Computing Stuttering Simulations

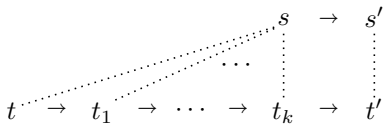
Francesco Ranzato Francesco Tapparo

Dipartimento di Matematica Pura ed Applicata
Università di Padova, Italy

Abstract. Stuttering bisimulation is a well-known behavioural equivalence that preserves CTL-X, namely CTL without the next-time operator X. Correspondingly, the stuttering simulation preorder induces a coarser behavioural equivalence that preserves the existential fragment ECTL-$\{\mathrm{X},\mathrm{G}\}$, namely ECTL without the next-time X and globally G operators. While stuttering bisimulation equivalence can be computed by the well-known Groote and Vaandrager's algorithm, to the best of our knowledge, no algorithm for computing the stuttering simulation preorder and equivalence is available. This paper presents such an algorithm for finite state systems.

1 Introduction

The Problem. Lamport's criticism [8] of the next-time operator X in CTL/CTL* arouse the interest in studying temporal logics like CTL-X/CTL*-X, obtained from CTL/CTL* by removing the next-time operator, and related notions of behavioural *stuttering*-based equivalences [1,4,6]. We are interested here in *divergence blind stuttering* simulation and bisimulation, that we call, respectively, stuttering simulation and bisimulation for short. We focus here on systems specified as Kripke structures (KSs), but analogous considerations hold for labeled transition systems (LTSs). Let $\mathcal{K} = \langle \Sigma, \rightarrow, \ell \rangle$ be a KS where $\langle \Sigma, \rightarrow \rangle$ is a transition system and ℓ is a state labeling function. A relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation on $\mathcal{K}$ when for any $s, t \in \Sigma$ such that $(s,t) \in R$: (1) s and t have the same labeling by ℓ and (2) if $s \rightarrow s'$ then $t \rightarrow^* t'$ for some t' in such a way that the following diagram holds:



where a dotted line between two states means that they are related by R. The intuition is that t is allowed to simulate a transition $s \rightarrow s'$ possibly through some initial "stuttering" transitions (τ-transitions in case of LTSs). R is called a stuttering bisimulation when it is symmetric. It turns out that the largest stuttering simulation R_{stsim} and bisimulation R_{stbis} relations exist: R_{stsim} is a preorder called the *stuttering simulation preorder* while R_{stbis} is an equivalence relation called the stuttering bisimulation equivalence. Moreover, the preorder R_{stsim} induces by symmetric reduction the *stuttering simulation equivalence* $R_{\mathrm{stsimeq}} = R_{\mathrm{stsim}} \cap R_{\mathrm{stsim}}^{-1}$. The partition of Σ corresponding to the equivalence R_{stsimeq} is denoted by P_{stsim}.

De Nicola and Vaandrager [4] showed that for finite KSs and for an interpretation of universal/existential path quantifiers over all the (possibly non-maximal and finite) paths, the stuttering bisimulation equivalence coincides with the state equivalence induced by the language CTL-X (this also holds for CTL*-X). This is not true with the standard interpretation of path quantifiers over maximal (possibly infinite) paths, since this requires a divergence sensitive notion of stuttering (see the details in [4]). Groote and Vaandrager [6] designed a well-known algorithm that computes the stuttering bisimulation equivalence R_{stbis} in $O(|\Sigma||\rightarrow|)$-time and $O(|\rightarrow|)$-space.

Clearly, stuttering simulation equivalence is coarser than stuttering bisimulation equivalence, i.e. $R_{\mathrm{stbis}} \subseteq R_{\mathrm{stsimeq}}$. As far as language preservation is concerned, it turns out that stuttering simulation equivalence coincides with the state equivalence induced by the language ECTL-$\{\mathrm{X}, \mathrm{G}\}$, namely the existential fragment of CTL without next-time X and globally G operators. Thus, on the one hand, stuttering simulation equivalence still preserves a significantly expressive fragment of CTL and, on the other hand, it may provide a significantly better state space reduction than simulation equivalence (and, in turn, bisimulation equivalence), and this has been shown to be useful in abstract model checking [9,10].

State of the Art. To the best of our knowledge, there exists no algorithm for computing stuttering simulation equivalence or, more in general, the stuttering simulation preorder. There is instead an algorithm by Bulychev et al. [2] for *checking* stuttering simulation, namely, this procedure checks whether a given relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation or not. This algorithm formalizes the problem of checking stuttering simulation as a two players game in a straightforward way and then exploits Etessami et al.'s [5] algorithm for solving such a game. The authors claim that this provides an algorithm for checking stuttering simulation on finite KSs that runs in $O(|\rightarrow|^2)$ time and space.

Main Contributions. In this paper we present an algorithm for computing simultaneously both the simulation preorder R_{stsim} and stuttering simulation equivalence R_{stsimeq} for finite KSs. This procedure is incrementally designed in two steps. We first put forward a basic procedure for computing the stuttering simulation preorder that relies directly on the notion of stuttering simulation. For any state $x \in \Sigma$, $\mathrm{StSim}(x) \subseteq \Sigma$ represents the set of states that are candidate to stuttering simulate x so that a family of sets $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ is maintained. A pair of states $(x, y) \in \Sigma \times \Sigma$ is called a refiner for StSim when $x \rightarrow y$ and there exists $z \in \mathrm{StSim}(x)$ that cannot stuttering simulate x w.r.t. y, i.e., $z \notin \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ where $\mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ is the set of all the states in $\mathrm{StSim}(x)$ that may reach a state in $\mathrm{StSim}(y)$ through a path of states in $\mathrm{StSim}(x)$. Hence, any such z can be correctly removed from $\mathrm{StSim}(x)$. It turns out that one such refiner (x, y) allows to refine StSim to StSim$'$ as follows: if $S = \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ then

$$\mathrm{StSim}'(w) := \begin{cases} \mathrm{StSim}(w) \cap S & \text{if } w \in S \\ \mathrm{StSim}(w) & \text{if } w \notin S \end{cases}$$

Thus, our basic algorithm consists in initializing $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ as $\{y \in \Sigma \mid \ell(y) = \ell(x)\}_{x\in\Sigma}$ and then iteratively refining StSim as long as a refiner exists. This provides

an *explicit* stuttering simulation algorithm, meaning that this procedure requires that for any explicit state $x \in \Sigma$, $\mathrm{StSim}(x)$ is explicitly represented as a set of states.

