ \text{PRefiner}(\mathcal{P}) &\triangleq \{C \in P \mid C \text{ is a partition refiner for } \mathcal{P}\}. \end{aligned}$$

Accordingly, \mathcal{P} is defined to be relation or partition *stable* when, respectively, $\text{RRefiner}(\mathcal{P}) = \emptyset$ or $\text{PRefiner}(\mathcal{P}) = \emptyset$. Then, Theorem 3.1 can be read as follows: \mathcal{P} is a simulation iff \mathcal{P} satisfies condition (i) and is both relation and partition stable.

If $C \in \text{PRefiner}(\mathcal{P})$ then P is first refined to $P' \triangleq \text{Split}(P, \text{pre}(\mu_{\mathcal{P}}(C)))$, i.e. P is split w.r.t. the splitter $S = \text{pre}(\mu_{\mathcal{P}}(C))$. Accordingly, the relation \sqsubseteq on P is transformed into the following relation \sqsubseteq' defined on P' :

$$\sqsubseteq' \triangleq \{(D, E) \in P' \times P' \mid \text{parent}_P(D) \sqsubseteq \text{parent}_P(E)\} \quad (1)$$

Hence, two blocks D and E of the refined partition P' are related by \sqsubseteq' if their parent blocks $\text{parent}_P(D)$ and $\text{parent}_P(E)$ in P were related by \sqsubseteq . Hence, if $\mathcal{P}' = \langle P', \sqsubseteq' \rangle$ then for all

```

1 ESim(PR  $\langle P, \sqsubseteq \rangle$ ) {
2   Initialize(); PStabilize(); bool PStable := RStabilize(); bool RStable := tt;
3   while  $\neg(\text{PStable} \ \& \ \text{RStable})$  do
4     if  $\neg \text{PStable}$  then {RStable := PStabilize(); PStable := tt;}
5     if  $\neg \text{RStable}$  then {PStable := RStabilize(); RStable := tt;}
6   }

7 bool PStabilize() {
8    $P_{\text{old}} := P$ ;
9   while  $\exists C \in \text{PRefiner}(\mathcal{P})$  do
10     $S := \text{pre}(\mu_{\mathcal{P}}(C))$ ;  $P := \text{Split}(S)$ ;
11    forall  $(D, E) \in P \times P$  do  $D \sqsubseteq E := \text{parent}_P(D) \sqsubseteq \text{parent}_P(E)$ ;
12  return  $(P = P_{\text{old}})$ ;
13 }

14 bool RStabilize() {
15   // Precondition: PStable = tt
16    $\sqsubseteq_{\text{old}} := \sqsubseteq$ ; Delete :=  $\emptyset$ ;
17   while  $\exists C \in \text{RRefiner}(\mathcal{P})$  do
18     Delete := Delete  $\cup \{(B, D) \in P \times P \mid B \sqsubseteq D, B \rightarrow^{\exists} C, D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)\}$ ;
19      $\sqsubseteq := \sqsubseteq \setminus \text{Delete}$ ;
20  return  $(\sqsubseteq = \sqsubseteq_{\text{old}})$ ;
21 }

```

Figure 1 Logical Simulation Algorithm.

$D \in P'$, we have that $\mu_{\mathcal{P}'}(D) = \mu_{\mathcal{P}}(\text{parent}_P(D))$. We will show that this refinement $\langle P', \sqsubseteq' \rangle$ of $\langle P, \sqsubseteq \rangle$ is correct, meaning that if $B \in P$ is split into two blocks $B \setminus S$ and $B \cap S$ of P' then any state in $B \setminus S$ is certainly not simulation equivalent to some state in $B \cap S$. Note that if $B \in P$ is split into $B \cap S$ and $B \setminus S$ then both $B \cap S \sqsubseteq' B \setminus S$ and $B \setminus S \sqsubseteq' B \cap S$ hold, and consequently \mathcal{P}' becomes relation unstable.

On the other hand, if \mathcal{P} is partition stable and $C \in \text{RRefiner}(\mathcal{P})$ then we will show that \sqsubseteq can be safely refined to the following relation \sqsubseteq' :

$$\begin{aligned}
\sqsubseteq' &\triangleq \sqsubseteq \setminus \{(B, D) \in P \times P \mid B \rightarrow^{\exists} C, B \sqsubseteq D, D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)\} \\
&= \{(B, D) \in P \times P \mid B \sqsubseteq D, (B \rightarrow^{\exists} C \Rightarrow D \rightarrow^{\exists} \mu_{\mathcal{P}}(C))\}
\end{aligned} \tag{2}$$

because if $(B, D) \in \sqsubseteq \setminus \sqsubseteq'$ then any state in D certainly does not simulate some state in B .

The above facts lead us to design a logical simulation algorithm ESim described in Figure 1. ESim maintains a PR $\mathcal{P} = \langle P, \sqsubseteq \rangle$, which initially is $\langle P_{\ell}, \text{id} \rangle$ and is iteratively refined as follows:

PStabilize(): If $\langle P, \sqsubseteq \rangle$ is not partition stable then the partition P is split for $\text{pre}(\mu_{\mathcal{P}}(C))$ as long as a partition refiner $C \in P$ for \mathcal{P} exists; when this split happens the relation \sqsubseteq is transformed into \sqsubseteq' as defined by (1); at the end of this loop (at lines 9-11 of *PStabilize*()), we obtain a PR $\mathcal{P}' = \langle P', \sqsubseteq' \rangle$ which is partition stable; if P has actually been refined, i.e. $P' \prec P$, then the current PR \mathcal{P}' becomes relation unstable.

RStabilize(): If $\langle P, \sqsubseteq \rangle$ is not relation stable then the relation \sqsubseteq is refined to \sqsubseteq' as described by the definition (2) as long as a relation refiner $C \in P$ for \mathcal{P} exists; hence, at the end of this

refinement loop (at lines 17-18 of $RStabilize()$), the PR $\langle P, \leq' \rangle$ becomes relation stable but possibly partition unstable.

Moreover, the following properties of the current PR for the algorithm ESim hold.

Lemma 3.2. *In any run of ESim, the following two conditions hold:*

- (i) *If $PStabilize()$ is called on a current PR $\langle P, \leq \rangle$ which is a partial order then at the exit we obtain a PR $\langle P', \leq' \rangle$ which is a preorder.*
- (ii) *If $RStabilize()$ is called on a current PR $\langle P, \leq \rangle$ which is a preorder then at the exit we obtain a PR $\langle P, \leq' \rangle$ which is a partial order.*

Proof. Let us first consider $PStabilize()$. Consider an input partial order PR $\mathcal{P} = \langle P, \leq \rangle$, a splitter S such that $P' = Split(P, S)$ and let \leq' be defined as in (1). Let us show that $\langle P', \leq' \rangle$ is a preorder PR.

(Reflexivity): If $B \in P'$ then, as \leq is reflexive, $P(B) \leq P(B)$ and thus $B \leq' B$.

