Computing Maximal Bisimulations

Alexandre Boulgakov, Thomas Gibson-Robinson, and A.W. Roscoe

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
{alexandre.boulgakov, thomas.gibson-robinson, bill.roscoe}@cs.ox.ac.uk

Abstract. We present and compare several algorithms for computing the maximal strong bisimulation, the maximal divergence-respecting delay bisimulation, and the maximal divergence-respecting weak bisimulation of a generalised labelled transition system. These bisimulation relations preserve CSP semantics, as well as the operational semantics of programs in other languages with operational semantics described by such GLTSs and relying only on observational equivalence. They can therefore be used to combat the space explosion problem faced in explicit model checking for such languages.

1 Introduction

Many different variations on bisimulation have been described in the literature of process algebra, for example [1–5]. They are typically used to characterise equivalences between nodes of a labelled transition system (LTS), but they can also be used to calculated state-reduced LTSs that can represent equivalent processes. They have the latter function in the CSP-based [6–8] refinement checker FDR [9], of which the third major version FDR3 has recently been released [10]. The present paper sets out the approaches to bisimulation reduction taken in FDR and especially FDR3.

FDR typically builds the transition system of a large process as the parallel composition (closely related to Cartesian product) of those of component processes, which are often sequential. One of the approaches it takes to the state explosion problem is to supply a number of compression functions that attempt to reduce the state spaces of these components. The set of compressions introduced in [11], which included *strong* bisimulation, has been extended by several other versions of bisimulation in the most recent versions of FDR.

The main purpose of this paper is to set out the bisimulation algorithms used by FDR3 and compare them with alternatives. Our strong bisimulation algorithm is related to Paige and Tarjan's bisimulation algorithm [12, 13], and is compared with that. When more compression is needed, other tools frequently use branching bisimulation [14] due to the existence of an efficient $O(nt)$ algorithm [15]. In contrast, FDR3 uses the even coarser delay and weak bisimulations; we present innovative algorithms to compute these bisimulations based on dynamic programming. These latter algorithms were introduced because, although

they typically achieve slightly poorer compression than FDR's existing compressions, bisimulations are more widely applicable. In Section 5.2 we compare these two classes of compressions.

2 Strong Bisimulation

FDR uses LTSs in which nodes sometimes have additional behaviours represented by labellings such as divergences or minimal acceptances.

Definition 1. A *generalised labelled transition system* (GLTS) is a tuple $(N, \Sigma, E, \Lambda, \lambda)$ where N is a set of nodes, Σ is a set of events, $\Sigma^\tau = \Sigma \cup \{\tau\}$, $\longrightarrow \subseteq N \times \Sigma^\tau \times N$ is a labelled transition relation (with $p \xrightarrow{a} q$ indicating a transition from p to q with action a), Λ is a set of labels, and $\lambda : N \to \Lambda$ is a total function labelling each node. The following shorthand is used:

- $\mathit{initials}(m) = \{e \mid \exists n \cdot m \xrightarrow{e} n\}$ denotes m's initial events;
- $\mathit{afters}(m) = \{(e, n) \mid m \xrightarrow{e} n\}$ denotes m's directly enabled transitions;
- $m \Uparrow \Leftrightarrow \exists m_0, m_1, \ldots \cdot m_0 = m \wedge \forall\, i \cdot m_i \xrightarrow{\tau} m_{i+1}$ denotes *divergence*, i.e. an infinite cycle of internal τ actions corresponding to *livelock*.

Definition 2. A relation $R \subseteq N \times N$ is a *strong bisimulation* of a GLTS S if and only if it satisfies all of the following, where $n_1, n_2, m_1, m_2 \in N$ and $x \in \Sigma^\tau$:

$$\forall\, n_1, n_2, m_1 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_1 \xrightarrow{x} m_1 \quad \Rightarrow \quad \exists m_2 \in N . n_2 \xrightarrow{x} m_2 \wedge m_1 R\, m_2$$
$$\forall\, n_1, n_2, m_2 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_2 \xrightarrow{x} m_2 \quad \Rightarrow \quad \exists m_1 \in N . n_1 \xrightarrow{x} m_1 \wedge m_1 R\, m_2$$
$$\forall\, n_1, m_1 \cdot n_1\, R\, n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$

Two nodes are *strongly bisimilar* if and only if there exists a strong bisimulation that relates them. The *maximal strong bisimulation* on a GLTS S is the relation that relates two nodes if and only if they are strongly bisimilar. The FDR function `sbisim` computes the maximal strong bisimulation on its input GLTS and returns a GLTS with a single node bisimilar to each equivalence class in the input. FDR has included the `sbisim` compression function since its early days. However the algorithm has not been described in the literature in detail (a brief outline is found in [8]) until now.

2.1 Naïve Iterative Refinement

The FDR2 implementation of `sbisim` first computes the desired equivalence relation as a two-directional one-to-many map between equivalence class and node identifiers. It then generates a new GLTS based on the input and the computed equivalence relation. This final step is straightforward to implement and dependent more on the internal GLTS format than the strong bisimulation algorithm and will not be discussed in this paper. Furthermore, it is not specific to strong bisimulation and can be used to factor a GLTS by an arbitrary relation.

It is computing the desired equivalence relation that requires the most effort both on the part of the algorithm designer and on the part of the computer.

