Under consideration for publication in Formal Aspects of Computing

Computing Maximal Weak and Other Bisimulations

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

Department of Computer Science, University of Oxford, 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. We concentrate on algorithms which work efficiently when implemented rather than on ones which have low asymptotic growth.

Keywords: CSP; FDR; Bisimulation; Strong bisimulation; Delay bisimulation; Weak bisimulation; Labelled transition systems; Model-checking

1. Introduction

Many different variations on bisimulation have been described in the literature on process algebra, for example [Par81, Mil81, vGW96, PU96, San96]. They are typically used to characterise equivalences between nodes of a labelled transition system (LTS), but they can also be used to calculate state-reduced LTSs that represent equivalent processes. They have the latter function in the CSP-based [Hoa85, Ros98, Ros10] refinement checker FDR [Ros94], of which the third major version FDR3 has recently been released [GRABR15]. The present paper sets out the approaches to bisimulation reduction taken in FDR and especially FDR3.

FDR typically views a large process as the parallel composition of a number of component processes, which are often sequential. The resulting LTS is closely related to the Cartesian product of the components' LTSs. One of the approaches it takes to combat the state explosion problem is to supply a number of compression functions that attempt to reduce the state spaces of these components[1]. The set of compressions described in [RGG$^+$95], which included *strong* bisimulation, has been extended by several other versions of bisimulation in the most recent versions of FDR.

Correspondence and offprint requests to: Alexandre Boulgakov, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK. e-mail: alexandre.boulgakov@cs.ox.ac.uk

[1] Necessarily, any technique for reducing the number of states will lose some information about the inputs, making presentation of counterexamples in terms of the input non-trivial. As discussed in [RGG$^+$95], FDR's debugger gets around this with the help of additional refinement checks.

The main purpose of this paper is to introduce novel algorithms for computing maximal *divergence-respecting delay bisimulations* (DRDB) and *divergence-respecting weak bisimulations* (DRWB) based on dynamic programming and incorporating the idea of *change tracking* introduced in the context of strong bisimulation in [BO05]. These algorithms are the first efficient algorithms for directly computing the maximal weak and delay bisimulation relations, without constructing an expensive intermediate form. For obvious reasons the choice of algorithms for FDR is primarily influenced by practical efficiency, and we include comprehensive benchmark results. These show that our new algorithms are much more efficient than existing algorithms used by other tools on many examples. We also compare the performance of a number of strong bisimulation algorithms.

Model checkers frequently use branching bisimulation [vGW96] for compression due to the existence of an efficient $O(nt)$ algorithm [GV90] and the absence, prior to our own work reported here, of a sufficiently efficient algorithm to compute the even coarser weak bisimulation directly. Even though DRDB and DRWB compress better than branching bisimulation they typically give less compression than FDR's pre-existing compressions. However the latter are not sound for some functionality of FDR3[2] whereas these bisimulations are, when strengthened so as not to identify any divergent process with a non-divergent one. In Section 4.5 we compare these two classes of compressions. The new algorithms are highly effective in practice. *Change tracking* can significantly reduce wasted effort. Their use of dynamic programming to compute *afters* on the fly for the divergence-respecting delay and weak bisimulations typically gives a vast reduction on memory usage and time for transition systems with many τs: this can be as much as several orders of magnitude.

This paper updates and extends the conference paper [BGRR14]. Since we believe that the most interesting contributions of our work relate to delay and weak bisimulation, the emphasis of the present paper has shifted in that direction. New algorithms are presented for DRDB and DRWB combining the change tracking and dynamic programming techniques. For all of the presented algorithms, the descriptions have been updated to include pseudo-code. The performance is investigated more thoroughly, with a comparison to CADP's BCG_MIN [GLMS13] and with the addition of benchmarks from the VLTS Benchmark Suite provided alongside the CADP Toolbox. For completeness, we adapt the Paige-Tarjan algorithm to compute DRDB and DRWB.

