Equivalence and Preorder Checking for Finite-State Systems

Rance Cleaveland
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
USA
rance@cs.sunysb.edu

Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
USA
sokolsky@saul.cis.upenn.edu

Abstract

This chapter surveys algorithms for computing semantic equivalences and refinement relations, or *preorders*, over states in finite-state labeled transitions systems. Methods for calculating a general equivalence, namely bisimulation equivalence, and a general preorder are described and shown to be useful as a basis for calculating other semantic relations as well. Two general classes of algorithms are considered: global ones, which require the *a priori* construction of the state space but are generally more efficient in the asymptotic case, and local, or on-the-fly ones, which avoid the construction of unnecessary states while incurring some additional computational overhead.

Keywords: process algebra, finite-state systems, labeled transition systems, bisimulation, equivalence checking, preorder checking.

Contents

1 Introduction

Research in process algebra has focused on the use of behavioral relations such as equivalences and refinement orderings as a basis for establishing system correctness (see Chapter 1.1 of this volume and [BBK86, BB87, Hoa85, Mil80, Mil89]). In the process-algebraic framework specifications and implementations are both given as terms in the same algebra; the intuition is that a specification describes the desired high-level behavior the system should exhibit, while the implementation details the proposed means for achieving this behavior. One then uses an appropriate equivalence or preorder to establish that the implementation conforms to the specification. In the case of equivalence-based reasoning, conformance means "has the same behavior as"; in this case an implementation is correct if its behavior is indistinguishable from that of the specification. Refinement (or preorder) relations, on the other hand, typically embody a notion of "better than": an implementation conforms to (or refines) a specification if the behavior of the former is "at least as good as" that stipulated by the specification. The benefits of such process-algebraic approaches include the following.

- Users need only learn one language in order to write specifications and implementations.
- The algebra provides explicit support for *compositional* specifications and implementations, allowing the specification (implementation) of a system to be built up from the specifications (implementations) of its components.
- Specifications include information about what is disallowed as well as what is allowed.

Consequently, a number of different process algebras have been studied, and a variety of different equivalences and refinement relations capturing different aspects of behavior have been developed.

A hallmark of process-oriented behavioral relations is that they are usually defined with respect to *labeled transition systems*, which form a semantic model of systems, rather than with respect to a particular syntax of process descriptions. This style of definition permits notions of equivalence and refinement to be applied to any algebra with a semantics given in terms of labeled transition systems. It also means techniques for establishing these relations may be given in terms of labeled transition systems. If, in addition, these labeled transition systems are finitary, then the relations may be calculated in a purely mechanical manner: tools may then be developed for automatically checking that (finitary) implementations conform to (finitary) specifications.

This paper surveys algorithms for calculating behavioral relations for a particular class of finitary labeled transition systems, namely, those consisting of a finite number of states and transition labels. We focus on relations that are sensitive only to the degree of nondeterminism systems may exhibit; we do not consider relations sensitive to other aspects of system behavior such as timing, probability, priority, or parallelism. See Chapters 4.1-4.3 and 5.1 of this volume, respectively, for a treatment of these features. This decision is due to the fact that models of nondeterminism have a longer history and hence are more settled, and to the fact that techniques for computing these relations find direct application in the computation of the others.

The remainder of this chapter has the following structure. The next section introduces the basic definitions and notations used throughout the remainder of the survey. The discussion then breaks into two major parts. Section 3 presents algorithms for calculating behavioral equivalences using *partition refinement* and *symbolic* techniques. Section 4 considers an algorithm for behavioral preorders. Section 5 introduces *local*, or *on-the-fly* techniques. Local algorithms aim to reduce the amount of work required by avoiding the *a priori* construction of the entire state space. Section 6 presents some of the tools that implement the algorithms described in this chapter, while Section 7 concludes.

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 S representing the set of states that systems may enter, 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 $s_I \in S$ 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 S and 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.

$$\begin{aligned}
\{\bullet \xrightarrow{a} S\} &= \bigcup_{s \in S} \{\bullet \xrightarrow{a} s\} \\
\{S \xrightarrow{a} \bullet\} &= \bigcup_{s \in S} \{s \xrightarrow{a} \bullet\} \\
\{S \xrightarrow{\bullet}\} &= \bigcup_{s \in S} \{s \xrightarrow{\bullet}\} \\
\{S \rightarrow^* \bullet\} &= \bigcup_{s \in S} \{s \rightarrow^* \bullet\}
\end{aligned}$$

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 [Plo81] (but also see Chapter 1.3 of 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 Chapter 1.1 of 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. Finally, it may be seen as the greatest fixpoint of the following function mapping relations to relations.

Definition 2.3 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS. Then $\mathcal{F}_{\mathcal{L}} : 2^{\mathcal{S}\times\mathcal{S}} \longrightarrow 2^{\mathcal{S}\times\mathcal{S}}$ is given by:*

$$\begin{aligned}\mathcal{F}_{\mathcal{L}}(R) = \{\, \langle s_1, s_2 \rangle \mid \forall a \in \mathcal{A}. \forall s' \in \mathcal{S}.\ \ & (s_1 \xrightarrow{a} s' \Rightarrow \exists t' \in \mathcal{S}. s_2 \xrightarrow{a} t' \wedge \langle s', t' \rangle \in R) \ \wedge \\ & s_2 \xrightarrow{a} s' \Rightarrow \exists r' \in \mathcal{S}. s_1 \xrightarrow{a} r' \wedge \langle r', s' \rangle \in R) \,\}\end{aligned}$$

Theorem 2.4 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS. Then $\sim = \mathcal{F}_{\mathcal{L}}(\sim)$, and for any R such that $R = \mathcal{F}_{\mathcal{L}}(R)$, $R \subseteq \sim$.*

The proof of this theorem relies on the Tarski-Knaster fixpoint theorem, which gives characterizations of the fixpoints of monotonic functions over complete lattices. In this case, the complete lattice in question is the set $2^{\mathcal{S}\times\mathcal{S}}$ of binary relations ordered by set inclusion; in this lattice it is easy to see that $\mathcal{F}_{\mathcal{L}}$ is monotonic (the more pairs there are in R, the more pairs there are in $F_{\mathcal{L}}(R)$), and hence the Tarski-Knaster theorem is applicable.

The characterization of bisimulation equivalence in Theorem 2.4 suggests that in order to compute the relation over a given LTS $\mathcal{L}$, it suffices to calculate the greatest fixpoint of $\mathcal{F}_{\mathcal{L}}$. The next result suggests how this might be done for finite-state LTSs.

Definition 2.5 *An LTS $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is* image-finite *if for every $s \in \mathcal{S}$ and $a \in \mathcal{A}$, the set $\{s \xrightarrow{a} \bullet\}$ is finite.*

In other words, an LTS is image-finite if every state has a finite number of outgoing transitions for any given action. Certainly, any finite-state LTS is also image-finite.

Now, when an LTS $\mathcal{L}$ is image-finite, the function $\mathcal{F}_{\mathcal{L}}$ turns out to be continuous, and its greatest fixpoint (i.e. $\sim$) has the following "iterative" characterization.

Theorem 2.6 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an image-finite LTS. Then $\sim = \bigcap_{i=0}^{\infty} \sim_i$, where the $\sim_i$ are defined as follows.*

$$\begin{aligned}\sim_0 &= \mathcal{S} \times \mathcal{S} \\ \sim_{i+1} &= \mathcal{F}_{\mathcal{L}}(\sim_i) \ \text{ for } i \geq 0\end{aligned}$$

This characterization provides the basis for the algorithms discussed later in the chapter.

Parameterized Bisimulation Equivalence. We will show that other semantic equivalences may be computed by combining appropriate transformations on labeled transition systems with a bisimulation algorithm. For this purpose, it turns out that a slight modification of the definition of bisimulation equivalence is useful.

Definition 2.7 *Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS, and let $E \subseteq \mathcal{S} \times \mathcal{S}$ be an equivalence relation.*

1. *A relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is an E-*bisimulation *if R is a bisimulation and $R \subseteq E$.*

2. *Two states $s_1, s_2 \in S$ are E-*bisimulation equivalent*, written $s_1 \sim^E s_2$, if there is an E bisimulation R with $\langle s_1, s_2 \rangle \in R$.*

The relation $\sim^E$ differs from $\sim$ in that it requires equivalent states to be related by E in addition to satisfying the transition conditions imposed by $\sim$. It is easy to show that $\sim$ coincides with $\sim^U$, where U is the universal relation relating every state to every other state. Theorems 2.4 and 2.6 have obvious analogs for $\sim^E$.

2.3 Simulation-Based Refinement Orderings

Refinement orderings relate states in an LTS on the basis of the relative "quality" of their behavior. The (forward) simulation ordering represents one such notion that plays an important algorithmic role. Its definition is as follows.

Definition 2.8 *Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS.*

- *A relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* forward simulation *if whenever $\langle s_1, s_2 \rangle \in R$ then the following holds for all $a \in \mathcal{A}$.*

  If $s_1 \xrightarrow{a} s_2$ then there is an s_2' such that $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in R$.

- *$s_1 \precsim s_2$ if there is a forward simulation R such that $\langle s_1, s_2 \rangle \in R$.*

Note that $s_1 \precsim s_2$ holds when s_2 is able to track, or "simulate", the transitions that s_1 is capable of. The relation $\precsim$ turns out to be a preorder (i.e. reflexive and transitive), and like bisimulation it turns out to be the greatest fixpoint of an appropriately defined function over relations.

Definition 2.9 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS. Then $\mathcal{G}_{\mathcal{L}} : 2^{\mathcal{S} \times \mathcal{S}} \longrightarrow 2^{\mathcal{S} \times \mathcal{S}}$ is defined as follows.*

$$\mathcal{G}_{\mathcal{L}}(R) = \{ \langle s_1, s_2 \rangle \mid \forall a \in \mathcal{A}. \forall s' \in \mathcal{S}. \quad (s_1 \xrightarrow{a} s' \Rightarrow \exists t'. s_2 \xrightarrow{a} t' \wedge \langle s', t' \rangle \in R) \}$$

That $\precsim$ is the greatest fixpoint of $\mathcal{G}_{\mathcal{L}}$ follows from the same line of reasoning offered for Theorem 2.4. In addition, one may give the following analog of Theorem 2.6, which provides a basis for algorithms presented later in the chapter.

Theorem 2.10 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an image-finite LTS. Then $\precsim = \bigcap_{i=0} \precsim_i$, where the $\precsim_i$ are defined as follows.*

$$\begin{aligned} \precsim_0 &= \mathcal{S} \times \mathcal{S} \\ \precsim_{i+1} &= \mathcal{G}_{\mathcal{L}}(\precsim_i) \text{ for } i \geq 0 \end{aligned}$$

The backward simulation ordering $\succsim$ is the inverse of $\precsim$: $s_1 \succsim s_2$ if and only if $s_2 \precsim s_1$.

2.4 A Parameterized Semantic Relation

We close this section with the definition of a parameterized semantic relation that proves useful as a basis for calculating other simulation and bisimulation relations.

