1991

H.P. Korver

Computing distinguishing formulas for branching bisimulation

Computer Science/Department of Software Technology Report CS-R9121 March

CWI, nationaal instituut voor onderzoek op het gebied van wiskunde en informatica

CWI is the research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a non-profit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands organization for scientific research (NWO).

Computing Distinguishing Formulas for Branching Bisimulation

Henri Korver
Department of software technology, CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
e-mail: henri@cwi.nl

Abstract

Branching bisimulation is a behavioral equivalence on labeled transition systems which has been proposed by Van Glabbeek and Weijland as an alternative to Milner's observational equivalence. This paper presents an algorithm which, given two branching bisimulation inequivalent finite state processes, produces a distinguishing formula in Hennessy-Milner logic extended with an 'until' operator. The algorithm, which is a modification of an algorithm due to Cleaveland, works in conjunction with a partition-refinement algorithm for deciding branching bisimulation equivalence. Our algorithm provides a useful extension to the algorithm for deciding equivalence because it tells a user *why* certain finite state systems are inequivalent.

Key Words & Phrases: branching bisimulation, Hennessy-Milner logic with Until, transition graph, distinguishing formulas, tools.
1985 Mathematics Subject Classification: 68Q10, 68Q25, 68Q55.
1987 CR Categories: D.2.4, D.2.5, F.1.2, F.2.0, F2.2, F.3.1, F.3.2.
Note: The research of the author is supported by the European Communities under RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS). This article does not necessarily reflect the view of the SPECS project.

1 Introduction

It is a well-known fact that descriptions of concurrent systems often are incorrect, in the sense that the behavior of the implemented system does not correspond to the behavior the designer had in mind. To overcome this problem a lot of research has been directed towards the development of formal verification methods for concurrent systems.

At the moment *behavioral equivalences* are one of the most popular criteria for guaranteeing correctness of concurrent systems. In this approach, concurrent systems are modeled as transition graphs, and verification amounts to establishing that the graph representing the implementation of the system is equivalent to (*behaves the same as*) the graph representing the specification of the system. The main advantage of this approach is that behavioral equivalences can be decided fully automatically on finite transition graphs and that several equivalences can be decided efficiently.

A number of equivalences have been proposed in the literature [2, 5, 9, 14, 15, 19, 23, 25], and several automated tools include facilities for computing them [4, 8, 13, 17, 18, 24].

One particularly interesting equivalence is *bisimulation equivalence* [25], which is a platform for a number of other equivalences that can be described in terms of it [7]. Bisimulation

Report CS-R9121
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

equivalence has a *logical* characterization: two systems are equivalent exactly when they satisfy the same formulas in a simple modal logic due to Hennessy and Milner [20]. This fact suggests a useful diagnostic methodology for tools that compute bisimulation equivalence: when two systems are found not to be equivalent, one may explain why by giving a formula satisfied by one and not by the other.

Hillerström was the first who came with the idea to compute Hennessy-Milner formulas effectively. In his thesis [21] he describes respectively a "maximal" and a "minimal" algorithm for generating distinguishing formulas. The maximal algorithm runs in polynomial time and usually generates very large formulas. The minimal algorithm is back-tracking based, but returns formulas that are considerably smaller. The latter has been implemented in the TAV-system (Tool for Automatic Verification) [17], which is up to now the only bisimulation tool known that explains why two systems are not equivalent.

Recently, Cleaveland developed a more advanced technique to generate distinguishing formulas. His method works in conjunction with a partition-refinement algorithm for computing bisimulation equivalence and is described in [6]. The formulas generated by this algorithm are often minimal in a precisely defined sense.

As a number of other behavioral equivalences may be characterized in terms of bisimulation equivalence, the technique of Cleaveland may be used to generate appropriate distinguishing information for these relations also. We mention here the well-known *observational equivalence* [25], this equivalence may be defined in terms of bisimulation equivalence on a suitably transformed transition system [3]. Observational equivalence has also a logical characterization, namely the Weak Hennessy-Milner Logic [20]. The method of Cleaveland can be used to generate distinguishing formulas for observational equivalence, just by applying the algorithm to the transformed transition system.

As an alternative for observational equivalence, *branching bisimulation equivalence* has been proposed in [15]. This equivalence resembles, but is finer than observational equivalence. In fact, the definition of branching bisimulation is just a natural restriction of the definition of observational equivalence, in the sense that all intermediate states have to be related as well [15]. Besides the naturalness of definition, there are a number of recent results that indicate that branching bisimulation has very nice properties. The following list of properties is taken from [18].

* The axiomatization of branching bisimulation is simpler than the axiomatization of observational equivalence [15].
* Unlike observational equivalence, the axiomatization of branching bisimulation can be transformed easily to a complete term rewriting system [1].
* Branching bisimulation is characterized by Hennessy-Milner Logic extended with a kind of until operator [10].
* Branching bisimulation may be characterized by *back and forth bisimulations* [10]. This characterization leads to a second modal characterization of branching bisimulation which is a variant of Hennessy-Milner logic extended with backward modalities [10].
* Branching bisimulation is the natural analogue of stuttering equivalence in case the transitions rather than the states are labeled. In this setting CTL and CTL* without the nexttime operator can be viewed as logics for branching bisimulation [10]. In [11]

action based versions of CTL and CTL* without nexttime operator are proposed which also correspond with branching bisimulation.

* In contrast with observational equivalence, branching bisimulation is preserved under refinement of actions [16].
* Branching bisimulation is computed by an algorithm that has been implemented [18] and turns out to be more efficiently computable than observational equivalence in practice [12, 18].
* For a large class of processes, branching bisimulation equivalence and observational equivalence are the same [15]. We do not know any real life protocol that can be verified by observational equivalence and not by branching bisimulation equivalence.

In this paper, we take Hennessy-Milner Logic extended with an Until-operator [10] as a characterization of branching bisimulation to compute logical formulas to differentiate between branching bisimulation-inequivalent systems. For this purpose, it is not possible to apply the algorithm of Cleaveland directly, because the efficient algorithm [18] to decide branching bisimulation is not based on bisimulation equivalence on a transformed transition system.