Section 2 first defines terms used throughout the rest of the paper. The rest of the section summarises *iterative refinement*, which is discussed in its naïve form in [BO02] and in an optimised form using *change tracking* in [BO05], and the Paige-Tarjan algorithm [PT87,Fer90] as implemented by FDR. Since our DRDB and DRWB algorithms build on these foundations, the reader is recommended to study this section before reading the subsequent sections. Divergence-respecting delay and weak bisimulations are closely related, and we present them together in Section 3. Finally, we compare the time and compression characteristics of each of our algorithms and a number of alternatives on various benchmarks in Section 4.

2. Background

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

Definition 2.1. A *generalised labelled transition system* (GLTS) is a tuple $(N, \Sigma, \longrightarrow, \Lambda, \lambda)$ where N is a set of nodes, Σ is a set of events, τ is a designated *invisible* event not in Σ, $\Sigma^\tau = \Sigma \cup \{\tau\}$, $\longrightarrow \subseteq N \times \Sigma^\tau \times N$ 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 node 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 series of internal τ actions corresponding to *livelock*.

The algorithms presented in this paper input and output GLTSs, and the definitions of the various bisimulations are modified to only identify states with identical node labels. However, regular LTSs are

[2] Use inside the `prioritise` operator and with semantic models richer than failures. This includes virtually all Timed CSP examples and most other real-time CSP models.

```
1: function IterativeRefinement(InitialApproximation, ComputeAfters, Refine)(G)
2:     $\rho \leftarrow InitialApproximation(G)$
3:     repeat
4:         $\rho' \leftarrow \rho$
5:         $cafters \leftarrow ComputeAfters(G, \rho')$
6:         $\rho \leftarrow Refine(G, \rho', cafters)$
7:     until $\rho = \rho'$
8:     ConstructMachine(G, $\rho$)
9: end function
```

Fig. 1. The iterative refinement skeleton used by most of our bisimulation algorithms. We present it as a curried higher-order function so that implementers can pass the functions specifying a certain algorithm and users would pass the GLTS it is to be run against.

supported as a special case. As input, an LTS $(N, \Sigma, \longrightarrow)$ is equivalent to the GLTS $(N, \Sigma, \longrightarrow, \{\emptyset\}, \lambda)$, where $\lambda(n) = \emptyset$ for all $n \in N$. The output GLTSs have the same node labellings as the input, and possibly divergence markings. If the input is an LTS, rather than a GLTS, and an output LTS is to be created then τ self-loops could be used in place of divergence markings.

2.1. Strong Bisimulation

Definition 2.2. Given a GLTS $G = (N, \Sigma, \longrightarrow, \Lambda, \lambda)$, a relation $R \subseteq N \times N$ is a *strong bisimulation* of G if and only if it satisfies all of the following:

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

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 the release of FDR2. Other bisimulation relations are defined similarly, differing in their use of a derived transition relation in place of $\longrightarrow$ and potentially additional constraints on related nodes beyond equivalence of their node labels.

2.2. 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. (This map allows us to quickly determine the index of any node's equivalence class, and the set of nodes denoted by any such index.) 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 bisimulation algorithms and so will not be discussed here in much detail. Furthermore, it is not specific to strong bisimulation and can be used to factor a GLTS by an arbitrary equivalence relation. Computing the desired equivalence relation is the more interesting problem and the topic of this paper.

We will use the term *coloured afters* to refer to the *afters* of a node under the last iteration's equivalence relation ρ, that is, $\{(e, \rho(n)) \mid m \xrightarrow{e} n\}$. This is similar to the *signatures* introduced by Blom and Orzan in [BO02]. Wimmer et al. present a list of *signatures* for a number of variants of bisimulation in [WHH+06]; we supplement them with node labels to support GLTSs and divergence-sensitive models.