Inspired by techniques used in algorithms that compute standard simulation preorders and equivalences (cf. Henzinger et al. [7] and Ranzato and Tapparo [11]) and in abstract interpretation-based algorithms for computing strongly preserving abstract models [12], our stuttering simulation algorithm, called SSA, is obtained by the above basic procedure by exploiting these two main ideas.

(1) The above explicit algorithm is made "symbolic" by representing the family of sets of states $\{\mathrm{StSim}(x)\}_{x\in\Sigma}$ as a family of sets of blocks of a partition P of the state space Σ. More precisely, we maintain a partition P of Σ together with a binary relation $\trianglelefteq \subseteq P \times P$ — a so-called *partition-relation pair* — so that: (i) two states x and y in the same block of P are candidate to be stuttering simulation equivalent and (ii) if B and C are two blocks of P and $B \trianglelefteq C$ then any state in C is candidate to stuttering simulate each state in B. Therefore, here, for any $x \in \Sigma$, if $B_x \in P$ is the block of P that contains x then $\mathrm{StSim}(x) = \mathrm{StSim}(B_x) = \cup\{C \in P \mid B_x \trianglelefteq C\}$.

(2) In this setting, a refiner of the current partition-relation $\langle P, \trianglelefteq\rangle$ is a pair of blocks $(B, C) \in P \times P$ such that $B{\rightarrow^{\exists}}C$ and $\mathrm{StSim}(B) \not\subseteq \mathbf{pos}(\mathrm{StSim}(B), \mathrm{StSim}(C))$, where $\rightarrow^{\exists}$ is the existential transition relation between blocks of P, i.e., $B{\rightarrow^{\exists}}C$ iff there exist $x \in B$ and $y \in C$ such that $x{\rightarrow}y$. We devise an efficient way for finding a refiner of the current partition-relation pair that allows us to check whether a given preorder R is a stuttering simulation in $O(|P||{\rightarrow}|)$ time and $O(|\Sigma||P|\log|\Sigma|)$ space, where P is the partition corresponding to the equivalence $R \cap R^{-1}$. Hence, this algorithm for checking stuttering simulation already significantly improves both in time and space Bulychev et al.'s [2] procedure.

Our algorithm SSA iteratively refines the current partition-relation pair $\langle P, \trianglelefteq\rangle$ by first splitting the partition P and then by pruning the relation $\trianglelefteq$ until a fixpoint is reached. Hence, SSA outputs a partition-relation pair $\langle P, \trianglelefteq\rangle$ where $P = P_{\mathrm{stsim}}$ and y stuttering simulates x iff $P(x) \trianglelefteq P(y)$, where $P(x)$ and $P(y)$ are the blocks of P that contain, respectively, x and y. As far as complexity is concerned, it turns out that SSA runs in $O(|P_{\mathrm{stsim}}|^2(|{\rightarrow}| + |P_{\mathrm{stsim}}||{\rightarrow^{\exists}}|))$ time and $O(|\Sigma||P_{\mathrm{stsim}}|\log|\Sigma|)$ space. It is worth remarking that stuttering simulation yields a rather coarse equivalence so that $|P_{\mathrm{stsim}}|$ should be in general much less than the size $|\Sigma|$ of the concrete state space.

2 Background

Notation. If $R \subseteq \Sigma \times \Sigma$ is any relation and $x \in \Sigma$ then $R(x) \triangleq \{x' \in \Sigma \mid (x, x') \in R\}$. Let us recall that R is called a preorder 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 set Σ is a set of nonempty subsets of Σ, called blocks, that are pairwise disjoint and whose union gives Σ. $\mathrm{Part}(\Sigma)$ denotes the set of partitions of Σ. If $P \in \mathrm{Part}(\Sigma)$ and $s \in \Sigma$ then $P(s)$ denotes the block of P that contains s. $\mathrm{Part}(\Sigma)$ is endowed with the following standard partial order $\preceq$: $P_1 \preceq P_2$, i.e. P_2 is coarser than P_1, iff $\forall B \in P_1.\exists B' \in P_2.\ B \subseteq B'$. For a given nonempty subset $S \subseteq \Sigma$ called

splitter, we denote by $Split(P, S)$ the partition obtained from P by replacing each block $B \in P$ with the nonempty sets $B \cap S$ and $B \smallsetminus S$, where we also allow no splitting, namely $Split(P, S) = P$ (this happens exactly when S is a union of some blocks of P). If $B \in P' = Split(P, S)$ then we denote by $\mathrm{parent}_P(B)$ (or simply by $\mathrm{parent}(B)$) the unique block in P that contains B (this may possibly be B itself).

A transition system $(\Sigma, \rightarrow)$ consists of a set Σ of states and a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$. The predecessor transformer $\mathrm{pre} : \wp(\Sigma) \rightarrow \wp(\Sigma)$ is defined as usual: $\mathrm{pre}(Y) \triangleq \{s \in \Sigma \mid \exists t \in Y.\ s \rightarrow t\}$. 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 $s_1 \rightarrow s_2$. Given a set AP of atomic propositions (of some specification language), a Kripke structure (KS) $\mathcal{K} = (\Sigma, \rightarrow, \ell)$ over AP consists of a transition system $(\Sigma, \rightarrow)$ together with a state labeling function $\ell : \Sigma \rightarrow \wp(AP)$. $P_\ell \in \mathrm{Part}(\Sigma)$ denotes the state partition induced by ℓ, namely, $P_\ell \triangleq \{\{s' \in \Sigma \mid \ell(s) = \ell(s')\}\}_{s \in \Sigma}$.

Stuttering Simulation. Let $\mathcal{K} = (\Sigma, \rightarrow, \ell)$ be a KS. A relation $R \subseteq \Sigma \times \Sigma$ is a *divergence blind stuttering simulation* on $\mathcal{K}$ if for any $s, t \in \Sigma$ such that $(s, t) \in R$:

(1) $\ell(s) = \ell(t)$;
(2) If $s \rightarrow s'$ then there exist $t_0, ..., t_k \in \Sigma$, with $k \geq 0$, such that: (i) $t_0 = t$; (ii) for all $i \in [0, k)$, $t_i \rightarrow t_{i+1}$ and $(s, t_i) \in R$; (iii) $(s', t_k) \in R$.

