An algorithm for Deciding Bisimulation Equivalence on Finite-State Systems

2. Basic definitions

This section contains definitions of concepts and notations used in the remainder of the chapter.

2.1. *Labeled transition systems*

Semantically, systems are modeled as labeled transition systems, which may be defined as follows.

DEFINITION 2.1. A *labeled transition system* (LTS) is a triple $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.

Intuitively, an LTS $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ defines a computational framework, with $\mathcal{S}$ representing the set of states that systems may enter, $\mathcal{A}$ the actions systems may engage in, and $\rightarrow$ the execution steps system undergo as they perform actions. In what follows we generally write $s \xrightarrow{a} s'$ in lieu of $\langle s, a, s' \rangle \in \rightarrow$, and we say that s' is an a-derivative of s. We use $\xrightarrow{a}{}^*$ to denote the transitive closure of $\xrightarrow{a}$. We define a *process* to be a quadruple $\langle \mathcal{S}, \mathcal{A}, \rightarrow, s_I \rangle$ where $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is an LTS and $s_I \in \mathcal{S}$ is the start state.

Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS, and let $s \in \mathcal{S}$ be a state and $a \in \mathcal{A}$ an action. We use the following terminology and notations in what follows.

- $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is finite-state if $\mathcal{S}$ and $\mathcal{A}$ are both finite sets.
- $s \xrightarrow{a}$ holds if $s \xrightarrow{a} s'$ for some $s' \in \mathcal{S}$.
- $\{\bullet \xrightarrow{a} s\} \subseteq \mathcal{S}$, the *preset* of s with respect to a, is the set $\{r \in \mathcal{S} \mid r \xrightarrow{a} s\}$.
- $\{s \xrightarrow{a} \bullet\} \subseteq \mathcal{S}$, the *postset* of s with respect to a, is the set $\{t \in \mathcal{S} \mid s \xrightarrow{a} t\}$.
- $\{s \xrightarrow{\bullet}\} \subseteq \mathcal{A}$, the *initial actions* of s, is the set $\{a \in \mathcal{A} \mid s \xrightarrow{a}\}$.
- $\{s \rightarrow^* \bullet\} \subseteq \mathcal{S}$, the *reachable* set of states from s, is the smallest set satisfying the following:
  - $s \in \{s \rightarrow^* \bullet\}$.
  - If $t \in \{s \rightarrow^* \bullet\}$ and $t \xrightarrow{a} t'$ for some $a \in \mathcal{A}$ then $t' \in \{s \rightarrow^* \bullet\}$.

These notions may be lifted to sets of states by taking unions in the obvious manner. Thus if $S \subseteq \mathcal{S}$ then we have the following:

$$\{\bullet \xrightarrow{a} S\} = \bigcup_{s \in S} \{\bullet \xrightarrow{a} s\},$$

$$\{S \xrightarrow{a} \bullet\} = \bigcup_{s \in S} \{s \xrightarrow{a} \bullet\},$$

$$\{S \xrightarrow{\bullet}\} = \bigcup_{s \in S} \{s \xrightarrow{\bullet}\},$$

$$\{S \rightarrow^* \bullet\} = \bigcup_{s \in S} \{s \rightarrow^* \bullet\}.$$

The traditional approach to defining the semantics of process algebras involves constructing an LTS in the following manner. Firstly, the syntax of the algebra includes a set $\mathcal{A}$ of actions and a set $\mathcal{P}$ of process terms. Then a transition relation $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ is defined inductively in the SOS style using proof rules [25] (but also see [1] in this Handbook). The structure $\langle \mathcal{P}, \mathcal{A}, \rightarrow \rangle$ constitutes an LTS that in essence encodes all possible behavior of all processes. Of course, this LTS is not usually finite-state, so one may wonder how algorithms for finite-state systems could be used for determining if two process terms in a given algebra are semantically related. The answer lies in the fact that in general, one does not need to consider the entire LTS of the algebra; it typically suffices to consider only the terms reachable from the ones in question. If this reachable set is finite (and typically one may give syntactic characterizations of terms satisfying this property) then one may apply the algorithms presented in this chapter to the LTS induced by the finite set of reachable states. We return to this point later.

2.2. *Bisimulation equivalence*

Bisimulation equivalence is interesting in its own right as a basis for relating processes; it also may be seen as a basis for defining other relations as well. Bisimulation and other behavioral equivalences are treated in more detail in [16] in this Handbook.

DEFINITION 2.2 (*Bisimulation equivalence*). Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS.

- A relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a *bisimulation* if whenever $\langle s_1, s_2 \rangle \in R$ then the following hold for all $a \in \mathcal{A}$:
  1. If $s_1 \xrightarrow{a} s_1'$ then there is an s_2' such that $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in R$.
  2. If $s_2 \xrightarrow{a} s_2'$ then there is an s_1' such that $s_1 \xrightarrow{a} s_1'$ and $\langle s_1', s_2' \rangle \in R$.
- Two states $s_1, s_2 \in \mathcal{S}$ are *bisimulation equivalent*, written $s_1 \sim s_2$, if there exists a bisimulation R such that $\langle s_1, s_2 \rangle \in R$.

Intuitively, two states in an LTS are bisimulation equivalent if they can "simulate" each other's transitions. Under this interpretation a bisimulation indicates how transitions from related states may be matched in order to ensure that the "bi-simulation" property holds.

