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 \((S, A, \rightarrow)\), where \(S\) is a set of states, \(A\) is a set of actions, and \(\rightarrow \subseteq S \times A \times S\) is the transition relation.

Intuitively, an LTS \((S, A, \rightarrow)\) defines a computational framework, with \(S\) representing the set of states that systems may enter, \(A\) the actions systems may engage in, and \(\rightarrow\) the execution steps systems undergo as they perform actions. In what follows we generally write \(s \xrightarrow{a} s'\) in lieu of \((s, a, s') \in \rightarrow\), and we say that \(s'\) is an \(a\)-derivative of \(s\). We use \(\rightarrow^*, \rightarrow^*\) to denote the transitive closure of \(\rightarrow\). We define a **process** to be a quadruple \((S, A, \rightarrow, s_1)\) where \((S, A, \rightarrow)\) is an LTS and \(s_1 \in S\) is the start state.

Let \((S, A, \rightarrow)\) be an LTS, and let \(s \in S\) be a state and \(a \in A\) an action. We use the following terminology and notations in what follows.
- \(s \xrightarrow{a} s'\) holds if \(s' = s \xrightarrow{a} s'\) for some \(s' \in S\).
- \(s \xrightarrow{\alpha} s\) is the preset of \(s\) with respect to \(\alpha\), is the set \(\{r \in S \mid r \xrightarrow{a} s\}\).
- \(s \xrightarrow{\alpha} s\) is the postset of \(s\) with respect to \(\alpha\), is the set \(\{r \in S \mid s \xrightarrow{a} r\}\).
- \(s \xrightarrow{\alpha} S\) is the initial actions of \(s\), is the set \(\{a \in A \mid s \xrightarrow{a} s\}\).
- \(s \xrightarrow{\alpha} S\) is the reachable set of states from \(s\), is the smallest set satisfying the following:
  - \(s \in S\).
  - If \(a \in (s \rightarrow \star)\) and \(r \xrightarrow{a} r'\) for some \(a \in A\) then \(r' \in (s \rightarrow \star)\).

These notions may be lifted to sets of states by taking unions in the obvious manner. Thus if \(S' \subseteq S\) then we have the following:

\[
\begin{align*}
(S \xrightarrow{\alpha} S) &= \bigcup_{s \in S} (s \xrightarrow{\alpha} s), \\
(S \xrightarrow{\alpha} S) &= \bigcup_{s \in S} (s \xrightarrow{\alpha} \star).
\end{align*}
\]

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

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 Kandallakis 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.

\[
\text{split}(B, a, P) \rightarrow ([B_1] \text{ a set of blocks})
\]

\[
\begin{align*}
\text{choose } s & \in B \\
\text{[} s \& B_1 \text{ contains states equivalent to } s \& B \text{]} \\
B_1 & = B \\
\text{[} s \& B_2 \text{ contains states inequivalent to } s \& B \text{]} \\
B_2 & = \emptyset \\
\text{for each } s' \in B & \text{ do} \\
\begin{align*}
\text{if } & [(s \xrightarrow{a} s')]_P = [(s' \xrightarrow{a} s')]_P \\
& \text{then } B_1 = B_1 \cup \{s'\} \\
& \text{else } B_2 = B_2 \cup \{s'\} \\
\end{align*} \\
\text{end} \\
\text{if } B_2 = \emptyset & \text{ then return } B_1 \\
\text{else return } & (B_1, B_2)
\end{align*}
\]

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

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

In presenting the procedure \(\text{split}\) 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, \([s \xrightarrow{a} s']_P\) 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 \(\text{split}\). 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.

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 transitions of all blocks of a block immediately before attempting to split the block. Procedure \(\text{SortTransitions}(a, B)\) uses lexicographic sorting to reorder the \(a\)-transitions of block \(B\).

Finally, we present the main loop of \(\text{KS_PARTITIONING}\) 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 \(\text{KS_PARTITIONING}\) relies on the fact that when \(\text{changed} = \text{false}\), there is no splitter for any of the blocks in \(P\). Therefore, \(P = \mathcal{F}_L(P)\) and, by Theorem 2.4, \(R \leq \sim\). Moreover, if we denote by \(P_i\) the partition after \(i\)th iteration of the main loop of \(\text{KS_PARTITIONING}\), we have \(\sim \leq \sim_{i-1} \leq P_i\). Thus we have that at termination of the algorithm, \(P = \sim\).

The complexity of \(\text{KS_PARTITIONING}\) is given by the following theorem.

**Theorem 3.1.** Given a finite-state LTS \((S, A, \rightarrow)\) with \(|S| = n\) and \(|\rightarrow| = m\), algorithm \(\text{KS_PARTITIONING}\) 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 \(\text{split}\) is called for each block at most once for each action \(a\).

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