A coarse approximation of the equivalence relation is first computed by using the first-step behaviour of each node, and each class in this relation is repeatedly refined using the first-step behaviours of the nodes under the current approximation. This is related to the formulation of strong bisimulation given in [2] as a series of experiments of increasing depth.

Initial Approximation. The initial approximation can most simply be computed by identifying all nodes. However, FDR employs a finer initial approximation that saves time later on.

Unlike the *afters* of a node, whose equivalence depends on the current equivalence relation, these are fixed labels and we can save time by only comparing them once. We can compute an initial approximation by comparing the nodes' labels and *initials* and need not look at the labels again. This is equivalent to identifying all nodes and then performing one refinement using the nodes' labels and their *afters*' equivalence classes.

Iteration. Assume that we have already separated the nodes into equivalence classes, whether from the initial approximation or from a previous refinement step. We will now attempt to refine these classes further. After computing the *afters* of each node under the latest equivalence relation, we sort the *afters* for the nodes in each class in order to reclassify them. A single in-order traversal through the sorted lists allows us to reclassify the nodes in each class.

If any nodes have changed class during this pass, we must proceed to refine the classes again. Otherwise, we are done. We can determine whether any nodes have changed class during the final reclassification traversal with very little additional work.

Construction. The final step is to construct the output GLTS. To do this, we first create a node for each equivalence class. Next, we can use the already computed *afters* to create the transition system, using an arbitrary representative from each class since the *afters* for each of the nodes in an equivalence class are guaranteed to be equivalent (since the refinement phase has terminated). Any node labels can also be copied from the representative directly as they are guaranteed to be equivalent.

Complexity. The initial approximation takes up to a constant factor more time than one iterative step and construction only requires a traversal of the output of the iterative step, so the run time is dominated by the iteration.

Assume an input GLTS with n nodes and t transitions. In FDR's representation, the transition set is sorted first by source and then by event, so recomputing the *afters* is a relatively inexpensive operation requiring simply an in-order traversal of the transition set and a random lookup per transition to compute the equivalence class of its destination, taking $O(t)$ time. The following sort can take $O(n \log(n))$ time in the worst case where the nodes are spread across few classes, and the reclassification is done in $O(n)$ time. Since we refine at least one class on each iteration except the final one, there can be no more than n iterations. The worst-case time complexity is therefore in $O(nt + n^2 \log(n))$.

2.2 Change-Tracking Iterative Refinement

We will now present an improvement on Naïve Iterative Refinement that is included in FDR3. With some bookkeeping, we can determine which states' *afters* could not have changed after the previous iteration. The proposed algorithm uses this information to avoid recomputing and sorting the *afters* for these states.

It is clear that in a process such as $P(n)$ where

$$P(0) = STOP \qquad P(n) = a \rightarrow P(n-1)$$

naïve iterative refinement would first identify all the states except $P(0)$, then split off $P(1)$, then $P(2)$, and so on, recalculating the *afters* for all $n+1$ nodes, and sorting a list of $n-i$ elements and $i+1$ lists of 1 element on the i^{th} iteration. However, we can note that only one node changes class on each iteration, so we need only to recompute the *afters* for that node.

We can also use this knowledge to reduce the number of nodes that have to be sorted. To do so, we must keep track of which nodes are affected by each node (that is, a copy of the transition system with the transitions reversed and the labels removed), and we must also keep track of which nodes change class on each iteration. As FDR represents states as consecutive integers and the transitions are stored in an array, we can easily construct a constant-time accessible map from nodes to their predecessors.

We will maintain as running state a bit vector *changed* containing the nodes whose equivalence class changed on the previous iteration, and a bit vector *affected* containing the nodes that might be affected by those changes. *affected* should be initialised with all nodes marked since we need to compute the *afters* for all of the nodes initially.

Following this initialisation, on each iteration we will perform the following sequence of actions. First, we will recompute the *afters* for each of the nodes in *affected*. All nodes that are not marked for update get to keep their *afters* from the previous iteration. Next, we compute the equivalence classes that contained the affected nodes in the previous iteration; these are the equivalence classes that might need to be refined, and this can be computed in linear time in the number of nodes by iterating over *affected*. We must also clear *changed* for the next step.

For each of the classes that we consider for refinement, we first separate the nodes that have not changed class from those that have (which are in *affected*). Next, we sort the nodes that are in *affected* and in this class in order to partition this class. At this point, we can go through the sorted nodes and assign each group a new class index. However, this does not perform well on examples like R (with the LTS shown in Figure 1) where

$$\begin{aligned} R &= (\Box_{i \in \{0..n\}}\ b \rightarrow b \rightarrow Q(i)) \,\Box\, a \rightarrow R' \\ R' &= \Box_{i \in \{0..n\}}\ a \rightarrow Q(i) \\ Q(i) &= a \rightarrow STOP \end{aligned}$$

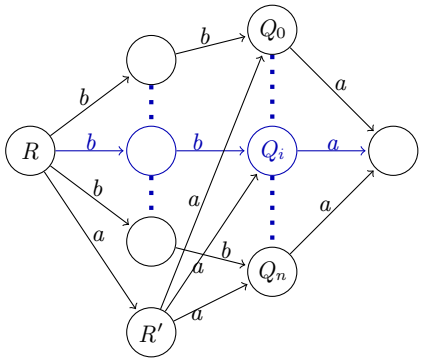


Fig. 1: R' has the same *initials* as each of the Q_i, but is in a different equivalence class from them. An initial classification based on *initials* would therefore place them in the same equivalence class, but a future refinement would reclassify either R' or all of the Q_i, which would change the *afters* of R only or all of the $b \to Q(i)$, respectively.