Definition 2.11 *Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS, let $\Pi \subseteq \mathcal{S} \times \mathcal{S}$ be a relation on states, and let $\Phi_1, \Phi_2 \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{A}$ be relations on states and actions.*

- *A relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* (Π, Φ_1, Φ_2)-bisimulation *if, whenever $\langle s_1, s_2 \rangle \in R$, then the following hold for all $a \in \mathcal{A}$.*

  1. *$\langle s_1, s_2 \rangle \in \Pi$*
  2. *If $\langle s_1, s_2, a \rangle \in \Phi_1$ and $s_1 \xrightarrow{a} s_1'$ then there is a s_2' such that $s_2 \xrightarrow{a} s_2'$ and $\langle s_1', s_2' \rangle \in R$.*
  3. *If $\langle s_1, s_2, a \rangle \in \Phi_2$ and $s_2 \xrightarrow{a} s_2'$ then there is a 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 are* (Π, Φ_1, Φ_2)-bisimilar *if there is a (Π, Φ_1, Φ_2)-bisimulation R with $\langle s_1, s_2 \rangle \in R$.*

The definition of (Π, Φ_1, Φ_2)-bisimilarity is somewhat complicated. Intuitively, the three parameters have the following roles.

- Π serves as an "initial condition" that related states must satisfy.
- Φ_1 and Φ_2 constrain when related states are required to track each others transitions.

We denote (Π, Φ_1, Φ_2)-bisimulation as $\sqsubseteq^{\Pi}_{\Phi_1, \Phi_2}$, , using the preorder symbol since in general, when $\Phi_1 \neq \Phi_2$, it is not an equivalence relation.

For example, if we let U be the (overloaded) symbol for the universal relation, then $\sim$ coincides with (U, U, U)-bisimilarity, while $\sim^E$ is (E, U, U)-bisimilarity. Similarly, $\sqsubseteq$ turns out to be $(U, U, \emptyset)$-bisimilarity, and $\sqsupseteq$ $(U, \emptyset, U)$-bisimilarity.

3 Global Equivalence Algorithms

In this section, we present several equivalence-checking algorithms which, given a finite-state LTS $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ and an equivalence relation E, iteratively compute the parameterized bisimulation equivalence $\sim^E$. Then, the equivalence of two given states can then be tested by checking whether they belong to the same equivalence class. The equivalence relation is represented by the set of equivalence classes, or a *partition* of the states in the LTS. A partition is a set of *blocks* $\{B_i \subseteq \mathcal{S} \mid B_i \cap B_j = \emptyset, i \neq j \;\wedge \bigcup_i B_i = \mathcal{S}\}_{i \in I}$. The algorithms iteratively compute the fixed point of $\mathcal{F}_{\mathcal{L}}$ as stipulated by Theorem 2.6. The starting point for the iteration is the partition induced by E. To simplify exposition, the algorithms below are presented for the important special case when $E = U$. In this case, the equivalence relation computed is bisimulation equivalence $\sim$, and the initial partition is $\{\mathcal{S}\}$.

The algorithms discussed in this section differ in the way the partition is represented, and in the data structures used in the fixed-point computation. We first concentrate on the algorithms that represent the partition explicitly, as a set of sets of individual states. Such algorithms are commonly known as *partition refinement* algorithms. Section 3.3 describes a symbolic approach based on boolean functions.

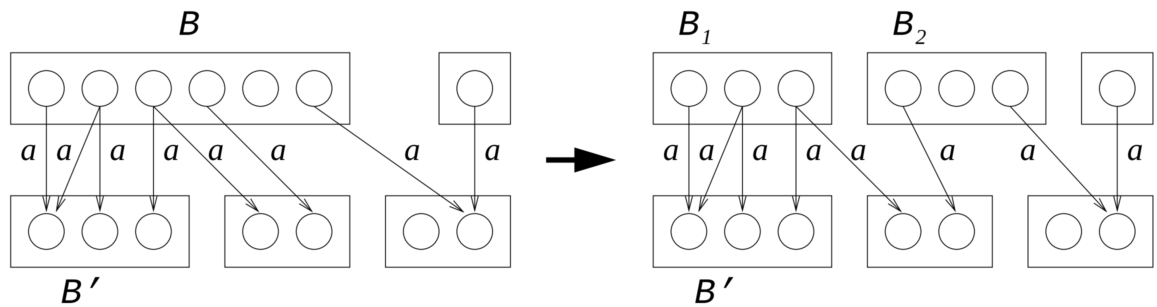

Figure 1: Splitting a block in the partition.

3.1 A Basic Partition-Refinement Algorithm for Bisimulation Equivalence

The first partition refinement algorithm for bisimulation equivalence is due to Kanellakis and Smolka [KS90]. Let $P = \{B_1, \ldots, B_n\}$ be a partition consisting of a set of blocks. 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 $a \in \mathcal{A}$, 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.

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

```
split(B, a, P) → ({B_i} a set of blocks)
   choose s ∈ B
   {* B_1 contains states equivalent to s *}
   B_1 = ∅
   {* B_2 contains states inequivalent to s *}
   B_2 = ∅
   for each s' ∈ B do
   begin
      if [{s -a-> •}]_P = [{s' -a-> •}]_P
      then B_1 = B_1 ∪ {s'}
      else B_2 = B_2 ∪ {s'}
   end
   if B_2 = ∅
   then return {B_1}
   else return {B_1, B_2}
```

Figure 2: The pseudo-code for procedure *split*.

```
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
```

Figure 3: Algorithm *KS_PARTITIONING*.

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* $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ *with* $|\mathcal{S}| = n$ *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. □

3.2 The Paige-Tarjan Algorithm for Bisimulation Equivalence

Performance of the basic partition refinement algorithm can be significantly improved through the use of more complex data structures. Paige and Tarjan [PT87] proposed an algorithm that utilizes information about previous splits to make future splits more efficient. To simplify the presentation of the algorithm, we describe the case where the alphabet of the LTS is a singleton. An extension of the algorithm to handle multiple actions is straightforward. First, we introduce the notion of *stability* of blocks and partitions.

Definition 3.2

- *A block* B *is stable with respect to a block* S *if either* $B \subseteq \{\bullet \rightarrow S\}$ *or* $B \cup \{\bullet \rightarrow S\} = \emptyset$.
- *A partition* P *is stable with respect to a block* S *if every* $B \in P$ *is stable with respect to* S.
- *A partition* P *is stable with respect to partition* Q *if* P *is stable with respect to every* $S \in Q$.
- *A partition is stable if it is stable with respect to itself.*

Clearly, $\sim$ is the coarsest stable partition.

The Paige-Tarjan algorithm is based on the following observation. Let B be stable with respect to S, and let S be partitioned into S_1 and S_2. Then, if $B \cup S = \emptyset$, B is stable with respect to both S_1 and S_2. Otherwise, B can be split into *three* blocks:

$$\begin{array}{lcl} B_1 & = & B - \{\bullet \rightarrow S_2\} \\ B_{12} & = & B \cap \{\bullet \rightarrow S_1\} \cap \{\bullet \rightarrow S_2\} \\ B_2 & = & B - \{\bullet \rightarrow S_1\} \end{array}$$

This three-way splitting is illustrated in Figure 4.

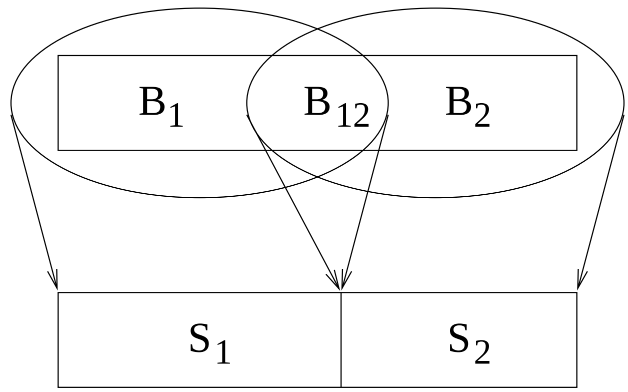

Figure 4: Efficient splitting of a block.

The improvement in complexity that the Paige-Tarjan algorithm provides over the basic partition refinement algorithm of the previous section stems from the fact that three-way splitting can be performed in time proportional to the size of the smaller of the two blocks S_1, S_2. To do so, with every state $s \in \mathcal{S}$ and for every block S, we keep a variable $count(s, S) = |\{s' \in S \mid s \stackrel{s'}{\rightarrow}\}|$. Now, we can decide in constant time to which of the sub-blocks of B a state $s \in B$ belongs. We have three cases: 1) if $count(s, S_1) = count(s, S)$, then $s \in B_1$; 2) if $0 < count(s, S_1) < count(s, S)$, then $s \in B_{12}$; 3) if $count(s, S_1) = 0$, then $s \in B_2$.

In addition to maintaining the *count* variables, we have to store the information about the history of prior splits. If a block has been used as a splitter, it is stable with respect to a partition. If such a block has been split itself, the smaller of its sub-blocks can be used in three-way splitting. We use an additional data structure to store split history: X is a forest of binary trees. Each node in a tree is a block satisfying the following conditions:

- For each leaf block B, $B \in P$.
- Each non-leaf block is the union of its children.
- For each root block B, P is stable with respect to B.

Roots of the trees are used as splitters for the current partition until X is empty. A splitter is called *compound* if it is the root of a non-trivial tree; otherwise it is called *simple*. For a compound splitter S, $compound(S)$ is the set of the non-trivial subtrees rooted at the children of S.

The algorithm, called *PT_PARTITIONING*, is shown in Figure 5. It repeatedly chooses a splitter Sp from X. A compound splitter is chosen if one exists. Then, every block $B \in P$ that has transitions into Sp is split with respect to the chosen splitter. We use the notation $B \in \{\bullet \rightarrow Sp\}$ to mean $\{B \in P \mid \exists s \in B. s \rightarrow s' \wedge s' \in Sp\}$. The operation is denoted $split(B, Sp)$. The result is a set of blocks P' containing one, two, or three blocks, depending on how B was split. By $S_{P'}$ we denote P' represented as a tree. The shape of $S_{P'}$ is given in Figure 6.

```
P := {𝒮}
X := P
while X ≠ ∅ do
begin
   choose S ∈ X
   X := X − S
   {* choose splitter *}
   if S is compound then
   begin
      Sp := the smaller of the children of S
      {* compound children of S stay in X *}
      X := X ∪ compound(S)
   end
   else
      Sp := S
   for each B ∈ {• → Sp} do
   begin
      P′ = split(B, Sp)
      updateX(B, P′)
   end end
```

Figure 5: Algorithm *PT_PARTITIONING*.

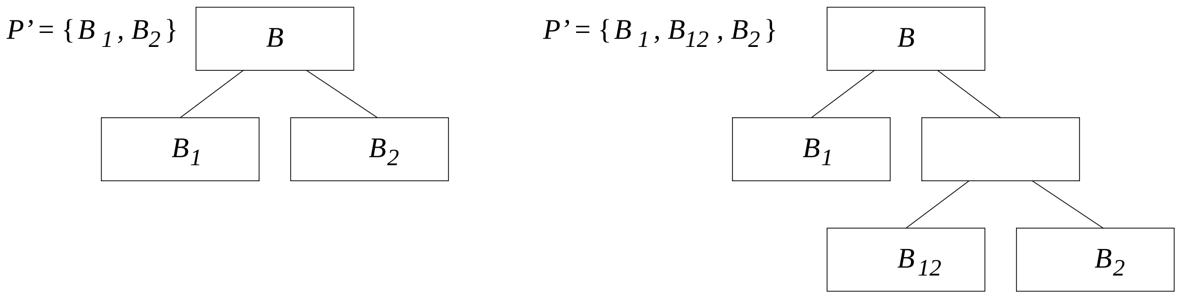

Figure 6: Splitting a block.