Observe that condition (2) allows the case $k = 0$ and this boils down to requiring that $(s', t) \in R$. With a slight abuse of terminology, R is simply called a *stuttering simulation*. If $(s, t) \in R$ for some stuttering simulation R then we say that t stuttering simulates s and we denote this by $s \leq t$. If R is a symmetric relation then it is called a stuttering bisimulation. The empty relation is a stuttering simulation and stuttering simulations are closed under union so that the largest stuttering simulation relation exists. It turns out that the largest simulation is a preorder relation called *stuttering simulation preorder* (on $\mathcal{K}$) and denoted by R_{stsim}. Thus, for any $s, t \in \Sigma$, $s \leq t$ iff $(s, t) \in R_{\mathrm{stsim}}$. *Stuttering simulation equivalence* R_{stsimeq} is the symmetric reduction of R_{stsim}, namely $R_{\mathrm{stsimeq}} \triangleq R_{\mathrm{stsim}} \cap R_{\mathrm{stsim}}^{-1}$, so that $(s, t) \in R_{\mathrm{stsimeq}}$ iff $s \leq t$ and $t \leq s$. $P_{\mathrm{stsim}} \in \mathrm{Part}(\Sigma)$ denotes the partition corresponding to the equivalence R_{stsimeq} and is called stuttering simulation partition.
Following Groote and Vaandrager [6], $\mathbf{pos} : \wp(\Sigma) \times \wp(\Sigma) \rightarrow \wp(\Sigma)$ is defined as:

$$\mathbf{pos}(S, T) \triangleq \{s \in S \mid \exists k \geq 0. \exists s_0, ..., s_k.\ s_0 = s\ \&\ \forall i \in [0, k).\ s_i \in S,\ s_i \rightarrow s_{i+1}\ \&\ s_k \in T\}$$

so that a relation $R \subseteq \Sigma \times \Sigma$ is a stuttering simulation iff for any $x, y \in \Sigma$, $R(x) \subseteq P_\ell(x)$ and if $x \rightarrow y$ then $R(x) \subseteq \mathbf{pos}(R(x), R(y))$.

It turns out [4] that P_{stsim} is the coarsest partition preserved by the temporal language ECTL-$\{\mathrm{X}, \mathrm{G}\}$. More precisely, ECTL-$\{\mathrm{X}, \mathrm{G}\}$ is inductively defined as follows:

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \mathrm{EU}(\phi_1, \phi_2)$$

where $p \in AP$ and its semantics is standard, namely $[\![p]\!] \triangleq \{s \in \Sigma \mid p \in \ell(s)\}$ and $[\![\mathrm{EU}(\varphi_1, \varphi_2)]\!] \triangleq [\![\varphi_2]\!] \cup \mathbf{pos}([\![\varphi_1]\!], [\![\varphi_2]\!])$. The coarsest partition preserved by the

```
BasicSSA(Partition P_ℓ) {
  forall x ∈ Σ do StSim(x) := P_ℓ(x);
  while (∃x, y ∈ Σ such that x→y & StSim(x) ⊈ pos(StSim(x), StSim(y))) do
    S := pos(StSim(x), StSim(y));
    forall w ∈ S do StSim(w) := StSim(w) ∩ S;
}
```

Fig. 1. Basic Stuttering Simulation Algorithm $BasicSSA$.

language ECTL-$\{\mathrm{X}, \mathrm{G}\}$ is the state partition corresponding to the following equivalence $\sim$ between states: for any $s, t \in \Sigma$, $s \sim t$ iff $\forall \phi \in \text{ECTL-}\{\mathrm{X}, \mathrm{G}\}$. $s \in [\![\phi]\!] \Leftrightarrow t \in [\![\phi]\!]$.

3 Basic Algorithm

For each state $x \in \Sigma$, the algorithm $BasicSSA$ in Figure 1 computes the stuttering simulator set $\mathrm{StSim}(x) \subseteq \Sigma$, i.e., the set of states that stuttering simulate x. The basic idea is that $\mathrm{StSim}(x)$ contains states that are candidate for stuttering simulating x. Thus, the input partition of $BasicSSA$ is taken as the partition P_ℓ determined by the labeling ℓ so that $\mathrm{StSim}(x)$ is initialized with $P_\ell(x)$, i.e., with all the states that have the same labeling of x. Following the definition of stuttering simulation, a refiner is a pair of states (x, y) such that $x \rightarrow y$ and $\mathrm{StSim}(x) \not\subseteq \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$. In fact, if $z \in \mathrm{StSim}(x) \smallsetminus \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ then z cannot stuttering simulate x and therefore can be correctly removed from $\mathrm{StSim}(x)$. On the other hand, if no such refiner exists then for any $x, y \in \Sigma$ such that $x \rightarrow y$ we have that $\mathrm{StSim}(x) \subseteq \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ so that any $z \in \mathrm{StSim}(x)$ actually stuttering simulates x. Hence, $BasicSSA$ consists in iteratively refining $\{\mathrm{StSim}(x)\}_{x \in \Sigma}$ as long as a refiner exists, where, given a refiner (x, y), the refinement of StSim by means of $S = \mathbf{pos}(\mathrm{StSim}(x), \mathrm{StSim}(y))$ is as follows:

$$\mathrm{StSim}(w) := \begin{cases} \mathrm{StSim}(w) \cap S & \text{if } w \in S \\ \mathrm{StSim}(w) & \text{if } w \notin S \end{cases}$$

Theorem 3.1 (Termination and Correctness). *For finite KSs, $BasicSSA$ terminates and is correct, i.e., if* StSim *is the output of $BasicSSA$ on input P_ℓ then for any $x, y \in \Sigma$, $y \in \mathrm{StSim}(x) \Leftrightarrow x \leq y$.*

4 Partition-Relation Pairs

A *partition-relation pair* $\langle P, \trianglelefteq \rangle$, PR for short, is given by a partition $P \in \mathrm{Part}(\Sigma)$ 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 $(B', C') \trianglelefteq (B, C)$ when $B' \trianglelefteq B$ and $C' \trianglelefteq C$. Our stuttering simulation algorithm relies on the idea of symbolizing the $BasicSSA$ procedure in order to maintain a PR $\langle P, \trianglelefteq \rangle$ in place of the family of explicit sets of states

$\{\mathrm{StSim}(s)\}_{s\in\Sigma}$. As a first step, $\mathcal{S} = \{\mathrm{StSim}(s)\}_{s\in\Sigma}$ induces a partition P that corresponds to the following equivalence $\sim_{\mathcal{S}}$: $s_1 \sim_{\mathcal{S}} s_2 \Leftrightarrow \mathrm{StSim}(s_1) = \mathrm{StSim}(s_2)$. Hence, the intuition is that if $P(s_1) = P(s_2)$ then s_1 and s_2 are "currently" candidates to be stuttering simulation equivalent. Accordingly, a relation $\trianglelefteq$ on P encodes stuttering simulation as follows: if $s \in \Sigma$ then $\mathrm{StSim}(s) = \{t \in \Sigma \mid P(s) \trianglelefteq P(t)\}$. Here, the intuition is that if $B \trianglelefteq C$ then any state $t \in C$ is "currently" candidate to stuttering simulate any state $s \in B$. Equivalently, the following invariant property is maintained: if $s \leq t$ then $P(s) \trianglelefteq P(t)$. Thus, a PR $\langle P, \trianglelefteq\rangle$ will represent the current approximation of the stuttering simulation preorder and in particular P will represent the current approximation of stuttering simulation equivalence.

