An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence

(Jan Friso Groote & Frits Vaandrager)

Abstract

This paper presents an efficient algorithm for the Relational Coarsest Partition with Stuttering problem (RCPS). The RCPS problem is closely related to the problem of deciding stuttering equivalence on finite state Kripke structures (see Browne, Clarke and Grumberg [3]), and to the problem of deciding branching bisimulation equivalence on finite state labelled transition systems (see van Glabbeek and Weijland [12]). If n is the number of states and m the number of transitions, then our algorithm has time complexity $O(n \cdot (n+m))$ and space complexity $O(n+m)$. The algorithm induces algorithms for branching bisimulation and stuttering equivalence which have the same complexity. Since for Kripke structures $m \leq n^2$, this confirms a conjecture of Browne, Clarke and Grumberg [3], that their $O(n^5)$-time algorithm for stuttering equivalence is not optimal.

1 Introduction

In this paper we present an efficient algorithm for the *Relational Coarsest Partition with Stuttering problem (RCPS)*. This problem is interesting because it is closely related to (1) deciding stuttering equivalence on finite Kripke structures, and (2) deciding branching bisimulation equivalence on finite labelled transition systems. Below we comment on these two problems separately.

Temporal logic model checking procedures have been successful in finding errors in relatively small network protocols and sequential circuits (for an overview see [5]). However, a serious problem for the model checking approach is the *state explosion problem*: in general, the number of states in the global state graph may grow exponentially with the number of components. In order to deal with this problem, it seems natural to hide details that do not need to be visible externally and merge those states that become indistinguishable. In the setting of temporal logic, hiding of 'details' is achieved by making it illegal to refer to these details in temporal logic formulas. If a program is constructed in a hierarchical fashion, then state explosion may be avoided by simplifying the components before computing the global state graph [6]. Our paper deals with the question how the idea of merging indistinguishable states can be implemented in the setting of the logic CTL*.

The computation tree logic CTL* [10] is a very powerful temporal logic that combines both branching time and linear time operators. CTL [4] is a restricted subset of CTL* that permits only branching time operators. One of the operators in CTL/CTL*, the nexttime operator X, has been subject to some criticism. Lamport [15] argues that in reasoning about concurrent systems, the nexttime operator may be dangerous since it refers to the *global* next state instead of the *local* next state. If, for this reason, one decides not to use the nexttime operator, then it becomes interesting to study the equivalences on states induced by the sets of formulas CTL$-X$ and CTL$^* - X$. The idea is that two states that satisfy the same CTL$-X$/CTL$^* - X$ formulas have the same relevant properties (and maybe some irrelevant ones as well) and are therefore identified. Browne, Clarke and Grumberg [3] introduce the notion of *stuttering equivalence* on Kripke structures and prove that this equivalence characterizes both the equivalence induced by CTL$-X$ and the equivalence induced by CTL$^* - X$.

They show that stuttering equivalence can be decided in polynomial time but give a rather high upper bound of $O(n^5)$ for the time complexity, where n is the number of states of the Kripke structure. They conjecture the existence of a faster algorithm. The present paper confirms this conjecture: our algorithm for the RCPS problem solves stuttering equivalence in $O(m \cdot n)$ time, where m is the number of transitions in the Kripke structure. Here, like in the rest of this paper, we assume $m \geq n$. This assumption may be dropped if $m + n$ is read wherever we write m. In Kripke structures always $m \leq n^2$.

VAN GLABBEEK and WEIJLAND [12] introduce the notion of *branching bisimulation equivalence* on labelled transition systems. This equivalence resembles, but is finer than the *observation equivalence* of MILNER [17]. They argue that, unlike observation equivalence, branching bisimulation preserves the branching structure of processes, in the sense that it preserves computations together with the potentials in all intermediate states that are passed through, even if silent moves are involved. Besides this conceptual aspect, a number of other results indicate that branching bisimulation equivalence has nice properties:

1. The corresponding congruence has a simple algebraic characterization [12]. In the setting of CCS, only a single axiom has to be added to the laws for strong bisimulation equivalence:
$$a \cdot (\tau \cdot (x+y) + x) \;=\; a \cdot (x+y).$$

2. In contrast with observation equivalence, the resulting axiom system can be turned easily into a complete term rewriting system.

3. A modal characterization of branching bisimulation equivalence can be obtained by adding to Hennessy-Milner logic a kind of until operator [9].

4. Branching bisimulation equivalence can be characterized in terms of *back and forth bisimulations* [8]. This characterization induces a second modal characterization of branching bisimulation which is a variant of Hennessy-Milner logic with backward modalities [9].

5. Branching bisimulation equivalence is the natural analogue of stuttering equivalence in a setting where the transitions rather than the states are labelled. It is even possible to view $\mathrm{CTL}^* - X$ as a logic for branching bisimulation [9].

6. Unlike observation equivalence, branching bisimulation is preserved under refinement of actions in a setting without parallelism [13].

7. For a large class of processes, branching bisimulation and observation equivalence are the same [12]. We are not aware of any protocol that can be verified in the setting of observation equivalence but not in the stronger setting of branching bisimulation equivalence.

In this paper we show how our algorithm for RCPS can be easily transformed to an $O(m \cdot n)$ algorithm for deciding branching bisimulation equivalence ($O(m \log m + m \cdot n)$ if the set of labels is infinite or not fixed).

The structure of this paper is as follows. In section 2 we present the RCPS problem and in section 3 our algorithm to solve it. Section 4 describes how the algorithm can be used to decide stuttering equivalence and in section 5 we show how a variant of the algorithm solves the problem of deciding branching bisimulation. Section 6 contains some concluding remarks. We conjecture that, at the price of a more complicated algorithm, the efficiency of our algorithm for RCPS can be slightly improved upon by incorporating ideas from the $O(m \log n)$ algorithm of PAIGE and TARJAN [18] for the *Relational Coarsest Partition problem (RCP).* Also in section 6, we compare the complexity of deciding branching bisimulation equivalence with the complexity of deciding observation equivalence.