In particular, the initial classification places R' and each of the $Q(i)$ into the same equivalence class, but the next iteration reclassifies each of the $Q(i)$. This forces each of the $b \to Q(i)$ to be reclassified as well. However, when splitting an equivalence class we are free to assign class indices in an arbitrary way; in particular, rather than changing the indices of the $Q(i)$, we could instead have changed the index of the single node R', which has the single predecessor R. To do this algorithmically, once we have sorted the classes of all of the affected nodes in a given class, we choose the *largest* sequence of nodes with the same class to keep the original class index and assign new indices to the rest, rather than picking the *first* such sequence. We must also record the nodes that had new indices assigned in *changed*.

Once we have refined each of the classes that needed to be refined, we can iterate through *changed* and add to *affected* each of their predecessors for the next iteration. If *changed* is empty, we can conclude that we have reached a fixed point, and we can terminate the algorithm, returning the bisimulation relation we have computed implicitly in the equivalence class indices of the nodes.

2.3 Paige-Tarjan Algorithm

The algorithm outlined in [13] is an adaptation of Paige and Tarjan's solution (described in Section 3 of [12]) to the relational coarsest partition problem (which is equivalent to single-action strong bisimulation) that works with LTSs by splitting with respect to each element of the alphabet in sequence whenever the original algorithm would split a class. In summary, each time a class is split, the resulting subclasses are recorded. Refinement is then performed with respect to the initial classes (separating nodes with edges into each class from those

	Naïve (s)	Change-Tracking (s)	P-T (s)
Total	**186**	**40**	**55**
1	22.51	3.38	4.32
2	22.03	2.89	3.84
3	21.99	2.84	3.68
4	21.80	2.70	3.56
5	17.10	2.69	3.07

Table 1: `sbisim` timings. Total runtime and the 5 longest invocations for each algorithm (not necessarily corresponding to the same inputs).

without) and with respect to each split class (separating nodes with edges into one subclass, the other, or both) using the inverse labelled transition relation.

Complexity. The worst-case time complexity for a graph with n nodes and t transitions is in $O(t \log(n))$. However, the cached in-counts (the *info maps* in [13]) necessary to achieve this bound can be unwieldy to manipulate, raising the implementation and runtime costs. In addition, as the algorithm requires frequent construction and traversal of sets, there is a time or space penalty depending on the set representation used.

2.4 Performance

We will now compare the performance of the three algorithms for `sbisim` on several real-world and generated examples. The test system used contains a medium-range CPU[1] and 4 GiB of memory, running 64-bit Ubuntu 12.10.

FDR3 Test Suite. To ensure proper operation of FDR3, we have developed a suite of regression and feature tests containing tests generated randomly at runtime, examples from [7] and [8], and assorted test files. Since `sbisim` is applied to all component processes by default, many of these tests exercise `sbisim`. There are about 60,000 invocations of `sbisim` over the test suite, and they are a good comparison of the algorithms' performance on small leaf components typical in a system that does not use `sbisim` explicitly. The move from naïve to change-tracking iterative refinement affords a nearly five-fold speedup, as evidenced by Table 1. The Paige-Tarjan algorithm is slightly slower than change-tracking iterative refinement, but part of this might be due to the heavy optimisation that our implementation of iterative refinement has gone through over the years.

Towers of Hanoi. The first example is a model of the classic Towers of Hanoi puzzle. The puzzle consists of three rods and N disks of varying sizes, with an invariant that the disks on any rod are arranged in ascending order. A move consists of moving the topmost disk from one rod to another, while preserving the invariant. The objective is to move all the disks from one rod to another. There are 3^N possible configurations, since each disk can be on any of the three

[1] The CPU is a quad-core Intel® Core™ i5-750 with 8 MB of cache. The number of cores is not relevant, since the strong bisimulation algorithm is single threaded.

rods at any time and its position on the rod is determined by the other disks on the rod (due to the invariant). From each configuration, either two or three others are reachable: if all the disks are on one rod, the two valid moves are to move the topmost disk to either of the two remaining rods, and if not all the disks are on one rod, the smallest of the topmost disks can be moved to either of the two other rods and the second smallest to one rod. In our model, if all the disks are on one rod, the system can also perform a completion event but remain in the same configuration, resulting in 3^{N+1} transitions. We have hidden (i.e., renamed to τ) all events except for one completion event (as one might do when solving the puzzle using FDR) and applied `sbisim` to the result.

As we can see from Table 2, there is a significant speedup due to change-tracking iterative refinement that grows even more pronounced as the problem size grows. The Paige-Tarjan algorithm is consistently faster, likely due to the small amount of branching.

Dining Philosophers. The next example is a model of the Dining Philosophers problem with N right-handed philosophers. We hide all visible events (so the states are now distinguished by how many events must occur before the inevitable deadlock) and apply `sbisim` to the result. We also do the same for two deadlock-free solutions (which therefore have a single state after hiding and strong bisimulation, obtained in one step of iterative refinement). The first solution involves introducing asymmetry by making one of the philosophers left-handed. The second solution introduces a butler who ensures there are never more than $N - 1$ of the philosophers seated.