More in detail, a PR $\mathcal{P} = \langle P, \trianglelefteq\rangle$ induces the following map $\mu_{\mathcal{P}} : \wp(\Sigma) \to \wp(\Sigma)$: for any $X \in \wp(\Sigma)$,

$$\mu_{\mathcal{P}}(X) \triangleq \cup\{C \in P \mid \exists B \in P.\, B \cap X \neq \varnothing,\ B \trianglelefteq C\}.$$

Note that, for any $s \in \Sigma$, $\mu_{\mathcal{P}}(s) = \mu_{\mathcal{P}}(P(s)) = \{t \in \Sigma \mid P(s) \trianglelefteq P(t)\}$, that is, $\mu_{\mathcal{P}}(s)$ represents the set of states that are currently candidates to stuttering simulate s. A PR $\mathcal{P}$ is therefore defined to be a stuttering simulation for a KS $\mathcal{K}$ when the relation $\{(s,t) \in \Sigma \times \Sigma \mid s \in \Sigma,\ t \in \mu_{\mathcal{P}}(s)\}$ is a stuttering simulation on $\mathcal{K}$.

Recall that in *BasicSSA* a pair of states $(s,t) \in \Sigma \times \Sigma$ is a refiner for StSim when $s{\to}t$ and $\mathrm{StSim}(s) \not\subseteq \mathbf{pos}(\mathrm{StSim}(s), \mathrm{StSim}(t))$. Accordingly, a pair of blocks $(B,C) \in P \times P$ is a refiner for $\mathcal{P}$ when $B{\to}^{\exists}C$ and $\mu_{\mathcal{P}}(B) \not\subseteq \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))$. Thus, by defining

$$\mathrm{Refiner}(\mathcal{P}) \triangleq \{(B,C) \in P \times P \mid B{\to}^{\exists}C,\ \mu_{\mathcal{P}}(B) \not\subseteq \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))\}$$

the following characterization holds:

Theorem 4.1. *$\mathcal{P} = (P, \trianglelefteq)$ is a stuttering simulation iff for any $s \in \Sigma$, $\mu_{\mathcal{P}}(s) \subseteq P_\ell(s)$ and* $\mathrm{Refiner}(\mathcal{P}) = \varnothing$.

4.1 A Symbolic Algorithm

The algorithm *BasicSSA* is therefore made symbolic as follows:

(1) $\langle P_\ell, \mathrm{id}\rangle$ is the input PR, where $(B,C) \in \mathrm{id} \Leftrightarrow B = C$;

(2) Find $(B,C) \in \mathrm{Refiner}(\mathcal{P})$; if $\mathrm{Refiner}(\mathcal{P}) = \varnothing$ then exit;

(3) Compute $S = \mathbf{pos}(\mu_{\mathcal{P}}(B), \mu_{\mathcal{P}}(C))$;

(4) $\mathcal{P}' := \langle P', \trianglelefteq'\rangle$, where $P' = \mathit{Split}(P, S)$ and $\trianglelefteq'$ is modified in such a way that for any $s \in \Sigma$, $\mu_{\mathcal{P}'}(P'(s)) = \mu_{\mathcal{P}}(P(s))$;

(5) $\mathcal{P}'' := \langle P', \trianglelefteq''\rangle$, where $\trianglelefteq'$ is modified to $\trianglelefteq''$ in such a way that for any $B \in P'$:

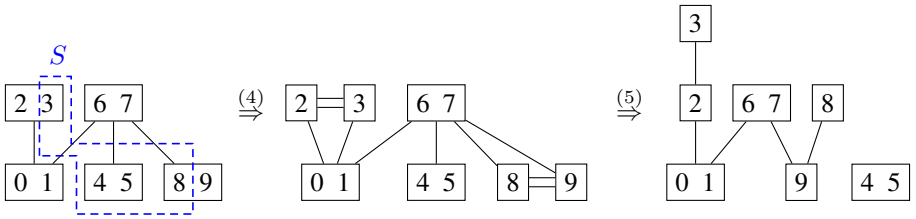
$$\mu_{\mathcal{P}''}(B) = \begin{cases} \mu_{\mathcal{P}'}(B) \cap S & \text{if } B \subseteq S \\ \mu_{\mathcal{P}'}(B) & \text{if } B \cap S = \varnothing \end{cases}$$

(6) $\mathcal{P} := \mathcal{P}''$ and go to (2).

```
1 SSA(PR ⟨P, Rel⟩) {
2   Initialize();
3   while ((B, C) := FindRefiner()) ≠ (null, null) do
4     list⟨State⟩ X := Image(⟨P, Rel⟩, B),  Y := Image(⟨P, Rel⟩, C);
5     list⟨State⟩ S := pos(X, Y);
6     SplittingProcedure(⟨P, Rel⟩, S);
7     Refine(⟨P, Rel⟩, S);
8 }
```

Fig. 2. Stuttering Simulation Algorithm SSA.

This leads to the symbolic algorithm SSA described in Figure 2, where: the input PR $\langle P, Rel\rangle$ at line 1 is $\langle P_\ell, \mathrm{id}\rangle$ of point (1); point (2) corresponds to the call $FindRefiner()$ at line 3; point (3) corresponds to lines 4-5; point (4) corresponds to the call $SplittingProcedure(\langle P, Rel\rangle, S)$ at line 6; point (5) corresponds to the call $Refine(\langle P, Rel\rangle, S)$ at line 7. The following graphical example shows how points (4) and (5) refine a PR $\langle\{[0,1],[2,3],[4,5],[6,7],[8,9]\}, \trianglelefteq\rangle$ w.r.t. the set $S = \{3,4,5,8\}$, where if $B \lhd C$ then B is drawed below C while if $B \lhd C$ and $C \lhd B$ then B and C are at same height and connected by a double line.



Theorem 4.2 (Correctness). *SSA is a correct implementation of $BasicSSA$, i.e., if* StSim *is the output function of $BasicSSA$ on input partition P_ℓ then SSA on input PR $\langle P_\ell, \mathrm{id}\rangle$ terminates with an output PR $\mathcal{P}$ such that for any $x \in \Sigma$,* $\mathrm{StSim}(x) = \mu_{\mathcal{P}}(x)$.