The intention of this paper is to develop a technique for determining a Hennessy-Milner formula with Until-operator that distinguishes two branching bisimulation inequivalent finite-state systems, using the idea of the advanced method of Cleaveland. To this end, we show how to use information generated by an adapted version of the partition-refinement algorithm of Groote and Vaandrager [18] to compute such a formula efficiently. On the basis of this result, tools using branching bisimulation may be modified to give users diagnostic information in the form of a distinguishing formula when a system is found not to be equivalent to its specification.

The remainder of the paper is organized as follows. The next section defines branching bisimulation equivalence and examines the connection between it and the Hennessy-Milner Logic with Until. Section 3.1 describes the algorithm of Groote and Vaandrager to compute branching bisimulation equivalence on the states of a transition graph. Then section 3.2 describes how to generate a block tree which retains information computed by the equivalence algorithm. Finally, in section 3.3 it is shown how to compute distinguishing formulas on the basis of this block tree; a small example is also presented to illustrate the working of the new algorithm.

A nice property of our algorithm is its unexpectedly simple presentation. In contrast with this, its correctness is not quite straightforward and even tricky at some places. A detailed proof is included as an appendix.

2 Transition Graphs, Branching Bisimulation and HMLU

Concurrent systems are often modeled by *transition graphs*. Vertices in these graphs correspond to the states a system may enter as it executes, with one vertex being distinguished as the start state. The edges, which are directed, are labeled with actions and represent the state transitions a system may undergo. The formal definition is the following.

Definition 2.1 *A labeled transition graph is a quadruple $<S, s, Act, \rightarrow>$, where:*

- *S is a set of states;*
- *$s \in S$ is the start state;*
- *Act is a set of actions; the silent action τ is not in Act; and*
- *$\rightarrow \subseteq S \times Act_\tau \times S$ is the transition relation where $Act_\tau = Act \cup \{\tau\}$. An element $(p, \alpha, q) \in \rightarrow$ is called a transition, and is usually written as $p \xrightarrow{\alpha} q$.*

The silent action τ is unobservable for the environment and is used to symbolize the internal behavior of the system.

When a graph does not have a start state indicated, we shall refer to the corresponding triple as a *transition system.* A state in a transition system gives rise to a transition graph in the obvious way: let the given state be the start state, with the three components of the transition graph coming from the transition system.

Transition graphs are often too concrete for representing concurrent systems. Mostly one is only interested in the observational behavior of a complicated system and one is not interested in the internal (low-level) computations. Branching bisimulation, which is an interesting alternative for the well-known observational equivalence [25], remedies this shortcoming. In [15] several definitions of branching bisimulation are given, which all lead to the same equivalence. The following definition is in our setting the most suitable one.

Definition 2.2 *(Branching bisimulation)*

- *Let $<S, Act, \rightarrow>$ be a transition system. A relation $R \subseteq S \times S$ is called a branching bisimulation if it is symmetric and satisfies the following transfer property:*

  If rRs and $r \xrightarrow{\alpha} r'$, then either $\alpha{=}\tau$ and $r'Rs$ or; $\exists s_0, .., s_n, s' \in S : s = s_0$, $[\forall_{0<i\leq n} : s_{i-1} \xrightarrow{\tau} s_i]$ and $s_n \xrightarrow{\alpha} s'$ such that $\forall_{0<i\leq n}\ rRs_i$ and $r'Rs'$.

- *Two states r and s are branching bisimilar, abbreviated $r \approx_B s$ or $s \approx_B r$, if there exists a branching bisimulation relating r and s.*

The arbitrary union of branching bisimulation relations is again a branching bisimulation; $\approx_B$ is the maximal branching bisimulation and is an equivalence relation.

Let $T_1 =< S_1, s_1, Act, \rightarrow_1>$ and $T_2 =< S_2, s_2, Act, \rightarrow_2>$ be two transition graphs satisfying $S_1 \cap S_2 = \emptyset$. Then T_1 and T_2 are branching bisimilar exactly when the two start states, s_1 and s_2 are branching bisimilar in the transition graph $< S_1 \cup S_2, Act, \rightarrow_1 \cup \rightarrow_2>$. In [10] it is proved that branching bisimulation has a *logical* characterization in terms of the Hennessy-Milner Logic with Until (HMLU): two states are equivalent exactly when they satisfy the same set of HMLU-formulas. This logic is a simple modal logic; the syntax of formulas is defined by the following grammar, where $\alpha \in Act_\tau$.

$$\Phi ::= tt \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi\langle\alpha\rangle\Phi$$

The formal semantics of the logic is given with respect to a transition system T = ⟨S, Act, →⟩

$$
\begin{aligned}
[\![tt]\!]_T &= S \\
[\![\neg\Phi]\!]_T &= S - [\![\Phi]\!]_T \\
[\![\Phi_1 \wedge \Phi_2]\!]_T &= [\![\Phi_1]\!]_T \cap [\![\Phi_2]\!]_T \\
[\![\Phi_1\langle\alpha\rangle\Phi_2]\!]_T &= \{s \in S |\ (\alpha = \tau \textit{ and } s \in [\![\Phi_2]\!]_T) \textit{ or} \\
&\qquad (\exists s_0, \ldots, s_n \in [\![\Phi_1]\!]_T, \exists s' \in [\![\Phi_2]\!]_T : s_0 = s, \\
&\qquad [\forall_{0<i\leq n} : s_{i-1} \xrightarrow{\tau} s_i] \textit{ and } s_n \xrightarrow{\alpha} s')\}
\end{aligned}
$$

Figure 1: The semantics of formulas in Hennessy-Milner Logic with Until.

and appears in Figure 1.

In Figure 1 each formula is mapped to the set of states for which the formula is "true". In the remainder of the paper we shall omit explicit reference to the transition system used to interpret formulas when it is clear from the context.

Intuitively, the formula tt holds in any state, and $\neg\Phi$ holds in a state if Φ does not. The formula $\Phi_1 \wedge \Phi_2$ holds in a state if both Φ_1 and Φ_2 do. The until-proposition $\Phi_1\langle\alpha\rangle\Phi_2$ holds in a state, if this state can reach via α, a state in which Φ_2 holds while moving through intermediate states in which Φ_1 holds.