Bisimulation equivalence enjoys a number of mathematical properties. Firstly, it is indeed an equivalence relation in that it is reflexive, symmetric and transitive. Secondly, it is itself a bisimulation, and in fact is the largest bisimulation with respect to set containment.

3.1. *A basic partition-refinement algorithm for bisimulation equivalence*

The first partition refinement algorithm for bisimulation equivalence is due to Kanellakis and Smolka [21]. Let $P = \{B_1, \ldots, B_n\}$ be a partition consisting of a set of blocks. The

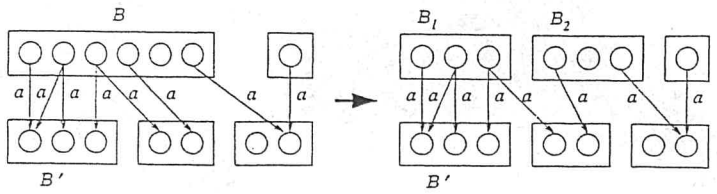

Fig. 1. Splitting a block in the partition.

```
split(B, a, P) → ({B_i} a set of blocks)
  choose s ∈ B
  {* B1 contains states equivalent to s *}
  B1 = ∅
  {* B2 contains states inequivalent to s *}
  B2 = ∅
  for each s' ∈ B do
  begin
    if [{s --a--> •}]P = [{s' --a--> •}]P
    then B1 = B1 ∪ {s'}
    else B2 = B2 ∪ {s'}
  end
  if B2 = ∅
  then return {B1}
  else return {B1, B2}
```

Fig. 2. The pseudo-code for procedure *split*.

algorithm is based around the notion of *splitting*. A *splitter* for a block $B \in P$ is the block $B' \in P$ such that some states in B have a-transitions, for some $a \in \mathcal{A}$, into B' and others do not. In this case, B can be split by B' with respect to a into blocks $B_1 = \{s \in B \mid \exists s' \in B'.s \xrightarrow{a} s'\}$, $B_2 = B - B_1$. Splitting is illustrated in Figure 1.

The algorithm uses splitting in the form of procedure $split(B, a, P)$, which detects whether the partition P contains a splitter for a given block $B \in P$ with respect to action $a \in \mathcal{A}$. If such splitter exists, *split* returns the blocks B_1 and B_2 that result from the split. Otherwise, B itself is returned. Efficient implementation of *split* is critical to the overall complexity of the algorithm. Therefore, we will discuss in more detail the implementation of *split* and the data structures necessary to make it efficient.

In presenting the procedure *split* we use the following notation: for a set of states S, $[S]_P = \{B \in P \mid \exists s \in S.s \in B\}$ is the minimal set of blocks in P that contain all states in S. Then, $[\{s \xrightarrow{a} \bullet\}]_P$ is the set of blocks that can be reached from s by an a-transition. We will abuse terminology and call this set the *postset of s in P with respect to a*. Figure 2 gives the pseudo-code for procedure *split*. The procedure chooses a state from B and compares its postset in P to the postsets in P of other states in B. Clearly, if the postsets of two states are different, then there exists a splitter that will put these states in different blocks.


```
P := {S}
changed := true
while changed do
begin
  changed := false
  for each B ∈ P do
  begin
    for each a ∈ A do
    begin
      SortTransitions(a, B)
      if split(B, a, P) ≠ {B}
      then begin
        P := P − {B} ∪ split(B, a, P)
        changed := true
        break
      end
    end
  end
end
```

Fig. 3. Algorithm *KS_PARTITIONING*.

In order to compare the postsets of the states of B efficiently, we need to order the transitions of s. For this purpose, we impose an ordering on the blocks of P. The transitions of s are lexicographically ordered by their labels. Further, for each label a, the transitions are ordered by the containing block of the target state of the transition. When a block is split, the ordering of transitions in states that have transitions into that block can be violated. Therefore, one needs to sort the a-transitions of all states of a block immediately before attempting to split the block. Procedure *SortTransitions*(a, B) uses lexicographic sorting to reorder the a-transitions of block B.

Finally, we present the main loop of *KS_PARTITIONING* in Figure 3. The algorithm iteratively attempts splitting of every block in P with respect to every $a \in A$ until no more blocks can be split.

Correctness of *KS_PARTITIONING* relies on the fact that when *changed* is *false*, there is no splitter for any of the blocks in P. Therefore, $P = \mathcal{F}_{\mathcal{L}}(P)$ and, by Theorem 2.4, $R \subseteq \sim$. Moreover, if we denote by P_i the partition after ith iteration of the main loop of *KS_PARTITIONING*, we have $\sim \subseteq \sim_i \subseteq P_i$. Thus we have that at termination of the algorithm, $P = \sim$.

The complexity of *KS_PARTITIONING* is given by the following theorem.

THEOREM 3.1. *Given a finite-state LTS* $\langle S, \mathcal{A}, \rightarrow \rangle$ *with* $|S| = n$ *and* $|\rightarrow| = m$, *algorithm KS_PARTITIONING takes* $O(n \cdot m)$ *time.*

PROOF. The main loop of the algorithm is repeated at most n times. Within one iteration of the main loop, procedure *split* is called for each block at most once for each action a.

In turn, *split* considers each transition of every state in the block at most once. Therefore, the calls to *split* within one iteration of the main loop take $O(m)$ time. The calls to *SortTransitions* collectively take $O(|\mathcal{A}| + m)$ time, or $O(m)$ when the set of labels is bounded by a constant. □