5 Bottom States

While it is not too hard to devise an efficient implementation of lines 2 and 4-7 of the SSA algorithm, it is instead not straightforward to find a refiner in an efficient way. In Groote and Vaandrager's [6] algorithm for computing stuttering bisimulations the key point for efficiently finding a refiner in their setting is the notion of *bottom state*. Given a set of states $S \subseteq \Sigma$, a bottom state of S is a state $s \in S$ that cannot go inside S, i.e., s can only go outside S (note that s may also have no outgoing transition). For any $S \subseteq \Sigma$, we therefore define:

$$\mathrm{Bottom}(S) \triangleq S \setminus \mathrm{pre}(S).$$

Bottom states allow to efficiently find refiners in KSs that do not contain cycles of states all having the same labeling. Following Groote and Vaandrager [6], a transition

$s \to t$ is called *inert* for a partition $P \in \mathrm{Part}(\Sigma)$ when $P(s) = P(t)$. Clearly, if a set of states S in a KS is strongly connected via inert transitions for the labeling partition P_ℓ then all the states in S are stuttering simulation equivalent, i.e., if $s, s' \in S$ then $P_{\mathrm{stsim}}(s) = P_{\mathrm{stsim}}(s')$. Thus, each strongly connected component (s.c.c.) S with respect to inert transitions for P_ℓ, called *inert s.c.c.*, can be collapsed to one single "symbolic state". In particular, if $\{s\}$ is one such inert s.c.c., i.e. if $s \to s$, then this collapse simply removes the transition $s \to s$. It is important to remark that a standard depth-first search algorithm by Tarjan [3], running in $O(|\Sigma| + |\to|)$ time, allows us to find and then collapse all the inert s.c.c.'s in the input KS. We can thus assume w.l.o.g. that the input KS $\mathcal{K}$ does not contain inert s.c.c.'s. The following characterization of refiners therefore holds.

Lemma 5.1. *Assume that $\mathcal{K}$ does not contain inert s.c.c.'s. Let $\mathcal{P} = \langle P, \trianglelefteq \rangle$ be a PR such that for any $B \in P$, $\mu_{\mathcal{P}}(B) \subseteq P_\ell(B)$. Consider $(B, C) \in P \times P$ such that $B \to^{\exists} C$. Then, $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff $\mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \not\subseteq \mu_{\mathcal{P}}(C) \cup \mathrm{pre}(\mu_{\mathcal{P}}(C))$.*

If $B \in P$ is any block then we define as *local bottom states* of B all the bottom states of $\mu_{\mathcal{P}}(B)$ that belong to B, namely

$$\mathrm{localBottom}(B) \triangleq \mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \cap B.$$

Also, we define $C \in P$ as a *bottom block* for B when $B \lhd C$ and C contains at least a bottom state of $\mu_{\mathcal{P}}(B)$, that is:

$$\mathrm{bottomBlock}(B) \triangleq \{C \in P \mid B \lhd C,\ C \cap \mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \neq \varnothing\}.$$

Local bottoms and bottom blocks characterize refiners for stuttering simulation as follows:

Theorem 5.2. *Assume that $\mathcal{K}$ does not contain inert s.c.c.'s. Let $\mathcal{P} = \langle P, \trianglelefteq \rangle$ be a PR such that $\trianglelefteq$ is a preorder and for any $B \in P$, $\mu_{\mathcal{P}}(B) \subseteq P_\ell(B)$. Consider $(B, C) \in P \times P$ such that $B \to^{\exists} C$ and for any (D, E) such that $D \to^{\exists} E$ and $(B, C) \lhd (D, E)$, $(D, E) \notin \mathrm{Refiner}(\mathcal{P})$. Then, $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff at least one of the following two conditions holds:*

(i) *$C \not\trianglelefteq B$ and $\mathrm{localBottom}(B) \not\subseteq \mathrm{pre}(\mu_{\mathcal{P}}(C))$;*
(ii) *There exists $D \in \mathrm{bottomBlock}(B)$ such that $C \not\trianglelefteq D$ and $D \not\to^{\exists} \mu_{\mathcal{P}}(C)$.*

We will show that this characterization provides the basis for an algorithm that efficiently finds refiners. Hence, this procedure also checks whether a given preorder R is a stuttering simulation or not. This can be done in $O(|P||\to|)$ time and $O(|\Sigma||P| \log |\Sigma|)$ space, where P is the partition corresponding to the equivalence $R \cap R^{-1}$. Thus, this algorithm for checking stuttering simulation already significantly improves Bulychev et al.'s [2] procedure that runs in $O(|\to|^2)$ time and space.

6 Implementation

6.1 Data Structures

SSA is implemented by exploiting the following data structures.

(i) A state s is represented by a record that contains the list $\mathrm{pre}(s)$ of its predecessors and a pointer s.block to the block $P(s)$ that contains s. The state space Σ is represented as a doubly linked list of states.

(ii) The states of any block B of the current partition P are consecutive in the list Σ, so that B is represented by two pointers begin and end: the first state of B in Σ and the successor of the last state of B in Σ, i.e., $B = [B.\mathrm{begin}, B.\mathrm{end}[$. Moreover, B contains a pointer B.intersection to a block whose meaning is as follows: after a call to $\mathit{Split}(P, S)$ for splitting P w.r.t. a set of states S, if $\varnothing \neq B \cap S \subsetneq B$ then B.intersection points to a block that represents $B \cap S$, otherwise B.intersection $=$ **null**. Finally, the fields localBottoms and bottomBlocks for a block B represent, respectively, the local bottom states of B and the bottom blocks of B. The current partition P is stored as a doubly linked list of blocks.

(iii) The current relation $\trianglelefteq$ on P is stored as a resizable $|P| \times |P|$ boolean matrix Rel: $\mathit{Rel}(B, C) =$ **tt** iff $B \trianglelefteq C$. Recall [3, Section 17.4] that insert operations in a resizable array (whose capacity is doubled as needed) take amortized constant time, and a resizable matrix (or table) can be implemented as a resizable array of resizable arrays. The boolean matrix Rel is resized by adding a new entry to Rel, namely a new row and a new column, for any block B that is split into two new blocks $B \smallsetminus S$ and $B \cap S$.