A very high level overview of iterative refinement is illustrated in Figure 1. A coarse approximation of the equivalence relation is first computed by *InitialApproximation*, usually 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 [Mil81]

```
1: function NullApproximation($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
2:     return $\{(n, 0) \mid n \in N\}$                    ▷ Put all the nodes in one class.
3: end function
```

(a) NullApproximation puts all nodes into the same equivalence class.

```
1: function FirstStepApproximation($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
2:     $\Phi \leftarrow \langle (n, \mathit{initials}(n), \lambda(n)) \mid n \in N \rangle$
3:     Sort $\Phi$ by $(n, a, l) \mapsto (a, l)$  ▷ We use this notation to mean sorting a sequence of $(n, a, l)$s by $(a, l)$.
           ▷ After the sort, nodes with equivalent initials and node labels are adjacent.
4:     $a', l' \leftarrow \mathit{INVALID}, \mathit{INVALID}$
5:     $c \leftarrow 0$
6:     for each $(n, a, l) \leftarrow \Phi$ do
7:         if $a \neq a' \vee l \neq l'$ then       ▷ Different initials or node label; start a new partition.
8:             $a', l' \leftarrow a, l$
9:             $c \leftarrow c + 1$
10:        end if
11:        $\rho \leftarrow \rho \cup \{(n, c)\}$
12:    end for
13:    return $\rho$                     ▷ $\rho$ partitions $N$ into the following equivalence classes:
14:        ▷ $\{\{n' \mid (n', a, l) \in \Phi \wedge a' = a \wedge l' = l\} \mid a \in \Sigma, l \in \Lambda \,.\, \exists n \,.\, (n, a, l) \in \Phi\}$
15: end function
```

(b) The more involved FirstStepApproximation instead classifies nodes based on their first-step behaviour: their node labels and initial events.

Fig. 2. Two initial approximation functions, each returning a partition function $\rho : N \to \mathbb{N}^+$.

as a series of experiments of increasing depth and is the naïve method mentioned by Kanellakis and Smolka in [KS83].

2.2.1. Algorithm

Initial approximation. The initial approximation can most simply be computed by identifying all nodes, as does NullApproximation in Figure 2(a). Instead, FDR uses the finer FirstStepApproximation in Figure 2(b) to compute an initial approximation $\rho_0 = \{\{n \in N \mid \lambda(n) = \lambda(m) \wedge \mathit{initials}(n) = \mathit{initials}(m)\} \mid m \in N\}$ based on the nodes' labels and *initials*. This is equivalent to identifying all nodes and then performing one refinement using the nodes' labels and their *coloured afters*. Unlike the *afters* of a node, whose colour and therefore equivalence depends on the current equivalence relation, these node labels are fixed and we can save time by only comparing them once. The time savings can be significant depending on the size of the node labels; in particular some GLTSs in FDR have nodes labelled with *minimal acceptances* drawn from the set $\Lambda = \mathcal{P}(\mathcal{P}(\Sigma))$. In addition to the time savings, a side effect of this finer initial approximation is that an algorithm that is not aware of node labellings, such as that given in [Fer90], can be used for the iteration phase.

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. We first compute the *afters* of each node *coloured* per the latest equivalence relation, as shown in ComputeAllAfters in Figure 3. Then, we use these *coloured afters* to produce a more refined equivalence relation as illustrated in RefineAll. This sorts the *coloured afters* for the nodes in each class (line 9), and a single in-order traversal through the sorted lists (lines 10-18) then 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 as outlined in Figure 4. To do this, we first create a node for each equivalence class (line 3). Any node labels can be copied from an arbitrary represen-

```
1: **function** ComputeAllAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
2:     $\mathit{cafters} \leftarrow \langle (n, \{(a, \emptyset) \mid \exists m. n \xrightarrow{a} m\}) \mid n \in N \rangle$
3:     **for each** $n \xrightarrow{a} m$ **do**
4:         $\mathit{cafters}(n)(a) \leftarrow \mathit{cafters}(n)(a) \cup \{\rho(m)\}$
5:     **end for**
6:     **return** $\mathit{cafters}$
7: **end function**

8: **function** RefineAll($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho, \mathit{cafters}$)
9:     Sort $\mathit{cafters}$ by $(n, A) \mapsto A$.
10:    $A' \leftarrow \mathit{INVALID}$
11:    $c \leftarrow \max(\rho)$        ▷ Instead of computing $\max(\rho)$, we could persist $c$ across iterations.
12:    **for each** $(n, A) \leftarrow \mathit{cafters}$ **do**
13:        **if** $A \neq A'$ **then**        ▷ Different *coloured afters*; start a new partition.
14:            $A' \leftarrow A$
15:            $c \leftarrow c + 1$
16:        **end if**
17:        $\rho(n) \leftarrow c$
18:    **end for**
19:    **return** $\rho$
20: **end function**

21: **function** SBisimNaïve := IterativeRefinement(
                FirstStepApproximation, ComputeAllAfters, RefineAll)
```