(Transitivity): Assume that $B, C, D \in P'$ and $B \leq' C$ and $C \leq' D$. Then, $P(B) \leq P(C)$ and $P(C) \leq P(D)$, so that by transitivity of \leq , $P(B) \leq P(D)$. Hence, $B \leq' D$.

Let us then take into account $RStabilize()$, consider an input preorder PR $\mathcal{P} = \langle P, \leq \rangle$ and let $\langle P, \leq' \rangle$ be the output PR of $RStabilize()$.

(Reflexivity): If $B \in P$ then, by reflexivity of \leq , $B \leq B$. If $B \rightarrow^{\exists} C$, for some $C \in P$, then since $C \leq C$ by reflexivity of \leq , we have that $B \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$. Hence, $B \leq' B$.

(Transitivity): Assume that $B \leq' C$ and $C \leq' D$. Then, $B \leq C$ and $C \leq D$, so that by transitivity of \leq , $B \leq D$. If $B \rightarrow^{\exists} E$ then, since $B \leq' C$, $C \rightarrow^{\exists} \mu_{\mathcal{P}}(E)$. Hence, there exists $F \in P$ such that $E \leq F$ and $C \rightarrow^{\exists} F$. Since $C \leq' D$, we have that $D \rightarrow^{\exists} \mu_{\mathcal{P}}(F)$. Since \leq is transitive and $E \leq F$, $\mu_{\mathcal{P}}(F) \subseteq \mu_{\mathcal{P}}(E)$. Thus, we have shown that $B \rightarrow^{\exists} E$ implies $D \rightarrow^{\exists} \mu_{\mathcal{P}}(E)$, namely $B \leq' D$.

(Antisymmetry): We observe that after calling $PStabilize()$ on a partial order PR, antisymmetry can be lost because for any block B which is split into $B_1 = B \cap S$ and $B_2 = B \setminus S$, where $S = \text{pre}(\mu_{\mathcal{P}}(C))$, we have that $B_1 \leq B_2$ and $B_2 \leq B_1$. In this case, $RStabilize()$ removes the pair $(B \cap S, B \setminus S)$ from the relation \leq : in fact, while $B \cap S \subseteq \text{pre}(\mu_{\mathcal{P}}(C))$ and therefore $B \cap S \rightarrow^{\exists} E$, for some block $E \subseteq \mu_{\mathcal{P}}(C)$, we have that $(B \setminus S) \cap \text{pre}(\mu_{\mathcal{P}}(C)) = \emptyset$, so that, since $\mu_{\mathcal{P}}(E) \subseteq \mu_{\mathcal{P}}(C)$, $(B \setminus S) \cap \text{pre}(\mu_{\mathcal{P}}(E)) = \emptyset$, i.e., $B \setminus S \not\rightarrow^{\exists} \mu_{\mathcal{P}}(E)$, and therefore $B \cap S \not\leq' B \setminus S$. Hence, $RStabilize()$ outputs a relation \leq' which is antisymmetric. \square

The main loop of ESim terminates when the current PR $\langle P, \leq \rangle$ becomes both partition and relation stable. By the above Lemma 3.2, the output PR \mathcal{P} of ESim is a partial order, and hence a preorder, so that Theorem 3.1 can be applied to \mathcal{P} which then results to be a simulation PR. It turns out that this algorithm is correct, meaning that the output PR \mathcal{P} actually represents the simulation preorder.

Theorem 3.3 (Correctness). *Let Σ be finite. Then, ESim is correct, i.e., ESim terminates on any input and if $\langle P, \leq \rangle$ is the output PR of ESim on input $\langle P_{\ell}, \text{id} \rangle$ then for any $s, t \in \Sigma$, $s \leq t \Leftrightarrow P(s) \leq P(t)$.*

Proof. Let us first note that ESim always terminates. In fact, if $\langle P, \trianglelefteq \rangle$ is the current PR at the beginning of some iteration of the while-loop of ESim and $\langle P', \trianglelefteq' \rangle$ is the current PR at the beginning of the next iteration then, since $\langle P', \trianglelefteq' \rangle$ is either partition or relation unstable, we have that either $P' \triangleleft P$ or $P' = P$ and $\trianglelefteq' \subsetneq \trianglelefteq$. Since the state space Σ is finite, at some iteration it must happen that $P' = P$ and $\trianglelefteq' = \trianglelefteq$ so that PStable & RStable = tt.

When ESim terminates, we have that RRefiner($\langle P, \trianglelefteq \rangle$) = \emptyset = PRefiner($\langle P, \trianglelefteq \rangle$). Also, let us observe that condition (i) of Theorem 3.1 always holds for the current PR $\langle P, \trianglelefteq \rangle$ because the input PR $\langle P_\ell, \text{id} \rangle$ initially satisfies condition (i) and this condition is clearly preserved at any iteration of ESim. Furthermore, at the beginning, we have that $\langle P, \trianglelefteq \rangle = \langle P_\ell, \text{id} \rangle$ and this is trivially a partial order. Thus, we can apply Lemma 3.2 for any call to *PStabilize()* and *RStabilize()*, so that we obtain that the output PR $\langle P, \trianglelefteq \rangle$ is a preorder. Hence, Theorem 3.1 can be applied to the output preorder PR $\langle P, \trianglelefteq \rangle$, which is then a simulation. Thus, $\text{Rel}(\langle P, \trianglelefteq \rangle) \subseteq R_{\text{sim}}$.

Conversely, let us show that if \mathcal{P} is the output PR of ESim then $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P})$. This is shown as follows: if \mathcal{P} is a preorder PR such that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P})$ and *RStabilize()* or *PStabilize()* are called on \mathcal{P} then at the exit of one of these calls we obtain a PR \mathcal{P}' such that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P}')$.

Let us first take into account *RStabilize()*, consider an input preorder PR $\mathcal{P} = \langle P, \trianglelefteq \rangle$ such that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P})$, and let $\mathcal{P}' = \langle P, \trianglelefteq' \rangle$ be the output PR of *RStabilize()*. We show that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P}')$, that is, for any $s, t \in \Sigma$, if $s \leq t$ then $P(s) \trianglelefteq' P(t)$. By hypothesis, from $s \leq t$ we obtain $P(s) \trianglelefteq P(t)$. Assume that $P(s) \rightarrow^{\exists} C$, for some $C \in P$. Hence, $P(s) \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$. Since the PR \mathcal{P} is partition stable, we have that $P(s) \subseteq \text{pre}(\mu_{\mathcal{P}}(C))$. Thus, there exists some $D \in P$ and $d \in D$ such that $C \trianglelefteq D$ and $s \rightarrow d$. Therefore, since t simulates s , there exists some state e such that $t \rightarrow e$ and $d \leq e$. By hypothesis, from $d \leq e$ we obtain $D = P(d) \trianglelefteq P(e)$. Hence, from $C \trianglelefteq D$ and $D \trianglelefteq P(e)$, since \trianglelefteq is transitive, we obtain $C \trianglelefteq P(e)$. Thus, $t \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$ and in turn $P(t) \rightarrow^{\exists} \mu_{\mathcal{P}}(C)$. We can thus conclude that $P(s) \trianglelefteq' P(t)$.