Let $\mathcal{H}(s)$ be the set of HMLU-formulas that are valid in state s:

$$\mathcal{H}(s) = \{\Phi | s \in [\![\Phi]\!]\}.$$

The next theorem is a specialization of a theorem proved in [10].

Theorem 2.3 *Let $< S, Act, \rightarrow >$ be a finite-state transition system, with $s_1, s_2 \in S$. Then $\mathcal{H}(s_1) = \mathcal{H}(s_2)$ if and only if $s_1 \approx_B s_2$.*

It follows that if two states in a (finite-state) transition system are inequivalent, then there must be a HMLU-formula satisfied by one and not the other. This is the basis of our definition for distinguishing formula, although we shall in fact use the following, slightly more general formulation taken from [6].

Definition 2.4 *Let $< S, Act, \rightarrow >$ be a transition system, and let $S_1 \subseteq S$ and $S_2 \subseteq S$. Then HMLU-formula Φ distinguishes S_1 from S_2 if the following hold.*

1. $S_1 \subseteq [\![\Phi]\!]$.
2. $S_2 \cap [\![\Phi]\!] = \emptyset$.

So Φ distinguishes S_1 from S_2 if every state in S_1, and no state in S_2, satisfies Φ. Theorem 2.3 thus guarantees the existence of a formula that distinguishes $\{s_1\}$ from $\{s_2\}$ if $s_1 \approx_B s_2$.

Finally, we take the following criterion from [6] to indicate whether a distinguishing formula contains extraneous information.

Definition 2.5 *Let Φ be a HMLU-formula distinguishing S_1 from S_2. Then Φ is minimal if no Φ' obtained by replacing a non-trivial subformula of Φ with the formula tt distinguishes S_1 from S_2.*

Intuitively, Φ is a minimal formula for S_1 with respect to S_2 if each of its subformulas plays a role in distinguishing the two.

3 Computing Distinguishing Formulas

In this section, we describe a partition refinement algorithm for computing branching bisimulation equivalence and show how to alter it to generate a block tree. Then given such a block tree, we describe how to generate distinguishing formulas. Finally, a small example is given that illustrates the use of the algorithm.

3.1 Computing Branching Bisimulation

Nowadays, "partition-refinement" is the most efficient method to compute bisimulation equivalences [22, 26]. A partition-refinement algorithm exploits the fact that an equivalence relation on the set of states may be represented as a partition, or a set of pairwise-disjoint subsets (called blocks) of the state set whose union is the whole state set. In this representation blocks correspond to the equivalence classes, so two states are equivalent exactly when they belong to the same block. Beginning with the partition containing one block (representing the trivial equivalence relation consisting of one equivalence class), the algorithm repeatedly *refines* a partition by splitting blocks until the associated equivalence relation becomes a bisimulation.

In [18] the refinement strategy to obtain branching bisimulation is described. To refine the current partition, the algorithm of Groote and Vaandrager looks at each block in turn. If a state in block B can reach via α, possibly after some initial stuttering, a state in block B' [1] and another state in B does not, then the algorithm splits B into two blocks. The first block contains all the states which can reach via α, possibly after some initial stuttering, a state in block B'. The second block contains all the other states. When no more splitting is possible, the resulting equivalence corresponds exactly to branching bisimulation on the given transition system.

Below we present the definitions and the algorithm in more formal notation; the description is a slight modification of the one in [18].

Definition 3.1
Let $\langle S, Act, \rightarrow \rangle$ be a transition system.

1. *A collection $\{B_j | j \in J\}$ of nonempty subsets of S is called a partition if $\bigcup_{j \in J} B_j = S$ and for $i \neq j : B_i \cap B_j = \emptyset$. The elements of a partition are called blocks.*
2. *If $\mathcal{P}$ and $\mathcal{P}'$ are partitions of S then $\mathcal{P}'$ refines $\mathcal{P}$, if any block of $\mathcal{P}'$ is included in a block of $\mathcal{P}$.*
3. *The equivalence $\sim_{\mathcal{P}}$ on S induced by a partition $\mathcal{P}$ is defined by: $r \sim_{\mathcal{P}} s \Leftrightarrow \exists B \in \mathcal{P} : r \in B \wedge s \in B$.*

[1] When $B = B'$, α is assumed not to be equal to τ.

4. *For B, B' we define the set $pos_\alpha(B, B')$ as the set of states in B from which, after some internal τ-stuttering, a state in B' can be reached:*
$pos_\alpha(B, B') = \{s \in B | \exists s_0, ..., s_n \in B, \exists s' \in B' : s_0 = s, [\forall_{0<i\leq n} : s_{i-1} \xrightarrow{\tau} s_i]$ *and* $s_n \xrightarrow{\alpha} s'\}$

5. *We say that B' is a splitter of B with respect to action α iff*

$B \neq B'$ *or* $\alpha \neq \tau$, *and* $\emptyset \neq pos_\alpha(B, B') \neq B$.

6. *If $\mathcal{P}$ is a partition of S and B' is a splitter of B with respect to α, then $Ref_p^\alpha(B, B')$ is the partition $\mathcal{P}$ where B is replaced by $pos_\alpha(B, B')$ and $B - pos_\alpha(B, B')$.*

7. *$\mathcal{P}$ is stable with respect to block B' if for no block B and for no action α, B' is a splitter of B with respect to α. $\mathcal{P}$ is stable if it is stable with respect to all its blocks.*

Algorithm 3.2 *The algorithm to compute branching bisimulation maintains a partition $\mathcal{P}$ that is initially $\mathcal{P}_0 = \{S\}$. It repeats the following step, until $\mathcal{P}$ is stable:*

> *Find blocks $B, B' \in \mathcal{P}$ and a label $\alpha \in Act_\tau$ such that B' is a splitter wrt. α;*
> $\mathcal{P} := Ref_p^\alpha(B, B')$.

The next theorem guarantees that the equivalence induced by the last partition computed by algorithm 3.2 corresponds exactly to branching bisimulation equivalence and is proved in [18].