Fig. 3. Naïve Iterative Refinement for strong bisimulation.

```
1: **function** ConstructMachine($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
2:     $\mathit{cafters} \leftarrow$ ComputeAllAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
3:     $N' \leftarrow \mathrm{range}(\rho)$        ▷ Create a node for each equivalence class.
4:     $\lambda' \leftarrow \emptyset$
5:     $T' \leftarrow \emptyset$
6:     **for each** $n \in N'$ **do**
7:         $n' = \mathit{pick}(\rho^{-1}(n))$        ▷ Compute a representative node for the class.
8:         $\lambda' \leftarrow \lambda' \cup (n, \lambda(n'))$
9:         **for each** $(a, M) \in \mathit{cafters}(n')$ **do**
10:            **for each** $m \in M$ **do**
11:                $T' \leftarrow T' \cup (n, a, m)$        ▷ Compute transitions.
12:            **end for**
13:        **end for**
14:    **end for**
15: **end function**
```

Fig. 4. The ConstructMachine function for strong bisimulation. The *pick* function chooses an arbitrary member of its nonempty set argument. Note that instead of recomputing *cafters*, we could use the *cafters* already computed by RefineAll.

tative in each class (the n' in line 7), as they are guaranteed to be equivalent by the initial approximation (line 8). Next, we output a transition corresponding to each input transition. This can generate duplicates, and we must take care to only output one copy of each. Instead of using the input transitions directly, we also have the option of using the already computed *coloured afters* to create the transitions (lines 9-13), using an arbitrary representative from each class since the *coloured afters* for each of the nodes in an equivalence class are guaranteed to be equivalent (since the refinement phase has terminated).

2.2.2. Representation of coloured afters

Blom and Orzan [BO02] state a worst-case complexity in $O(nt)$, where the input has n nodes and t transition. They are able to achieve this by assuming a bounded fanout, which allows the representation of *coloured afters* to be ignored. Since our primary objective is developing algorithms that perform well on practical examples as typified by our extensive benchmarks on real machines, rather than seeking the best asymptotic behaviour on notional ones, we must choose a data structure that works well in practice.

Asymptotically a tree representation of *coloured afters* sets seems most efficient, with $O(\log c)$ time for each insertion, where c is the number of equivalence classes in the output GLTS. However, in practice, we have observed in nearly all cases a significant speedup from using sorted arrays instead, with $O(c)$ time for each insertion. This is likely due to the *coloured afters* sets often being much smaller than c, and due to the x86 architecture being optimised for operations on contiguous blocks of memory. For this reason, most of the performance results in Section 4 refer only to implementations using sorted arrays.

2.3. Change-Tracking Iterative Refinement

We will now present the strong bisimulation algorithm FDR3 uses, an improvement on Naïve Iterative Refinement. With some bookkeeping, we can determine which states' *coloured afters* could not possibly have changed after the previous iteration. The algorithm shown in Figure 5 uses this information to avoid recomputing and sorting the *coloured afters* for these states, in a similar fashion to the optimisation used by Blom and Orzan in [BO05]. 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. A node then might change class at the $n+1^{th}$ iteration only when it is one step back from a node that has changed class on the n^{th}.

We will maintain the following items 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. Before the first iteration, *changed* should be initialised with all nodes marked since the initial approximation classified all the nodes.