Let us now consider *PStabilize()*. Consider an input preorder PR $\mathcal{P} = \langle P, \trianglelefteq \rangle$ (which, by Lemma 3.2, actually is a partial order PR) such that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P})$. Consider a splitter S such that $P' = \text{Split}(P, S)$ and let \trianglelefteq' be defined as in (1). Let $\mathcal{P}' = \langle P', \trianglelefteq' \rangle$ and let us check that $R_{\text{sim}} \subseteq \text{Rel}(\mathcal{P}')$, i.e., if $s \leq t$ then $P'(s) \trianglelefteq' P'(t)$. By hypothesis, if $s \leq t$ then $P(s) \trianglelefteq P(t)$. Moreover, by definition of \trianglelefteq' and since $P' \preceq P$, $P(s) \trianglelefteq P(t)$ iff $P'(s) \trianglelefteq' P'(t)$.

To sum up, we have shown that for the output PR $\langle P, \trianglelefteq \rangle$, $R_{\text{sim}} = \text{Rel}(\langle P, \trianglelefteq \rangle)$, so that $s \leq t$ iff $P(s) \trianglelefteq P(t)$. \square

4 Efficient Implementation

4.1 Data Structures

The simulation algorithm ESim is implemented by relying on the following data structures.

States: A state s is represented by a record that contains the list $\text{post}(s)$ of its successors in the transition system, a pointer $s.\text{block}$ to the block $P(s)$ that contains s and a boolean flag used for marking purposes. The whole state space Σ is represented as a doubly linked list of states. Therefore, the input transition system is represented by all the successor sets in $\{\text{post}(s)\}_{s \in \Sigma}$.

Partition: The states of any block B of the current partition P are consecutive in the list that represents Σ , so that B is represented by two pointers begin and end : $B.\text{begin}$ is the first state of B in Σ and $B.\text{end}$ is the successor of the last state of B in Σ , i.e., $B = [B.\text{begin}, B.\text{end}]$.

Moreover, B stores a boolean flag $B.intersection$ and a pointer $B.brother$ to some partition block, whose meanings are as follows: after a call to $Split(P, S)$ for splitting P w.r.t. a set of states S , if $B_1 = B \cap S$ and $B_2 = B \setminus S$, for some block $B \in P$ that has been split by S , then $B_1.intersection = \mathbf{tt}$ and $B_2.intersection = \mathbf{ff}$, while $B_1.brother$ points to B_2 and $B_2.brother$ points to B_1 . If instead some block B has not been split by S then $B.intersection = \mathbf{null}$ and $B.brother = \mathbf{null}$. Also, any block B stores in $Rem(B)$ a list of blocks of P , which is used by the function $RStabilize()$, and in $B.preE$ the list of blocks $C \in P$ such that $C \rightarrow^{\exists} B$. Finally, any block B stores in a field $B.size$ the size of B , in $B.count$ an integer counter bounded by $|P|$, which is used by the function $PStabilize()$, and a pair of boolean flags used for marking purposes. The current partition P is stored as a doubly linked list of blocks.

Relation: The current relation \sqsubseteq on P is stored as a resizable $|P| \times |P|$ boolean matrix. Recall [11, Section 17.4] that insert operations in a resizable array (whose capacity is doubled as needed) take amortized constant time and that a resizable matrix (or table) can be implemented as a resizable array of resizable arrays. The boolean matrix \sqsubseteq is resized by adding a new entry to \sqsubseteq , namely a new row and a new column, for any block B that is split into two new blocks $B \setminus S$ and $B \cap S$. The old entry B becomes the entry for the new block $B \setminus S$ while the new entry is used for the new block $B \cap S$.

Auxiliary Data Structures: We store and maintain a resizable boolean matrix $BCount$ and a resizable integer matrix $Count$, both indexed over P , whose meanings are as follows:

$$BCount(B, C) \triangleq \begin{cases} 1 & \text{if } B \rightarrow^{\exists} C \\ 0 & \text{if } B \not\rightarrow^{\exists} C \end{cases}$$

$$Count(B, C) \triangleq \sum_{E \supseteq C} BCount(B, E)$$

Hence, $Count(B, C)$ stores the number of blocks E such that $C \sqsubseteq E$ and $B \rightarrow^{\exists} E$. The table $Count$ allows to implement the test $B \not\rightarrow^{\exists} pre(\mu_{\mathcal{P}}(C))$ in constant time as $Count(B, C) = 0$.

The data structures $BCount$, $preE$, $Count$ and Rem are initialized by a function $Initialize()$ called at line 2 of $ESim$, which is described in Figure 2.

4.2 Partition Stability

Our implementation of $ESim$ will exploit the following logical characterization of partition refiners.

Theorem 4.1. *Let $\langle P, \sqsubseteq \rangle$ be a partial order PR. Then, $PRefiner(\langle P, \sqsubseteq \rangle) \neq \emptyset$ iff there exist blocks $B, C \in P$ such that the following three conditions hold:*

- (i) $B \rightarrow^{\exists} C$;
- (ii) for any $C' \in P$, if $C \triangleleft C'$ then $B \not\rightarrow^{\exists} C'$;
- (iii) $B \not\sqsubseteq pre(C)$.

Proof. Let $\mathcal{P} = \langle P, \sqsubseteq \rangle$.

(\Leftarrow) From condition (i) we have that $B \cap pre(\mu_{\mathcal{P}}(C)) \neq \emptyset$. From conditions (ii) and (iii), $B \not\sqsubseteq pre(\mu_{\mathcal{P}}(C))$. Thus, $C \in PRefiner(\mathcal{P})$.

(\Rightarrow) Assume that $PRefiner(\mathcal{P}) \neq \emptyset$. Since $\langle P, \sqsubseteq \rangle$ is a partial order, we consider a partition refiner $C \in \max(PRefiner(\mathcal{P}))$ which is maximal w.r.t. the partial order \sqsubseteq . Since C is a partition

```

1 Initialize() {
2   // Initialize BCount
3   forall B ∈ P do
4     forall C ∈ P do BCount(B, C) := 0;
5   forall B ∈ P do
6     forall x ∈ B do
7       forall y ∈ post(x) do
8         if (BCount(B, y.block) = 0) then BCount(B, y.block) := 1;
9   // Initialize preE
10  updatePreE(); // In Figure 5
11  // Initialize Count
12  forall B ∈ P do
13    forall C ∈ P do Count(B, C) := 0;
14  forall D ∈ P do
15    forall B ∈ D.preE do
16      forall C ∈ P such that C ≤ D do Count(B, C)++;
17  // Initialize Rem
18  forall C ∈ P do
19    forall B ∈ P do
20      forall D ∈ B.preE do
21        if (Count(D, C) = 0) then Rem(C).append(D);
22 }

```

Figure 2 Initialization of data structures.