2 The RCPS problem

Let S be a set. A collection $\{B_j \mid j \in J\}$ of nonempty subsets of S is called a *partition* of S 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*. If P and P' are partitions of S then P' *refines* P, and P is *coarser* than P', if any block of P' is included in a block of P. The equivalence $\sim_P$ on S induced by a partition P is defined by: $r \sim_P s$ iff $\exists B \in P : r \in B \wedge s \in B$.

The *Relational Coarsest Partition with Stuttering problem (RCPS)* can now be specified as follows:

Given: a nonempty, finite set S of *states*, a relation $\longrightarrow \subseteq S \times S$ of *transitions* and an *initial partition* P_0 of S.

Find: the coarsest partition P_f satisfying:

(i) P_f refines P_0;

(ii) if $r \sim_{P_f} s$ and $r \longrightarrow r'$, then there is an $n \geq 0$ and there are $s_0, ..., s_n \in S$ such that:

- $s_0 = s$;
- for all $0 \leq i < n$: $r \sim_{P_f} s_i$ and $s_i \longrightarrow s_{i+1}$;
- $r' \sim_{P_f} s_n$.

Below we show that a coarsest partition satisfying (i) and (ii) always exists; if it exists, then clearly it is unique.

3 The Algorithm

Next we describe our algorithm for the RCPS problem. We fix a nonempty, finite set S of states, a transition relation $\longrightarrow$ and an initial partition P_0. Let $|S| = n$ and $|\longrightarrow| = m$. For $B, B' \subseteq S$ we define the set $pos(B, B')$ as the set of states in B from which, after some initial stuttering, a state in B' can be reached:

$$pos(B,B') \;=\; \{s \in B \mid \exists n \geq 0 \; \exists s_0, ..., s_n \;:\; s_0 = s, \; [\forall i < n : \; s_i \in B \;\wedge\; s_i \longrightarrow s_{i+1}] \text{ and } s_n \in B'\}.$$

Call B' a *splitter* of B and (B, B') a *splitting pair* iff $\emptyset \neq pos(B, B') \neq B$. Since $pos(B, B) = B$, a block can never be a splitter of itself. If P is a partition of S and B' a splitter of B, define $\mathit{Ref}_P(B, B')$ as the partition obtained from P by replacing B by $pos(B, B')$ and $B - pos(B, B')$. P is *stable* with respect to a block B' if for no block B, B' is a splitter of B. P is *stable* if it is stable with respect to all its blocks. Thus the RCPS problem consists of finding the coarsest stable partition that refines P_0. Our algorithm maintains a partition P that is initially P_0. The following refinement step is repeated as long as P is not stable:

find $B, B' \in P$ such that B' is a splitter of B;
$P := \mathit{Ref}_P(B, B')$

Theorem 3.1. *The above algorithm terminates after at most $n - |P_0|$ refinement steps. The resulting partition P_f is the coarsest stable partition refining P_0.*
Proof. In order to see that the algorithm terminates, observe that after each iteration of the refinement step the number of blocks of P has increased by one. Since a partition of S can have at most n blocks, termination will occur after at most $n - |P_0|$ iterations.

Next we show that the algorithm solves the RCPS problem. By induction on the number of refinement steps we prove that any stable refinement of P_0 is also a refinement of the current partition.

Clearly the statement holds initially. Suppose it is true before a refinement step that refines a partition P to a partition Q, using a splitting pair (B, B'). Let R be any stable refinement of P_0 and let C be a block of R. It is enough to show that C is included in a block from Q. By the induction hypothesis, we can assume that C is included in a block D of P. If $D \neq B$, then D is a block of Q and we are done. So suppose $D = B$. We have to show that either $C \subseteq pos(B, B')$ or $C \subseteq B - pos(B, B')$. Suppose that there are $r, s \in C$ with $r \in pos(B, B')$ and $s \notin pos(B, B')$. We derive a contradiction. There are $r_0, ..., r_n$ such that $r = r_0$, for all $i < n$: $r_i \in B \wedge r_i \longrightarrow r_{i+1}$ and $r_n \in B'$. Let $C_0, ..., C_n$ be the blocks of R such that $r_i \in C_i$. Then $C_0 = C$ and, by induction, for all $i < n$: $C_i \subseteq B$ and $C_n \subseteq B'$. Now use the fact that R is stable to construct a sequence $s_0, ..., s_m$ with $s_0 = s$, for $i < m$: $s_i \in B \wedge s_i \longrightarrow s_{i+1}$ and $s_m \in B'$. This contradicts $s \notin pos(B, B')$. Thus we have proved the induction step. □

Below we describe an implementation of our algorithm. We show how to compute in $O(m)$ time whether or not a partition is stable. The computation is organized in such a way that if the partition is not stable, a counterexample, i.e. a splitting pair (B, B'), is produced. Next we show how to compute $Ref_P(B, B')$ in $O(m)$ time. Since the number of iterations of the main loop is $O(n)$, this establishes a complexity of $O(m \cdot n)$ for the RCPS problem.

Efficient implementation of the algorithm requires some preprocessing. Let P be a partition. We call a transition $s \longrightarrow s'$ *inert* with respect to P, or P-*inert*, iff $s \sim_P s'$. If in the initial partition a set of states is strongly connected via inert transitions, then these states will be in the same block of the final partition: by definition of inert they are in the same block of the initial partition, and no refinement step will place two states from the set in a different block. As a preprocessing step in our algorithm we look for strongly connected components with respect to inert transitions in the initial partition and 'collapse' these components to one state. Here we can use the well-known $O(m)$ algorithm for finding strongly connected components in a directed graph (see for instance Aho, Hopcroft and Ullman [1]). Thus it is sufficient to solve the RCPS problem in the case where P_0 contains no cycles of inert transitions.