Following this initialisation, on each iteration we will perform the following sequence of actions. First, we need to compute *affected* by iterating through *changed* and adding each of their predecessors (lines 6-9). We then clear (lines 12-14) and recompute (lines 15-17) the *coloured afters* for each of the nodes in *affected*. All nodes that are not marked for update get to keep their *coloured afters* from the previous iteration.

Next, we compute the equivalence classes that contained the affected nodes in the previous iteration (line 24); 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, the latter being in *affected* (line 26). Next, we sort the nodes that are in *affected* and in this class in order to partition this class (line 27). Once we have sorted the *coloured afters* of all of the affected nodes in a given class, we choose the *largest* sequence of nodes with the same *coloured afters* to keep the original class index (line 28). We assign new indices to the rest (lines 29-37), rather than picking the *first* such sequence. We must also record the nodes that had new indices assigned in *changed* (line 36).

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.

2.4. Paige-Tarjan Algorithm

We have also implemented the algorithm outlined in [Fer90]. This is an adaptation of Paige and Tarjan's solution (described in Section 3 of [PT87]) 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 without) and with respect to each split class

```
global *changed*                 ▷ Output from RefineChanged to ComputeChangedAfters.
global *affected*                ▷ Output from ComputeChangedAfters to RefineChanged.
global *cafters*     ▷ Read and written by ComputeChangedAfters; persisted for optimisation.
global $T^{-1}$                  ▷ Read by ComputeChangedAfters; persisted for optimisation.

function ComputeChangedAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
    $affected \leftarrow \emptyset$
    for each $n \in changed$ do
        $affected \leftarrow affected \cup T^{-1}(n)$
    end for

    for each $n \in affected$ do
        for each $(a, M) \in cafters(n)$ do
            $cafters(n)(a) \leftarrow \emptyset$
        end for
        for each $(a, m) \in \{n \xrightarrow{a} m\}$ do
            $cafters(n)(a) \leftarrow cafters(n)(a) \cup \{\rho(m)\}$
        end for
    end for
    return *cafters*
end function

function RefineChanged($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho, cafters$)
    $changed \leftarrow \emptyset$
    $c \leftarrow \max(\rho)$               ▷ As in *RefineAll*, we could persist $c$ across iterations.
    for each $class \in \{\rho(n) \mid n \in affected\}$ do
        $nodes \leftarrow \rho^{-1}(class)$
        Reorder *nodes* by moving those in *affected* to the front.
        Sort remaining nodes (those not in *affected*) by $n \mapsto cafters(n)$.
        Find largest run with equivalent *cafters* and remove it from *nodes*.
        $A' \leftarrow INVALID$
        for each $n \in nodes$ do
            if $A \neq A'$ then               ▷ Different *coloured afters*; start a new partition.
                $A' \leftarrow A$
                $c \leftarrow c + 1$
            end if
            $\rho(n) \leftarrow c$
            $changed \leftarrow changed \cup \{n\}$
        end for
    end for
    return $\rho$
end function

function SBisimCT($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
    $changed \leftarrow N$
    $affected \leftarrow \emptyset$
    $cafters \leftarrow \langle(n, \{(a, \emptyset) \mid \exists m . n \xrightarrow{a} m\}) \mid n \in N\rangle$
    $T^{-1} \leftarrow \{(n, \{m \mid \exists a . n \xrightarrow{a} m\}) \mid n \in N\}$
    return IterativeRefinement(
                    FirstStepApproximation, ComputeChangedAfters, RefineChanged)
            $((N, \Sigma, \longrightarrow, \Lambda, \lambda))$
end function
```

Fig. 5. Change-Tracking Iterative Refinement for strong bisimulation. Note that the functions rely on a number of global variables.

(separating nodes with edges into one subclass, the other, or both) using the inverse labelled transition relation.

We have produced two implementations of this algorithm for performance comparison. The first implementation stays true to the original formulation, which makes heavy use of linked lists. An alternative implementation uses arrays like our other algorithms.