Theorem 3.3 *Let $\langle S, Act, \rightarrow \rangle$ be a finite transition system. Let $\mathcal{P}_f$ be the final partition obtained by the algorithm above. Then $\sim_{\mathcal{P}_f} = \approx_B$.*

In [18] the following complexity bounds are given.

Theorem 3.4 *The time complexity of algorithm 3.2 is $O(|S| * | \rightarrow |)$. And the space complexity is $O(| \rightarrow |)$.*

These complexity measures are not obvious from the conceptual description of the algorithm above. Therefore a more implementation oriented view of the algorithm is given in [18]. However in this paper these details are not relevant.

3.2 Generating The Block Tree

In addition to computing the partition as described above, we now retain information about *how* and *why* the blocks are split by construction of a labeled block "tree". The following definitions are used to describe the generation procedure of such a block tree.

Definition 3.5 *(Parent and its children)*
$\mathcal{P}(B)$ is the parent of block B in the block tree. $\mathcal{L}(B)$ is the left child of block B. $\mathcal{R}(B)$ is the right child of block B. In case block B has a copy of itself as single child then $\mathcal{L}$ and $\mathcal{R}$ are applied to the highest [2] copy (B^*) of B in the tree. This means that $\mathcal{L}(B) = \mathcal{L}(B^*)$ and $\mathcal{L}(B) = \mathcal{L}(B^*)$. When P has no children then $\mathcal{L}(B)$ and $\mathcal{R}(B)$ are undefined.

[2] See definition 3.6.

Definition 3.6 *(Height of a block)*
The height of a block B in the block tree is defined as follows:
$h(B) := 0$ *where B is the root block.*
$h(B) := 1 + h(\mathcal{P}(B))$.

Definition 3.7 *(Parent Partition)*
The Parent Partition of block B in the block tree, is the partition where B is created.
$\mathcal{PP}(B) := \{C |\ h(C) = h(\mathcal{P}(B))\}$.

Definition 3.8 *(Blocks that can be reached from block B)*
Let B be a block in the block tree.

- $r_\alpha(B) := \{C \in \mathcal{PP}(B) |\ \exists s \in B, s' \in C : s \xrightarrow{\alpha} s'\}$; $r_\alpha(B)$ *contains all the blocks in the parent partition of B that can be reached from a state in B via α.*
- $r^p_\tau(B) := \{C \in \mathcal{PP}(B) |\ \exists s \in B, s' \in C : s \xrightarrow{\tau} s' \wedge C \neq \mathcal{P}(B)\}$; $r^p_\tau(B)$ *contains all the blocks in the parent partition of B that can be reached from a state in B via τ. The superscript "p" indicates that the parent of B is not included.*

Algorithm 3.2 is modified as follows. Rather than discarding an old partition after it is refined, the new procedure constructs a tree of blocks as follows. The children of a block are the new blocks that result when the algorithm splits the block; accordingly, the root is labeled with the block S, and after each refinement the leaves of this tree represents the current partition.

When a block P is split due to splitter block B' and action α, we position the new block $L{=}pos_\alpha(P, B')$ as the left child and the new block $R{=}P - pos_\alpha(P, B')$ as the right child, and we label the arc connecting P to L with α and B'. We label the arc connecting P to R with $r^p_\tau(R)$ and $r_\alpha(R)$, these block-sets are given in definition 3.8. The blocks in $r^p_\tau(R)$ bear witness to the states in R that cannot evolve in internal stuttering. The blocks in $r_\alpha(R)$ bear witness to the states in R that cannot reach the splitter block by an α-step.

Recall that every state in L can reach via α, possibly after some initial stuttering, a state in B' and no state in R does. If a block is not split during a refinement, it is assigned a copy of itself as its only child Figure 3 contains an example of such a tree.

The construction of the block tree during the partition-refinement algorithm does not influence the time complexity. The space complexity has changed slightly from $O(|\rightarrow|)$ to $O(|S|^2)$ due to the following theorem (note that $|\mathcal{P}_f| \leq |S|$).

Theorem 3.9 *The space requirement of the labeled block tree is $O(|S| + |P_f|^2)$.*

Proof. Strictly speaking, only the leaves in the tree need to be labeled with the corresponding sets of states, and therefore the recording of state-sets requires $O(|S|)$ space.

The space requirement of the labels on the arcs is bounded by $O(|P_f|^2)$, due to the following observation. The number of left and right arcs is always $O(|P_f|)$, because the nodes that have a single child may be left childless [3]. The labels on a left arc always require a constant amount of memory. A right arc may contain at worst all the blocks of the current partition for each

[3] We included spurious children in the definition to simplify our inductive argument of correctness.

label; this means a space requirement of no more than $2 * |P_f|$ block-pointers. Hence, the space requirement of the labels on the arcs is bounded by $O(|P_f|^2)$.

All together we have a total memory requirement of $O(|S| + |P_f|^2)$ for the labeled block tree. □

3.3 Generating Distinguishing Formulas

Given a block tree computed by the extended partition-refinement algorithm above, and two disjoint blocks B_1 and B_2, the following postprocessing step builds a formula $\Delta(B_1, B_2)$ that distinguishes the states in B_1 from those in B_2.

First we compute the lowest common ancestor of B_1 and B_2 (and call it P). By lemma 3.11 we know that a formula distinguishing the children of P, also distinguishes B_1 from B_2.

Definition 3.10 *(Lowest Common Ancestor)*
The function $\mathcal{LCA}$ returns the Lowest Common Ancestor of two disjoint blocks B_1 and B_2 in the block tree.

Lemma 3.11

$$\left.\begin{array}{l} P = \mathcal{LCA}(B_1, B_2) \\ \Phi \text{ distinguishes } \mathcal{L}(P) \text{ and } \mathcal{R}(P) \end{array}\right\} \wedge \implies \Phi \text{ distinguishes } B_1 \text{ and } B_2.$$

Proof. B_1 and B_2 are subsets of respectively $\mathcal{L}(P)$ and $\mathcal{R}(P)$.

For the sake of short notation, let $L = \mathcal{L}(P)$ be the left child and $R = \mathcal{R}(P)$ the right child of P. The arc connecting blocks P and L is labeled with α and B'; and the arc connecting blocks P and R is labeled with respectively the block-sets $r^p_\tau(R)$ and $r_\alpha(R)$ (call these block-sets respectively r_1 and r_2).