For $B \subseteq S$, define the set $bottom(B)$ of *bottom* states of B by:

$$bottom(B) \;=\; \{r \in B \mid \forall s : \; r \longrightarrow s \;\Rightarrow\; s \notin B\}.$$

If P is a partition, then the set $bottom(P)$ of *bottom* states of P is given by:

$$bottom(P) \;=\; \bigcup_{B \in P} bottom(B).$$

The following two observations play a crucial role in the implementation of our algorithm:

Lemma 3.2. *Let P be a refinement of P_0 and let $B, B' \in P$. Then B' is a splitter of B iff*

(1) $B \neq B'$,

(2) for some $r \in B$ and $r' \in B'$: $r \longrightarrow r'$, and

(3) there is an $s \in bottom(B)$ such that for no $s' \in B'$: $s \longrightarrow s'$.

Proof. "⇒" Suppose B' is a splitter of B. Then $B \neq B'$ because a block can never split itself. By definition of a splitter: $\emptyset \neq pos(B, B')$. Thus $r \longrightarrow r'$ for some $r \in B$ and $r' \in B'$. Suppose that for every bottom state s of B there is an $s' \in B'$ with $s \longrightarrow s'$. We derive a contradiction. Pick an element $t \in B$. Since P_0 contains no cycle of inert transitions, P does not contain such a cycle either. Thus there must be a path of inert transitions from t to a bottom state t' of B. Since for some $t'' \in B'$ we have $t' \longrightarrow t''$, t is in $pos(B, B')$. But since t was chosen arbitrarily, this means that $pos(B, B') = B$. This contradicts the fact that B' is a splitter of B.

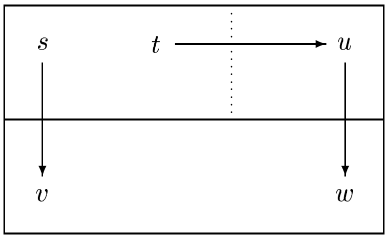


Figure 1: Stability is not inherited under refinement

"$\Leftarrow$" Suppose that B and B' satisfy condition (1), (2) and (3). Then B' is a splitter of B: $pos(B, B') \neq \emptyset$ because of (2), and $pos(B, B') \neq B$ because of (1) and (3). □

Lemma 3.3. *Let P, R be partitions such that R refines P, and P and R have the same bottom states. Let B be a block of both P and R such that P is stable with respect to B. Then R is stable with respect to B.*

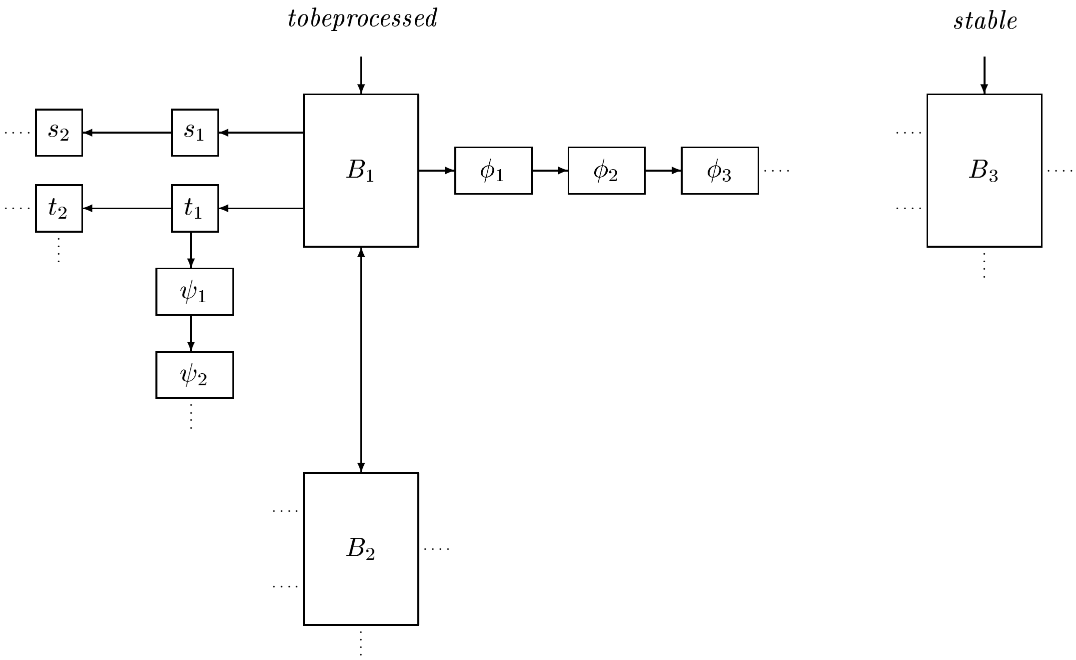
Proof. Let P, R and B be as above. Pick a block B' of R. Suppose that B is a splitter for B'. We will derive a contradiction. Application of lemma 3.2 gives: (1) $B' \neq B$, (2) for some $r \in B'$ and $r' \in B$: $r \longrightarrow r'$, and (3) there is an $s \in bottom(B')$ such that for no $s' \in B$: $s \longrightarrow s'$. Now use that B' is included in some block C of P. Clearly $C \neq B$. Moreover we can find $r \in C$ and $r' \in B$ with $r \longrightarrow r'$. Since $bottom(P) = bottom(R)$ we have $bottom(B') \subset bottom(C)$. Thus we can find an $s \in bottom(C)$ such that for no $s' \in B$: $s \longrightarrow s'$. Now apply lemma 3.2 to conclude that B is a splitter for C, which is a contradiction. □

Remark 3.4. In the setting of the Relational Coarsest Partition problem, stability is inherited under refinement in general; that is, if R is a refinement of P and P is stable with respect to B, then so is R. The $O(m \log n)$ algorithm of Paige and Tarjan [18] depends crucially on this property.