refiner, there exists some $B \in P$ such that $B \cap \text{pre}(\mu_{\mathcal{P}}(C)) \neq \emptyset$ and $B \not\subseteq \text{pre}(\mu_{\mathcal{P}}(C))$. If $C' \in P$ is such that $C \triangleleft C'$ then C' cannot be a partition refiner because C is a maximal partition refiner. Hence, if $B \rightarrow^{\exists} C'$ then $B \subseteq \text{pre}(\mu_{\mathcal{P}}(C'))$, because C' is not a partition refiner, so that, since $\text{pre}(\mu_{\mathcal{P}}(C')) \subseteq \text{pre}(\mu_{\mathcal{P}}(C))$, $B \subseteq \text{pre}(\mu_{\mathcal{P}}(C))$, which is a contradiction. Hence, for any $C' \in P$ if $C \triangleleft C'$ then $B \not\rightarrow^{\exists} C'$. Therefore, from $B \cap \text{pre}(\mu_{\mathcal{P}}(C)) \neq \emptyset$ we obtain that $B \rightarrow^{\exists} C$. Moreover, from $B \not\subseteq \text{pre}(\mu_{\mathcal{P}}(C))$ we obtain that $B \not\subseteq \text{pre}(C)$. \square

Notice that the above characterization of partition refiners requires that the current PR is a partial order relation and, by Lemma 3.2, for any call to $PStabilize()$, this is actually guaranteed by the ESIm algorithm.

The algorithm in Figure 4 is an implementation of the $PStabilize()$ function that relies on Theorem 4.1 and on the data structures described in Section 4.1. The function $FindPRefiner()$ implements the conditions of Theorem 4.1: it returns a partition refiner for the current PR $\mathcal{P} = \langle P, \leq \rangle$ when this exists, otherwise it returns a null pointer. Given a block $B \in P$, the function $Post(B)$ returns a list of blocks $C \in P$ that satisfy conditions (i) and (iii) of Theorem 4.1, i.e., those blocks C such that $B \rightarrow^{\exists} C$ and $B \not\subseteq \text{pre}(C)$. This is accomplished through the counter $C.count$ that at the exit of the for-loop at lines 18-23 in Figure 4 stores the number of states in B having (at least) an outgoing transition to C , i.e., $C.count = |B \cap \text{pre}(C)|$. Hence, we have that:

$$B \rightarrow^{\exists} C \text{ and } B \not\subseteq \text{pre}(C) \Leftrightarrow 1 \leq C.count < B.size.$$

```

1 list(State) preμ(Block C) {
2   list(State) S := ∅;
3   forall x ∈ Σ do
4     forall y ∈ post(x) do
5       if (C ≤ y.block & unmarked(x)) then {S.append(x); mark(x);}
6   forall x ∈ S do unmark(x);
7   return S;
8 }

9 list(Block) Split(list(State) S) {
10  list(Block) split;
11  forall B ∈ P do B.intersection := null;
12  forall x ∈ S do
13    if (x.block.intersection = null) then
14      x.block.intersection := ff;
15      Block B := new Block; x.block.brother := B;
16      B.brother := x.block; B.intersection := tt;
17      split.append(x.block);
18  move x in list Σ from x.block at the end of B;
19  if (x.block = ∅) then // x.block ⊆ S
20    x.block.begin := B.begin; x.block.end := B.end;
21    x.block.brother := null; x.block.intersection := null;
22    split.remove(x.block); delete B;
23  return split;
24 }

```

Figure 3 Split algorithm.

Then, for any candidate partition refiner $C \in Post(B)$, it remains to check condition (ii) of Theorem 4.1. This condition is checked in $FindPRefiner()$ by testing whether $Count(B, C) = 1$: this is correct because $Count(B, C) \geq 1$ holds since $C \in Post(B)$ and therefore $B \rightarrow^{\exists} C$, so that

$$Count(B, C) = 1 \text{ iff } \forall C' \in P. C \triangleleft C' \Rightarrow B \not\rightarrow^{\exists} C'.$$

Hence, if $Count(B, C) = 1$ holds at line 13 of $FindPRefiner()$ then, by Theorem 4.1, C is a partition refiner. Once a partition refiner C has been returned by a call $Post(B)$, $PStabilize()$ splits the current partition P w.r.t. the splitter $S = pre(\mu_P(C))$ by calling the function $Split(S)$, then updates the relation \leq as defined by (1) in Section 3 by calling $updateRel()$, next updates the data structures $BCount$, $preE$, $Count$ and Rem , and finally check again whether a partition refiner exists. At the exit of the main while-loop of $PStabilize()$, the current PR $\langle P, \leq \rangle$ is partition stable.

$PStabilize()$ calls the functions $pre\mu()$ and $Split()$ in Figure 3. Recall that the states of a block B of P are consecutive in the list of states Σ , so that B is represented as $B = [B.begin, B.end]$. The implementation of $Split(S)$ is quite standard (see e.g. [21, 31]): this is based on a linear scan of the states in S and for each state in S performs some constant time operations. Hence, $Split(S)$ takes $O(|S|)$ time. Also, $Split(S)$ returns the list $split$ of blocks $B \setminus S$ such that $\emptyset \subsetneq B \setminus S \subsetneq B$ (i.e., $B.intersection = ff$). Let us remark that a call $Split(S)$ may affect the ordering of the states in the list Σ because states are moved from old blocks to newly generated blocks.

```

1 bool PStabilize() {
2   list(Block) split := ∅;
3   while (C := FindPRefiner()) ≠ null do
4     list(State) S := preμ(C); split := Split(S);
5     updateRel(split); updateBCount(split); updatePreE();
6     updateCount(split); updateRem(split);
7   return (split = ∅);
8 }

9 Block FindPRefiner() {
10  forall B ∈ P do
11    list(Block) p := Post(B);
12    forall C ∈ p do
13      if (Count(B, C) = 1) then return C;
14  return null;
15 }

16 list(Block) Post(Block B) {
17  list(Block) p := ∅;
18  forall b ∈ B, do
19    forall c ∈ post(b) do
20      Block C := c.block;
21      if unmarked1(C) then {mark1(C); C.count = 0; p.append(C);}
22      if unmarked2(C) then {mark2(C); C.count++;}
23    forall C ∈ p do unmark2(C);
24  forall C ∈ p do
25    unmark1(C);
26    if (C.count = B.size) then p.remove(C);}
27  return p;
28 }

```

Figure 4 *PStabilize*() Algorithm.

We will show that the overall time complexity of *PStabilize*() along a whole run of ESim is in $O(|P_{\text{sim}}| \rightarrow | \cdot |)$.

4.3 Updating Data Structures

In the function *PStabilize*() in Figure 4, after calling *Split*(*S*), firstly we need to update the boolean matrix that stores the relation \sqsubseteq in accordance with definition (1) in Section 3. After that, since both *P* and \sqsubseteq are changed we need to update the data structures BCount, preE, Count and Rem. The implementations of these update functions *updateRel*(), *updateBCount*(), *updatePreE*() and *updateRem*() are quite straightforward and are described in Figure 5.