(iv) SSA additionally stores and maintains a resizable integer table Count and a resizable integer matrix BCount. Count is indexed over Σ and P and has the following meaning: $\mathrm{Count}(s, C) \triangleq |\{(s, t) \mid C \trianglelefteq D, t \in D, s{\to}t\}|$. BCount is indexed over $P \times P$ and has the following meaning: $\mathrm{BCount}(B, C) \triangleq \sum_{s \in B} \mathrm{Count}(s, C)$. The table Count allows to implement the test $s \notin \mathrm{pre}(\mu_{\mathcal{P}}(C))$ in constant time as $\mathrm{Count}(s, C) = 0$, while BCount allows to implement the test $B \not\to^{\exists} \mu_{\mathcal{P}}(C)$ in constant time as $\mathrm{BCount}(B, C) = 0$.

6.2 *FindRefiner* Algorithm

The algorithm $\mathit{FindRefiner}()$ in Figure 3 is an implementation of the characterization of refiners provided by Theorem 5.2. In particular, lines 8-10 implement condition (i) of Theorem 5.2 and lines 11-12 implement condition (ii). The correctness of this implementation depends on the following key point. Given a pair of blocks $(B, C) \in P \times P$ such that $B{\to}^{\exists}C$, in order to ensure the equivalence: $(B, C) \in \mathrm{Refiner}(\mathcal{P})$ iff (i) $\vee$ (ii), Theorem 5.2 requires as hypothesis the following condition:

$$\forall (D, E) \in P \times P.\ D{\to}^{\exists}E\ \&\ (B, C) \lhd (D, E)\ \Rightarrow (D, E) \notin \mathrm{Refiner}(\mathcal{P}) \qquad (*)$$

In order to ensure this condition $(*)$, we guarantee throughout the execution of SSA that the list P of blocks is stored in reverse topological ordering w.r.t. $\trianglelefteq$, so that if $B \lhd B'$ then B' precedes B in the list P. The reverse topological ordering of P initially holds because the input PR is the DAG $\langle P_\ell, \mathrm{id}\rangle$ which is trivially topologically ordered (whatever the ordering of P_ℓ is). More in general, for a generic input PR $\langle P, \mathit{Rel}\rangle$ to SSA the function $\mathit{Initialize}()$ achieves this reverse topological ordering by a standard algorithm [3, Section 22.4] that runs in $O(|P|^2)$ time. Then, the reverse topological

```
1  Precondition: The list P is stored in reverse topological ordering wrt Rel
2  ⟨Block, Block⟩ FindRefiner() {
3    matrix⟨bool⟩ Refiner;
4    forall B ∈ P do  forall C ∈ P do Refiner(B,C) := maybe;
5    forall C ∈ P do
6      forall B ∈ P such that B→∃C do
7        if (Refiner(B,C) = maybe) then
8          if (Rel(C, B) = ff) then
9            forall s ∈ B.localBottoms do
10             if (Count(s, C) = 0) then return (B, C);
11         forall D ∈ B.bottomBlocks do
12           if (Rel(C, D) = ff & BCount(D, C) = 0) then return (B, C);
13         forall E ∈ P do
14           if (Rel(E, C) = tt) then Refiner(B,E) := ff;
15   return (null,null);
16 }
```

Fig. 3. $FindRefiner()$ algorithm.

ordering of P is always maintained throughout the execution of SSA. In fact, if the partition P is split w.r.t. a set S and a block B generates two new descendant blocks $B \cap S$ and $B \setminus S$ then our *SplittingProcedure* in Figure 5 modifies the ordering of the list P as follows: B is replaced in P by inserting $B \cap S$ immediately followed by $B \setminus S$. This guarantees that at the exit of $Refine(\langle P, Rel\rangle, S)$ at line 7 of SSA the list P is still in reverse topological ordering w.r.t. Rel. This is a consequence of the fact that at the exit of $Refine(\langle P, Rel\rangle, S)$, by point (5) in Section 4.1, we have that $\mu_{\langle P,Rel\rangle}(B \cap S) = \mu_{\langle P,Rel\rangle}(B) \cap S$, i.e., $\mu_{\langle P,Rel\rangle}(B \cap S) \cap (B \setminus S) = \varnothing$ so that $B \cap S \not\trianglelefteq B \setminus S$. The reverse topological ordering of P w.r.t. $\trianglelefteq$ ensures that if $(B, C) \triangleleft (B', C')$ then the pair (B, C) is scanned by $FindRefiner$ after the pair (B', C'). Since $FindRefiner()$ exits as soon as a refiner is found, we have that (B', C') cannot be a refiner, so that condition $(*)$ holds for (B, C).

When $FindRefiner()$ determines that a pair of blocks (B, C), with $B \rightarrow^{\exists} C$, is not a refiner, it stores this information in a local boolean matrix Refiner that is indexed over $P \times P$ and initialized to **maybe**. Thus, the meaning of the matrix Refiner is as follows: if $\mathrm{Refiner}(B, C) = \textbf{ff}$ then $(B, C) \notin \mathrm{Refiner}(\mathcal{P})$. If $(B, C) \notin \mathrm{Refiner}(\mathcal{P})$ then both (i) and (ii) do not hold, therefore $FindRefiner()$ executes the for-loop at lines 13-14 so that any (B, E) with $E \trianglelefteq C$ is marked as $\mathrm{Refiner}(B, E) = \textbf{ff}$. This is correct because if $(B, C) \notin \mathrm{Refiner}(\mathcal{P})$ and $(B, E) \trianglelefteq (B, C)$ then $(B, E) \notin \mathrm{Refiner}(\mathcal{P})$: in fact, by Lemma 5.1, $\mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \subseteq \mu_{\mathcal{P}}(C) \cup \mathrm{pre}(\mu_{\mathcal{P}}(C))$, and since $E \trianglelefteq C$ implies, because $\trianglelefteq$ is transitive, $\mu_{\mathcal{P}}(C) \subseteq \mu_{\mathcal{P}}(E)$, we have that $\mathrm{Bottom}(\mu_{\mathcal{P}}(B)) \subseteq \mu_{\mathcal{P}}(E) \cup \mathrm{pre}(\mu_{\mathcal{P}}(E))$, so that, by Lemma 5.1, $(B, E) \notin \mathrm{Refiner}(\mathcal{P})$. The for-loop at lines 13-14 is therefore an optimization of Theorem 5.2 since it determines that some pairs of

```
1  Precondition: TS(S, →, P_ℓ)  &  ∀x, y ∈ S. P_ℓ(x) = P_ℓ(y)
2  list⟨State⟩ pos(list⟨State⟩ S, list⟨State⟩ T) {
3     list⟨State⟩ R := ∅;
4     forall s ∈ S do mark1(s);
5     forall t ∈ T do
6        forall s ∈ pre(t) such that marked1(s) do
7           mark2(s); R.append(s);
8     forall y ∈ S backward such that marked2(y) do
9        forall x ∈ pre(y) such that marked1(x) & unmarked2(x) do
10          mark2(x); R.append(x);
11    forall x ∈ S do unmark1(x); forall x ∈ R do unmark2(x);
12    return R;
13 }
```

Fig. 4. Computation of $\mathbf{pos}$.

blocks are not a refiner without resorting to the condition ¬(i) ∧ ¬(ii) of Theorem 5.2. This optimization and the related matrix Refiner turn out to be crucial for obtaining the overall time complexity of SSA.