Operation $split(B, Sp)$ also updates the variables $count(s, S)$ for all $s \in B$. To do this, *split* scans the set $\{\bullet \to S\}$ twice. On the first pass, counts are computed, and splitting itself is done on the second pass.

When P' is a singleton, no splitting has occurred and no further work is necessary. Otherwise, X needs to be updated to reflect the splitting of B. Three cases are possible:

1. B is a simple splitter in X. In this case, B is removed from X and all elements of P' are added to X as simple splitters.

2. B is a leaf in one of the trees in X. The tree is extended at B with $S_{P'}$.

3. B does not appear in any tree in X. $S_{P'}$ is added to X as a new tree.

The three cases are illustrated in Figure 7. We denote by $updateX(B, P')$ the operation of updating X as just described.

Correctness of *PT_PARTITIONING* is established by means of the following invariant. For $B \in P$, if B is not a simple splitter in X and B is not a leaf of a compound splitter in X, then P is stable with respect to B. Indeed, every newly split block is placed in one of the trees in X, and it remains there until it is used in splitting. At every iteration of the outer loop of the algorithm, at most two blocks are removed from X: the one used as a splitter in that iteration, and the sibling of the splitter, if one exists. After the iteration is complete, P is stable with respect to both blocks. Subsequent refinements do not alter stability with respect to a block: if P is stable with respect to a block B and P' is a refinement of P, then P' is stable with respect to B. Therefore, when X is empty, P is stable with respect to itself.

The worst-case running time of *PT_PARTITIONING* is $O(m \cdot \log_2 n)$. For each state in a splitter Sp, the algorithm visits every incoming edge twice, performing $O(1)$ work for each edge. Because the smaller of the two sub-nodes of a compound splitter is used, each state can appear in at most $\log_2 n + 1$ splitters.

3.3 OBDD-Based Equivalence Checking

A major drawback of the algorithms presented in the preceding sections is that they require the LTS to be fully constructed in advance. When a process term is constructed from n subprocesses by parallel composition, the size of the resulting LTS may grow exponentially large in n, rendering equivalence checking infeasible. Several approaches have been proposed

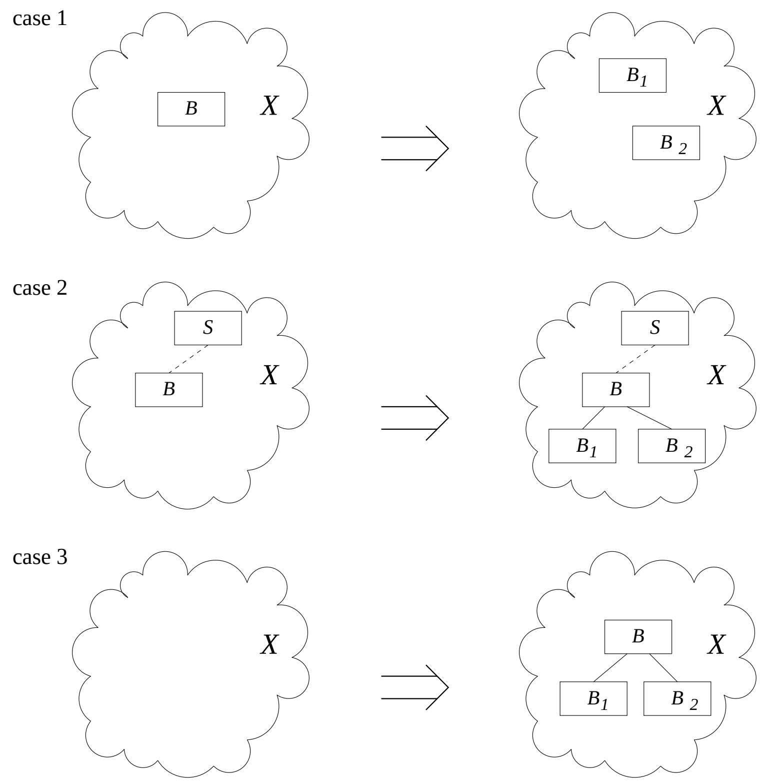

Figure 7: Updating the splitters.

to alleviate this problem. A group of algorithms called "local," which construct only as much of the LTS as needed to determine equivalence (or inequivalence), will be introduced later in this chapter. This section is concerned with a different approach, one that uses *ordered binary decision diagrams* (OBDDs) to succinctly represent the LTS. OBDDs [Bry86] are widely used in symbolic analysis algorithms such as model checking. We present an OBDD-based bisimulation checking algorithm based on the one by Bouali and de Simone [BdS92].

An OBDD is a representation of a boolean function by a rooted acyclic directed graph with respect to a fixed variable ordering. Terminal nodes of the graph are labeled with boolean constants, non-terminal nodes are labeled with input variables. Each non-terminal node has two outgoing edges for the two possible values of the variable labeling the node, indicating which node is evaluated next. Every path from the root of the OBDD to a terminal node has to respect the ordering of variables. In addition, an OBDD does not contain duplicate terminals or non-terminals, nor redundant tests (i.e., nodes with both outgoing edges leading to the same node). Figure 8 shows the OBDD for the function $(x_1 \vee x_2) \wedge x_3$ with the ordering $x_1 < x_2 < x_3$.

For a fixed variable ordering, OBDDs offer a canonical representation of boolean functions; that is, two OBDDs representing the same function are isomorphic. Commonly used operations on boolean functions, such as boolean operations, functional composition, and variable quantification can be computed with OBDDs in time polynomial in the size of the OBDD representations for the component functions. The size of the OBDD for a boolean function depends on the chosen variable ordering. In the worst case, the size is exponential

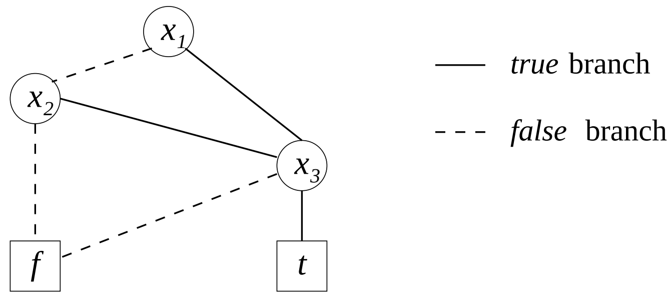

Figure 8: An example OBDD.

in the number of variables. Still, many functions have much more compact representations. For a detailed discussion of OBDD properties we refer the reader to [Bry92].

The OBDD representation of an LTS. An LTS $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is represented by OBDDs in the following way. The states of the LTS are represented by an encoding of their enumeration, that is, by means of a function $\sigma : \mathcal{S} \longrightarrow \{0,1\}^k$ that associates with each state a distinct boolean vector. Thus, representation of n states in an LTS requires $k = log_2 n$ boolean variables. We let $\vec{s}$ represent the encoding of a state s. The alphabet of the LTS is encoded in the same way, and $\vec{a}$ is the encoding of a label a. $States(x)$ is the characteristic function of the set of states in the LTS. The transition relation of an LTS is represented by a characteristic function $\delta(\vec{a}, \vec{s}, \vec{s'})$, which returns $true$ when $s \xrightarrow{a} s'$. Since all boolean functions considered in this section operate on encodings of state, labels, etc., and not on the objects themselves, we will use x for $\vec{(x)}$ where no confusion can arise.

Symbolic computation of the bisimulation equivalence. Like the algorithms presented previously, the algorithm for symbolic bisimulation checking is based on the characterization of Theorem 2.6. In order to compute the iterative fixed point symbolically, we need to express the function $\mathcal{F}_{\mathcal{L}}$ in terms of efficient OBDD operations.

First, we introduce several auxiliary operations on boolean functions. A *restriction* of function f with respect to variable x is $f|_{x \leftarrow k}(x_1, \ldots, x, \ldots, x_n) = f(x_1, \ldots, k, \ldots, x_n)$. The *smoothing* operator is defined by $S_x(f) = f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1}$. We have $S_x(f) = \exists x\ f$. The smoothing operator is extended to sets of variables in the obvious way. *Substitution* of a vector $X = \{x_1, \ldots, x_n\}$ by a vector $Y = \{y_1, \ldots, y_n\}$ within the OBDD for the function f is the simultaneous replacement of variables from X by respective variables from Y. Substitution is defined as

$$[Y \leftarrow X]f = S_X(\bigvee_{i \in \{1,\ldots,n\}} (y_i \leftrightarrow x_i) \wedge f)$$

.

With these definitions, we can give the definition of $\mathcal{F}_{\mathcal{L}}(R)$ suitable for symbolic computation. We represent R as the characteristic function of the cartesian product of the equivalence classes of R, denoted $R(x, y)$. $R(x, y)$ returns $true$ if x and y belong to the same equivalence class.

$$
\begin{array}{l}
R(x,y) := States(x) \wedge States(y) \\
New(x,y) := R(x,y) \\
\texttt{while } New(x,y) \neq \mathit{false} \texttt{ do} \\
\texttt{begin} \\
\quad R^{+}(x,y) := \mathcal{F}_{\mathcal{L}}(R) \\
\quad New(x,y) := R^{+}(x,y) \wedge \overline{R(x,y)} \\
\quad R(x,y) := R^{+}(x,y) \\
\texttt{end}
\end{array}
$$

Figure 9: Symbolic computation of bisimulation.

An auxiliary function

$$E_a(x,z) = S_y(R(x,y) \wedge \delta(a,z,y))$$

represents the relationship "z has an a-transition into the equivalence class of x." With this function, we can express the relationship that two states cannot be in the same equivalence class:

$$Bad(x,y) = [x \leftarrow y](\bigvee_{a \in \mathcal{A}} S_x([y \leftarrow z]E_a(x,z) \leftrightarrow \overline{E_a(x,z)})).$$

Finally, we have

$$\mathcal{F}_{\mathcal{L}}(R) = R(x,y) \wedge \overline{Bad(x,y)}.$$

Note that the use of substitutions in the definition of $Bad(x,y)$ allows us to construct E_a once per the application of $\mathcal{F}_{\mathcal{L}}$. Construction of E_a, which computes the inverse of the transition relation of the LTS, is a much more expensive operation than substitution.

The algorithm to compute the bisimulation equivalence symbolically, shown in Figure 9, is now a straightforward iterative fixed point computation that applies $\mathcal{F}_{\mathcal{L}}$ to the current partition in each iteration. Initially, $R = \mathcal{S} \times \mathcal{S}$ or, in a functional representation, $R(x,y) = States(x) \wedge States(y)$.

Performance issues. The time and space efficiency of the OBDD-based algorithm depend on the OBDD variable ordering and certain details of the LTS encoding. Significant performance improvements can be achieved in the case when the LTS under consideration is obtained as a product of several communicating processes. Then, in addition to representing and manipulating the global LTS, we need to represent the LTSs of the components, which we refer to as *local* LTSs, and compute the global LTS from the local LTSs symbolically.