As we can see in Table 2, change-tracking iterative refinement is significantly faster than naïve iterative refinement for the deadlocking problem, and increasingly so for larger numbers of philosophers, but somewhat slower for the non-deadlocking variants. This is likely due to the fact that the additional bookkeeping it must perform does not have a chance to become useful – there is no second iteration after all the nodes are identified. However, the slowdown is not significant (less than twofold). The modified Paige-Tarjan algorithm is intermediate to the two variants of iterative refinement for the deadlocking cases and slower than both for the non-deadlocking variants.

Matrix. The final example is a matrix of $N + 1$ by $N + 1$ nodes, each being able to transition to the node on its right or the node below it with an event a. This is process $Q(N)$ where

$$\begin{aligned} P(0) &= STOP \\ P(n) &= a \rightarrow P(n-1) \\ Q(n) &= P(n) \;|||\; P(n) \end{aligned}$$

It has $(N + 1)^2$ states and $2N * (N + 1)$ transitions, but `sbisim` can reduce this to $2N + 1$ states and $2N$ transitions, since each node simply performs a number of as and deadlocks, and the number is no more than $2N$, N from each of the component $P(N)$.

Summary. For most of our experiments, change-tracking iterative refinement and the modified Paige-Tarjan algorithm both outperformed naïve iterative re-

Problem	States		Transitions		Naïve	CTIR	P-T
	Output	Input	Output	Input			
Hanoi $N = 8$	1,645	6,561	4,927	19,683	1.15	0.099	0.038
Hanoi $N = 9$	4,926	19,683	14,769	59,049	9.37	0.467	0.184
Hanoi $N = 10$	14,768	59,049	44,294	177,147	79.8	2.96	0.631
Hanoi $N = 11$	44,293	177,147	132,868	531,441	702	18.2	2.91
5 Phils (deadlock)	1,558	7,774	6,825	34,241	0.283	0.034	0.087
6 Phils (deadlock)	7,825	46,656	41,054	246,613	3.06	0.399	1.13
7 Phils (deadlock)	39,994	279,934	246,549	1,726,257	29.6	2.99	11.2
7 Phils (butler)	1	218,751	2	1,266,616	0.354	0.597	0.852
7 Phils (asymm)	1	266,604	2	1,641,653	0.454	0.759	1.126
$Q(100)$	201	10,201	201	20,201	1.03	0.064	0.024
$Q(300)$	601	90,601	601	180,601	66.1	1.77	0.418
$Q(1000)$	2,001	1,002,001	2,001	2,002,001	4540	60.2	16.3

Table 2: `sbisim` statistics. Times for Naïve Iterative Refinement, Change-Tracking Iterative Refinement, and the Paige-Tarjan algorithm in seconds.

finement and exhibited similar performance. For problems where only a small number of refinements were required, they were slightly slower due to bookkeeping overhead, but not significantly so.

3 Divergence-Respecting Delay Bisimulation

While FDR has long supported strong bisimulation, it has only recently supported variants of weak bisimulation. This was because the weak bisimulation of [2] is not compositional for most CSP models and because FDR already had compressions (e.g., `diamond` and `normal`) that successfully eliminated τ actions. However the implementation of priority, Timed CSP, and semantic models such as refusal testing in FDR created the need for further compressions, since `diamond` is not compositional with these and `normal` is problematic. The first compression introduced for this reason is `dbisim` (called `wbisim` when it was introduced in FDR 2.94), which returns the maximal divergence-respecting delay bisimulation (DRDB) of its input.

Given the transition relation $\longrightarrow$ of a GLTS S, let us define a binary relation $\Longrightarrow$ such that $p \Longrightarrow q$ if and only if there is a sequence $p_0, ..., p_n$ (with n possibly 0) such that $p = p_0$, $q = p_n$, and $\forall\, i < n.p_i \xrightarrow{\tau} p_{i+1}$. Let us further define a ternary relation $\hookrightarrow$ with $p \overset{a}{\hookrightarrow} q$ for $a \in \Sigma$ if and only if $\exists p'.p \Longrightarrow p' \wedge p' \xrightarrow{a} q$, and $p \overset{\tau}{\hookrightarrow} q$ if and only if $p \Longrightarrow q$. We will refer to this relation as the delayed transition relation, since the visible events are delayed by 0 or more τs.

Definition 3. A relation $R \subseteq N \times N$ is a *divergence-respecting delay bisimulation* of a GLTS S if and only if it satisfies all of the following requirements, where $n_1, n_2, m_1, m_2 \in N$ and $x \in \Sigma^\tau$:

$$\forall\, n_1, n_2, m_1 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_1 \xhookrightarrow{x} m_1 \quad \Rightarrow \quad \exists m_2 \in N . n_2 \xhookrightarrow{x} m_2 \wedge m_1 R\, m_2$$
$$\forall\, n_1, n_2, m_2 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_2 \xhookrightarrow{x} m_2 \quad \Rightarrow \quad \exists m_1 \in N . n_1 \xhookrightarrow{x} m_1 \wedge m_1 R\, m_2$$
$$\forall\, n_1, n_2 \cdot n_1\, R\, n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$
$$\forall\, n_1, n_2 \cdot n_1\, R\, n_2 \quad \Rightarrow \quad n_1 \Uparrow \Leftrightarrow n_2 \Uparrow$$

Note that the definition is very similar to that of strong bisimulation. The differences are the use of the delayed transition relation and the added clause about divergence, which is necessary to make the compression compositional for CSP. However, if we precompute divergence information and record it in each node's label, the requirement that $n_1 \Uparrow \Leftrightarrow n_2 \Uparrow$ will be absorbed into the requirement that $\lambda(n_1) = \lambda(n_2)$.