The function *updateCount*() is in Figure 6 and deserves special care in order to design a time efficient implementation. The core of the *updateCount*() algorithm follows Hopcroft’s “process the smaller half” principle [24] for updating the integer matrix Count. Let *P'* be the partition which is obtained by splitting the partition *P* w.r.t. the splitter *S*. Let *B* be a block of *P* that has

```

1  updateRel(list(Block) split) {
2  forall B ∈ split do addNewEntry(B) in matrix ≤;
3  forall B ∈ P do
4  |   forall C ∈ split do
5  |   |   if (B.intersection = tt) then B ≤ C := B.brother ≤ C.brother;
6  |   |   else B ≤ C := B ≤ C.brother;
7  forall C ∈ P do
8  |   forall B ∈ split do
9  |   |   if (C.intersection = ff) then B ≤ C := B.brother ≤ C;
10 }
11 updateBCount(list(Block) split) {
12 forall B ∈ split do addNewEntry(B) in matrix Count;
13 forall B ∈ P do
14 |   forall x ∈ B do
15 |   |   forall y ∈ post(x) do BCount(B, y.block) := 0;
16 forall B ∈ P do
17 |   forall x ∈ B do
18 |   |   forall y ∈ post(x) do
19 |   |   |   if (BCount(B, y.block) = 0) then BCount(B, y.block) := 1;
20 }
21 updatePreE() {
22 forall B ∈ P do B.preE := ∅;
23 forall B ∈ P do
24 |   forall x ∈ B do
25 |   |   forall y ∈ post(x) do {unmark(B); y.block.preE.append(B);}
26 forall C ∈ P do
27 |   forall B ∈ C.preE do
28 |   |   if unmarked(B) then mark(B);
29 |   |   else C.preE.remove(B);
30 |   forall B ∈ C.preE do unmark(B);
31 }
32 updateRem(list(Block) split) {
33 forall B ∈ split do Rem(B) := Rem(B.brother);
34 }

```

Figure 5 Update functions.

been split into two new blocks $B \cap S$ and $B \setminus S$. Thus, we need to update $\text{Count}(B \cap S, C)$ and $\text{Count}(B \setminus S, C)$ for any $C \in P'$ by knowing $\text{Count}(B, \text{parent}_P(C))$. Let us first observe that after the initializations at lines 3-10 of $\text{updateCount}()$, we have that for any $B, C \in P'$, $\text{Count}(B, C) = \text{Count}(\text{parent}_P(B), \text{parent}_P(C))$. Let X be the block in $\{B \cap S, B \setminus S\}$ with the smaller size, and let Z be the other block, so that $|X| \leq |B|/2$ and $|X| + |Z| = |B|$. Let C be any block in P' . We set $\text{Count}(X, C)$ to 0, while $\text{Count}(Z, C)$ is left unchanged, namely $\text{Count}(Z, C) = \text{Count}(B, C)$. We can correctly update both $\text{Count}(Z, C)$ and $\text{Count}(X, C)$ by just scanning all the outgoing transitions from X . In fact, if $x \in X$, $x \rightarrow y$ and the block $P(y)$

is scanned for the first time then for all $C \trianglelefteq P(y)$, $\text{Count}(X, C)$ is incremented by 1, while if $Z \not\rightarrow^{\exists} P(y)$, i.e. $\text{BCount}(Z, P(y)) = 0$, then $\text{Count}(Z, C)$ is decremented by 1. The correctness of this procedure goes as follows:

- (1) At the end, $\text{Count}(X, C)$ is clearly correct because the for-loop at lines 22-28 recomputes its value from scratch.
- (2) At the end, $\text{Count}(Z, C)$ stores the correct value because:
 - (a) $\text{Count}(Z, C)$ initially stores the value $\text{Count}(B, C)$;
 - (b) if there exists some block D such that $C \trianglelefteq D$, $B \rightarrow^{\exists} D$ whereas $Z \not\rightarrow^{\exists} D$ then necessarily $X \rightarrow^{\exists} D$, because B has been split into X and Z ;
 - (c) the test $Z \not\rightarrow^{\exists} D$ is correctly implemented at line 28 as $\text{BCount}(Z, D) = 0$, since at that line the data structure BCount is up to date;
 - (d) therefore, $D = P(y)$ for some $y \in \text{post}(X)$, namely D has been taken into account by some increment $\text{Count}(X, C)++$ at line 27 and consequently $\text{Count}(Z, C)$ is decremented by 1 at line 28.

Moreover, if some block $D \in P' \setminus \{B \cap S, B \setminus S\}$ is such that both $D \rightarrow^{\exists} X$ and $D \rightarrow^{\exists} Z$ hold then for all the blocks $C \in P$ such that $C \trianglelefteq X$ (or, equivalently, $C \trianglelefteq Z$), we need to increment $\text{Count}(D, C)$ by 1. This is done at lines 30-32 by relying on the updated data structures preE and BCount .

Let us observe that the time complexity of a single call of $\text{updateCount}(\text{split})$ is

$$|P|(|\text{split}| + \sum_{X \in \text{split}} (|\{(x, y) \mid x \in X, y \in \Sigma, x \rightarrow y\}| + |\{(X, D) \mid D \in P, X \rightarrow^{\exists} D\}|)).$$

Hence, let us calculate the overall time complexity of all the calls to $\text{updateCount}()$. If X and X' are two blocks that are scanned in two different calls of $\text{updateCount}()$ and $X' \subseteq X$ then it turns out that $|X'| \leq |X|/2$. Consequently, any transition $x \rightarrow y$ at line 23 and $D \rightarrow^{\exists} X$ at line 30 can be scanned in some call of $\text{updateCount}()$ at most $\log_2 |\Sigma|$ times. Thus, the overall time complexity of $\text{updateCount}()$ is in $O(|P_{\text{sim}}| \rightarrow |\log |\Sigma|)$.

4.4 Relation Stability

The logical procedure $R\text{Stabilize}()$ in Figure 1 is implemented by the algorithm described in Figure 7. Let $\mathcal{P}^{\text{in}} = \langle P, \trianglelefteq^{\text{in}} \rangle$ be the current PR when calling $R\text{Stabilize}()$. For each relation refiner $C \in P$, the function $R\text{Stabilize}()$ must iteratively refine the initial relation $\trianglelefteq^{\text{in}}$ in accordance with definition (2) in Section 3. Hence, if $B \rightarrow^{\exists} C$, $B \trianglelefteq D$ and $D \not\rightarrow^{\exists} \mu_{\mathcal{P}^{\text{in}}}(C)$, the entry $B \trianglelefteq D$ of the boolean matrix that represents the relation \trianglelefteq must be set to **ff**. Thus, the idea is to store and incrementally maintain for each block $C \in P$ a list $\text{Rem}(C)$ of blocks $D \in P$ such that: (Inv

- If C is a relation refiner for \mathcal{P}^{in} then $\text{Rem}(C) \neq \emptyset$;
- If $D \in \text{Rem}(C)$ then necessarily $D \not\rightarrow^{\exists} \mu_{\mathcal{P}^{\text{in}}}(C)$.