States in the global LTS are represented as tuples of local states. There are two natural ways to order the variables used in the representation of local states. One ordering groups together all variables representing a state in a local LTS and uses the order of local states in the tuple to separate the states from different local LTSs. The other ordering groups local states together bit by bit. That is, the ordering places the first bit of the encodings of all local states before the second bit, etc. Experimental results presented in [BdS92] suggest that the former ordering yields more compact representation during the initial stages of the algorithm, when the equivalence relation is coarse. When there are many equivalence classes, the latter ordering is a better choice. Dynamic reordering of variables may allow one to take advantage of both orderings.

In addition, the order of local states within global state tuples has its effect on the size of OBDDs. As a rule of thumb, it is advantageous to group together the processes that actively communicate with each other. For example, let the global process be composed of three local processes, p_1, p_2, p_3. Assume that p_1 communicates with both p_2 and p_3, but there is little or no communication between p_2 and p_3. Then, the optimal order of local processes is to place p_1 in the middle, for example $\langle p_2, p_1, p_3 \rangle$.

3.4 Computing Other Equivalences via Process Transformations

The algorithms presented above can be used to decide several other behavioral equivalences in addition to bisimulation equivalence. Examples of equivalences that can be decided by partition refinement include weak bisimulation [Mil89] and testing equivalence [Hen88]. In order to obtain a partition refinement algorithm for an equivalence relation R, we need to define a suitable *process transformation* $\mathcal{T}_R$ and an equivalence relation E_R that will be used by the algorithm.

Given an LTS $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$, a process transformation is a function $\mathcal{T}_R(\mathcal{L}) = \langle \mathcal{S}', \mathcal{A}', \rightarrow' \rangle$ from the set of LTSs to itself. Abusing notation, we will also use $\mathcal{T}_R$ to denote a mapping from $\mathcal{S}$ to $\mathcal{S}'$ that relates the states of P and $\mathcal{T}_R(\mathcal{L})$. A process transformation can be used to decide an equivalence relation R if there is an equivalence relation E_R such that the following condition holds:

$$s_1 \, R \, s_2 \text{ iff } \mathcal{T}_R(s_1) \sim^{E_R} \mathcal{T}_R(s_2)$$

In this case, we can construct $\mathcal{T}_R(\mathcal{L})$ and then apply an equivalence-checking algorithm to compute $\sim^{E_R}$.

Weak bisimulation equivalence. As an example of process transformation, we present $\mathcal{T}_\approx$ that is used to decide $\approx$, *weak bisimulation* equivalence or *observational* equivalence. Weak bisimulation equivalence abstracts away from the special unobservable action $\tau \in \mathcal{A}$ by allowing transitions to be matched by sequences of transitions having the same observable (non-τ) content.

Definition 3.3 *Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS.*

- *A symmetric relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* weak bisimulation *if whenever $\langle s_1, s_2 \rangle \in R$ and $s_1 \xrightarrow{a} s_1'$ then one of the following holds:*
  - *$a = \tau$ and there is an s_2' such that $s_2 \xrightarrow{\tau}{}^* s_2'$ and $\langle s_1', s_2' \rangle \in R$, or*
  - *there is an s_2' such that $s_2 \xrightarrow{\tau}{}^* \xrightarrow{a} \xrightarrow{\tau}{}^* s_2'$ and $\langle s_1', s_2' \rangle \in R$.*
- *Two states $s_1, s_2 \in \mathcal{S}$ are* weak bisimulation equivalent*, written $s_1 \approx s_2$, if there exists a weak bisimulation R such that $\langle s_1, s_2 \rangle \in R$.*

The transformation used to decide weak bisimulation is $\mathcal{T}_{\approx}(\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle) = \langle \mathcal{S}, \mathcal{A} - \{\tau\} \cup \epsilon, \Rightarrow \rangle$. The *weak*transition relation $\Rightarrow$ is defined as follows:

- $s \stackrel{\epsilon}{\Rightarrow} s'$ when $s \stackrel{\tau}{\rightarrow}^* s'$
- for $a \neq \tau$, $s \stackrel{a}{\Rightarrow} s'$ when $s \stackrel{\tau}{\rightarrow}^* \stackrel{a}{\rightarrow} \stackrel{\tau}{\rightarrow}^* s'$.

We compute $\Rightarrow$ in two steps. First, $\stackrel{\epsilon}{\Rightarrow}$ is constructed by applying a transitive-closure algorithm to the τ-transitions of the original LTS. Then, the composition of relations $\stackrel{\epsilon}{\Rightarrow}$ and $\stackrel{a}{\rightarrow}$, $a \neq \epsilon$, produces all observable weak transitions.

Weak bisimulation equivalence can now be computed as U-bisimulation over $\mathcal{T}_{\approx}(\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle)$. The correctness of this approach follows directly from Definition 3.3 and the construction of $\Rightarrow$.

Other equivalences. Several other process transformations for equivalence checking have been presented in the literature. In particular, *observational congruence* [Mil89] is computed as U-bisimulation over *congruence transition systems* (see [CPS93] for details).

Testing equivalence [Hen88] can also be computed by process transformation. It turns out to be an $\mathcal{A}$-bisimulation over *acceptance graphs*. Both the process transformation and the acceptance set equivalence $\mathcal{A}$ are described in [CH93].

3.5 Computing Branching Bisimulation Equivalence

Branching bisimulation equivalence, introduced by [vGW96], is similar to weak bisimulation equivalence in the sense that it abstracts away from the unobservable action τ. However, branching bisimulation equivalence preserves the branching structure of processes; that is, the non-deterministic choices are resolved by the equivalent processes in the same order. Consequently, branching bisimulation equivalence is finer than weak bisimulation equivalence and possesses an appealing set of algebraic properties.

Despite its close relation to bisimulation equivalence, branching bisimulation equivalence requires a modification of the partition refinement algorithm. The extended algorithm *BB_PARTITIONING* has been presented by Groote and Vaandrager [GV90].

Definition 3.4 *Let $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ be an LTS.*

- *A symmetric relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* branching bisimulation *if whenever $\langle s_1, s_2 \rangle \in R$ and $s_1 \stackrel{a}{\rightarrow} s_1'$ then one of the following holds:*
  - *$a = \tau$ and $\langle s_1', s_2 \rangle \in R$, or*
  - *there exist s_2', s_2'' such that $s_2 \stackrel{\tau}{\rightarrow}^* s_2' \stackrel{a}{\rightarrow} s_2''$ and $\langle s_1', s_2'' \rangle \in R$*
- *Two states $s_1, s_2 \in \mathcal{S}$ are* branching bisimilar*, written $s_1 \underline{\leftrightarrow} s_2$, if there exists a branching bisimulation R such that $\langle s_1, s_2 \rangle \in R$.*

Before presenting the algorithm, we introduce some additional terminology. Given a partition P, a transition $s \xrightarrow{\tau} s'$ is called *P-inert* if s and s' belong to the same block in P. A *bottom* state s in a block B is such that for all outgoing transitions of s, $s \xrightarrow{\tau} s' \Rightarrow s' \notin B$.

We will assume that the LTS does not contain cycles of unobservable events. This assumption is not a restriction. Indeed, if $s \xrightarrow{\tau}{}^{*} s'$ and $s' \xrightarrow{\tau} s$, then $s \leftrightarrow s'$. Then, states strongly connected by unobservable transitions will always be within the same equivalence class, and we can collapse them into the same state before the algorithm is run.

A new notion of splitter is taken directly from Definition 3.4. For $B, B' \in P$ and $a \in \mathcal{A}$, let

$$\begin{aligned} reach_a(B, B') \ = \{s \in B \ \ &| \ \exists n \geq 0 \exists s_0, \ldots, s_n \exists s' \in B'. \\ & s_0 = s \wedge (\forall 0 < i \leq n. s_{i-1} \xrightarrow{\tau} s_i) \wedge (s_n \xrightarrow{a} s' \vee (a = \tau \wedge s_n = s))\}. \end{aligned}$$

A block B' is an a-splitter for B if $\emptyset \neq reach_a(B, B') \neq B$.

A partition refinement algorithm that uses this definition of splitter directly would be impractical. It would have to maintain information about sequences of inert τ-transitions in each block and update this information as blocks are split. Instead, a different characterization of splitter is used, as stipulated by the following theorem.

Theorem 3.5 *Let P be a partition of $\mathcal{S}$, $B, B' \in P$, and $a \in \mathcal{A}$. Then, B' is an a-splitter of B if*

1. *$a \neq \tau$ or $B \neq B'$;*
2. *there exists $s \in B$ such that for some $s' \in B'$, $s \xrightarrow{a} s'$;*
3. *there exists a* bottom *state $s \in B$ such that for no $s' \in B'$, $s \xrightarrow{a} s'$.*

Proof. Suppose B' is an a-splitter of B. If $a = \tau$, then clearly $B \neq B'$, since $reach_\tau(B, B')$ is trivially equal to B, satisfying the first condition. By definition of a splitter, $reach_\tau(B, B') \neq \emptyset$, which means that condition 2 holds. Finally, assume that for every bottom state $s \in B$ there is an a-transition into B'. Choose an arbitrary state $t \in B$. Since there are no τ-cycles in the LTS, then there is a (possibly trivial) τ-path from t to a bottom state s. This means that $t \in reach_1(B, B')$. Since t was chosen arbitrarily, $reach_a(B, B') = B$, which is a contradiction, and condition 3 holds. For the reverse direction, $reach_a(B, B') \neq \emptyset$ because of condition 2, and $reach_a(B, B') \neq B$ because of conditions 1 and 3. $\square$

This characterization of a splitter allows us to find splitters in P efficiently. To do this, we need the following data structures. For each block B, we keep two lists of states. One list holds bottom states of the block, the other holds non-bottom ones. Moreover, we assume that non-bottom states are topologically sorted according to τ-transitions. That is, is $s \xrightarrow{\tau} s'$, then s' precedes s in the list of non-bottom states. This ordering is always well-defined since there are no τ-cycles. Each non-bottom state s contains a list of inert transitions $inert(s)$ originating in this state. Each block B, in addition to the lists of states, contains a list $in(B)$ of non-inert transitions that end in B. Non-inert transitions are lexicographically sorted by their label. Every state $s \in \mathcal{S}$ has a boolean variable $mark(s)$, initialized to *false*.

Blocks of P are stored in two disjoint lists, *stable* and *working*. P is stable with respect to blocks in the first list; blocks in the second list can be splitters. When *working* becomes empty, the partition is stable and the algorithm terminates.

The pseudo-code of the refinement step is shown in Figure 10, where *pre* is the set of blocks that have a transition into B. Procedure *setMarks* accomplishes two things: 1) if s is the source state of transition t, set $mark(s)$ to *true*; 2) add the block to which s belongs to *pre*. For each $B \in pre$, S is *not* a splitter if all bottom states of B are marked.

If there are unmarked bottom states in B, procedure $split(B, B_1, B_2)$ partitions B into two blocks B_1, B_2 in the following way. First, marked bottom states of B become bottom states of B_1 and unmarked states become bottom states of B_2. Next, non-bottom states of B are scanned. If a state s is unmarked and does not have an outgoing transition that leads to a state in B_1, then s becomes a non-bottom state of B_2. All transitions in $inert(s)$ must lead to a state in B_2 and thus no adjustment of $inert(s)$ is needed. Here, the ordering of the non-bottom states is important (note that the ordering is preserved by splitting). Otherwise, s is placed in B_1. Its inert transitions are scanned and, if a transition leads to a state in B_2, it is removed from $inert(s)$ and placed into $in(B_2)$. If $inert(s)$ becomes empty, s is placed into the list of bottom states of B_1. Otherwise, it remains a non-bottom states of B_1. As each state is moved from B to its new block, its mark is cleared. Finally, $in(B)$ is distributed to $in(B_1)$ and $in(B_2)$ according to the target state of each transition.