The FDR compression function `dbisim` computes the maximal DRDB on its input GLTS and returns a GLTS with a single node DRD-bisimilar to each equivalence class in the input. It is an important compression because it preserves semantics in all CSP models, while potentially offering a significantly higher amount of compression than strong bisimulation. FDR has included this compression since version 2.94 as an effective compression for CSP models richer than the failures model [16]. However the algorithm has not been described in the literature until now.

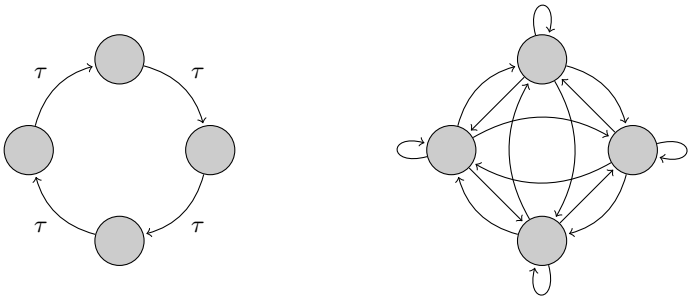
3.1 Reduction to Strong Bisimulation

FDR2 employs an adaptation of the naïve iterative refinement discussed in 2.1 to compute a maximal DRDB. A naïve implementation can apply the algorithm directly to an input with nodes containing divergence information, but for any of the requested properties (*initials* or labels) consider the τ-closure of the node (all nodes reachable from the given node by a sequence of τs) and allow the behaviours of each of the nodes in the τ-closure for the given node.

For an input GLTS S, we can compute a GLTS $\widehat{S}$ with a transition for each delayed transition of the input and mark each node with divergence information computed from S. Care is required not to introduce divergences not present in S due to the τ self-loops introduced in $\widehat{S}$ because the original node can take an empty sequence of τs to itself. The maximal strong bisimulation of $\widehat{S}$ is the maximal DRDB of S by construction.

Complexity. A significant problem with this approach is the high worst-case space complexity. $\widehat{S}$ can have up to An^2 transitions if the input has n nodes and an alphabet of size A, even if S has $o(An^2)$ transitions. For example, a process that performs N τs before recursing exhibits this worst-case behaviour. Since all nodes are mutually τ-reachable, a transition system with N^2 transitions is constructed. Figure 2 demonstrates this quadratic explosion for $N = 4$.

Construction of $\widehat{S}$ can take a correspondingly significant amount of time. For example, using an adaptation of the Floyd-Warshall algorithm [17] requires $O(n^3)$ operations. The strong bisimulation step after this transformation takes up to $O(n^3)$ operations since the number of transitions can grow to $O(n^2)$ and dominate the $n \log(n)$ term.



(a) The input, $P(4)$, has only four transitions and four nodes.

(b) The output has sixteen transitions for the same four nodes. Labels have been omitted for clarity.

Fig. 2: The constructed LTS can be quadratically larger than the input.

3.2 Dynamic Programming Approach

Rather than constructing $\widehat{S}$ and keeping it in memory (which is often the limiting factor for such computations, since main memory is limited and the hard disk is prohibitively slow given the random nature of the accesses required by parts of the strong bisimulation algorithm), FDR3 instead recomputes the relevant information using the original transition system on each refinement iteration.

Algorithm. First, noting that two nodes on a τ loop are both DRD-bisimilar and divergent, we factor the input GLTS S by the relation that identifies nodes on a τ loop. FDR has a function built in that does this, `tau_loop_factor`. We will not discuss it in detail here, but it uses Tarjan's algorithm for finding strongly connected components [18] via a single depth-first search and runs in $O(n + t)$ time for a system with n nodes and t transitions. In addition to eliminating τ loops, it marks each node as divergent or stable. Now that we have ensured there are no τ loops, the τ-transition relation can be used to topologically sort the nodes with another depth-first search [19], so that there are no upstream τ-transitions.

The topological sort allows us to obtain the transitions of the $\widehat{S}$ described in Section 3.1 using a dynamic programming approach. The last node in this topological sort has no outgoing τ transitions, so its new *initials* and *afters* are precisely those in S with the addition of itself after τ. We then proceed upstream and for each node compute the union of its own *afters* (with the inclusion of a self-transition under τ) and the *afters* of each of the nodes it can reach under a single τ transition. Of course, since we are doing this in a topological order, these nodes have been processed already, so we have computed the union of the *afters* of all τ-reachable nodes from the given node.

We can apply a modified Naïve Iterative Refinement (Section 2.1) to compute the maximal strong bisimulation of $\widehat{S}$, which is itself never constructed. The faster CTIR or modified Paige-Tarjan algorithms require the inverted transition relation, and we have not found a way to do this dynamically. We compute the *initials* and labels for the initial approximation using dynamic programming on the topologically sorted nodes. For each refinement, we compute the equivalence classes of the *afters* using the dynamic programming approach described above, but keeping track of equivalence classes rather than node identifiers for each *after*. For the construction step, we compute the equivalence classes of the *afters* as above, but without inserting the τ self-transition.

