On Automatically Explaining Bisimulation Inequivalence*

Rance Cleaveland
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-8206
USA

Abstract

This paper describes a technique for generating a logical formula that differentiates between two bisimulation-inequivalent finite-state systems. The method works in conjunction with a partition-refinement algorithm for computing bisimulation equivalence and yields formulas that are often minimal in a precisely defined sense.

1 Introduction

A popular technique for verifying finite-state systems involves the use of a *behavioral equivalence*. In this approach, specifications and implementations are formalized as finite-state machines, and verification consists of establishing that an implementation is equivalent to, in the sense of *behaving the same as*, its specification. A number of equivalences have been proposed in the literature [1, 3, 7, 8, 12, 14], and several automated tools include facilities for computing them [2, 5, 6, 13].

One particularly interesting equivalence is *bisimulation equivalence* [14]. In addition to the fact that a number of other equivalences may be described in terms of it [4], the relation 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 [9]. 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 the other.

The purpose of this paper is to develop a technique for determining a Hennessy-Milner formula that distinguishes two bisimulation-inequivalent finite-state systems. To this end, we show how to use information generated by the partition-refinement algorithm of Kanellakis and Smolka [11] to compute such a formula efficiently. On the basis of our results, the

*Research supported by NSF/DARPA research grant CCR-9014775.

tools mentioned above may be modified to give users diagnostic information in the form of a formula when a system is found not to be equivalent to its specification.

Hillerström [10] also gives a technique for computing a Hennessy-Milner formula. However, his method relies on the use of a backtracking-based algorithm that is less efficient that the more popular partition-based algorithms we are interested in.

The remainder of the paper is organized as follows. The next section defines bisimulation equivalence and examines the connection between it and the Hennessy-Milner Logic. Section 3 then describes a modification of Kanellakis-Smolka bisimulation algorithm that computes distinguishing formulas; a small example is also presented to illustrate the workings of the new algorithm. The final section contains our conclusions and directions for further research.

2 Transition Graphs, Bisimulations and Hennessy-Milner Logic

Finite-state systems may be represented as *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 the actions and represent the state transitions a system may undergo. The formal definition is the following.

Definition 2.1 *A* transition graph *is a quadruple* $\langle Q, q, Act, \rightarrow \rangle$*, where:*

- Q *is a set of states (vertices);*
- $q \in Q$ *is the start state;*
- Act *is a set of actions; and*
- $\rightarrow \subseteq Q \times Act \times Q$ *is the derivation relation (set of labeled edges).*

We shall often write $q_1 \xrightarrow{a} q_2$ to indicate that there is an edge labeled a from state q_1 to state q_2; in this case, we shall sometimes say that q_2 is an *a-derivative* of q_1. 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 other three components of the transition graph coming from the transition system.

Reactive systems [16] compute by interacting with their environment. For such systems, the traditional language equivalence of automata theory is insufficiently discriminating, since the resolution of nondeterministic choices may leave a system in states that react differently to stimuli offered by the environment. Bisimulation equivalence remedies this shortcoming by requiring that equivalent systems have state sets that "match up" appropriately: the start states must be matched, and if two states are matched then they must have matching a-derivatives for any action a either is capable of. These intuitions may be formalized in terms of *bisimulations* on a *single* transition system.

Definition 2.2 *Let $\langle Q, Act, \rightarrow\rangle$ be a transition system. Then a relation $R \subseteq Q \times Q$ is a* bisimulation *if, whenever $q_1 \, R \, q_2$, the following hold.*

1. *If $q_1 \xrightarrow{a} q_1'$ then there is a q_2' such that $q_2 \xrightarrow{a} q_2'$ and $q_1' \, R \, q_2'$.*
2. *If $q_2 \xrightarrow{a} q_2'$ then there is a q_1' such that $q_1 \xrightarrow{a} q_1'$ and $q_1' \, R \, q_2'$.*

Two states and in a transition system are *bisimulation equivalent* if there is a bisimulation relating them. When q_1 and q_2 are bisimulation equivalent we shall write $q_1 \sim q_2$.