Procedure *split* returns a boolean value, which is *true* if the set of bottom states changed. It can be easily seen that, unless the set of bottom states in a partition has changed, stability of a block is preserved after refinement. Otherwise, every block in *stable* has to be considered as a splitter again. Thus, if *split* returns *true*, the contents of *stable* are moved to *working*.

Theorem 3.6 *Given a finite-state LTS* $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ *with* $|\mathcal{S}| = n$ *and* $|\rightarrow| = m$*, algorithm* BB_PARTITIONING *computes the stable partition in* $O(|\mathcal{A}| + m \cdot n)$ *time.*

Proof. Initialization of the data structures in the algorithm is accomplished in $O(|\mathcal{A}| + m)$ time. This includes the computation of strongly connected components of $\mathcal{S}$ and topological sorting of inert transitions in each block of the initial partition. Both can be accomplished in $O(m)$ time. Lexicographic sorting of non-inert transitions takes $O(|A| + m)$ time. There can be at most $n - 1$ refinement steps, each of which takes $O(m)$ time. $\Box$

4 Global Preorder Algorithms

In this section we present an efficient algorithm to compute the parameterized semantic relations $\sqsubseteq^{\Pi}_{\Phi_1,\Phi_2}$ introduced in Section 2.4. The algorithm is due to Celikkan and Cleaveland [CC95]. We first introduce an auxiliary function on relations similar to the one given in Definition 2.3.

Definition 4.1 *Let* $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ *be an LTS and* $R \subseteq \mathcal{S} \times \mathcal{S}$ *be a relation between states.*

```
select S ∈ working
pre = ∅
for each a ∈ 𝒜 do
begin
   for each a-transition t ∈ in(S) do
      setMarks(t)
   for each B ∈ pre do
      if S is a splitter of B then
      begin
         remove B from its list
         changed := split(B, B1, B2)
         add B1, B2 to working
         if changed then
            append stable to working
      end
end
working := working − {S}
stable := stable ∪ {S}
```

Figure 10: Refinement step of *BB_PARTITIONING*.

Then $\mathcal{F}^{\sqsubseteq}_{\mathcal{L}} : 2^{\mathcal{S}\times\mathcal{S}} \longrightarrow 2^{\mathcal{S}\times\mathcal{S}}$ *is given by:*

$$\begin{aligned}\mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R) = \{\langle p,q\rangle |\quad & p\Pi q \wedge \forall a \in \mathcal{A}. \\ & \langle p,q,a\rangle \in \Phi_1 \Rightarrow [p \xrightarrow{a} p' \Rightarrow \exists q'.\ q \xrightarrow{a} q' \wedge \langle p',q'\rangle \in R] \wedge \\ & \langle p,q,a\rangle \in \Phi_2 \Rightarrow [q \xrightarrow{a} q' \Rightarrow \exists p'.\ p \xrightarrow{a} p' \wedge \langle p',q'\rangle \in R]\}\end{aligned}$$

Note that relation R is a $\langle \Pi, \Phi_1, \Phi_2\rangle$-bisimulation *iff* $R \subseteq \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$. Correspondingly, if $R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R) \neq \emptyset$ then R is not a $\langle \Pi, \Phi_1, \Phi_2\rangle$-bisimulation. It is easy to see that the algorithm $EFF\_PREORDER$ in Figure 11 computes $\sqsubseteq^{\Pi}_{\Phi_1,\Phi_2}$ over $\mathcal{S} \times \mathcal{S}$.

In order to efficiently implement this algorithm, one must maintain the value of $ToDelete = R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$, as removing an element from R may change the value of $\mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$. Now from the definition of $\mathcal{F}^{\sqsubseteq}_{\mathcal{L}}$ it follows that if $\langle p,q\rangle \in R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$ then one of the three conditions below must hold.

1. $\langle p,q\rangle \notin \Pi$
2. $\langle p,q,a\rangle \in \Phi_1 \wedge \exists p'.\ p \xrightarrow{a} p' \wedge \forall q'.(q \xrightarrow{a} q' \Rightarrow \langle p',q'\rangle \notin R)$
3. $\langle p,q,a\rangle \in \Phi_2 \wedge \exists q'.\ q \xrightarrow{a} q' \wedge \forall p'(p \xrightarrow{a} p' \Rightarrow \langle p',q'\rangle \notin R)$

To understand how we may exploit these conditions, suppose that $\langle p,q\rangle \in R$, $\langle p,q,a\rangle \in \Phi_1$, and p has an a-derivative p' such that $\langle p',q'\rangle$ for only one a-derivative q' of q. Now if $\langle p',q'\rangle$

$R := \mathcal{S} \times \mathcal{S}$;
$ToDelete := R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$
while $ToDelete \neq \emptyset$ **do begin**
  $Choose\ \langle p, q\rangle \in ToDelete$;
  $R := R - \{\langle p, q\rangle\}$;
  $ToDelete := R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$;
end;

Figure 11: Algorithm $EFF\_PREORDER$ for computing the $\sqsubseteq^{\Pi}_{\Phi_1,\Phi_2}$ relation.

is removed from R, it follows that $\langle p, q\rangle \notin \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$ and hence $\langle p, q\rangle \in R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$. Thus $\langle p, q\rangle$ must be removed from R, as R cannot be a bisimulation if $\langle p, q\rangle \in R$. This suggests that if we know how many a-derivatives of q that p' is related to, then when this number reaches 0 we can remove pairs involving q and states (like p) having an a-transition into p'. To record this information, $EFF\_PREORDER$ maintains two arrays of counters. We describe these in turn.

- $HighCount(a, p', q)$ is the number of a-derivatives of the state q that are "higher" than p'.
- $LowCount(a, p, q')$ is the number of a-derivatives of the state p that are "lower" than q'.

Formally they are defined as follows.

$$HighCount(a, p', q) = |\{\, q' \mid q \xrightarrow{a} q' \wedge \langle p', q'\rangle \in R \,\}|$$

$$LowCount(a, p, q') = |\{\, p' \mid p \xrightarrow{a} p' \wedge \langle p', q'\rangle \in R \,\}|$$

The task of implementing $EFF\_PREORDER$ can now be divided into an initialization phase and an iteration phase. The former involves the computation of the initial value of R, $ToDelete$, $HighCount$ and $LowCount$. The initialization code is given in Figure 12. Note that $HighCount(a, p', q)$ is set to the number of a-derivatives of q since we are assuming that all states are related (that is, we start with the coarsest partition), and similarly for $LowCount$. The set R holds the intermediate approximations to the preorder, and it will contain the desired preorder upon termination of the algorithm.

After the initialization phase, the algorithm repeatedly removes elements from $ToDelete$, deletes them from R, and incrementally updates $ToDelete = R - \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$ using the counters $HighCount$ and $LowCount$. (As a practical matter, note that when an element is added to $ToDelete$, one may immediately remove it from R without affecting the correctness of the algorithm.) These counter values determine when a pair of states is not an element of $\mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$ and thus should be added to $ToDelete$ for eventual removal from R. The algorithm terminates when $ToDelete = \emptyset$, in which case $R = \mathcal{F}^{\sqsubseteq}_{\mathcal{L}}(R)$ and R is therefore the largest $\langle \Pi, \Phi_1, \Phi_2\rangle$-bisimulation on $\mathcal{S}$. Figures 13 and 14 illustrate how $HighCount$ counters are used

$ToDelete := \{ \langle p, q \rangle \mid p \not\Pi q \}$;
$R := \mathcal{S} \times \mathcal{S}$;

`foreach` $\langle p, q \rangle \in R$ `do begin`
    `foreach` $a \in (\{p \xrightarrow{\bullet}\} \cup \{q \xrightarrow{\bullet}\})$ `do begin`
        $HighCount(a, p, q) = |\{ q' \mid q \xrightarrow{a} q' \}|$;
        $LowCount(a, p, q) = |\{ p' \mid p \xrightarrow{a} p' \}|$;

{* *If q does not have an a-transition, pair $\langle p, q \rangle$ is inserted into ToDelete.* *}
        `if` $HighCount(a, p, q) = 0$ and $\langle p, q, a \rangle \in \Phi_1$ and $\langle p, q \rangle \notin ToDelete$ `then`
            $ToDelete := ToDelete \cup \{\langle p, q \rangle\}$;

{* *If p does not have an a-transition, pair $\langle p, q \rangle$ is inserted into ToDelete.* *}
        `if` $LowCount(a, p, q) = 0$ and $\langle p, q, a \rangle \in \Phi_2$ and $\langle p, q \rangle \notin ToDelete$ `then`
            $ToDelete := ToDelete \cup \{\langle p, q \rangle\}$;
    `end`;
`end`;

Figure 12: Initialization of $ToDelete$ and R.

during the algorithm. A $HighCount(a, p', q)$ value of 0 means that there is no a-derivative of q related to p'; thus all pairs $\langle p, q \rangle$ such that $p \xrightarrow{a} p'$ should be inserted into the set $ToDelete$. Note that the insertion of a pair $\langle p, q \rangle$ into the set $ToDelete$ is guarded by the predicates Φ_1 and Φ_2. A $HighCount(a, p', q)$ value of 0 causes a pair $\langle p, q \rangle$ to be inserted into $ToDelete$ only if $\langle p, q, a \rangle$ is in Φ_1. $LowCount$ is employed analogously. Figure 15 contains the code for the iteration phase of the algorithm.

We now remark on the time complexity of the preorder-checking algorithm given in Figures 12 and 15. If $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ is an LTS, let $|\mathcal{L}| = |\mathcal{S}| + |\rightarrow|$. Also let t_{init} be the time spent on computing Π, Φ_1 and Φ_2. We also assume that sets are implemented using hash tables and thus that insertion, deletion and set membership may be computed in $O(1)$ time.

Theorem 4.2 *The code given in Figure 12 takes $O(t_{init} + |\mathcal{S}| \times |\mathcal{L}|)$ time.*

Proof. Initialization of $ToDelete$ and R takes $O(t_{init} + |\mathcal{S}|^2)$ time units. The nested "**foreach**" loops require $O(|\mathcal{S}| \times |\mathcal{L}|)$ set membership operations. This follows from the fact that

$$\begin{aligned}
\sum_{p \in \mathcal{S}} \sum_{q \in \mathcal{S}} \sum_{a \in (\{p \xrightarrow{\bullet}\} \cup \{q \xrightarrow{\bullet}\})} O(1) &\leq \sum_{p \in \mathcal{S}} \sum_{q \in \mathcal{S}} |\{p \xrightarrow{\bullet}\}| + |\{q \xrightarrow{\bullet}\}| \\
&\leq \sum_{p \in \mathcal{S}} \sum_{q \in \mathcal{S}} |\{q \xrightarrow{\bullet}\}| + \sum_{p \in \mathcal{S}} \sum_{q \in \mathcal{S}} |\{p \xrightarrow{\bullet}\}|
\end{aligned}$$

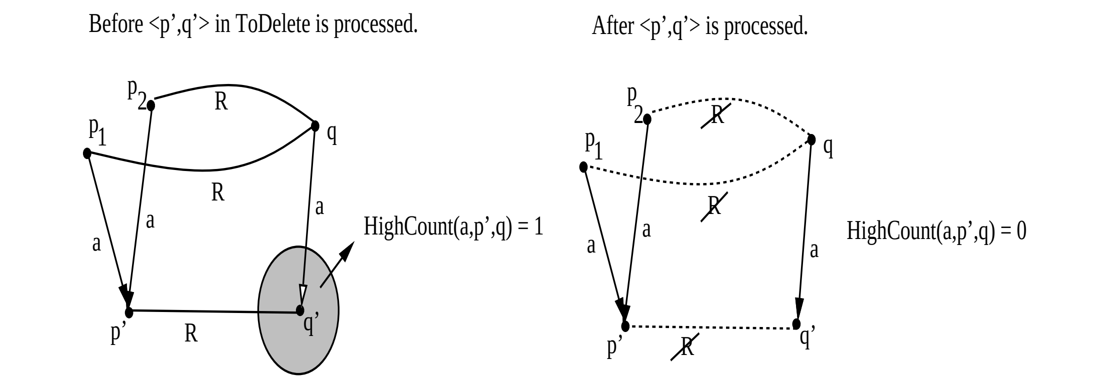

Figure 13: Illustration of the main loop where *HighCount* value is 1.