Complexity. The space complexity for this algorithm is never significantly higher than that of the explicit reduction, and can be significantly lower. The only additional information we have is the transient DFS stack and bookkeeping information, and the sorted node list. The *afters* we compute for each node take no more space than the exploded transition system, and will take less if any nodes are identified – and if the user is running the algorithm there is reason to believe that they will be. In addition, since the *afters* are recomputed at each iteration, the working set for each refinement iteration can be smaller than the peak working set required by the final one. For example, for the process $P(N)$ portrayed in Figure 2, the initial classification will identify all nodes, and the first *afters* computation will have a single *after* for each node: equivalence class 0 under τ.

We still traverse the entire transition set a single time (split across nodes). But now, for each node, we have to take the union of its *afters* and the ones preceding it. Provided we keep these sorted, and use a merge sort for union, we will have in the worst case $O(Acn)$ operations for each node, where A is the size of the alphabet, c is the number of classes in this iteration, and n is the number of nodes, since Ac is the maximal number of *afters* a node could have and we could have $O(n)$ nodes following this one. This means an upper bound on the overall worst-case runtime is $O(An^4)$.

However, in practice the time complexity is much lower. Removing τ loops ensures that the graph is not fully connected and reduces the number of unions for each node significantly. The number of classes c is often much less than n. In addition, there are further optimisations that could be made to reduce the runtime, the union operation can be made faster by keeping metadata that allows us to avoid unioning duplicate *afters* sets. Section 5.2 demonstrates that the dynamic programming approach is faster on many examples with a large number of τs than the explicit reduction approach.

4 Divergence-Respecting Weak Bisimulation

FDR3 adds support for compression by an even weaker equivalence relation, divergence-respecting weak bisimulation (DRWB).

Given the transition relation $\longrightarrow$ of a GLTS S and the binary relation $\Longrightarrow \equiv \xrightarrow{\tau}{}^{*}$, let us define a ternary relation $\Longrightarrow$ with $p \stackrel{a}{\Longrightarrow} q$ for $a \in \Sigma$ if and

only if $\exists p', q'.p \Longrightarrow p' \wedge p' \xrightarrow{a} q' \wedge q' \Longrightarrow q$, and $p \xrightarrow{\tau} q$ if and only if $p \Longrightarrow q$. We will refer to this relation as the observed transition relation.

Definition 4. A relation $R \subseteq N \times N$ is a *divergence-respecting weak bisimulation* of a GLTS S if and only if it satisfies all of the following requirements, where $n_1, n_2, m_1, m_2 \in N$ and $x \in \Sigma^\tau$:

$$\begin{array}{lcl}
\forall\, n_1, n_2, m_1 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_1 \overset{x}{\Longrightarrow} m_1 & \Rightarrow & \exists m_2 \in N . n_2 \overset{x}{\Longrightarrow} m_2 \wedge m_1 R\, m_2 \\
\forall\, n_1, n_2, m_2 \cdot \forall\, x \cdot n_1 R\, n_2 \wedge n_2 \overset{x}{\Longrightarrow} m_2 & \Rightarrow & \exists m_1 \in N . n_1 \overset{x}{\Longrightarrow} m_1 \wedge m_1 R\, m_2 \\
\forall\, n_1, n_2 \cdot n_1\, R\, n_2 & \Rightarrow & \lambda(n_1) = \lambda(n_2) \\
\forall\, n_1, n_2 \cdot n_1\, R\, n_2 & \Rightarrow & n_1 \Uparrow \Leftrightarrow n_2 \Uparrow
\end{array}$$

Note that the definition is very similar to that of divergence-respecting delay bisimulation. The only difference is the use of the observed transition relation in place of the delayed transition relation.

The FDR3 compression function `wbisim` computes the maximal DRWB on its input GLTS and returns a GLTS with a single node DRW-bisimilar to each equivalence class in the input. It is an important compression because, like `sbisim` and `dbisim` it preserves semantics in all CSP models, while potentially offering a higher amount of compression than `dbisim`. This compression is new in FDR3 and is the strongest implemented compression for CSP models richer than the failures model.

4.1 Algorithm

We proceed in a manner similar to that described in Section 3.2. Noting that two nodes on a τ loop are both DRW-bisimilar and divergent, we factor the input GLTS by the relation that identifies nodes on a τ loop using `tau_loop_factor`. We then topologically sort the nodes by the τ-transition relation.

The topological sort allows us to obtain the observed transitions using a two-pass dynamic programming approach. One pass as in delay bisimulation is not sufficient here since we need to determine the τ^* *afters* of the visible *afters* of each node, and these visible *afters* might not have been previously explored. In the first pass, we compute the τ^* *afters* of each node. The last node in this topological sort has no outgoing τ transitions, so its only τ^* *after* is itself. We then proceed upstream and for each node compute the union of its own τ *afters* (with the inclusion of itself) and the previously computed τ^* *afters* of each of the nodes it can reach under a single τ transition. The second pass computes the visible observed transitions. For each node, these are the union of the τ^* *afters* of its visible *afters* and the visible observed transitions of its τ *afters*. If we proceed in topological order, the visible observed transitions of each node's τ *afters* will have already been computed by the time they are needed.

We can apply a modified Naïve Iterative Refinement to compute the maximal strong bisimulation of the induced GLTS as in Section 3.2, removing the τ self-transition from each node in the construction step.