In the case of the RCPS problem stability is in general not inherited under refinement; the condition in lemma 3.3 that P and R have the same bottom states cannot be dropped. An example is presented in figure 1. If $P = \{\{s, t, u\}, \{v, w\}\}$ and $R = \{\{s, t\}, \{u\}, \{v, w\}\}$, then P is stable w.r.t. $\{v, w\}$ but R is not. As a consequence the idea behind the Paige and Tarjan [18] algorithm cannot be applied to the RCPS problem in the same way.

In the implementation of the algorithm, there is for each block, state and transition a corresponding record of type *block*, *state* resp. *transition* (see figure 2). We identify a block, state and transition with the record representing it. There are two doubly linked lists, *tobeprocessed* and *stable*, of blocks. A block B is in *stable* when the current partition is stable with respect to B. Otherwise B is in the list *tobeprocessed*. Initially, all blocks of P_0 are in the list *tobeprocessed* and the list *stable* is empty. Each block B contains a list of bottom states in B and a list of non-bottom states in B. We assume that whenever $s \longrightarrow s'$ for $s, s' \in B - bottom(B)$, s is after s' in the list of non-bottom states. Initially, the division of states in bottom states and non-bottom states, and also the ordering on the non-bottom states, can be accomplished by a standard depth first search algorithm using $O(m)$ time and space (see for instance Aho, Hopcroft and Ullman [1]). Each state contains a pointer to the block of which it is an element. Each transition contains two pointers to resp. its starting state and its target state. Each non-bottom state contains a list of the inert transitions starting in this state. Each block points to a list of the non-inert transitions that end in this block. Each state and each block has an auxiliary field *flag* of type boolean, which is 0 initially. Moreover we use an auxiliary list *BL* of blocks, which is empty initially.

Next we describe how to find out in $O(m)$ time whether a partition is stable. Let B' be a block in *tobeprocessed*. Scan the list of non-inert transitions which end in B'. When some transition is visited, the flag of the starting state is raised (i.e. the value 1 is assigned to the field *flag* of the starting state).



- $B_1, B_2, B_3, ...$ are the blocks of the current partition.
- $s_1, s_2, ...$ are the bottom states in block B_1.
- $t_1, t_2, ...$ are the non-bottom states in block B_1.
- $\phi_1, \phi_2, \phi_3, ...$ are the non-inert transitions that end in block B_1.
- $\psi_1, \psi_2, ...$ are the inert transitions that start in state t_1.

Figure 2: The data structure of the implementation

If the flag of the block to which the starting state belongs has not yet been raised, then we do so and add a copy of this block to the list BL. After having scanned all non-inert transitions ending in B', we consider the list BL. This list contains all blocks, different from B', which contain a state from which a state from B' can be reached. There is at most one reference to each block in the list. Remove the first block B from the list BL. By lemma 3.2, B' is a splitter of B iff there is a bottom state s of B such that for no $s' \in B'$: $s \longrightarrow s'$. So in order to find out whether B' is a splitter of B we only have to check whether the flag of all bottom states in B is raised.

Suppose that B' is a splitter of B. In this case we remove B from the linked list of blocks in which it occurs (in general this can be either the list *tobeprocessed* or the list *stable*), and insert two new blocks B_1 and B_2 in the list *tobeprocessed*. The *flag* fields of the new blocks are set to 0. All bottom states of B with a raised flag become bottom states of B_1, the other bottom states of B become bottom states of B_2. Next we scan the non-bottom states of B. If for some non-bottom state the flag is not raised and if none of the outgoing P-inert transitions leads to a state in B_1, then this state becomes a non-bottom state of B_2 (here we use the ordering on the list of non-inert states in the old partition). In this case the outgoing inert transitions of this state remain the same. Otherwise, the state is placed in B_1. It may be that certain transitions which were inert in the old partition lead to a state in B_2. In that case they have to be moved to the list of non-inert transitions which end in B_2. It may be that a state which is not a bottom state in the old partition becomes a bottom state in the new partition, just because no inert transitions are left. If, in a refinement, a non-bottom state becomes a bottom state, then (cf. lemma 3.3) we append the list *stable* to the list *tobeprocessed* and make *stable* empty. The non-inert transitions ending in B are distributed in the obvious way over B_1 and B_2.

If B' is not a splitter of B, or if it is a splitter and we have carried out the splitting as described above, then we consider the next first block of the list BL and check whether B' is splitter for that block, etc. When we have dealt with all blocks of BL then we know that for no block in the current partition B' is a splitter. We move B' from the list *tobeprocessed* to the list *stable*, reinitialize all flags by an additional scan of the non-inert transitions with an end state in B', and we apply the same procedure for a next block in *tobeprocessed*, etc. If *tobeprocessed* is empty then we know that the current partition is stable.

One can easily check that in $O(m)$ time we either have found a splitter and refined the partition, or we have established that the current partition is stable. Moreover the space complexity is $O(m)$. Thus we have the following theorem:

Theorem 3.5. *The RCPS problem can be decided in $O(m \cdot n)$ time, using $O(m)$ space.*

Remark 3.6. The implementation may be simplified slightly by eliminating the *stable* list. However, since the *stable* list provides a very simple way to avoid a lot of work (in our trial implementation the time performance increased with more than a factor 2), we decided to include it in the above description.