6.3 Computing pos

Given two lists of states S and T, we want to compute the set of states that belong to $\mathbf{pos}(S, T)$. This can be done by traversing once the edges of the transition relation $\rightarrow$ provided that the list Σ of states satisfies the following property:

$$\text{For all } x, y \in \Sigma, \text{ if } x \text{ precedes } y \text{ in the list } \Sigma \text{ and } \ell(x) = \ell(y) \text{ then } y \not\rightarrow x.$$

We denote this property by $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$. Hence, this is a topological ordering of Σ w.r.t. the transition relation $\rightarrow$ that is local to each block of the labeling partition P_ℓ. As described in Section 5, as an initial pre-processing step of SSA, we find and collapse inert s.s.c.'s. After this pre-processing step, Σ is successively topologically ordered w.r.t. $\rightarrow$ locally to each block of P_ℓ in $O(|\Sigma| + |\rightarrow|)$ time in order to initially establish $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$. We will see in Section 6.4 that while the ordering of the list Σ of states changes across the execution of SSA, the property $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$ is always maintained invariant.

The computation of $\mathbf{pos}(S, T)$ is done by the algorithm in Figure 4. The result R consists of all the states in S that are marked2. We assume that all the states in S have the same labeling by ℓ: this is clearly true when the function $\mathbf{pos}$ is called from the algorithm SSA. The for-loop at lines 5-7 makes the states in $S \cap \mathrm{pre}(T)$ marked2. Then, the for-loop at lines 8-10 scans backward the list of states S and when a marked2 state y is encountered then all the states in $S \cap \mathrm{pre}(y)$ are marked2. It is clear that the property $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$ guarantees that this procedure does not miss states that are in $\mathbf{pos}(S, T)$.

```
1  list⟨Block⟩ Split(list⟨Block⟩ P, list⟨State⟩ S) {
2    list⟨Block⟩ split;
3    forall x ∈ S do
4      if (x.block.intersection = null) then
5        Block B := new Block;
6        x.block.intersection := B;
7        split.append(x.block);
8      move x in the list Σ from x.block at the end of B;
9      if (x.block = ∅) then x.block := copy(B); x.block.intersection := null;
10   forall B ∈ split do
11     if (B.intersection = null) then split.remove(B); delete B;
12     else insert B.intersection in P in front of B;
13   return split;
14 }

15 void SplittingProcedure(PR ⟨P, Rel⟩, list⟨State⟩ S) {
16   list⟨Block⟩ split := Split(P, S);
17   if (split ≠ ∅) then
18     resize Rel; // update Rel
19     forall B ∈ P do forall C ∈ split do Rel(C.intersection, B) := Rel(C, B);
20     forall B ∈ split do forall C ∈ P do Rel(C, B.intersection) := Rel(C, B);
21     Update(); // update Count, BCount, localBottoms, bottomBlocks
22     forall B ∈ P do B.intersection := null;
23 }
```

Fig. 5. Splitting Procedure.

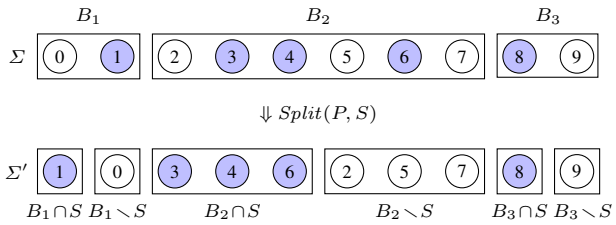
6.4 *SplittingProcedure*

SSA calls $SplittingProcedure(\langle P, Rel\rangle, S)$ at line 6 with the precondition $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$ and needs to maintain this invariant property at the exit (as discussed in Section 6.3 this is crucial for computing $\mathbf{pos}$). This function must modify the current PR $\mathcal{P} = \langle P, Rel\rangle$ to $\mathcal{P}' = \langle P', Rel'\rangle$ as follows:

(A) P' is the partition obtained by splitting P w.r.t. the splitter S;
(B) Rel is modified to Rel' in such a way that for any $x \in \Sigma$, $\mu_{\mathcal{P}'}(P'(x)) = \mu_{\mathcal{P}}(P(x))$.

Recall that the states of a block B of P are consecutive in the list Σ, so that B is represented as $B = [B.\mathrm{begin}, B.\mathrm{end}[$. An implementation of the splitting operation $Split(P, S)$ that only scans the states in S, i.e. that takes $O(|S|)$ time, is quite easy and standard (see e.g. [6,11]). However, this operation affects the ordering of the states in the list Σ because states are moved from old blocks to newly generated blocks. It turns out that this splitting operation can be implemented in a careful way that preserves the invariant property $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$. The idea is rather simple. Observe that the list of states $S = \mathbf{pos}(\mu_{\mathcal{P}}(X), \mu_{\mathcal{P}}(Y))$ can be (and actually is) built as a sublist of Σ so that the following property holds: If x precedes y in S and $P_\ell(x) = P_\ell(y)$ then $y \not\rightarrow x$. The

following picture shows the idea of our implementation of $Split(P, S)$, where states within filled circles determine the splitter set S.



The property $\mathrm{TS}(\Sigma', \rightarrow, P_\ell)$ still holds for the modified list of states Σ'. In fact, from the above picture observe that it is enough to check that: if B has been split into $B \cap S$ and $B \setminus S$ by preserving the relative orders of the states in Σ then if $x \in B \cap S$ and $y \in B \setminus S$ then $y \not\rightarrow x$. This is true because if $y \rightarrow x$ and $x \in S = \mathbf{pos}(\mu_{\mathcal{P}}(X), \mu_{\mathcal{P}}(Y))$ then, since x and y are in the same block of P and $\mu_{\mathcal{P}}(X)$ is a union of some blocks of P, by definition of $\mathbf{pos}$ we would also have that $y \in S$, which is a contradiction.