It turns out that C is a relation refiner for \mathcal{P}^{in} iff there exist blocks B and D such that $B \rightarrow^{\exists} C$, $D \in \text{Rem}(C)$ and $B \trianglelefteq D$. Hence, the set of blocks $\text{Rem}(C)$ is reminiscent of the set of states $\text{remove}(s)$ used in Henzinger et al.'s [22] simulation algorithm, since each pair (B, D) which must be removed from the relation \trianglelefteq is such that $D \in \text{Rem}(C)$, for some block C .

Initially, namely at the first call of $R\text{Stabilize}()$ by ESim , $\text{Rem}(C)$ is set by the function

```

1 // Precondition: BCount and preE are updated with the current PR
2 updateCount(list(Block) split) {
3   forall B ∈ split do addNewEntry(B) in matrix Count;
4   forall B ∈ P do
5     forall C ∈ split do
6       if (B.intersection = tt) then Count(B, C) := Count(B.brother, C.brother);
7       else Count(B, C) := Count(B, C.brother);
8   forall C ∈ P do
9     forall B ∈ split do
10      if (C.intersection = ff) then Count(B, C) := Count(B.brother, C);
11  forall C ∈ P do unmark(C);
12  forall B ∈ split do
13    // Update Count(B, ·) and Count(B.brother, ·)
14    Block X, Z;
15    if (B.size ≤ B.brother.size) then
16      {X := B; Z := B.brother;}
17    else
18      {X := B.brother; Z := B;}
19    forall C ∈ P do
20      Count(X, C) := 0;
21      // Count(Z, C) := Count(B, C);
22    forall x ∈ X do
23      forall y ∈ post(x) do
24        if unmarked(y.block) then
25          mark(y.block);
26          forall C ∈ P such that C ⊆ y.block do
27            Count(X, C)++;
28            if (BCount(Z, y.block) = 0) then Count(Z, C)--;
29    // For all D ∉ {B, B.brother}, updateCount(D, ·)
30    forall D ∈ X.preE do
31      if (D ≠ X & D ≠ Z & BCount(D, Z) = 1) then
32        forall C ∈ P such that C ⊆ X do Count(D, C)++;
33 }

```

Figure 6 *updateCount()* function.

Initialize() to $\{D \in P \mid D \rightarrow^{\exists} \Sigma, D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)\}$. Hence, *RStabilize()* scans all the blocks in the current partition P and selects those blocks C such that $Rem(C) \neq \emptyset$, which are therefore candidate to be relation refiners. Then, by scanning all the blocks $B \in C.preE$ and $D \in Rem(C)$, if $B \sqsubseteq D$ holds then the entry $B \sqsubseteq D$ must be set to **ff**. However, the removal of the pair (B, D) from the current relation \sqsubseteq may affect the function $\mu_{\mathcal{P}}$. This is avoided by making a copy *oldRem(C)* of all the $Rem(C)$'s at the beginning of *RStabilize()* and then using this copy. During the main for-loop of *RStabilize()*, $Rem(C)$ must satisfy the following invariant property:

$$(Inv): \forall C \in P. Rem(C) = \{D \in P \mid D \rightarrow^{\exists} \mu_{\mathcal{P}}^{in}(C), D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)\}.$$

```

1 bool RStabilize() {
2   //  $\mu_{\mathcal{P}}^{\text{in}} := \mu_{\mathcal{P}}$ ;
3   forall  $C \in P$  do {oldRem( $C$ ) := Rem( $C$ ); Rem( $C$ ) =  $\emptyset$ ;}
4   bool Removed := ff;
5   forall  $C \in P$  such that oldRem( $C$ )  $\neq \emptyset$  do
6     // Invariant (Inv):  $\forall C \in P. \text{Rem}(C) = \{D \in P \mid D \rightarrow^{\exists} \mu_{\mathcal{P}}^{\text{in}}(C), D \not\rightarrow^{\exists} \mu_{\mathcal{P}}(C)\}$ 
7     forall  $B \in C.\text{preE}$  do
8       forall  $D \in \text{oldRem}(C)$  do
9         if ( $B \leq D$ ) then
10           $B \leq D := \text{ff}$ ; Removed := tt;
11          // update Count and Rem
12          forall  $F \in D.\text{preE}$  do
13            Count( $F, B$ ) := Count( $F, B$ ) - 1;
14            if (Count( $F, B$ ) = 0 & Rem( $B$ ) =  $\emptyset$ ) then //  $F \rightarrow^{\exists} \mu_{\mathcal{P}}^{\text{in}}(B)$  &  $F \not\rightarrow^{\exists} \mu_{\mathcal{P}}(B)$ 
15              Rem( $B$ ).append( $F$ );
16 }

```

Figure 7 RStabilize() algorithm.

This means that at the beginning of *RStabilize()*, any $\text{Rem}(C)$ is set to empty, and after the removal of a pair (B, D) from \leq , since $\mu_{\mathcal{P}}(B)$ has changed, we need: (i) to update the matrix Count, for all the entries (F, B) where $F \rightarrow^{\exists} D$, and (ii) to check if there is some block F such that $F \not\rightarrow^{\exists} \mu_{\mathcal{P}}(B)$, because any such F must be added to $\text{Rem}(B)$ in order to maintain the invariant property (Inv).

4.5 Complexity

The time complexity of the algorithm ESim relies on the following key properties:

- (1) The overall number of partition refiners found by a complete run of ESim is in $O(|P_{\text{sim}}|)$. Moreover, the overall number of newly generated blocks by the splitting operations performed by calling *Split(S)* at line 4 of *PStabilize()* is in $O(|P_{\text{sim}}|)$. In fact, let $\{P_i\}_{i \in [0, n]}$ be the sequence of different partitions computed by ESim where P_0 is the initial partition P_ℓ , P_n is the final partition P_{sim} and for all $i \in [1, n]$, P_i is the partition after the i -th call to *Split(S)*, so that $P_i \prec P_{i-1}$. The number of new blocks which are produced by a call *Split(S)* that refines P_i to P_{i+1} is $2(|P_{i+1}| - |P_i|)$. Thus, the overall number of newly generated blocks is $\sum_{i=1}^n 2(|P_i| - |P_{i-1}|) = 2(|P_{\text{sim}}| - |P_\ell|) \in O(|P_{\text{sim}}|)$.
- (2) The invariant (Inv) of the sets $\text{Rem}(C)$ guarantees the following property: if C_1 and C_2 are two blocks that are selected by the for-loop at line 5 of *RStabilize()* in two different calls of *RStabilize()*, and $C_2 \subseteq C_1$ (possibly $C_1 = C_2$) then $(\cup \text{Rem}(C_1)) \cap (\cup \text{Rem}(C_2)) = \emptyset$.

Theorem 4.2. ESim runs in $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}| + |\Sigma| \log |\Sigma|)$ -space and $O(|P_{\text{sim}}| \rightarrow \log |\Sigma|)$ -time.