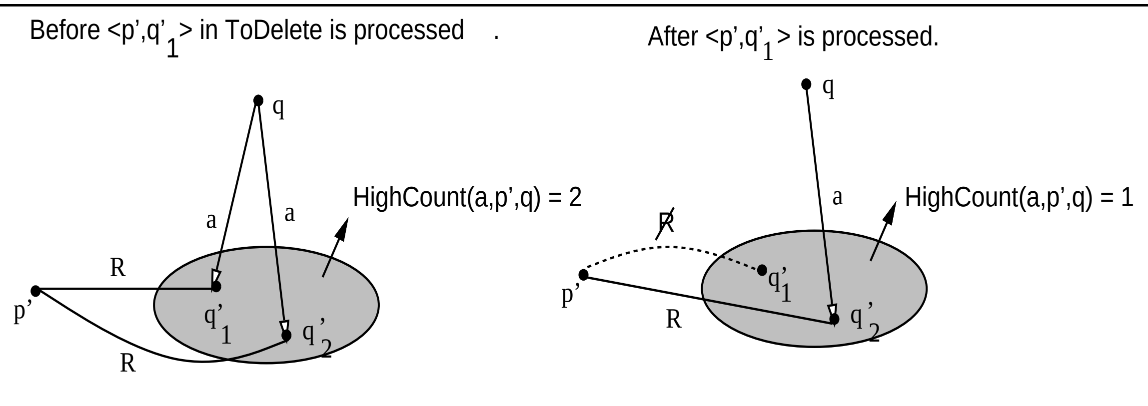

Figure 14: Illustration of the main loop where *HighCount* value is greater than 1.

```
while ToDelete ≠ ∅ do begin
    Choose ⟨p', q'⟩ ∈ ToDelete;
    R := R − {⟨p', q'⟩};
/* First condition */
    foreach a ∈ {•→ q'} do begin
        foreach q ∈ {• -a-> q'} such that ⟨p, q, a⟩ ∈ Φ1 do begin
            HighCount(a, p', q) := HighCount(a, p', q) − 1;
            if HighCount(a, p', q) = 0 then begin
                foreach p ∈ {• -a-> p'} do
                    if ⟨p, q⟩ ∈ R and ⟨p, q⟩ ∉ ToDelete then
                            ToDelete := ToDelete ∪ {⟨p, q⟩};
            end;
        end;
    end;
/* Second condition */
    foreach a ∈ {•→ p'} do begin
        foreach p ∈ {• -a-> p'} such that ⟨p, q, a⟩ ∈ Φ2 do begin
            LowCount(a, p, q') := LowCount(a, p, q') − 1;
            if LowCount(a, p, q') = 0 then begin
                foreach q ∈ {• -a-> q'} do
                    if⟨p, q⟩ ∈ R and ⟨p, q⟩ ∉ ToDelete then
                            ToDelete := ToDelete ∪ {⟨p, q⟩};
            end;
        end;
    end;
    ToDelete := ToDelete − {⟨p', q'⟩};
end
```

Figure 15: Main loop of algorithm *EFF_PREORDER*.

$$
\begin{aligned}
&\leq \sum_{p\in\mathcal{S}} |\mathcal{L}| + \sum_{q\in\mathcal{S}} |\mathcal{L}| \\
&\leq O(|\mathcal{S}| \times |\mathcal{L}| + |\mathcal{S}| \times |L|)
\end{aligned}
$$

and checking predicates Φ_1 and Φ_2 may be done in constant time after initializing them properly. Then the total amount of time spent is $O(t_{init} + |\mathcal{S}|^2 + |\mathcal{S}| \times |\mathcal{L}|)$. This reduces to $O(t_{init} + |\mathcal{S}| \times |\mathcal{L}|)$. $\Box$

Theorem 4.3 *The code given in Figure 15 takes $O(|\mathcal{S}|^2 + |\mathcal{S}| \times |\mathcal{L}|)$ time.*

Proof. We first note that in the worst case, the outer loop executes at most $O(|\mathcal{S}|^2)$ times. Let T_1 represent the total amount of time spent executing the first "`foreach`" loop over all iterations of the outermost "`while`" loop and T_2 represent the time spent in the second "`foreach`" loop. T_1 may be further decomposed into:

$$T_1 = T_i + T_r$$

where T_i represents the total amount of time spend in the innermost "`foreach` $p \in \{\bullet \xrightarrow{a} p'\}$ `do begin`" loop and T_r represents the time spent in the rest of the loop (e.g. in decrementing $HighCount(a, p, q')$ and performing the test in the `if-then` statement). From the structure of the loops, we have the following.

$$
\begin{aligned}
T_r &= \sum_{\langle p',q'\rangle\in\mathcal{S}\times\mathcal{S}} \; \sum_{a\in\{\xrightarrow{\bullet} q'\}} \; \sum_{q\in\{\bullet\xrightarrow{a} q'\}} O(1) \\
&= \sum_{p'\in\mathcal{S}} \sum_{q'\in\mathcal{S}} \; \sum_{a\in\{\xrightarrow{\bullet} q'\}} \; \sum_{q\in\{\bullet\xrightarrow{a} q'\}} O(1) \\
&= \sum_{p'\in\mathcal{S}} O(|\mathcal{L}|) \\
&= O(|\mathcal{S}| \times |\mathcal{L}|)
\end{aligned}
$$

Regarding T_i, first note that for any a, p' and q, $HighCount(a, p', q)$ has its value changed to zero at most once. Thus the total amount of time spent in the innermost loop is:

$$
\begin{aligned}
T_i &= \sum_{\langle a,p',q\rangle\in Act\times\mathcal{S}\times\mathcal{S}} \; \sum_{p\in\{\bullet\xrightarrow{a} p'\}} O(1) \\
&= \sum_{q\in\mathcal{S}} \; \sum_{\langle a,p'\rangle\in Act\times\mathcal{S}} \; \sum_{p\in\{\bullet\xrightarrow{a} p'\}} O(1) \\
&= \sum_{q\in\mathcal{S}} O(|\mathcal{L}|) \\
&= O(|\mathcal{S}| \times |\mathcal{L}|)
\end{aligned}
$$

Thus $T_1 = O(|\mathcal{S}| \times |\mathcal{L}|)$. Using a similar argument, we may also infer that $T_2 = O(|\mathcal{S}| \times |\mathcal{L}|)$, and the theorem then follows. $\Box$

Note that the time spent in the initialization phase of this algorithm asymptotically dominates the time devoted to the second phase. We thus have the following.

Theorem 4.4 *Given a finite-state LTS $\langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ with $|\mathcal{S}| = n$ and $| \rightarrow | = m$, algorithm $EFF\_PREORDER$ takes $O(t_{init} + |\mathcal{S}| \times |\mathcal{L}|)$ time.*

Speeding up the Algorithm

In general, when computing $\sqsubseteq^{\Pi}_{\Phi_1,\Phi_2}$ we are in fact really interested in whether the two given states, p and q, are related. If they are found not to be, then it is possible to terminate the algorithm immediately. Therefore if $p \not\sqsubseteq^{\Pi}_{\Phi_1,\Phi_2} q$, then we would like to determine this as quickly as possible. This immediately suggests an optimization: if $\langle p, q\rangle$ is ever inserted into $ToDelete$ then the algorithm can terminate.

Another heuristic for early termination involves processing elements in $ToDelete$ in a particular order. Note that when $\langle p', q'\rangle \in ToDelete$ is processed, it may induce the removal of $\langle p, q\rangle$ from R, where $p \stackrel{a}{\rightarrow} p'$ and $q \stackrel{a}{\rightarrow} q'$. Therefore while processing the pairs in $ToDelete$, the pair that is "closest" to the pair $\langle p, q\rangle$ that we are interested in will be the most promising candidate to process. Formally, define $l(p')$ to be the length of the shortest path from p to p', and $l(q')$ to be the length of the shortest path from q to q'. Also let $h(p', q') = l(p') + l(q')$. Then among the pairs in $ToDelete$ the pair that minimizes $h(p', q')$ should be chosen for processing.

Another way to speed up the algorithm is to avoid entirely the corresponding loops when either Φ_1 or Φ_2 is empty. If one of these relations is empty there is no need check the condition involving this relation, since that condition is trivially satisfied.

5 Local Algorithms

Algorithms for computing semantic equivalences and refinement replations usually consist of two steps [CPS93]. In the first, the state space is generated and stored (possibly symbolically), while the second then manipulates the state space to determine whether an appropriate relation exists. The algorithms presented in the previous sections are examples of such "global" algorithms. Global algorithms often perform poorly in practice because of the requirement that the state space be generated in advance. In particular, in many cases, one may be able to determine that one process fails to be related to another by examining only a fraction of the state space. In a design setting where one is repeatedly changing a design and checking it against a specification, one would like a verification algorithm exploiting this fact.

On-the-fly verification algorithms combine the checking of a system's correctness with the generation of the system's state space [CVWY90]. In this section we present the efficient on-the-fly preorder-checking algorithm proposed in [Cel95]. To determine whether two states p_0, q_0 are related, the algorithm attempts to construct a relation relating the states of the LTS incrementally starting with the pair $\langle p_0, q_0\rangle$. In order to achieve the desired complexity we represent the relation being built as an edge-labeled graph whose vertices are pairs of related states. Formally let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow\rangle$ be an LTS and $p_0, q_0 \in \mathcal{S}$. We have the following.

Theorem 5.1 $p_0 \sqsubseteq^{\Pi}_{\Phi_1,\Phi_2} q_0$ *iff there exists a graph $\mathcal{G} = \langle V, E\rangle$ with $V \subseteq \mathcal{S} \times \mathcal{S}$ and $E \subseteq V \times \mathcal{A} \times V$ such that*

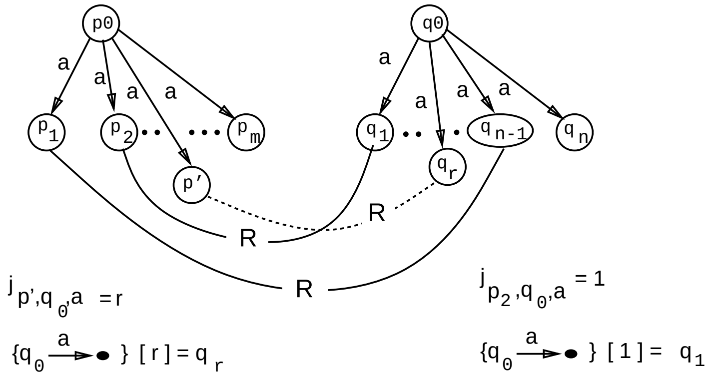

Figure 16: Illustration of the pointers $high_{p',q,a}$.