4 Stuttering equivalence

In this section we show how a solution of the the RCPS problem can be used to decide stuttering equivalence on finite Kripke structures. Let **AP** be a set of *atomic proposition names*.

Definition 4.1. A *Kripke structure* is a triple $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ where S is a set of *states*, $\longrightarrow \subseteq S \times S$ is the *transition relation* and $\mathcal{L} : S \to 2^{AP}$ is the *proposition labelling*. A Kripke structure is *finite* if the set of states is finite and for each state the set of associated proposition names is finite.

Definition 4.2. Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a Kripke structure. A relation $R \subseteq S \times S$ is called a *divergence blind stuttering bisimulation* if it is symmetric and whenever $r\,R\,s$ then:

(i) $\mathcal{L}(r) = \mathcal{L}(s)$ and

(ii) if $r \longrightarrow r'$ then there exist $s_0, ..., s_n \in S$ $(n \geq 0)$ such that $s = s_0$, for all $0 \leq i < n$: $s_i \longrightarrow s_{i+1} \wedge r \, R \, s_i$, and $r' R \, s_n$.

Two states $r, s \in S$ are *divergence blind stuttering equivalent*, notation $\mathcal{K} : r \leftrightarrows_{dbs} s$ or just $r \leftrightarrows_{dbs} s$, iff there exists a divergence blind stuttering bisimulation relation relating r and s.

One can easily check that divergence blind stuttering equivalence is indeed an equivalence relation.

Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a finite Kripke structure with $|S| = n$ and $|\longrightarrow| = m$. In order to determine whether two states in S are divergence blind stuttering equivalent, one can use our RCPS algorithm as follows.

The initial partition P_0 is constructed by putting all states with the same labels in the same block. Assuming that the set **AP** is finite and fixed, the initial partition can be computed in $O(n)$ time using a lexicographic sorting method [1]. If lexicographic sorting is not feasible, it can be computed in $O(n \log n)$ time. Next the RCPS algorithm is used to compute the coarsest stable partition P_f that refines P_0. This takes $O(m \cdot n)$ time. The following theorem says that partition P_f solves our problem.

Theorem 4.3. *Two states are in the same block of P_f exactly when they are divergence blind stuttering equivalent.*

Proof. Suppose $r, s \in B$ for some block B in P_f. Let R_f be the relation that relates two states iff they are in the same block of P_f. Clearly $r \, R_f \, s$. We show that R_f is a divergence blind stuttering bisimulation. Let $p, q \in S$ with $p \, R_f q$. As P_f refines P_0, $\mathcal{L}(p) = \mathcal{L}(q)$. Moreover, condition (ii) of definition 4.2 holds as it exactly coincides with the condition (ii) in the RCPS problem. Hence R_f is a divergence blind stuttering bisimulation and r and s are divergence blind stuttering equivalent. Let P_s be the partition of S induced by $\leftrightarrows_{dbs}$. By definition of divergence blind stuttering P_s refines P_0. Moreover, P_s is stable. As P_f is the coarsest stable partition refining P_0, P_s refines P_f. □

Thus the time complexity of deciding divergence blind stuttering equivalence is at most $O(m \cdot n + n \log n) = O(m \cdot n)$ (remember $m \geq n$).

In De Nicola and Vaandrager [9] it is shown that for finite Kripke structures divergence blind stuttering equivalence coincides with the equivalence induced by CTL$-X$/CTL$^* - X$ formulas if one quantifies over all paths in the Kripke structure (the finite as well as the infinite ones). If one only quantifies over the infinite paths, then this leads to the following *stuttering equivalence*:

Definition 4.4. Let $\mathcal{K} = (S, \longrightarrow, \mathcal{L})$ be a Kripke structure. Let s_0 be a state not in S and let p_0 be an atomic proposition such that for all s in S: $p_0 \notin \mathcal{L}(s)$. Define a Kripke structure $\mathcal{K}' = (S', \longrightarrow', \mathcal{L}')$ by:

- $S' = S \cup \{s_0\}$,
- $\longrightarrow' = \longrightarrow \cup \{(s, s_0) \mid s \in S$ has no outgoing transition or occurs on a cycle of states which all have the same label$\}$,
- $\mathcal{L}' = \mathcal{L} \cup \{(s_0, \{p_0\})\}$.

Two states $r, s \in S$ are *stuttering equivalent* if in $\mathcal{K}'$: $r \leftrightarrows_{dbs} s$ (note that this definition does not depend on the particular choice of state s_0 and atomic proposition p_0).

For finite Kripke structures the stuttering equivalence as defined above coincides with the equivalence induced by CTL$-X$ and CTL$^* - X$ if one quantifies over infinite paths [9]. Since Browne, Clarke and Grumberg [3] proved the same result for their version of stuttering equivalence, both notions agree. For finite Kripke structures the transformation of definition 4.4 can be accomplished in $O(m)$ time. Thus our algorithm for RCPS can be used to decide stuttering equivalence in $O(m \cdot n)$ time.

5 Branching bisimulation equivalence

The RCPS algorithm cannot be used directly for deciding branching bisimulation equivalence. We have to generalize RCPS to the case where transitions have labels.

The *Generalized Relational Coarsest Partition with Stuttering problem (GRCPS)* is given by:

Given: A nonempty, finite set S of *states*, a finite set A of *labels* containing the *silent step* τ, a relation $\longrightarrow \subseteq S \times A \times S$ of *transitions* and an *initial partition* P_0 of S.

Find: the coarsest partition P_f satisfying:

(i) P_f refines P_0,

(ii) if $r \sim_{P_f} s$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $r' \sim_{P_f} s$, or there is an $n \geq 0$ and there are $s_0, ..., s_n, s'$ such that $s_0 = s$, for all $0 < i \leq n$: $[r \sim_{P_f} s_i \ \wedge \ s_{i-1} \xrightarrow{\tau} s_i]$, $s_n \xrightarrow{a} s'$ and $r' \sim_{P_f} s'$.