Let $G_1 = \langle Q_1, q_1, Act, \rightarrow_1\rangle$ and $G_2 = \langle Q_2, q_2, Act, \rightarrow_2\rangle$ be two transition graphs satisfying $Q_1 \cap Q_2 = \emptyset$. Then G_1 and G_2 are bisimulation equivalent exactly when the two start states, q_1 and q_2, are equivalent in the transition system $\langle Q_1 \cup Q_2, Act, \rightarrow_1 \cup \rightarrow_2\rangle$. This definition may be generalized to arbitrary transition systems (i.e., ones whoses state sets are not disjoint), at the cost of a slightly more complicated definition for the transition system in which bisimulation equivalence is to be computed. In the remainder of the paper we only consider the problem of determining a formula that distinguishes two inequivalent states in a transition system.

A number of behavioral equivalences may be characterized in terms of bisimulation equivalence on suitably transformed transition systems [4, 5, 6]. For example, if the transition system is *deterministic*, meaning that every state has at most one a-derivative for a given a, then bisimulation equivalence coincides with *language equivalence* from formal language theory. To determine if two states in an arbitrary transition system are language equivalent, it suffices to apply a "determinizing" transformation to the transition system and then determine whether their corresponding states in this new system are bisimulation equivalent. Other equivalences, including testing equivalence [7, 8], failures equivalence [3], and observational equivalence [14], may be computed in an analogous fashion.

Bisimulation equivalence also has a *logical* characterization in terms of Hennessy-Milner Logic (HML) [9]: two states are equivalent exactly when they satisfy the same HML formulas. The syntax of HML is defined as follows, where $a \in Act$.

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

Given a transition system $T = \langle Q, Act, \rightarrow\rangle$, the interpretation of the logic maps each formula to the set of states for which the formula is "true"; Figure 1 gives the formal definition. 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 of any state, and $\neg\Phi$ holds of a state if Φ does not. $\Phi_1 \wedge \Phi_2$ holds of a state if both Φ_1 and Φ_2 do, while the modal proposition $\langle a\rangle\Phi$ holds if the state has an a-derivative for which Φ holds. We shall say that a state q in transition system T *satisfies* formula Φ if $q \in [\![\Phi]\!]_T$.

Let $H(q)$ be the set of HML formulas that a state q satisfies:

$$H(q) = \{\, \Phi \mid q \in [\![\Phi]\!] \,\}.$$

The next theorem is a corollary of a result proved in [9].

$$
\begin{aligned}
[\![tt]\!]_T &= Q \\
[\![\neg\Phi]\!]_T &= Q - [\![\Phi]\!]_T \\
[\![\Phi_1 \wedge \Phi_2]\!]_T &= [\![\Phi_1]\!]_T \cap [\![\Phi_2]\!]_T \\
[\![\langle a\rangle\Phi]\!]_T &= \{\, q \in Q \mid \exists q'.\ q \xrightarrow{a} q' \wedge q' \in [\![\Phi]\!]_T \,\}
\end{aligned}
$$

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

Theorem 2.3 *Let $\langle Q, Act, \rightarrow\rangle$ be a finite-state transition system, with $q, q' \in Q$. Then $H(q) = H(q')$ if and only if $q \sim q'$.*

It follows that if two states in a (finite-state) transition system are inequivalent, then there must be a HML formula satisfied by one and not the other. This is the basis of the following definition of distinguishing formula.

Definition 2.4 *Let $\langle Q, Act, \rightarrow\rangle$ be a transition system, and let $S_1 \subseteq Q$ and $S_2 \subseteq Q$. Then HML 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 a formula that distinguishes $\{q_1\}$ from $\{q_2\}$ if $q_1 \not\sim q_2$.

Finally, we shall adopt the following criterion in assessing whether a distinguishing formula contains extraneous information.

Definition 2.5 *Let Φ be a HML 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 distinguishing formula for S_1 with respect to S_2 if each of its subterms plays a role in distinguishing the two.