Complexity. In the typical case this algorithm will require more space to store the *afters* than the DRD-bisimulation algorithm since it must follow the

τ transitions after a visible event in addition to the ones tracked by the DRD-bisimulation algorithm. However, the worst-case space complexity for this algorithm is the same, since in the worst case all the nodes are mutually reachable under both the delayed transition relation and the observed transition relation. The time complexity is a constant factor greater since at each iteration two passes through the topologically sorted nodes must be performed.

However, in practice we have found that `wbisim` is nearly as fast as `dbisim`, and produces identical results on all example files other than ones we have contrived to prove that the two are in fact different.

5 Performance

5.1 Diamond Elimination

It is interesting to compare `dbisim` with alternatives available in FDR prior to its introduction. The most widely used compression was `sbisim`(`diamond`(P)), which we will call `sbdia`. In all the following examples `sbdia` is valid.

5.2 Timing

This section is primarily to compare the runtimes of `sbdia` and the algorithms we have presented for `dbisim` and `wbisim`. We will use the same system as for the `sbisim` tests, described in 2.4. Reduction to strong bisimulation has only been implemented in FDR2 and the dynamic programming approach has only been implemented in FDR3, so the timings are not directly comparable due to differences in other components such as the compiler, which is single-threaded in FDR2 and multi-threaded in FDR3. However, the examples have been designed to heavily use `dbisim`, and most of the runtime will be due to `dbisim` rather than these other components. We will use the same examples here as in the `sbisim` tests, so section 2.4 should be consulted for more details.

Towers of Hanoi. As in the `sbisim` test, we have hidden all events except for one completion event, resulting in a strongly connected network of τs, with a single visible transition. `dbisim` reduces this to a system with one node and two transitions in one iteration. However, FDR2 does not always reach this iteration – 4 GiB of RAM is not enough for an exploded transition system corresponding to $N \geqslant 8$, and it uses 537 MiB for the $N = 7$ problem, while FDR3 uses only 700 MiB for the 729 times larger $N = 13$ problem and 1.9 GiB for the 2187 times larger $N = 14$ problem. The situation is similar for the Dining Philosophers.

Matrix. The matrix example perhaps shows best the difference between the two algorithms using P and Q as defined in 2.4 and $R(n) = Q(n) \setminus a$.

We have tested both $Q(N)$ and $R(N)$ for various values of N. The FDR2 algorithm, which explicitly constructs an LTS representing the delay transitions performs better on Q, which doesn't contain any τs (so the exploded transition system is the same size as the original one), since the dynamic programming approach performs unnecessary work at each iteration as well as at the start.

Problem	States	Transitions	Explicit `dbisim`	Dynamic `dbisim`	`wbisim`	`sbdia`
Hanoi (7)	2,187	6,561	13	0.03	0.05	0.04
Hanoi (8)	6,561	19,683	–	0.08	0.08	0.08
Hanoi (12)	531,441	1,594,323	–	2.49	2.49	2.49
Hanoi (13)	1,594,323	4,782,969	–	6.83	6.85	6.72
Hanoi (14)	4,782,969	14,348,907	–	24.1	26.21	24.1
$Q(10)$	2,601	5,101	0.01	0.01	0.01	0.01
$Q(100)$	10,201	20,201	1.29	2.05	2.18	0.10
$Q(300)$	90,601	180,601	27.5	109	109	2.28
$R(10)$	2,601	5,101	0.02	0.01	0.01	0.01
$R(100)$	10,201	20,201	69.4	0.03	0.03	0.02
$R(300)$	90,601	180,601	–	0.38	0.39	0.11
$R(1000)$	1,002,001	2,002,001	–	4.63	4.89	1.35

Table 3: `dbisim`, `wbisim`, and `sbdia` timings in seconds.

However, the FDR3 algorithm performs vastly better on R which has a lot of τ^*-connectivity, but relatively few τ transitions (so the FDR2 algorithm constructs an LTS with $\Theta(N^4)$ transitions, when the input only has $\Theta(N^2)$). We were unable to obtain FDR2 timings for $R(300)$ and $R(1000)$ due to insufficient memory ($R(100)$ used 2.2 GiB), while FDR3 coped with these examples very well.

Summary. For computing `dbisim`, the explicit reduction approach is prohibitively memory-intensive for large graphs with a high degree of τ-connectivity. The dynamic programming approach, on the other hand, is somewhat slower for problems with few τs. Little difference was observed between `wbisim` and `dbisim`, both in terms of runtime and output. The latter is not surprising given that delay bisimulation lies between weak and branching bisimulation, which are known to frequently coincide in a non divergence-respecting context.

5.3 Amount of Compression

We will examine the performance and effectiveness of `dbisim` and `sbdia` on the *bully* algorithm (the FDR implementation is outlined in Section 14.4 of [8]) with 5 processors and an implementation of Lamport's bakery algorithm (Section 18.5 of [8]) with either 3 or 4 threads and integers ranging from 0 to 7. These are typical examples composed of a variable number of parallel processes, with many τs and symmetry that can be reduced by either `dbisim` or `sbdia`. We will compress these processes *inductively*[2] (as described in Section 8.8 of [8]); that is, add them to the composition one at a time, compressing at every step. This is a common technique that allows a large portion of the system to be compressed while keeping each compression's inputs manageable. Table 5 shows that `sbdia` runs much faster than `dbisim` and Table 4 shows that it is more effective at reducing state counts, but can add transitions, whereas `dbisim` cannot by design.