Our algorithm for the GRCPS-problem is a minor modification of the algorithm for the RCPS-problem. Therefore, we will not describe it in detail but only sketch the differences. We fix S, A, $\longrightarrow$ and P_0 and, as usual, write $|S| = n$ and $|\longrightarrow| = m$. For $B, B' \subseteq S$ and $a \in A$, the set $pos_a(B, B')$ is defined by:

$$pos_a(B, B') = \{s \in B \mid \exists n \geq 0 \, \exists s_0, ..., s_n \in B \, \exists s' \in B' : \ s_0 = s, \ [\forall 0 < i \leq n : \ s_{i-1} \xrightarrow{\tau} s_i] \text{ and } s_n \xrightarrow{a} s'\}.$$

We say that B' is a *splitter of B with respect to a* iff $B \neq B'$ or $a \neq \tau$, and $\emptyset \neq pos_a(B, B') \neq B$. If P is a partition of S and B' is a splitter of B with respect to a, then $Ref^a_P(B, B')$ is the partition P where B is replaced by $pos_a(B, B')$ and $B - pos_a(B, B')$. P is *stable* with respect to a block B' if for no block B and for no action a, B' is a splitter of B w.r.t. a. P is *stable* if it is stable with respect to all its blocks.

The algorithm maintains a partition P that is initially P_0. It repeats the following step, until P is stable:

> find blocks $B, B' \in P$ and a label $a \in A$ such that B' is a splitter of B with respect to a;
> $P := Ref^a_P(B, B')$.

Theorem 5.1. *The above algorithm for the GRCPS problem terminates after at most $n - |P_0|$ refinement steps. The resulting partition P_f is the coarsest stable partition refining P_0.*
Proof. Similar to the proof of theorem 3.1. □

We must now show that one can find a splitter B' with respect to some label a in time $O(m)$ or find in $O(m)$ time that no such splitter exists. Moreover, a refinement must be carried out in $O(m)$ time. To this purpose we use the data structure of the RCPS algorithm. But now a transition $s \xrightarrow{a} s'$ is called *(P-)inert* if $s \sim_P s'$ and $a = \tau$, and a state $s \in B$ is a *bottom state* of B if $s \in B$ and there is no $s' \in B$ such that $s \xrightarrow{\tau} s'$. The data structure is initialized in the same way as for the RCPS algorithm. However, the non-inert transitions ending in a block B are grouped on label, i.e. all transitions with the same label are in subsequent records in the list. If there are non-inert transitions with a label τ ending in a block B, then they are at the beginning of the list. This facilitates adding inert transitions that become non-inert after a refinement at the beginning of the transition list. Grouping of the transitions has time complexity $O(m \log m)$ (heapsort) or $O(|A| + m)$ (bucket sort).

The following lemmas are the counterparts of theorem 3.1 and lemma 3.2. As the proofs are similar, they are omitted.

Lemma 5.2. *Let P be a refinement of P_0 and let $B, B' \in P$ and $a \in A$. Then B' is a splitter of B with respect to a iff*

1) $a \neq \tau$ or $B \neq B'$,

2) for some $r \in B$ *and* $r' \in B'$: $r \xrightarrow{a} r'$, *and*

3) there is a bottom state s *of* B *such that for no* $s' \in B'$: $s \xrightarrow{a} s'$.

Lemma 5.3. *Let* P, R *be partitions such that* R *refines* P*, and* P *and* R *have the same bottom states. Let* B *be a block of both* P *and* R *such that* P *is stable with respect to* B*. Then* R *is stable with respect to* B*.*

A splitter can be found in the same way as in the RCPS algorithm. Continue the following step until the list *tobeprocessed* is empty or a splitter has been found. Consider a block B from the list *tobeprocessed*. Consider subsequently all groups Φ of non-inert transitions ending in B with the same label a, set the *flag* field of the starting states of transitions in Φ and construct BL. A copy of Φ is maintained for resetting the flags. Then check stability of all blocks B' in BL with respect to B and label a and split B' if necessary. Due to lemma 5.2 and lemma 5.3 this can be performed in exactly the same way as in the RCPS case. Reset the flags of the states using the copy of Φ. If B splits itself into blocks B_1 and B_2, it is not necessary to check more transitions ending in B, as they must again be checked for B_1 and B_2. If all incoming transitions in block B have been checked, if B is not split and if there is no new bottom state, move B from *tobeprocessed* to *stable*.

Branching bisimulation is mostly defined on labelled transition systems (LTS's). The GRCPS-algorithm can be used to decide branching bisimulation on finite LTS's.

Definition 5.4. A *labelled transition system (LTS)* is a triple $\mathcal{L} = (S, A, \longrightarrow)$ with S a set of *states*, A a set of *labels* containing the *silent step* τ, and $\longrightarrow \subseteq S \times A \times S$ a *transition relation*. $\mathcal{L}$ is called *finite* if both S and A are finite.

Definition 5.5 *([12])*. Let $\mathcal{L} = (S, A, \longrightarrow)$ be an LTS. Let $\Rightarrow$ be the transitive and reflexive closure of $\xrightarrow{\tau}$. A relation $R \subseteq S \times S$ is a *branching bisimulation* iff it is symmetric and whenever $r\, R\, s$ and $r \xrightarrow{a} r'$, then either $a = \tau$ and $r'\, R\, s$, or there exist s_1, s' such that $s \Rightarrow s_1 \xrightarrow{a} s'$ and $r\, R\, s_1$ and $r'\, R\, s'$. Two states $r, s \in S$ are *branching bisimilar*, notation $r \leftrightarroweq_b s$, iff there exists a branching bisimulation relation relating r and s.