3 Computing Distinguishing Formulas

In this section, we describe an algorithm for computing bisimulation equivalence [11] and show how to alter it to compute distinguishing formulas. We then consider a small example that illustrates the behavior of the modified algorithm.

```
function split(B, a, B') =
  {{ s ∈ B | ∃s' ∈ B'. s -a-> s' }, { s ∈ B | ¬∃s' ∈ B'. s -a-> s' }} − {∅};

algorithm bisim(Q, Act, →);
  begin
    P1 := {Q};
    P2 := ∅;
    while P1 ≠ P2 do begin
      P2 := P1;
      P1 := ∅;
      foreach B ∈ P2 do P1 := P1 ∪ split(B, a, B');
    end
  end
```

Figure 2: The partition refinement algorithm for bisimulation equivalence.

3.1 The Kanellakis-Smolka Algorithm

The Kanellakis-Smolka algorithm exploits the fact that an equivalence relation on a set of states may be viewed as a *partition*, or set of pairwise-disjoint subsets (called *blocks*) of the state set whose union is the 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* this partition (by splitting blocks) until the associated equivalence relation becomes a bisimulation. In order to determine whether the partition needs further refining, the algorithm looks at each block in turn. If a state in a block B has an a-derivative in a block B' and another state in B does not, then the algorithm splits B into two blocks, one containing the states having an a-derivative in B' and the other containing the states that do not. When no more splitting is possible, the resulting equivalence relation corresponds exactly to bisimulation equivalence on the given transition system. The algorithm is given in Figure 2. Function `split` is used to split one block with respect to another; notice that `split`$(B, a, B') = \{B\}$ (i.e. B is not split with respect to a and B') if either all the states in B, or none of them, have an a-derivative in B'. It should also be pointed out that $P_1 = P_2$ exactly when no more splits in P_1 are possible. The worst-case complexity of *bisim* is $O(|\rightarrow| * |Q|)$.

3.2 Generating Distinguishing Formulas

One straightforward way to compute distinguishing formulas is to associate a formula, $\Phi(B)$, with each block B in the partition in such a way that that the following hold.

- $B \subseteq [\![\Phi(B)]\!]$.

- $B' \cap [\![\Phi(B)]\!] = \emptyset$ if $B' \neq B$.

In the initial partition, $\{Q\}$, $\Phi(Q)$ is set to *tt*. Now suppose a block B is split, i.e. suppose there is an action a and another block B' such that $\mathit{split}(B, a, B') = \{B_1, B_2\}$, with every state in B_1 having a transition into B' and no state in B_2 having one. Then $\Phi(B_1)$ may be set to $\Phi(B) \wedge \langle a \rangle \Phi(B')$, while $\Phi(B_2)$ becomes $\Phi(B) \wedge \neg\langle a \rangle \Phi(B')$. Arguing inductively, it is easy to establish that for any block B, a state satisfies $\Phi(B)$ exactly when it is contained in B. Since two states that are not bisimulation equivalent will eventually wind up in different blocks, it is a simple matter to compute a formula that distinguishes such states: just return the formula associated with one of the containing blocks.

Although intuitively appealing, this approach has a drawback; it generates very large formulas. In general, the size of a formula associated with a block will grow in size as 2^r, where r is defined as the number of iterations of the `while` loop in *bisim*. In certain cases $r = |Q|$, so the formulas obtained using this method may be exponential in the size of the state space. There is, however, a polynomial-size representation using a set of propositional equations, so this complexity is not as severe as it seems; moreover, it is the case that an exponential-size formula may also be minimal. More importantly, the formulas include a large amount of extraneous information: not only does such a formula distinguish one state from another inequivalent state, it also distinguishes it from every state to which it is inequivalent. In fact, the formulas generated this way are rarely minimal, and because of this, they are not useful from a diagnostic standpoint.

Another Approach

We now describe a better technique for generating distinguishing formulas. The method uses information computed by a slightly altered version of *bisim* that, in addition to computing the partition as described above, retains information about how blocks are split. Then, a postprocessing step constructs a formula that generates a formula distinguishing the states in one block from the states in another.

Bisim 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 block is split; accordingly, the root is labeled with the block Q, and after each iteration of the `foreach` loop the leaves of this tree represents the current partition. When a block B is split (by $\mathit{split}(B, a, B')$), we place the new block $B_1 = \{\, s \in B \mid \exists s' \in B'.\ s \xrightarrow{a} s' \,\}$ as the left child and the new block $B_2 = \{\, s \in B \mid \neg\exists s' \in B'.\ s \xrightarrow{a} s' \,\}$ as the right child, and we label the arc connecting B to B_1 with a and B'. Recall that every state in B_1 has an a-transition into B' and that no state in B_2 does. If a block is not split during an iteration of the `foreach` loop, it is assigned a copy of itself as its only child.[1] Figure 5 contains an example of such a tree.