From the way that the block tree is generated, we know that every state in L can reach via α, possibly after some initial stuttering, a state in B' and that no state in R does. Accordingly, one recursively builds formulas that distinguish P and blocks in r_1, and takes their conjunction (call it Φ_1). And, if one also recursively builds formulas that distinguish B' from each block in r_2 and also takes their conjunction (call it Φ_2), then every state in L satisfies $\Phi_1\langle\alpha\rangle\Phi_2$ (call this formula Φ) and no state in R does. In case $\alpha = \tau$, one has to add the extra conjunct $\Delta(B', R)$, to ensure that Φ is a distinguishing formula; this is caused by the first disjunct at the right hand side of the last mapping in figure 1. The details are given below.

Algorithm 3.12
When B_1 and B_2 are disjoint, $\Delta(B_1, B_2)$ can be computed recursively as follows.

1. *Compute $P := \mathcal{LCA}(B_1, B_2)$.*
2. *Let $L := \mathcal{L}(P)$; $R := \mathcal{R}(P)$.*
   (Notice that $B_1 \subseteq L$ and $B_2 \subseteq R$, or $B_2 \subseteq L$ and $B_1 \subseteq R$.)
3. *Let α and B' be the labels on the arc connecting P and L; and*
   let $r_1 := r^p_\tau(R)$ and $r_2 := r_\alpha(R)$ be the labels on the arc connecting P and R;

- *if* $r_1 = \emptyset$ *then* $\Phi_1 := tt$ *else* $\Phi_1 := \bigwedge_{C \in r_1} \Delta(P, C)$;
- *if* $r_2 = \emptyset$ *then* $\Phi_2 := tt$ *else* $\Phi_2 := \bigwedge_{C \in r_2} \Delta(B', C)$.

4. *If* $\alpha \neq \tau$ *then* $\Phi := \Phi_1 \langle \alpha \rangle \Phi_2$.
   else $\Phi_3 := \Delta(B', R)$;
   $\Phi := \Phi_1 \langle \tau \rangle (\Phi_2 \wedge \Phi_3)$.

5. *If* $B_1 \subseteq L$ *then return* Φ *else return* $\neg\Phi$.

We now have the following theorem.

Theorem 3.13 $B_1 \cap B_2 = \emptyset \implies \Delta(B_1, B_2)$ *distinguishes* B_1 *and* B_2.

Proof. By induction on the depth of B_1 and B_2 in the block tree (see appendix).

It should be noted that exponential length formulas may be generated. However, one may present such a formula (as a set of propositional equations) in space proportional to $|\mathcal{P}_f|^2$, where $\mathcal{P}_f$ is the final partition computed by the algorithm (note that $|\mathcal{P}_f| \leq |S|$). This results from the fact that there can be at most $|\mathcal{P}_f| - 1$ recursive calls generated by the above procedure and the fact that each distinguishing formula is of the form $(\neg)\Phi_1 \langle \alpha \rangle \Phi_2$, where Φ_1 and Φ_2 contain together at most $|\mathcal{P}_f| - 1$ conjuncts, each of the form $\Delta(B_i, B_j)$ for some B_i and B_j.

Theorem 3.14 *An equational representation of* $\Delta(B_1, B_2)$ *may be calculated in* $O(|\mathcal{P}_f|^2)$ *time, once the tree of blocks has been computed.*

Proof. At each recursive call, computing the lowest common ancestor requires at most $O(|\mathcal{P}_f|)$ work. □

In general a formula $\Delta(s_1, s_2)$ will not be minimal in the sense of definition 2.5. In [6] the following straightforward procedure is proposed to minimize $\Delta(s_1, s_2)$ once it has been computed. Repeatedly replace subformulas in the formula by *tt* and see if the resulting formula still distinguishes s_1 from s_2. If so, the subformula may either be omitted (if it is one of several conjuncts in a larger conjunction) or left at *tt*. The result of this would be a minimal formula. The computational tractability of this procedure remains to be examined.

We close this subsection with a general remark about our method. Our method generates a formula that distinguishes blocks that may contain more than one state, but mostly one is only interested in a formula that distinguishes two particular states. In this case, algorithm 3.12 can also be used to construct a formula distinguishing two inequivalent states s_1 and s_2; first locate the disjoint blocks B_1 and B_2 such that $s_i \in B_i$ $(i = 1, 2)$, then build $\Delta(B_1, B_2)$.

3.4 An Example

To illustrate our algorithm we consider two transition graphs that are not branching bisimulation equivalent. Figure 2 shows the transition system that includes the two transition graphs. It is interesting to notice that these two graphs are an instance of the second τ-law of observational equivalence (see e.g. [25]); so they are not differentiated by HML without

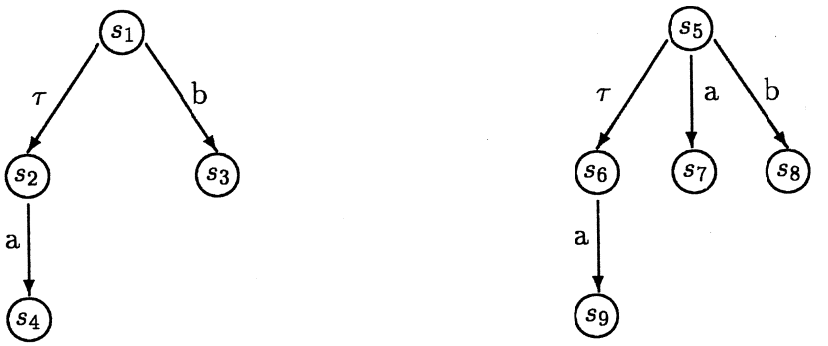

Figure 2: Two branching bisimulation inequivalent transition graphs.