We could have strengthened this definition by requiring *all* intermediate states in $s \Rightarrow s_1$ to be related with r. The following lemma implies that this would lead to the same equivalence relation.

Lemma 5.6 *(cf. lemma 1.3 of [12]). Let* $\mathcal{L} = (S, A, \longrightarrow)$ *be an LTS and let for some* $n > 0$, $r_0 \xrightarrow{\tau} r_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_{n-1} \xrightarrow{\tau} r_n$ *be a path in* $\mathcal{L}$ *with* $r_0 \leftrightarroweq_b r_n$*. Then for all* $0 \leq i \leq n$: $r_0 \leftrightarroweq_b r_i$.

Theorem 5.7. *Let* $\mathcal{L} = (S, A, \longrightarrow)$ *be a finite LTS. Let* P_f *be the final partition obtained after applying the GRCPS algorithm on an initial partition containing only block* S*. Then* $\sim_{P_f} = \leftrightarroweq_b$.

Proof. "$\subseteq$" Using theorem 5.1 it follows that $\sim_{P_f}$ is a branching bisimulation relation.
"$\supseteq$" $\leftrightarroweq_b$ induces a stable partition on S (use lemma 5.6). As P_f is the coarsest stable partition, $\sim_{P_f} \supseteq \leftrightarroweq_b$. □

So in order to compute whether two states in a finite LTS are branching bisimilar we can apply the GRCPS algorithm with as initial partition the partition containing the set of states as only block. This takes $O(m \log m + m \cdot n)$ resp. $O(|A| + m \cdot n)$ time.

6 Concluding remarks

Is our $O(m \cdot n)$ algorithm for the RCPS problem optimal? We do not think so. In fact we conjecture that our algorithm can be slightly improved upon by incorporating ideas behind the $O(m \log n)$ algorithm of PAIGE and TARJAN [18] for the RCP problem. Let m_i be the number of inert transitions in the initial partition and let n_i be the number of states which have an outgoing inert transition in the initial partition but are bottom states in the final partition. We have the following conjecture:

Conjecture 6.1. *The RCPS problem can be decided in $O(m \cdot n_i + m_i \cdot n + m \log n)$ time, using $O(m)$ space.*

As already observed in remark 3.4, stability is not inherited under refinement in general: problems arise when, in a refinement, a non-bottom state becomes a bottom state. This situation can occur n_i times. The summand $m \cdot n_i$ in the expression above corresponds to the additional amount of work that has to be done to deal with these situations. At present we do not see how to avoid scanning all inert transitions when we do a refinement step. This explains the summand $m_i \cdot n$. If there are no inert transitions, then the algorithm which we conjecture is as efficient as the PAIGE and TARJAN [18] algorithm for the RCP problem. However, often m_i will be of the same order as m. In that case the order of complexity equals the one of our $O(m \cdot n)$ algorithm. For this reason, and also because the algorithm which we conjecture is rather complex (it combines the techniques of PAIGE and TARJAN [18] with the techniques of our $O(m \cdot n)$ algorithm), we decided to concentrate first on a clear exposition of the $O(m \cdot n)$ algorithm.

In a sense, branching bisimulation equivalence can be viewed as an alternative to observation equivalence. Thus it is interesting to compare the complexities of deciding these equivalences. First consider the situation where the set A of labels is fixed (so $O(m) \leq O(n^2)$). All known algorithms for deciding observation equivalence (see e.g. [2, 14]) work in two phases. First a transitive closure algorithm is used to compute the so-called double arrow relation. With a simple algorithm (see e.g. [1]) this takes $O(n^3)$ time. The result of the transitive closure is a new LTS with at most $O(n^2)$ more edges than the original LTS. Next a variant of the PAIGE and TARJAN [18] algorithm is used to decide *strong bisimulation equivalence* on the new LTS. This takes $O(n^2 \log n)$ time. The resulting complexity in this case for deciding observation equivalence is $O(n^3)$, which is the same as the complexity of our algorithm. Now there are numerous sub-cubic transitive closure algorithms in the literature (see e.g. [7] for an $O(n^{2.376})$ algorithm). These algorithms tend to be practical only for large values of n. Still we have that if the set of labels is fixed, the number of states is large and the number of transitions is of order $O(n^2)$, observation equivalence can be decided faster than branching bisimulation if one uses these sub-cubic algorithms.

However, things change if one does not fix the set of labels. Since one has to compute the double arrow relation for all labels that occur in the LTS, the complexity of computing the double arrow relation then becomes $O(m \cdot n^{2.376})$ (at least, we do not know any faster solution). In that case our algorithm for branching bisimulation is more efficient.

Clearly, the issue of comparing the complexities of observation equivalence and branching bisimulation is nontrivial and the analysis above does not give very much insight into the performance of our algorithm in practical applications. Therefore, we wrote a trial implementation in Pascal and compared the performance of this implementation with the performance of AUTO [19] and Aldébaran [11], as far as we know the two fastest tools currently available for deciding observation equivalence.