Given a block tree computed by the new version of *bisim*, and two states s_1 and s_2 that are inequivalent and hence in different blocks, the postprocessing step builds a formula

[1] Strictly speaking, this is not necessary; these blocks may be left childless. We include these spurious children to simplify our inductive argument of correctness.

```
Size := ∞;
S_L := { s' | s_L -a-> s' } ∩ B';
S_R := { s' | s_R -a-> s' };
foreach s'_L ∈ S_L do begin
  Γ := ∅;
  foreach s'_R ∈ S_R do begin
    Φ' := δ(s'_L, s'_R);
    Γ := Γ ∪ {Φ'};
  end;
  foreach Φ_i ∈ Γ do begin
    S_i := { s' ∈ S_R | s' ∉ ⟦Φ_i⟧ ∧ ∀Φ_j ∈ Γ. i ≠ j ⇒ s' ∈ ⟦Φ_j⟧ };
    if S_i = ∅ then Γ := Γ − {Φ_i};
  end;
  if |⋀Γ| < Size then begin
    Size := |⋀Γ|;
    Φ := ⋀Γ;
  end;
end;
```

Figure 3: Code for generating conjunctions.

$\delta(s_1, s_2)$ that distinguishes $\{s_1\}$ from $\{s_2\}$. Although this formula will not necessarily be minimal either, it will in general be much smaller than the formula computed using the method described above; it is guaranteed to be no larger. The details are as follows.

1. Determine the deepest block P in the block tree such that $s_i \in P$ for $i = 1, 2$. Let L and R be the left and right children of P, with a, B' labeling the arc from P to L. Note that either $s_1 \in L$ and $s_2 \in R$, or vice versa. Let s_L be the state in L, and s_R the state in R.

2. Execute the code in Figure 3. The idea is the following. For each state in B' that is an a-derivative of s_L we will generate a minimal set of formulas satisfied by s_L whose conjunction is satisfied by no derivative of of s_R. We will then take the set yielding the smallest conjunction. *Size* is the variable used to record the size of the current shortest conjunction, while Γ contains the current collection of formulas being built.

3. If $s_L = s_1$ then return $\langle a \rangle \Phi$; otherwise, return $\neg \langle a \rangle \Phi$.

Theorem 3.1 *The formula $\delta(s_1, s_2)$ distinguishes $\{s_1\}$ from $\{s_2\}$.*

Proof. By induction on the depth of the deepest block in the block tree containing both s_1 and s_2. $\square$

In general, $\delta(s_1, s_2)$ will not be minimal. However, it is possible to characterize situations when $\delta(s_1, s_2)$ will be minimal, as the following result indicates.

Theorem 3.2 *Suppose that in each recursive call to δ generated by $\delta(s_1, s_2)$ ($s_1 \not\sim s_2$) the following holds.*

1. *Let Γ be the set of conjuncts used to build Φ and s'_L the state in s_L that was used to create Γ. Then for each $\Phi' \in \Gamma$ there is an $s_j \in S_R$ such that Φ minimally distinguishes $\{s'_L\}$ from $\{s_j\}$, and $s_j \in [\![\Phi'']\!]$ for all other $\Phi'' \in \Gamma$.*

2. *$\{\, s' \mid s_L \xrightarrow{a} s' \,\} - B' \subseteq S_R$.*

Then $\delta(s_1, s_2)$ is minimal

Proof. By contradiction. Of importance is the fact that Φ is the shortest length formula that δ can build to distinguish an a-derivative of s_L from all a-derivatives of s_R. $\square$

It is also the case that a minimizing procedure can be applied to $\delta(s_1, s_2)$ once it has been computed; the result of this would be a minimal formula. The minimizing procedure is straightforward: 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 computational tractability of this procedure remains to be examined, however.