[2] We used inductive compression to increase the time spent on the compressions. This is not necessarily the most efficient approach to checking these systems in FDR.

Problem	States			Transitions		
	Uncompressed	`dbisim`	`sbdia`	**Uncompressed**	`dbisim`	`sbdia`
Bully	492,548	140,776	105,701	3,690,716	1,280,729	3,872,483
Bakery (3)	9,164,958	29,752	17,787	27,445,171	85,217	64,283
Bakery (4)	–	1,439,283	716,097	–	5,327,436	3,408,420

Table 4: State and transition counts with no compression, `dbisim`, and `sbdia`.

Problem	Compilation Time (s)			Exploration Time (s)		
	Uncompressed	`dbisim`	`sbdia`	**Uncompressed**	`dbisim`	`sbdia`
Bully	0.06	185.46	25.17	1.76	0.36	0.88
Bakery (3)	0.37	0.57	0.36	137.52	0.93	1.07
Bakery (4)	–	105.88	9.54	–	3.63	1.64

Table 5: Timings with no compression, `dbisim`, and `sbdia`.

6 Conclusions

We have presented a number of GLTS compression algorithms, including novel algorithms as well as ones that had been implemented previously, but not characterised until now. Our change-tracking iterative refinement algorithm for `sbisim` showed comparable performance to the Paige-Tarjan algorithm (the current state of the art) and offered a significant improvement over the naïve iterative refinement used by previous versions of FDR. We have shown that explicitly constructing a τ-closed transition relation for weak bisimulations, the current state of the art, is prohibitively memory-intensive and provided an efficient alternative based on dynamic programming. Comparing `dbisim` and `wbisim`, we have noticed that they produce identical output on all the real-world examples we have tested, and exhibit a similar runtime.

Future Work. We plan to explore implementing DRD-bisimulation by reduction to strong bisimulation for FDR3 for those cases where this approach is more efficient. We can provide the alternatives to the user, but we would like to find and implement a heuristic that would allow FDR3 to automatically select of the two algorithms the one that is likely to be faster for the given problem. We would also like to find heuristics for deciding which compression to use, in particular for inductively compressing large parallel compositions.

It would be interesting to come up with versions of the dynamic DRD-bisimulation or DRW-bisimulation algorithms that use Change-Tracking Iterative Refinement or The Paige-Tarjan Algorithm.[3] This is challenging due to the difficulty of inverting the delayed and observed transition relations dynamically.

Despite the multi-threaded core of FDR3, compressions are still single threaded, though independent compressions can be run in parallel. Iterative refinement consists of massively parallel *afters* computations and parallel sorts of a number

[3] Since submitting the first version of the paper, the authors have developed such an algorithm, and intend to publish results once it is characterised.

of *afters* lists of arbitrary size. Both phases could be sped up by a multi-threaded implementation. The naïve parallelisation has the nice property that the transition set can be partitioned across threads and only the node to equivalence class map needs to be shared. This could allow for an efficient GPU implementation.

References

1. D. Park, *Concurrency and automata on infinite sequences*. Springer, 1981.
2. R. Milner, "A modal characterisation of observable machine-behaviour," in *CAAP'81*, pp. 25–34, Springer.
3. R. J. van Glabbeek and W. P. Weijland, "Branching time and abstraction in bisimulation semantics," *J. ACM*, vol. 43, pp. 555–600, May 1996.
4. I. Phillips and I. Ulidowski, "Ordered SOS rules and weak bisimulation," *Theory and Formal Methods*, 1996.
5. D. Sangiorgi, "A theory of bisimulation for the π-calculus," *Acta informatica*, vol. 33, no. 1, pp. 69–97, 1996.
6. C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
7. A. W. Roscoe, *The Theory and Practice of Concurrency*. 1998.
8. A. W. Roscoe, *Understanding Concurrent Systems*. Springer, 2010.
9. A. W. Roscoe, "Model-Checking CSP," *A Classical Mind: Essays in Honour of CAR Hoare*, 1994.
10. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3—A Modern Refinement Checker for CSP," 2014.
11. A. W. Roscoe, P. Gardiner, M. Goldsmith, J. Hulance, D.M.Jackson, and J. Scattergood, "Hierarchical compression for model-checking CSP, or How to check 10^{20} dining philosophers for deadlock," in *Proceedings of TACAS 1995*, BRICS.
12. R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987.
13. J.-C. Fernandez, "An implementation of an efficient algorithm for bisimulation equivalence," *Science of Computer Programming*, vol. 13, no. 2, pp. 219–236, 1990.
14. R. Van Glabbeek and W. Weijland, "Branching time and abstraction in bisimulation semantics: extended abstract," *Rep./Centrum voor wiskunde en informatica. Computer science; CS-R8911*, 1989.
15. J. Groote and F. Vaandrager, "An efficient algorithm for branching bisimulation and stuttering equivalence," in *Automata, Languages and Programming* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*, pp. 626–638, Springer Berlin Heidelberg, 1990.
16. P. Armstrong, M. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell, "Recent developments in FDR," *CAV 2012*.
17. R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, pp. 345–, June 1962.
18. R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
19. R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica*, vol. 6, no. 2, pp. 171–185, 1976.