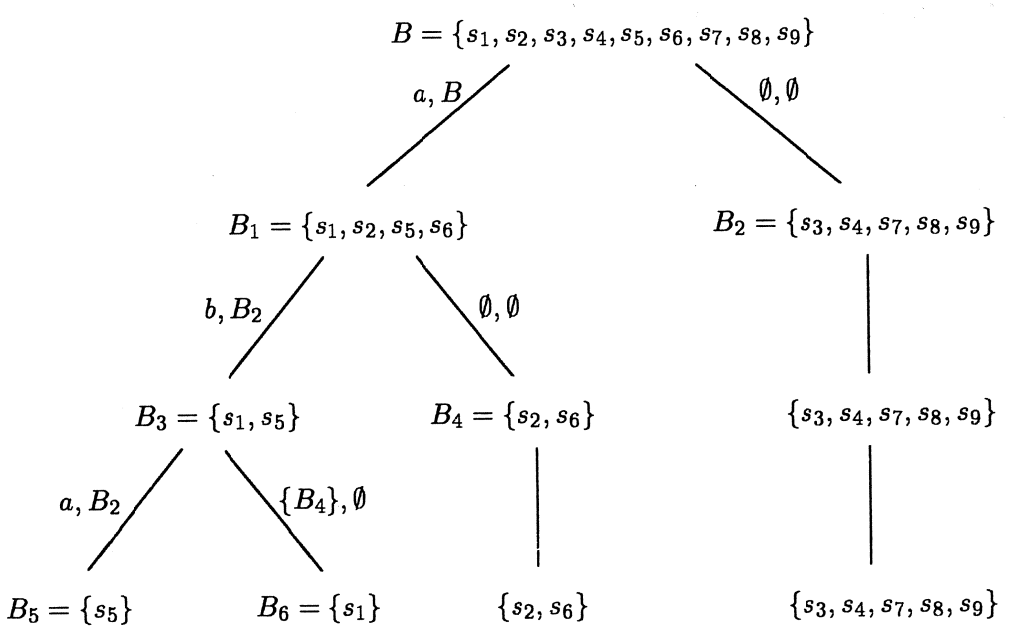

Figure 3: The generated tree of blocks.

Until-operator. State s_1 is the start state of one graph, while state s_5 is the start state of the other. Figure 3 contains a tree of blocks generated by the altered partition-refinement algorithm. Notice that $s_1 \not\approx_B s_5$, as they are in different blocks. In order to build a formula that s_1 satisfies and s_5 does not, it suffices to generate $\Delta(B_6, B_5)$, the formula that distinguishes block B_6 and B_5. To do so, the algorithm first locates the lowest common ancestor of the two blocks (B_3, in this case). The left child is B_5 and the right child is B_6. The labels on the left arc indicate that the action causing the split is a, and the splitter block is B_2. The labels on the right arc indicate that $r^p_\tau(R)=\{B_4\}$ and $r_\alpha(R)=\emptyset$. The formula that will be returned, then, will be

$$\neg(\Delta(B_3, B_4)\langle a\rangle tt);$$

this formula holds of s_1 and not of s_5. By repeating this process, it turns out that

$$\Delta(B_3, B_4) \quad = \quad tt\langle b\rangle tt$$

So the formula distinguishing s_1 from s_5 is

$$\neg((tt\langle b\rangle tt)\langle a\rangle tt).$$

This formula explains why s_1 and s_5 are inequivalent because s_5 may engage in an a-transition while in all the intermediate states (only s_5 here) a b-transition is available. This is not the case for state s_1. Note that this formula is minimal.

4 Conclusions and Future Work

This paper has shown how it is possible to alter the partition-refinement of Groote and Vaandrager for computing branching bisimulation equivalence to compute a formula in the Hennessy-Milner Logic with Until that distinguishes two inequivalent states. The generation of the formula relies on a postprocessing step that is invoked on a tree-based representation of the information computed by the equivalence algorithm. The postprocessing step has no effect on the worst-case complexity of the equivalence-checking algorithm, only the space complexity has changed slightly from $O(|\rightarrow|)$ to $O(|S|^2)$.

The most important direction for future work is tackling the problem of generating minimal formulas and moreover its complexity. Clearly, the complexity of the minimization procedure mentioned in passing at the end of section 3.3 needs to be analyzed fully; if this procedure is efficient enough, then it may be incorporated into the distinguishing formula generation procedure.

Another area of investigation would be an implementation of our technique, as an extension of the equivalence-checking algorithm of Groote and Vaandrager which is already implemented successfully.

Acknowledgements

I would like to thank Jos Baeten, Rance Cleaveland and Frits Vaandrager for their helpful comments. Special thanks to Jan Friso Groote for spotting bugs in previous drafts of this paper.

References

[1] G.J. Akkerman and J.C.M. Baeten: Term Rewriting Analysis in Process Algebra. In *report P9006*, Programming Research Group, Univerisity of Amsterdam, 1990.

[2] B. Bloom, S. Istrail and A. Meyer: Bisimulation Can't Be Traced. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988.

[3] T. Bolognesi and S.A. Smolka: Fundamental results for the verification of observational equivalence: a survey. In H. Rudin and C. West, editors, *Proceedings 7^{th} IFIP WG6.1 International Symposium on Protocol Specification, Testing, and Verification,* Zürich, Switserland, May 1987. North-Holland, 1987.

[4] G. Boudol, V. Roy, R. de Simone and D. Vergamini: Process Algebras and Systems of Communicating Processes. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, Lecture Notes in Computer Science 407, 1–10. Springer-Verlag, Berlin, 1990.

[5] S.D. Brookes, C.A.R Hoare and A.W. Roscoe: A Theory of Communicating Sequential Processes. In *Journal of the ACM*, v. 31, n. 3, pages 560-599, 1984.

[6] R. Cleaveland: On Automatically Distinguishing Inequivalent Processes. In *Proceedings: 1990 Workshop on Computer-Aided Verification (R. Kurshan and E.M. Clarke, editors)*, DIMACS technical report 90-31, Vol. 2, New Yersey, 1990. To appear in Lecture Notes in Computer Science.

[7] R. Cleaveland and M. Hennessy: Testing Equivalence as a Bisimulation Equivalence. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, Lecture Notes in Computer Science 407, 11–23. Springer-Verlag, Berlin, 1990.

[8] R. Cleaveland, J. Parrow and B. Steffen: A Semantics-Based Tool for the Verification of Finite-State Systems. In *Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing and Verification*, 287–302. North-Holland, Amsterdam, 1990.