- $(p_0, q_0) \in V$
- *whenever* $(p, q) \in V$ *then* $\langle p, q\rangle \in \Pi$
- *whenever* $(p, q) \in V$ *and* $\langle p, q, a\rangle \in \Phi_1$ *and* $p \xrightarrow{a} p'$ *then there exists a* q' *such that* $q \xrightarrow{a} q'$ *with* $(p', q') \in V$ *and* $((p', q'), a_1, (p, q)) \in E$*.*
- *whenever* $(p, q) \in V$ *and* $\langle p, q, a\rangle \in \Phi_2$ *and* $q \xrightarrow{a} q'$ *then there exists a* p' *such that* $p \xrightarrow{a} p'$ *with* $(p', q') \in V$ *and* $((p', q'), a_2, (p, q)) \in E$

If such a graph exists then V represents a $\langle\Pi, \Phi_1, \Phi_2\rangle$-bisimulation relating p_0 and q_0. If $\langle p, q\rangle \in V(\mathcal{G})$, then the edges leading into $\langle p, q\rangle$ may be thought of as the "justification" for including $\langle p, q\rangle$ in the preorder. The index i of the action a indicates which of the conditions of Definition 4.1 the justification is based on. If $i = 1$ then the basis of justification is condition 2; if $i = 2$ then it is condition 3.

Given LTS $\mathcal{L} = \langle\mathcal{S}, \mathcal{A}, \rightarrow\rangle$ and two states $p_0, q_0 \in \mathcal{S}$, the algorithm works by attempting to build a graph as described in Theorem 5.1. This construction proceeds incrementally using a *depth-first search* of $\mathcal{S} \times \mathcal{S}$, starting with $\langle p_0, q_0\rangle$. When the algorithm is invoked on p and q, a vertex for $\langle p, q\rangle$ is added to the graph (if one does not exist already) provided that $\langle p, q\rangle \in \Pi$. The algorithm then attempts to add nodes relating each a-derivative of p to some a-derivative of q whenever $\langle p, q, a\rangle \in \Phi_1$, with edges being added from these nodes to $\langle p, q\rangle$ to indicate that the inclusion of $\langle p, q\rangle \in \mathcal{G}$ currently depends on the presence of these nodes. If no matching a-derivative of q can be found for some a-derivative p' of p, then p and q cannot be related. In this case $\langle p, q\rangle$ must be removed from the graph, and the nodes depending on $\langle p, q\rangle$ must be re-examined to determine whether they should also be removed. Note that once p and q are found not to be related, they need not be processed again; no graph can be constructed that includes $\langle p, q\rangle$ as a node. A similar action is taken while matching the each a-derivative of q to some a-derivative of p. Note that if $\langle p, q\rangle \in \Pi$ and $\langle p, q, a\rangle \notin \Phi_1$ and $\langle p, q, a\rangle \notin \Phi_2$ then $\langle p, q\rangle$ becomes a leaf vertex of $\mathcal{G}$.

The following three data structures are used:

- The graph $\mathcal{G}$.

- A set $\overline{R}$ that stores all the state pairs that have been determined not to be related.

- A set of pointers of the form $high_{p',q,a}$ and $low_{q',p,a}$. Intuitively, $high_{p',q,a}$ is the index of the state in $\{q \xrightarrow{a} \bullet\}$ that is currently matched to p' and $low_{q',p,a}$ is the index of the state in $\{p \xrightarrow{a} \bullet\}$ that is currently matched to q'. By using these pointers we ensure that p' will be compared to each a-transition of q at most once and similarly q' will be compared to each a-transition of p at most once.

- A set A which records the pairs that need to be re-examined. It contains tuples of the sort $\langle r \xrightarrow{a} r', s, type\rangle$. r, r', and s are states, and $type$ indicates which condition needs to be rechecked. If $type$ is equal to 1 then r and s are candidates for being related by $r \sqsubseteq\!\!\!\sim^{\Pi}_{\Phi_1,\Phi_2} s$ and r' needs to be matched to some a-derivative of s using the $high$ pointers, namely $high_{r',s,a}$. Dually, if $type$ is equal to 2 then r and s are supposed to be related by $s \sqsubseteq\!\!\!\sim^{\Pi}_{\Phi_1,\Phi_2} r$ and low pointers will be used to find a match for r'.

The algorithm consists of three procedures. Given a transition $p \xrightarrow{a} p'$ and state q, procedure $SEARCH\_HIGH$ tries to match p' to some a-derivative of q, under the assumptions that $high_{p',q,a}$ is the index of the first potential candidate and $\langle p, q, a\rangle \in \Phi_1$. If it finds a match then it adds the edge $\langle\langle p', q'\rangle, a_1, \langle p, q\rangle\rangle$ into $\mathcal{G}$ to indicate that the status of $\langle p, q\rangle$ depends on the status of $\langle p', q'\rangle$— any status change of $\langle p', q'\rangle$ requires the re-analysis of state $\langle p, q\rangle$. Procedure $SEARCH\_LOW$ performs the dual task of procedure $SEARCH\_HIGH$. Given states p and q, main procedure $PREORDER$ tries to match each a-derivative of p with some a-derivative of q by calling $SEARCH\_HIGH$ and each a-derivative of q with some a-derivative of p by calling $SEARCH\_LOW$. $PREORDER$ does not invoke $SEARCH\_HIGH$ or $SEARCH\_LOW$ if $\langle p, q\rangle \notin \Pi$. When $SEARCH\_HIGH$ or $SEARCH\_LOW$ returns a $not\_related$ status ($SEARCH\_HIGH$ returns a $not\_related$ status if some a-derivative of p cannot be matched to any a-derivative of q) then vertex $\langle p, q\rangle$ is removed from $\mathcal{G}$ and inserted into $\overline{R}$. Then all the vertices having an incoming edge from $\langle p, q\rangle$ are re-analyzed. These vertices are stored in the set A. Pseudo-code for these procedures may be found in Figures 17, 18 and 19.

Theorem 5.2 *Let p, q be states in a finite-state LTS, and assume $\overline{R} = \emptyset$ and $\mathcal{G} = \langle\emptyset, \emptyset\rangle$. Then $PREORDER(p, q)$ terminates, and $PREORDER(p, q) = related$ iff $p \sqsubseteq\!\!\!\sim^{\Pi}_{\Phi_1,\Phi_2} q$.*

Proof. Each possible vertex and edge is added and deleted at most once from $\mathcal{G}$. Since the number of potential edges and vertices is bounded, $PREORDER$ eventually terminates. As for correctness, we first note that when $PREORDER(p, q)$ terminates $\mathcal{G}$ satisfies the conditions laid out in Theorem 5.1. Also, $PREORDER(p, q)$ returns a $related$ result if and only if $\langle p, q\rangle$ is a vertex in $\mathcal{G}$. Therefore, if $PREORDER(p, q) = related$ then $p \sqsubseteq\!\!\!\sim^{\Pi}_{\Phi_1,\Phi_2} q$. Now note that $PREORDER(p, q) = not\_related$ if and only if $\langle p, q\rangle \in \overline{R}$ at termination. Also, $\langle p, q\rangle$ is added to $\overline{R}$ only if some a-transition of p can not be matched to any a-transition of q in such a way that the resulting $\mathcal{G}$ satisfies Theorem 5.1. Therefore, if $PREORDER(p, q) = not\_related$ then $p \not\sqsubseteq\!\!\!\sim q$. $\square$

The time complexity of $PREORDER$ may now be characterized as follows. We assume that the calculation of the transitions from a state takes unit time (that is, the set of transitions of a state is stored in the state rather than being computed by the algorithm).

$\overline{R} := \emptyset$;
$\mathcal{G} := \langle \emptyset, \emptyset \rangle$;
$PREORDER(p, q) \rightarrow (result : \{related, not\_related\})$
{* *p and q are not related.* *}
  if $\langle p, q \rangle \in \overline{R}$ **then return** $(not\_related)$;
  if $\langle p, q \rangle \notin \Pi$ **then**
    $\overline{R} := \overline{R} \cup \{\langle p, q \rangle\}$;
    return $(not\_related)$;
  end;
{* *Pair $\langle p, q \rangle$ has already been inserted into $V(\mathcal{G})$. Its status is assumed related.* *}
  if $\langle p, q \rangle \in V(\mathcal{G})$ **then return** $(related)$;
{* *Pair $\langle p, q \rangle$ is going to be analyzed for the first time.* *}
  $\mathcal{G} := \langle V(\mathcal{G}) \cup \{\langle p, q \rangle\}, E(\mathcal{G}) \rangle$;
  $status := related$;
{* *Match each a-derivative of p with some a-derivative of q.* *}
  foreach $a \in \{p \xrightarrow{\bullet}\}$ **while** $status = related$ **do**
    foreach $p' \in \{p \xrightarrow{a} \bullet\}$ **while** $status = related$ **do**
      $status := not\_related$;
      if $high_{p',q,a}$ *does not exist* **then**
        create $high_{p',q,a}$
        $high_{p',q,a} := 1$;
      end;
{* *Match p' with some a-derivative of q.* *}
      $status := SEARCH\_HIGH(p \xrightarrow{a} p', q)$;
      if $status = not\_related$ **then**
{* *A contains vertices that are going to be affected as a result of the removal of $\langle p, q \rangle$ from $\mathcal{G}$.* *}
        $A := \{\, \langle r \xrightarrow{a} p, s, i \rangle \mid \langle \langle p, q \rangle, a_i, \langle r, s \rangle \rangle \in E(\mathcal{G}) \,\}$;
    end;
  end;
{* *Match each a-derivative of q with some a-derivative of p.* *}
  foreach $a \in \{q \xrightarrow{\bullet}\}$ **while** $status = related$ **do**
    foreach $p' \in \{q \xrightarrow{a} \bullet\}$ **while** $status = related$ **do**
      $status := not\_related$;
      if $low_{q',p,a}$ *does not exist* **then**
        create $low_{q',p,a}$
        $low_{q',p,a} := 1$;
      end;
{* *Match q' with some a-derivative of p.* *}
      $status := SEARCH\_LOW(q \xrightarrow{a} q', p)$;
      if $status = not\_related$ **then** $A := \{\, \langle s \xrightarrow{a} q, r, i \rangle \mid \langle \langle p, q \rangle, a_i, \langle r, s \rangle \rangle \in E(\mathcal{G}) \,\}$;
    end;
  end;