Proof. Space Complexity. The input transition system is represented by the post relation, so that the size of $\{\text{post}(s)\}_{s \in \Sigma}$ is not taken into account in the space complexity of ESim.

The doubly linked list of states in Σ takes $O(|\Sigma| \log |\Sigma|)$ space while all the pointers $s.\text{block}$ take $O(|\Sigma| \log |P_{\text{sim}}|)$. The partition P and the pointers stored in each block of P overall take $O(|P_{\text{sim}}| \log |\Sigma|)$. The binary relation \triangleleft takes $O(|P_{\text{sim}}|^2)$. The auxiliary data structures Rem , preE and BCount overall take $O(|P_{\text{sim}}|^2)$ space, while the integer matrix Count takes $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}|)$. Hence, the overall bit space complexity for storing the above data structures is $O(|P_{\text{sim}}|^2 \log |P_{\text{sim}}| + |\Sigma| \log |\Sigma|)$.

Time Complexity. The time complexity bound of ESim depends on the following points.

- (A) The initialization function $\text{Initialize}()$ takes $|P|^2 + |\rightarrow| + |P| \cdot \{(B, D) \mid B, D \in P, B \rightarrow^{\exists} D\}$ time. Observe that $|P| \leq |P_{\text{sim}}| \leq |\rightarrow|$ so that the time complexity of $\text{Initialize}()$ is in $O(|P_{\text{sim}}| |\rightarrow|)$.
- (B) A call to $\text{pre}\mu(C)$ takes $O(|\rightarrow|)$ time. A call to $\text{Split}(S)$ takes $|S|$ time. Since S is returned by $\text{pre}\mu(C)$, $|S| \leq |\rightarrow|$ holds so that the time complexity of a call to $\text{Split}(S)$ is in $O(|\rightarrow|)$. A call to $\text{Post}(B)$ takes $|\{(b, c) \mid b \in B, c \in \Sigma, b \rightarrow c\}|$ time, so that a call to $\text{FindPRefiner}()$ takes $O(|\rightarrow|)$ time. Moreover, let us observe that $\text{FindPRefiner}()$ returns **null** just once, because when $\text{FindPRefiner}()$ returns **null** the current PR of ESim is both partition and relation stable and therefore ESim terminates and outputs that PR. Consequently, since, by point (1) above, the overall number of partition refiners is in $O(|P_{\text{sim}}|)$, the overall number of function calls for $\text{FindPRefiner}()$ is in $O(|P_{\text{sim}}|)$ and, in turn, the overall time complexity of $\text{FindPRefiner}()$ is in $O(|P_{\text{sim}}| |\rightarrow|)$ time. Also, the overall time complexity of $\text{pre}\mu(C)$ and $\text{Split}(S)$ is in $O(|P_{\text{sim}}| |\rightarrow|)$.
- (C) Let us observe that the calls $\text{updateRel}(\text{split})$ and $\text{updateRem}(\text{split})$ take $O(|P| |\text{split}|)$ time, while $\text{updatePreE}()$ and $\text{updateBCount}(\text{split})$ take $O(|\rightarrow|)$ time. Since the overall number of calls for these functions is in $O(|P_{\text{sim}}|)$ and since $\sum_{i \in \text{Iterations}} |\text{split}_i|$ is in $O(|P_{\text{sim}}|)$, it turns out that their overall time complexity is in $O(|P_{\text{sim}}| (|P_{\text{sim}}| + |\rightarrow|))$; therefore, since $|P_{\text{sim}}| \leq |\rightarrow|$, it is in $O(|P_{\text{sim}}| |\rightarrow|)$. Moreover, as already shown in Section 4.3, the overall time complexity of $\text{updateCount}(\text{split})$ is in $O(|P_{\text{sim}}| |\rightarrow| \log |\Sigma|)$.
- (D) Hence, by points (B) and (C), the overall time complexity of $\text{PStabilize}()$ turns out to be in $O(|P_{\text{sim}}| |\rightarrow| \log |\Sigma|)$.
- (E) Let $C \in P_{\text{in}}$ be some block of the initial partition and let $\langle C_i \rangle_{i \in I_C}$, for some set of indices I_C , be a sequence of blocks selected by the for-loop at line 5 of $\text{RStabilize}()$ such that: (a) for any $i \in I_C$, $C_i \subseteq C$ and (b) for any i , C_{i+1} has been selected after C_i and C_{i+1} is contained in C_i . Observe that C is the parent block in P_{in} of all the C_i 's. Then, by the property (2) above, it turns out that the corresponding sets in $\{\cup \text{Rem}(C_i)\}_{i \in I_C}$ are pairwise disjoint so that $\sum_{i \in I_C} |\text{Rem}(C_i)| \leq |P_{\text{sim}}|$. This property guarantees that if $D \in \text{oldRem}(C_i)$ at line 8 then for all the blocks $D' \subseteq D$ and for any $j \in I_C$ such that $i < j$, $D' \notin \text{oldRem}(C_j)$. Moreover, if the test $B \triangleleft D$ at line 9 is true for some iteration k , so that $B \triangleleft D$ is set to **ff**, then for all the blocks D' and B' such that $D' \subseteq D$ and $B' \subseteq B$ the test $D' \triangleleft B'$ will be always false for all the iterations which follow k . From these observations, we derive that the overall time complexity of the code of the for-loop at lines 7-10 is $\sum_C \sum_{i \in I_C} \sum_{B \rightarrow^{\exists} C} |\text{Rem}(C_i)| \leq |P_{\text{sim}}| |\{(B, C) \mid B, C \in P_{\text{sim}}, B \rightarrow^{\exists} C\}| \leq |P_{\text{sim}}| |\rightarrow|$. Moreover, the overall time complexity of the code of the for-loop at lines 12-15 is

$$\sum_B \sum_D \sum_{F \rightarrow^{\exists} D} 1 \leq |P_{\text{sim}}| |\{(F, D) \mid F, D \in P_{\text{sim}}, F \rightarrow^{\exists} D\}| \leq |P_{\text{sim}}| |\rightarrow|.$$

We also observe that the overall time complexity of the for-loop at line 3 of *RStabilize()* is in $O(|P_{\text{sim}}|^2)$. Thus, the overall time complexity of *RStabilize()* is in $O(|P_{\text{sim}}|(|P_{\text{sim}}| + |\rightarrow|))$, so that, since $|P_{\text{sim}}| \leq |\rightarrow|$, it is in $O(|P_{\text{sim}}||\rightarrow|)$.

Summing up, by points (A), (D) and (E), we have shown that the overall time complexity of ESIm is in $O(|P_{\text{sim}}||\rightarrow| \log |\Sigma|)$. \square

5 Conclusion and Further Work

We have introduced a new algorithm, called ESIm, for efficiently computing the simulation preorder on finite Kripke structures which: (i) reaches the space bound of the simulation algorithm GPP [16, 17] — that features the currently best space complexity — while significantly improving its time bound; (ii) significantly improves the space bound of the simulation algorithm RT [29, 31] — that features the currently best time complexity — while closely approaching (up to a multiplicative logarithmic factor) its time bound. Moreover, the space complexity of ESIm is quasi-optimal, meaning that it differs only for logarithmic factors from the size of the output.