[9] R. DeNicola and M. Hennessy: Testing Equivalences for Processes. *Theoretical Computer Science*, v. 34, pages 83-133, 1983

[10] R. DeNicola and F.W. Vaandrager: Three logics for branching bisimulation (extended abstract). In *Proceedings 5^{th} Annual Symposium on Logic in Computer Science,* Philadelphia, USA, pages 118–129, Los Alamitos, CA, 1990. IEEE Computer Society Press. Full version appeared as CWI Report CS-R9012.

[11] R. DeNicola and F.W. Vaandrager: Action versus state based logics for transition systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, April 90, (I. Guessarian), Springer-Verlag, LNCS, 1990.

[12] P. Ernberg, L. Fredlund and B. Jonsson: Specification and Validation of a Simple Overtaking Protocol using LOTOS, *SICS technical report*, T90006, ISSN 1100-3154, 1990.

[13] J.C. Fernandez: Aldébaran, Un Système de Vérification par Réduction de Processus Communicants. *Ph.D. Thesis*, Université de Grenoble, 1988.

[14] R.J. van Glabbeek: The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[15] R.J. van Glabbeek and W.P. Weijland: Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989.

[16] R.J. van Glabbeek and W.P. Weijland: Refinement in branching time semantics. *Report CS-R8922*, CWI, Amsterdam, 1989. Also appeared in *Proceedings AMAST Conference*, Iowa, USA, pp. 197–201.1989.

[17] J. Godskesen, K. Larsen and M. Zeeberg: TAV User Manual. *Technical report R 89-19*, 1989.

[18] J.F. Groote and F.W. Vaandrager: An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *LNCS*, pages 626–638. Springer-Verlag, 1990.

[19] M. Hennessy: *Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[20] M. Hennessy and R. Milner: Algebraic Laws for Nondeterminism and Concurrency. *Journal of the Association for Computing Machinery*, v. 32, n. 1, pages 137-161, January 1985.

[21] M. Hillerström: Verification of CCS-processes. *M.Sc. Thesis*, Computer Science Department, Aalborg University, 1987.

[22] P. Kanellakis and S.A. Smolka: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second ACM Symposium on the Principles of Distributed Computing*, 1983.

[23] K. Larsen and A. Skou: Bisimulation through Probabilistic Testing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1989.

[24] J. Malhotra, S.A. Smolka, A. Giacalone and R. Shapiro: Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Proceedings of the Workshop on Specification and Verification of Concurrent Systems*, University of Stirling, Scotland, 1988.

[25] R. Milner: *Communication and Concurrency.* Prentice Hall, 1989.

[26] R. Paige and R.E. Tarjan: Three Partition Refinement Algorithms. In *SIAM Journal of Computing*, v. 16, n. 6, pages 973-989, December 1987.

Appendix: Proof

The main theorem 3.13 will be proved by induction on the minimal height of B_1 and B_2.

Definition 1
The minimal height of two blocks in the block tree is defined as:
$h_m(B_1, B_2) := min(h(B_1), h(B_2))$.

The following lemmas are simple, but crucial for our proof. The first lemma is used to show that recursive calls are at least one level lower in the tree.

Lemma 1

- $h(\mathcal{LCA}(B_1, B_2)) < h(B_1)$
- $h(\mathcal{LCA}(B_1, B_2)) < h(B_2)$

Proof. By construction of the block tree.

Lemma 2
$h_m(B_1, B_2) = 0 \implies B_1$ *or* B_2 *is the root block.*

Proof. By construction of the block tree.

Lemma 3
$$\left.\begin{array}{l} \Phi \text{ distinguishes } B_1 \text{ and } B_2 \quad \wedge \\ s \in [\![\Phi]\!] \end{array}\right\} \implies s \notin B_2.$$

Proof. Directly by definition 2.4.

Lemma 4
$$\left.\begin{array}{l} B_1 \cap B_2 = \emptyset \qquad \wedge \\ P = \mathcal{LCA}(B_1, B_2) \end{array}\right\} \implies \mathcal{L}(P) \text{ and } \mathcal{R}(P) \text{ are defined.}$$

Proof. By construction of the block tree.

Lemma 5
Let P be a block in the block tree and $L = \mathcal{L}(P)$ and $R = \mathcal{R}(P)$.
$\forall s \in R: \; s \xrightarrow{\tau} s'$ *implies* $s' \notin L$.

Proof. By definition of splitting.

Lemma 6
Let P be a block in the block tree.
If P and α are the labels on the arc from P to $\mathcal{L}(P)$ then $\alpha \neq \tau$.

Proof. By definition 3.1 (v).

Lemma 7
Let D_1 and D_2 be blocks in the block tree.
(i) $h(D_1) = h(D_2) + 1 \quad \wedge \quad D_1 \neq \mathcal{P}(D_2) \quad \Longrightarrow \quad D_1 \cap D_2 = \emptyset.$
(ii) $h(D_1) = h(D_2) \quad \wedge \quad D_1 \neq D_2 \quad \Longrightarrow \quad D_1 \cap D_2 = \emptyset.$

Proof. By construction of the block tree.

Lemma 8
Let P be a block in the block tree, and let $R = \mathcal{R}(P)$.
$\bigcup r_\tau(R) \subseteq \bigcup r_\tau^p(R) \cup \{R\}$

Proof. By lemma 5.

Main Theorem $B_1 \cap B_2 = \emptyset \implies \Delta(B_1, B_2)$ *distinguishes* B_1 and B_2.

Proof. Given two blocks B_1 and B_2 in a block tree generated by the extended partition refinement algorithm, then in case B_1 and B_2 are disjoint, algorithm 3.12 returns a formula that distinguishes B_1 and B_2. This will be proved by induction on the minimal height of two blocks in the block tree.

Basis: The base case is that $h_m(B_1, B_2) = 0$. Then by lemma 2, block B_1 or block B_2 must be the root block of the block tree. This means that B_1 and B_2 can never be disjoint and therefore the implication of the theorem is always true.

Induction Hypothesis: We assume that all formulas $\Delta(B_1, B_2)$ with $h_m(B_1, B_2) < h$ and $B_1 \cap B_2 = \emptyset$, distinguish B_1 and B_2.