It should be noted that δ may still generate exponential length formulas. However, one may represent such a formula (as a set of propositional equations) in space proportional to $|Q|^3$. This results from the fact that there can be at most $|Q|^2$ total recursive calls generated by the above procedure and the fact that each distinguishing formula is of the form $(\neg)\langle a\rangle\Phi$, where Φ contains at most $|Q|$ conjuncts, each of the form $\delta(s_i, s_j)$ for some s_i and s_j. By saving information appropriately and modifying the procedure for δ so that the semantic information of the formula computed is also returned, we may establish the following bound on the amount of computation needed to compute such a series of equations.

Theorem 3.3 *An equational representation of $\delta(s_1, s_2)$ may be calculated in $O(|Q|^5)$ time, once the tree of blocks has been computed.*

Proof. Follows from the fact that determining the equation for each recursive call of δ requires $O(|Q|^5)$ work. $\square$

We close this subsection with some general remarks about our method. One feature of our approach is that the overhead involved in maintaining the block tree is minimal; nodes need not be labeled with the corresponding sets of states, except at the leaves. Also, the postprocessing step is only invoked after equivalence is computed—so if the states of interest are found to be equivalent, then the postprocessing may be avoided altogether.

3.3 An Example

To illustrate our algorithm we consider a well-known example of two transition graphs that are not bisimulation equivalent. Figure 4 shows the transition system that includes the two transition graphs. State q_1 is the start state of one graph, while state q_4 is the start state of the other. Figure 5 contains the tree of blocks generated by the altered *bisim* algorithm.

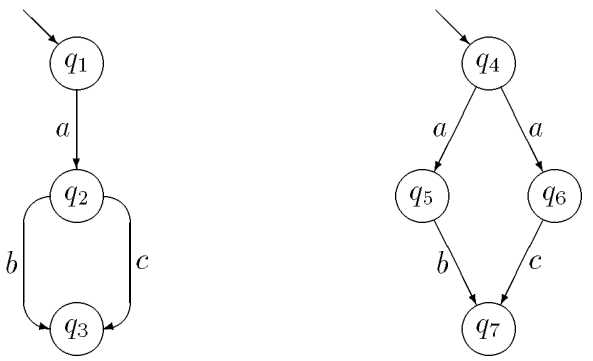

Figure 4: Two inequivalent transition systems.

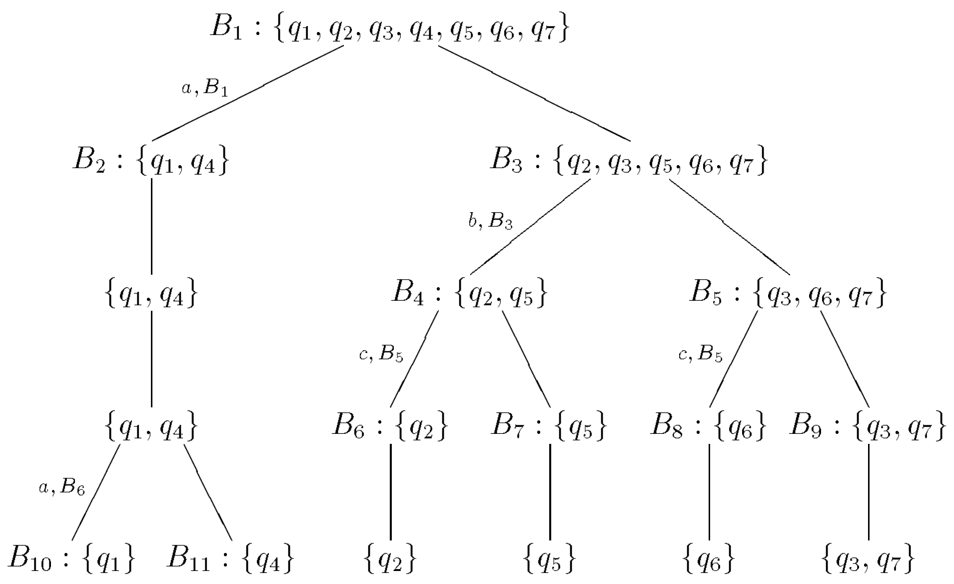

Figure 5: The tree of blocks generated by *bisim*.