Figure 17: "On-the-fly" $PREORDER$.

```
{* Re-analyze all the states that are affected from the removal of vertex ⟨p,q⟩ from 𝒢.
   The set A contains all such pairs. *}
   if status = not_related then
{* Remove incoming and outgoing edges from vertex ⟨p,q⟩. *}
      𝒢 := ⟨V(𝒢) − {⟨p,q⟩}, E(𝒢) − { ⟨π1, a_i, π2⟩ | (π1 = ⟨p,q⟩ ∨ π2 = ⟨p,q⟩) ∧ i ∈ {1,2} }⟩;
{* Add ⟨p,q⟩ to the set of unrelated state pairs. *}
      R̄ := R̄ ∪ {⟨p,q⟩};
      while A ≠ ∅ do
         Choose ⟨r -a-> r', s, type⟩ ∈ A;
         A := A − {⟨r -a-> r', s, type⟩};
         if type = 1 then begin
            high_{r',s,a} := high_{r',s,a} + 1;
            {* Match r' to the next a-derivative of s. *}
            status := SEARCH_HIGH(r -a-> r', s);
            if status = not_related then
               A := A ∪ { ⟨rr -b-> r, ss, i⟩ | ⟨⟨r,s⟩, b_i, ⟨rr,ss⟩⟩ ∈ E(𝒢) };
               𝒢 := ⟨V(𝒢) − {⟨r,s⟩}, E(𝒢) − { ⟨π1, b_i, π2⟩ | π1 = ⟨r,s⟩ ∨ π2 = ⟨r,s⟩ }⟩;
               R̄ := R̄ ∪ {⟨r,s⟩};
            end;
         end;
         if type = 2 then begin
            low_{r',s,a} := low_{r',s,a} + 1;
            status := SEARCH_LOW(r -a-> r', s);
            if status = not_related then
               A := A ∪ { ⟨rr -b-> r, ss, i⟩ | ⟨⟨s,r⟩, b_i, ⟨ss,rr⟩⟩ ∈ E(𝒢) };
               𝒢 := ⟨V(𝒢) − {⟨s,r⟩}, E(𝒢) − { ⟨π1, b_i, π2⟩ | π1 = ⟨s,r⟩ ∨ π2 = ⟨s,r⟩ }⟩;
               R̄ := R̄ ∪ {⟨s,r⟩};
            end;
         end;
      end;
   endif;
   return(status);
end PREORDER;
```

Figure 18: "On-the-fly" $PREORDER$ (continued).

$SEARCH\_HIGH(p \xrightarrow{a} p', q) \rightarrow (result : \{related, not\_related\})$

    $status := not\_related;$

    if $\langle p, q, a\rangle \notin \Phi_1$ **then** $status := related;$

    while $status = not\_related$ **and** $high_{p',q,a} \leq |\{q \xrightarrow{a} \bullet\}|$ **do**

{* *Start searching beginning from the first potential candidate referenced by* $high_{p',q,a}$. *}

        $status := PREORDER(p', \{q \xrightarrow{a} \bullet\}[high_{p',q,a}]);$

        if $status = related$ **then**

            $\mathcal{G} := (V(\mathcal{G}), E(\mathcal{G}) \cup \{\langle\langle p', \{q \xrightarrow{a} \bullet\}[high_{p',q,a}]\rangle, a_1, \langle p, q\rangle\rangle\};$

        else

{* *A match is not found. Process the next candidate.* *}

            $high_{p',q,a} := high_{p',q,a} + 1;$

    end;

    return $(status);$

end $SEARCH\_HIGH;$

$SEARCH\_LOW(q \xrightarrow{a} q', p) \rightarrow (result : \{related, not\_related\})$

    $status := not\_related;$

    if $\langle p, q, a\rangle \notin \Phi_2$ **then** $status := related;$

    while $status = not\_related$ **and** $low_{q',p,a} \leq |\{p \xrightarrow{a} \bullet\}|$ **do**

{* *Start searching beginning from the first potential candidate referenced by* $low_{q',p,a}$. *}

        $status := PREORDER(\{p \xrightarrow{a} \bullet\}[low_{q',p,a}], q');$

        if $status = related$ **then**

            $\mathcal{G} := (V(\mathcal{G}), E(\mathcal{G}) \cup \{\langle\langle \{p \xrightarrow{a} \bullet\}[low_{q',p,a}], q'\rangle, a_2, \langle p, q\rangle\rangle\};$

        else

{* *A match is not found. Process the next candidate.* *}

            $low_{q',p,a} := low_{q',p,a} + 1;$

    end;

    return $(status);$

end $SEARCH\_LOW;$

Figure 19: Procedures $SEARCH\_HIGH$ and $SEARCH\_LOW$.

Theorem 5.3 *Let $\mathcal{L} = \langle \mathcal{S}, \mathcal{A}, \rightarrow \rangle$ with $|\mathcal{L}| = |\mathcal{S}| + |\rightarrow|$. PREORDER takes time proportional to that required by $O(t_{init} + m)$ set membership operations, where $m \leq |\mathcal{L}|^2$ and t_{init} is the time required to compute Π, Φ_1, and Φ_2.*

Proof. Ignoring the time consumed by recursive calls, the time spent to match each a-derivative of a state p to some a-derivative of q and to match each a-derivative of a state q to some a-derivative of p can be characterized as

$$\sum_{a \in \mathcal{A}} |\{p \xrightarrow{a} \bullet\}| * t(SEARCH\_HIGH) + \sum_{a \in \mathcal{A}} |\{q \xrightarrow{a} \bullet\}| * t(SEARCH\_LOW)$$

where $t(SEARCH\_HIGH) = O(|\{q \xrightarrow{a} \bullet\}|)$ and $t(SEARCH\_LOW) = O(|\{p \xrightarrow{a} \bullet\}|)$. Then the total time spent to execute the nested `foreach` loops in Figure 17 over all the recursive calls is bounded by

$$\sum_{(p,q) \in \mathcal{S} \times \mathcal{S}} \sum_{a \in \mathcal{A}} (|\{p \xrightarrow{a} \bullet\}| * |\{q \xrightarrow{a} \bullet\}| + |\{q \xrightarrow{a} \bullet\}| * |\{p \xrightarrow{a} \bullet\}|) \leq O(|\mathcal{L}|^2).$$

When a state pair (p, q) is found to be unrelated, all the state pairs that depend on this state pair need to be re-analyzed. The time spent to re-analyze these state pairs ignoring the time consumed by recursive calls is

$$O(\sum_{a \in \mathcal{A}} |\{\bullet \xrightarrow{a} p\}| * |\{\bullet \xrightarrow{a} q\}|).$$

In the worst case each state pair needs to be re-analyzed, and total time required to re-analyze the state pairs is bounded by

$$\sum_{(p,q) \in \mathcal{S} \times \mathcal{S}} \sum_{a \in \mathcal{A}} |\{\bullet \xrightarrow{a} p\}| * |\{\bullet \xrightarrow{a} q\}| \leq O(|\mathcal{L}|^2).$$

Then it follows that the time it takes to execute $PREORDER$ is proportional to that required by $O(t_{init} + m)$ set membership operations. $\square$

As noted in Section 2.4, $\sqsubseteq^{\Pi}_{\Phi_1, \Phi_2}$ can represent various behavioral equivalences and preorders by choosing Φ_1, Φ_2, and Π appropriately. Therefore, *PREORDER* yields an efficient local checking algorithm for all these relations.

6 Tools

Over the last decade, a number of tools that use process-algebraic formalisms for specification and analysis of systems have been developed. Several of these tools implement one or more of the algorithms presented in this chapter.

The first successful tool in this category was the Concurrency Workbench (CWB) [CPS93], developed as a joint project between the University of Edinburg, the University of Sussex, and Uppsala University. The CWB implements global algorithms for several behavioral equivalences and preorders, including bisimulation, observational equivalence, and branching bisimulation.

A continuation of the Concurrency Workbench effort resulted in CWB-NC, the Concurrency Workench of the New Century [CS96]. In addition to improved efficiency and an extended set of algorithms, CWB-NC features the ability to "plug in" user-defined system description languages, and uses language-independent analysis algorithms on a uniform internal representation of processes.

The Fc2 toolset [BRRdS96] gives the user the choice between explicit representation of the state space with partition refinement analysis techniques and symbolic OBDD-based representation. A graphical user interface provided by the Autograph visual editor gives a more intuitive specification language compared with text-based process algebra terms. The Fc2 *interchange format makes* it possible to share specifications with other tools that support this format, such as Aldébaran, described below.

Cadp (Cæsar/Aldébaran Development Package) [FGK+96] also supports both partition refinement techniques based on the Paige-Tarjan algorithm and symbolic BDD-based algorithms. The toolset is oriented towards analysis of systems expressed in the high-level process-algebraic languages Lotos and E-Lotos [GS98]. A number of interchange formats are supported, allowing the user to work with other high-level languages such as Sdl as well as low-level representations.

The commercially available tool FDR2 [Ros94] is based on the CSP [Hoa85] process algebra. FDR2's main analysis technique is based on establishing refinements (behavioral preorders) between processes. Analysis of real-time systems, including schedulability and resource requirements, is supported by the PARAGON toolset [SLBA99]. PARAGON calculates a resource-sensitive version of bisimulation equivalence using a partition-refinement algorithm.

7 Conclusions

This chapter presented several algorithms for the analysis of finite-state process-algebraic specifications. The algorithms calculate behavioral equivalences and preorders; equivalences can be used to compare two processes for indistinguishable behavior and to minimize the state space of a system in a behavior-preserving manner. Preorders can be used to establish refinements, that is, specification/implementation relationships between processes.

Finite-state systems represent an important class of system specifications, amenable to fully automatic formal analysis. Finite-state systems are used in the specification and verification of hardware and software designs, communication and security protocols, real-time systems, etc. Even when a specification has an infinite state space, some of its components may be finite-state. In this case, the inherent modularity of process-algebraic specifications allows one to apply automatic equivalence and preorder checking techniques to these components.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BBK86] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt in mechanism process algebra. *Fundamenta Informatica*, 9:127–168, 1986.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimization. In *Proceedings of CAV '91*, number 663 in LNCS, pages 96–108, 1992.

[BRRdS96] A. Bouali, A. Ressouche, V. Roy, and R. de Simone. The Fc2Tools set. In *Proceedings of Computer-Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 441–445, July 1996.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CC95] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995.

[Cel95] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems*. PhD thesis, North Carolina State University, Raleigh, 1995.

[CH93] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *Proceedings of Computer-Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, July 1996.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proceedings of Computer-Aided Verification (CAV '90)*, June 1990.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of Computer-Aided Verification (CAV '96)*, number 1102 in Lecture Notes in Computer Science, pages 437–440, July 1996.

[GS98] H. Garavel and M. Sighireanu. Towards a second generation of formal description techniques - rationale for the design of e-lotos. In *Proceedings of FMICS'98*, pages 187–230, May 1998.

[GV90] J. F. Groote and F. W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of 17th International Colloquium on Automata, Languages and Programming,*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer-Verlag, 1990.

[Hen88] M. Hennessy. *An Algebraic Theory of Processes.* MIT Press, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall Intl., 1985.

[KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* LNCS 92, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall Intl., 1989.

[Plo81] G. Plotkin. A structural approach to operational semantics. Aarhus University, Computer Science Department, 1981.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[Ros94] W. R. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C.A.R. Hoare.* Prentice-Hall Intl., 1994.

[SLBA99] O. Sokolsky, I. Lee, and H. Ben-Abdallah. Specification and analysis of real-time systems with paragon. *Annals of Software Engineering*, 7:211–234, 1999.

[vGW96] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

Index