The functions in Figure 5 sketch a pseudo-code that implements the above described splitting operation (the $Update()$ function that updates data structures is omitted). The above point (B), i.e., the modification of Rel to Rel' so that for any $x \in \Sigma$, $\mu_{\mathcal{P}'}(P'(x)) = \mu_{\mathcal{P}}(P(x))$ is straightforward and is implemented at lines 18-20 of $SplittingProcedure()$.

6.5 *Refine* Function

SSA calls $Refine(\langle P, Rel\rangle, S)$ at line 7 with the precondition that S is a union of blocks of the current partition P. The function $Refine(\langle P, Rel\rangle, S)$ in Figure 6 implements the point (5) of Section 4.1. This function must modify the current PR $\mathcal{P} = \langle P, Rel\rangle$ to $\mathcal{P}' = \langle P, Rel'\rangle$ by pruning the relation Rel in such a way that for any $B \in P$:

$$\mu_{\mathcal{P}'}(B) = \begin{cases} \mu_{\mathcal{P}}(B) \cap S & \text{if } B \subseteq S \\ \mu_{\mathcal{P}}(B) & \text{if } B \cap S = \varnothing \end{cases}$$

This is done by the $Refine()$ function at lines 5-7 by reducing the relation Rel to Rel' as follows: if $B, C \in P$ and $Rel(B, C) = \mathbf{tt}$ then $Rel'(B, C) = \mathbf{ff}$ iff $B \subseteq S$ and $C \cap S = \varnothing$, while the rest of the code updates the data structures Count, BCount, localBottoms and bottomBlocks accordingly.

6.6 Auxiliary Functions

The implementation of the remaining functions $Initialize()$ and $Image()$ is easy and is omitted. It is just worth remarking that $Initialize()$ in particular initially establishes the property $\mathrm{TS}(\Sigma, \rightarrow, P_\ell)$ and provides an initial reverse topological order of P w.r.t. Rel when the input partial order Rel is not the identity relation id.

```
1  void Refine(PR ⟨P, Rel⟩, list⟨State⟩ S) {
2    list⟨Block⟩ L := ∅;
3    forall s ∈ S such that unmarked(s.block) do mark(s.block); L.append(s.block);
4    forall B ∈ L do
5      forall C ∈ P do
6        if (Rel(B, C) = tt & unmarked(C)) then
7          Rel(B, C) := ff;
8          forall y ∈ C do
9            forall x ∈ pre(y) do Count(x, B) − −; BCount(x.block, B) − −;
10         if (C ∈ B.bottomBlocks) then B.bottomBlocks.erase(C);
11         forall y ∈ C do
12           forall x ∈ pre(y) do
13             if (x.block ≠ B & Rel(B, x.block) = tt & Count(x, B) = 0)
               then
14               mark2(x.block);
15               if unmarked2(x.block) then
16                 B.bottomBlocks.append(x.block);
17             else if (x.block = B & Count(x, B) = 0) then
18               B.localBottoms.append(x);
19   forall B ∈ P do unmark(B); unmark2(B);
20 }
```

Fig. 6. *Refine* function.

6.7 Complexity

Time and space bounds for SSA are as follows. In the following statement we assume, as usual in model checking, that the transition relation $\rightarrow$ is total, i.e., for any $s \in \Sigma$ there exists $t \in \Sigma$ such that $s \rightarrow t$, so that the inequalities $|\Sigma| \leq |\rightarrow|$ and $|P_{\mathrm{stsim}}| \leq |\rightarrow^{\exists}|$ hold, where $\rightarrow^{\exists}$ is the existential transition relation between blocks of P_{stsim}, and this allows us to simplify the expression of the time bound.

Theorem 6.1 (Complexity). *SSA runs in $O(|P_{\mathrm{stsim}}|^2(|\rightarrow| + |P_{\mathrm{stsim}}||\rightarrow^{\exists}|))$-time and $O(|\Sigma||P_{\mathrm{stsim}}| \log |\Sigma|)$-space.*

6.8 Adapting SSA for LTSs

The algorithms SSA computes the stuttering simulation preorder on KSs, but it can be modified to work over LTSs by following the adaptation to LTSs of Groote and Vaandrager's algorithm [6] for KSs. Due to lack of space the details are here omitted. We just mention that we have a parametric $\mathbf{pos}_a$ operator for any action $a \in Act$ so that the notions of splitting and refinement of the current PR are parameterized w.r.t. the action a.

7 Conclusion

We presented an algorithm, called SSA, for computing the stuttering simulation preorder and equivalence on Kripke structures (or labeled transition systems). To the best of our knowledge, this is the first algorithm for computing this behavioural preorder. The only available algorithm related to stuttering simulation is a procedure by Bulychev et al. [2] that checks whether a given relation is a stuttering simulation. Our procedure SSA includes an algorithm for checking whether a given relation is a stuttering simulation that significantly improves Bulychev et al.'s one both in time and in space.

Acknowledgements. We are grateful to Silvia Crafa for numerous helpful discussions. This work was partially supported by the PRIN 2007 Project "*AIDA2007: Abstract Interpretation Design and Applications*" and by the University of Padova under the Projects "*Formal methods for specifying and verifying behavioural properties of software systems*" and "*Analysis, verification and abstract interpretation of models for concurrency*".

References

1. M.C. Browne, E.M. Clarke and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comp. Sci.*, 59:115-131, 1988.
2. P.E. Bulychev, I.V. Konnov and V.A. Zakharov. Computing (bi)simulation relations preserving CTL*-X for ordinary and fair Kripke structures. *Mathemathical Methods and Algorithms*, Institute for System Programming, Russian Academy of Sciences, vol. 12, 2007. Available from http://lvk.cs.msu.su/∼peterbul.
3. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd ed., 2001.
4. R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995
5. K. Etessami, T. Wilke, R.A. Schuller. Fair simulation relations, parity games, and state space reduction for Buchi automata. *SIAM J. Comput.*, 34(5):1159-1175, 2001.
6. 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.
7. M.R. Henzinger, T.A. Henzinger and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36th FOCS*, 453-462, 1995.
8. L. Lamport. What good is temporal logic? In *Information Processing '83*, pp. 657-668, IFIP, 1983.
9. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001.
10. S. Nejati, A. Gurfinkel and M. Chechik. Stuttering abstraction for model checking. In *3rd IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM'05)*, pp. 311-320, 2005
11. F. Ranzato and F. Tapparo. A new efficient simulation equivalence algorithm. In *Proc. 22nd IEEE Symp. on Logic in Computer Science (LICS'07)*, pp. 171–180, IEEE Press, 2007.
12. F. Ranzato and F. Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Information and Computation*, 206(5):620-651, 2008.