Notice that $q_1 \not\sim q_4$, as they are in different blocks.

In order to build $\delta(q_1, q_4)$, the algorithm first locates the lowest common ancestor of the two blocks (B_2, in this case). The left child is B_{10}, the right child is B_{11}, the action causing the split is a, and the block causing the split is B_6. The formula that will be returned, then, will be

$$\langle a \rangle (\delta(q_2, q_6) \wedge \delta(q_2, q_5));$$

this formula holds of q_1 and not q_4. By repeating this process, it turns out that

$$\begin{aligned} \delta(q_2, q_6) &= \langle c \rangle tt \text{ and} \\ \delta(q_2, q_5) &= \langle b \rangle tt. \end{aligned}$$

So the formula distinguishing q_1 from q_4 is

$$\langle a \rangle (\langle c \rangle tt \wedge \langle b \rangle tt).$$

This formula states that q_1 and q_4 are inequivalent because q_1 may engage in an a-transition and evolve into a state from which both b- and c-transitions are available. Note that this formula is minimal. By way of contrast, the formula generated by the first naive method would be the following.

$$\langle a \rangle tt \wedge \langle a \rangle (\neg \langle a \rangle tt \wedge \langle b \rangle \neg \langle a \rangle tt \wedge \langle c \rangle (\neg \langle a \rangle tt \wedge \neg \langle b \rangle \neg \langle a \rangle tt))$$

This formula is clearly not minimal, since, for example, the formula obtained by substituting tt for subformula $\langle a \rangle tt$ is still distinguishes q_1 from q_4.

4 Conclusions and Future Work

This paper has shown how it is possible to alter partition-refinement based algorithms for computing bisimulation equivalence to compute a formula in the Hennessy-Milner Logic 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 formulas are often minimal in a certain sense, and the postprocessing step has an $O(|Q|^2)$ effect on the worst-case complexity of the equivalence-checking algorithm.

There are several avenues for future work to be pursued. Clearly, the complexity of the minimization procedure mentioned in passing at the end of Section 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 involve an implementation of our technique; we plan to incorporate this distinguishing formula capability into the Concurrency Workbench [5, 6], a tool for the analysis of finite-state systems. Yet another involves determining appropriate ways of using formulas computed in the course of checking equivalences other than bisimulation equivalence. Of particular interest is *testing (or failures) equivalence* [3, 7, 8]. These equivalences may be characterized in terms of the tests a process may pass and must pass. One method for distinguishing states that are not testing equivalent would be to build a test based on the formula computed by the bisimulation equivalence checker that one state may (or must) pass and that the other must (or may) not. Finally, it may be possible to extend our techniques to the computation of distinguishing formulas in the context of *preorder* checking. Another method of verifying processes involves the use of a behavioral preorder; in this setting, an implementation satisfies a specification if the implementation is "greater than" (intuitively: "behaves at least as well as") the specification. One interesting preorder is the *prebisimulation preorder*, which has a logical characterization in terms of an intuitionistic variant of the Hennessy-Milner logic: one state is "greater than" another if it satisfies all the formulas satisfied by the latter. This property could serve as the theoretical basis for computing diagnostic information in the same way that the logical characterization of bisimulation equivalence served as the theoretical basis for the techniques described in this paper.

Acknowledgement

I would like to thank Henri Korver for spotting errors in, and for his helpful comments on, previous drafts of this paper.

References

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

[2] Boudol, G., 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.

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

[4] Cleaveland, R. 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.

[5] Cleaveland, R., 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.

[6] Cleaveland, R., J. Parrow and B. Steffen. "The Concurrency Workbench." In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, Lecture Notes in Computer Science 407, 24–37. Springer-Verlag, Berlin, 1989.

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

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

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

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

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

[12] Larsen, K. and A. Skou. "Bisimulation through Probabilistic Testing." Proceedings of the ACM Symposium on Principles of Programming Languages, 1989.

[13] Malhotra, J., Smolka, S.A., Giacalone, A. and Shapiro, R. "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.

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

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

[16] Pnueli, A. "Linear and Branching Structures in the Semantics and Logics of Reactive Systems." Lecture Notes in Computer Science 194, 14-32. Springer-Verlag, Berlin, 1985.