The process we used for our tests was the 'scheduler' as described by MILNER [16]. This scheduler schedules k processes in succession modulo k, i.e. after process k process 1 is reactivated again. However, a process must never be reactivated before it has terminated. The scheduler is constructed of k cyclers $C_1, ..., C_k$, where cycler C_i takes care of process i. The left part of figure 3 shows the transition system for cycler C_i. In the right part the architecture of the scheduler is depicted. The dotted lines indicate where the cyclers synchronize. Cycler C_i first receives a signal $\bar{c}_i$ which indicates

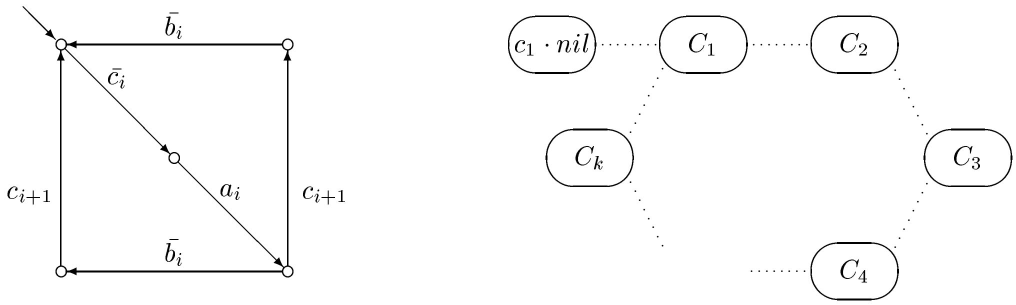


Figure 3: A cycler and a scheduler

			both a_i and $\bar{b}_i$ visible				only a_i visible			
k	states	trans.	AU	AB	BB	eq.cl.	AU	AB	BB	eq.cl.
4	97	241	0.5s	0.26s	0.07s	64	0.4s	0.15s	0.02s	4
5	241	721	1.9s	0.88s	0.3s	160	1.1s	0.6s	0.07s	5
6	577	2017	8.0s	2.6s	0.9s	384	3.3s	1.9s	0.2s	6
7	1345	5377	38s	7.2s	2.5s	896	12s	6.9s	0.5s	7
8	3073	13825	201s	21s	7.7s	2048	57s	24s	1.2s	8
9	6913	34561	-	56s	23s	4608	-	80s	2.9s	9
10	15361	84481	-	160s	67s	10240	-	-	7.4s	10
11	33793	202753	-	*	214s	22528	-	-	19s	11
12	73729	479233	-	*	1254s	49152	-	-	53s	12

Table 1: Some test results

that it may start. It then activates process i via an action a_i. Next, it waits for termination of process i, indicated by $\bar{b}_i$, and in parallel, using c_{i+1}, the next cycler is informed to start. Afterwards the cycler is back in its initial state. In a CCS-like language cycler C_i is described by:

$$C_i \;=\; \bar{c}_i \cdot a_i \cdot (\bar{b}_i | c_{i+1}) \cdot C_i.$$

The complete scheduler for k processes is described by:

$$Sch_k \;=\; (c_1 \cdot nil | C_1 | ... | C_k) \backslash c_1 ... \backslash c_k.$$

Here $c_1 \cdot nil$ is an auxiliary process that starts the first cycler.

The results of experiments with schedulers of different size are given in table 1. Here k is the number of cyclers per scheduler. The second and third column give the number of states resp. transitions of the corresponding transition system. Then we give the time necessary to calculate the bisimulation equivalence classes of the schedulers for AUTO (AU), Aldébaran (AB) and our trial implementation (BB). We give these figures not only for the case where labels a_i and $\bar{b}_i$ are both visible, but also for the case where actions $\bar{b}_i$ are hidden (i.e. renamed into τ). The figures for AUTO and our trial implementation have been obtained using a SUN 3/60 with 16 MB of memory. The figures for Aldébaran, which are taken from [11], were obtained with a 50 MB SUN 3/60. It is important to note that these figures refer only to the second phase of the algorithm where the strong bisimulation equivalence classes are computed. So the time it takes to carry out the first phase (the transitive closure) is not included. This means that, roughly speaking, the figures for Aldébaran must be multiplied by 2. The figures for AUTO refer to the time needed for the complete algorithm. In separate columns the number of resulting equivalence classes of both experiments are given. They are the same for branching bisimulation and observation equivalence. In the table, "-" means that no outcome was obtained due to lack of memory and "*" means that no outcome is available.

Our implementation improves the performance of Aldébaran and AUTO considerably, especially when a lot of τ's are around. For the space requirements this is directly reflected in the fact that, in the case where only the a_i-actions are visible, we can handle 12 cyclers on a 16 MB machine, whereas Aldébaran, on a 50 MB machine, can handle only 9 cyclers and AUTO, on a 16 MB machine, only 8. The figures about the time performance also show a considerable improvement (up to a factor 47). So it appears that in practical situations our algorithm is doing better than the usual algorithms for observation equivalence. However, we find it hard to draw firm conclusions from the results of our experiments since they are influenced by many uncontrollable factors.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] 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.

[3] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1,2):115–131, 1988.

[4] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[5] E.M. Clarke and O. Grümberg. Research on automatic verification of finite state concurrent systems. *Ann. Rev. Comput. Sci.*, 2:269–290, 1987.

[6] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings 4^{th} Annual Symposium on Logic in Computer Science,* Asilomar, California, pages 353–362, Washington, 1989. IEEE Computer Society Press.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19^{th} Annual ACM Symposium on Theory of Computing,* New York City, pages 1–6, 1987.

[8] R. De Nicola, U. Montanari, and F.W. Vaandrager. Back and forth bisimulations. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer-Verlag, 1990. Also appeared as CWI Report CS-R9021.

[9] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. To appear in *Proceedings Ecole de Printemps on Semantics of Concurrency*, April 90, (I. Guessarian), Springer-Verlag, LNCS, 1990.

[10] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not Never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[11] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.

[12] 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.

[13] 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, May 1989, Iowa, USA, pp. 197–201.

[14] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the* 2^{nd} *Annual ACM Symposium on Principles of Distributed Computing,* Montreal, Quebec, Canada, 1983.

[15] L. Lamport. What good is temporal logic? In R.E. Mason, editor, *Information Processing 83*, pages 657–668. North-Holland, 1983.

[16] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[17] R. Milner. *Communication and concurrency.* Prentice Hall International, 1989.

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

[19] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.