Induction: We show that formulas $\Delta(B_1, B_2)$ with $h_m(B_1, B_2) \leq h$ and $B_1 \cap B_2 = \emptyset$, distinguish B_1 and B_2. In step 1 of the algorithm 3.12, block P is the lowest common ancestor of blocks B_1 and B_2. According to step 2 and lemma 4, block P is connected to blocks L and R by respectively a left and a right arc. According to step 3, action α and splitter B' are the labels on the left arc; $r_1 = r_\tau^p(R)$ and $r_2 = r_\alpha(R)$ are the labels of the right arc. If r_1 is empty then formula Φ_1 is "true", else formula Φ_1 distinguishes block P and all blocks in r_1 by the induction hypothesis. The application of the induction hypothesis is correct because its conditions hold. The first condition holds by lemma 1 and the fact that all blocks in r_1 have the same height as lowest common ancestor P. The second condition holds by lemma 7 (ii); note that the second conjunct of this lemma holds because P cannot be in r_1 by definition of r_1.

In the same way, if r_2 is empty then formula Φ_2 is "true", else formula Φ_2 distinguishes block B' and all blocks in r_2 by the induction hypothesis. The application of the induction hypothesis is correct because its conditions hold. The first condition holds by lemma 1 and the fact that block B' has the same height as lowest common ancestor P. The second condition holds by lemma 7 (ii); note that B' is disjoint with all the blocks in r_2 by definition of splitting. Step 4 of the algorithm distinguishes two cases.

I) In case $\alpha \neq \tau$, it is obvious that Φ holds in all the states of L. However, Φ does not hold in any state of R. We prove this by reductio ad absurdum. Suppose $\exists s \in R : s \in [\![\Phi]\!]$. Then by definition of $[\![\Phi]\!]$ this state s has to satisfy the following property:

(i) $\exists s_0, ..., s_n \in [\![\Phi_1]\!], \exists s' \in [\![\Phi_2]\!] : s_0 = s, [\forall_{0<i\leq n} : s_{i-1} \xrightarrow{\tau} s_i]$ *and* $s_n \xrightarrow{\alpha} s'$.

To derive the contradiction, we first prove that s_n must always be in R by induction on the index of state s_n.

> Basis: $s_0 = s \in R$.
> Induction Hypothesis: $s_{i-1} \in R$.
> Induction: By the i.h. s_{i-1} is in R. The state s_i is a τ-successor of s_{i-1}, and therefore s_i must be in a block of $r_\tau(R)$. By lemma 8, s_i must also be in $\bigcup r^p_\tau(R) \cup \{R\}$. Now we show that s_i is not in a block of $r^p_\tau(R)$.
> By property (i), s_i must also be in $[\![\Phi_1]\!]$. If $\Phi_1 = tt$ then $\bigcup r^p_\tau(R) = \emptyset$, according to step 3. Otherwise, by the main induction hypothesis formula Φ_1 distinguishes P and blocks in $r^p_\tau(R)$. Then by lemma 3, $s_i \notin \bigcup r^p_\tau(R)$.

Now that state s_n must always be in R, we know by the last conjunct of property (i) that s' must be in $\bigcup r_\alpha(R)$. Moreover, s' must be in $[\![\Phi_2]\!]$. If $\Phi_2 = tt$ then $\bigcup r_\alpha(R) = \emptyset$, according to step 3. This contradicts the fact that s' must be in $\bigcup r_\alpha(R)$. Otherwise by the induction hypothesis formula Φ_2 distinguishes B' and blocks in $r_\alpha(R)$, but then by lemma 3: $s' \notin \bigcup r_\alpha(R)$. This also contradicts the fact that s' must be in $r_\alpha(R)$. Hence, if $\alpha \neq \tau$ then Φ is a distinguishing formula for L and R.

II) Otherwise, in case $\alpha = \tau$, the extra formula Φ_3 distinguishes block B' and block R by the induction hypothesis. The application of the induction hypothesis is correct because its conditions hold. The first condition holds by lemma 1 and the fact that block B' has the same height as common ancestor P. The second condition holds by lemma 7 (i); for this purpose we prove that $B' \neq P$ by reductio ad absurdum. Suppose that $B' = P$, then lemma 6 contradicts the fact that α is equal to τ.

Now it is obvious that Φ holds in all the states of L. However, Φ does not hold in any state of R. We will prove this by reductio ad absurdum. Suppose $\exists s \in R : s \in [\![\Phi]\!]$. Then by definition of $[\![\Phi]\!]$, state s must satisfy one of the following properties:

(ii) $s \in [\![\Phi_2 \wedge \Phi_3]\!]$ or;

(iii) $\exists s_0, ..., s_n \in [\![\Phi_1]\!], \exists s' \in [\![\Phi_2 \wedge \Phi_3]\!] : s_0 = s, [\forall_{0<i\leq n} : s_{i-1} \xrightarrow{\tau} s_i]$ *and* $s_n \xrightarrow{\alpha} s'$.

Now we derive a contradiction in case state s satisfies property (ii) or property (iii). When state s satisfies property (ii) then s must be in $[\![\Phi_3]\!]$ because $[\![\Phi_2 \wedge \Phi_3]\!]$ is defined as $[\![\Phi_2]\!] \cap [\![\Phi_3]\!]$. We already know that formula Φ_3 distinguishes B' and R by the induction hypothesis. But then by lemma 3, $s \notin R$. This contradicts the fact that s is in R.

When state s satisfies property (iii), state s' must also be in $[\![\Phi_2 \wedge \Phi_3]\!] = [\![\Phi_2]\!] \cap [\![\Phi_3]\!]$ and thus s' must be in $[\![\Phi_2]\!]$. This means that s has to satisfy property (i) and this will lead to a contradiction as before. Hence, if $\alpha = \tau$ then Φ is a distinguishing formula for L and R.

Finally, in step 5 we can conclude by lemma 3.11 that when $B_1 \subseteq L$ then formula Φ distinguishes B_1 and B_2. Otherwise, formula $\neg\Phi$ distinguishes B_1 and B_2 by negation. □