We see a number of viable avenues for further work. A first natural question arises: can the time complexity $O(|P_{\text{sim}}||\rightarrow| \log |\Sigma|)$ of ESIm be further improved and reaches the time complexity $O(|P_{\text{sim}}||\rightarrow|)$ of RT? That is, can ESIm be modified in order to eliminate the multiplicative factor $\log |\Sigma|$ from its time complexity (while retaining its space complexity)? Presently, this goal seems to us quite hard to achieve. More in general, it would be interesting to investigate whether some lower space and time bounds can be stated for the simulation preorder problem.

ESIm has been designed for Kripke structures. While an adaptation of a simulation algorithm from Kripke structures to labeled transition systems (LTSs) can be conceptually simple, unfortunately such a shift may lead to significant losses both in space and time complexities, as argued in [6]. We mention the works [1, 23] and [25] that provide simulation algorithms for LTSs by adapting, respectively, RT and GPP. It is thus worth investigating whether and how ESIm can be efficiently adapted to work with LTSs. In particular, one could follow the basic idea used by Cécé in the simulation algorithm for LTSs in [6]. A simple replication of a data structure like $Rem(C)$ for any label l of the LTS leads to a label-indexed structure $\{Rem_l(C)\}_{l \in Labels}$. This space (and consequently time) loss could be avoided as follows. Since any $Rem_l(C)$ is always contained in the complement $P \setminus \mu_{\mathcal{P}}(C)$, which does not depend on the label l , one could instead maintain the unindexed complement $P \setminus \mu_{\mathcal{P}}(C)$ and compute $Rem_l(C)$ only when needed by restricting this search scan to the blocks in $P \setminus \mu_{\mathcal{P}}(C)$. As shown in [6], this kind of adaptation to LTSs may yield significant savings.

Finally, since the number and quality of available simulation algorithms have grown and their theoretical time and space complexities are often uncomparable, it would certainly be a worthwhile task to conduct a practical experimental comparison of the simulation algorithms listed in Table 1. In particular, this would allow to assess the practical impact of the constants hidden in their asymptotic complexities. It is here worth mentioning that a similar experimental comparison between bisimulation algorithms is already available [14, 15].

Acknowledgements. We acknowledge the contribution of Francesco Tapparo to a preliminary stage of this research which was informally presented in [30]. This work was partially supported by Microsoft Research SEIF 2013 Award and by the University of Padova under the project BECOM.

References

1. P.A. Abdulla, A. Bouajjani, L. Holík, L. Kaati and T. Vojnar. Computing simulations over tree automata. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, LNCS 4963, pp. 93–108, Springer, 2008.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. B. Bloom. *Ready simulation, bisimulation, and the semantics of CCS-like languages*. Ph.D. thesis, Massachusetts Institute of Technology, 1989.
4. B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comp. Program.*, 24(3):189-220, 1995.
5. D. Bustan and O. Grumberg. Simulation-based minimization. *ACM Trans. Comput. Log.*, 4(2):181-204, 2003.
6. G. Cécé. Three simulation algorithms for labelled transition systems. Preprint cs.arXiv:1301.1638, arXiv.org, 2013.
7. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back, 10 Years Ahead*. LNCS 2000, pp. 176-194, Springer, 2001.
8. E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. The MIT Press, 1999.
9. R. Cleaveland, J. Parrow and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Systems*, 15(1):36-72, 1993.
10. R. Cleaveland and O. Sokolsky. Equivalence and preorder checking for finite-state systems. In *Handbook of Process Algebra*, chapter 6, pp. 391-424, Elsevier, 2001.
11. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd ed., 2001.
12. S. Crafa, F. Ranzato and F. Tapparo. Saving space in a time efficient simulation algorithm. *Fundam. Inform.*, 108(1-2):23-42, 2011.
13. W. Fan, X. Wang and Y. Wu. Incremental graph pattern matching. *ACM Trans. on Database Systems*, 38(3), article no. 18, 2013.
14. K. Fisler and M.Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Proc. Int. Conf. on Formal Methods in Computer-Aided Verification*, LNCS 1522, pp. 115-132, Springer, 1998.
15. K. Fisler and M.Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39-78, 2002.
16. R. Gentilini, C. Piazza and A. Policriti. Simulation as coarsest partition problem. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, LNCS 2280, pp. 415-430, Springer, 2002.
17. R. Gentilini, C. Piazza and A. Policriti. From bisimulation to simulation: coarsest partition problems. *J. Automated Reasoning*, 31(1):73-103, 2003.
18. R. Gentilini, C. Piazza and A. Policriti. Rank and simulation: the well-founded case. *J. Logic and Computation*, to appear, 2013.
19. R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In *Handbook of Process Algebra*, chapter 1, pp. 3-99, Elsevier, 2001.
20. R. van Glabbeek and B. Ploeger. Correcting a space-efficient simulation algorithm. In *Proc. 20th Int. Conf. on Computer Aided Verification (CAV'08)*, LNCS 5123, pp. 517-529, Springer, 2008.
21. J.F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proc. 17th ICALP*, LNCS 443, pp. 626-638, Springer, 1990.
22. M.R. Henzinger, T.A. Henzinger and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36th IEEE FOCS*, 453-462, IEEE Press, 1995.
23. L. Holík and J. Šimáček. Optimizing an LTS-simulation algorithm. *Computing and Informatics*, 29(6+):1337-1348, 2010.
24. J.E. Hopcroft. A $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz eds., *Theory of Machines and Computations*, pp. 189-176, Academic Press, 1971.
25. J. Markovski. Saving time in a space-efficient simulation algorithm. In *Proc. 11th Int. Conf. on Quality Software*, pp. 244-251, IEEE Press, 2011.
26. M.U. Nisar, A. Fard and J.A. Miller. Techniques for graph analytics on big data. In *Proc. 2013 IEEE International Congress on Big Data*, pp. 255-262, IEEE Press, 2013.
27. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973-989, 1987

28. F. Ranzato. A more efficient simulation algorithm on Kripke structures. In *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS'13)*, LNCS 8087, pp. 753-764, Springer, 2013.
29. F. Ranzato and F. Tapparo. A new efficient simulation equivalence algorithm. In *Proc. 22nd Annual IEEE Symp. on Logic in Computer Science (LICS'07)*, pp. 171-180, IEEE Press, 2007.
30. F. Ranzato and F. Tapparo. A time and space efficient simulation algorithm. Short talk at *24th Annual IEEE Symposium on Logic in Computer Science (LICS'09)*, 2009.
31. F. Ranzato and F. Tapparo. An efficient simulation algorithm based on abstract interpretation. *Inf. Comput.*, 208(1):1-22, 2010.
32. L. Tan and R. Cleaveland. Simulation revisited. In *Proc. 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pp. 480-495, Springer, 2001.