2.4.1. Complexity

The worst-case time complexity for a GLTS with n nodes and t transitions is in $O(t\log(n))$. However, the cached in-counts (the *info maps* of [Fer90]) 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. In fact, in our performance tests (Section 4.2 and Table 2), change-tracking iterative refinement outperforms the Paige-Tarjan algorithm despite the latter's superior asymptotic time complexity. This is consistent with Blom and Orzan's observations in [BO05].

3. Divergence-Respecting Delay and Weak Bisimulations

While FDR has long supported strong bisimulation, it has only recently added support for variants of weak bisimulation. This was because the weak bisimulation of [Mil81] is not compositional for most CSP models and because FDR already had compressions that successfully eliminated τ actions, the most notable of which are `normal` and `diamond`. `normal` was originally provided as an important component of refinement checking in FDR, but has since also proven to be a useful compression function in its own right. However, it does sometimes create an exponential increase in the size of a LTS. `diamond` [RGG+95] is a compression that was designed to remove redundant *diamonds* of transitions in LTSs, but is not semantics-preserving when combined with prioritisation, Timed CSP, and stronger semantic models such as refusal testing. Due to the problems with both `diamond` and `normal`, FDR 2.94 [AGL+12] implemented a new `dbisim` compression (originally called `wbisim`), which returns the maximal divergence-respecting delay bisimulation (DRDB) of its input.

Definition 3.1. 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 $\forall\, i < n \cdot p_i \xrightarrow{\tau} p_{i+1}$. Let us further define a ternary relation $\hookrightarrow$ with $p \overset{a}{\hookrightarrow} q$ for $a \in \Sigma$ if and only if $\exists p' \cdot p \Longrightarrow p' \wedge p' \xrightarrow{a} q$, and $p \overset{\tau}{\hookrightarrow} q$ if and only if $p \Longrightarrow q$. This is the *delayed transition relation*, since the visible events are delayed by 0 or more τs.

Definition 3.2. 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:

$$\begin{array}{lll}
\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1 \; R \; n_2 \wedge n_1 \overset{x}{\hookrightarrow} m_1 & \Rightarrow & \exists m_2 \in N \cdot n_2 \overset{x}{\hookrightarrow} m_2 \wedge m_1 \; R \; m_2 \\
\forall\, n_1, n_2, m_2 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1 \; R \; n_2 \wedge n_2 \overset{x}{\hookrightarrow} m_2 & \Rightarrow & \exists m_1 \in N \cdot n_1 \overset{x}{\hookrightarrow} m_1 \wedge m_1 \; R \; m_2 \\
\forall\, n_1, n_2 \in N \cdot n_1 \; R \; n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2) & & \\
\forall\, n_1, n_2 \in N \cdot n_1 \; R \; n_2 \quad \Rightarrow \quad n_1{\Uparrow} \Leftrightarrow n_2{\Uparrow} & &
\end{array}$$

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[3]. 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 $\lambda(n_1) = \lambda(n_2)$.

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

[3] Of all the semantic models of CSP, only the simple traces model $\mathcal{T}$ identifies an LTS node with no action and one whose only action is τ to itself. These two nodes are both weakly and delay bisimilar under the usual definitions.

```
1: function DRDBisimulationReduction($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
2:     $\lambda' = \emptyset$
3:     for each $n \in N$ do
4:         $\lambda'(n) \leftarrow (\lambda(n), \mathit{CheckDivergence}(n))$
5:     end for
6:     Compute $\hookrightarrow$ from $\longrightarrow$.
7:     return SBisimCT($(N, \Sigma, \hookrightarrow, \Lambda \times \{\mathit{Divergent}, \mathit{NotDivergent}\}, \lambda')$)    ▷ Or SBisimNaïve.
8: end function
```

Fig. 6. An implementation of DRD-bisimulation by reduction to strong bisimulation. The DRW-bisimulation is similar, but uses $\Longrightarrow$ instead of $\hookrightarrow$.

Definition 3.3. 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 $p \overset{a}{\Longrightarrow} q$ for $a \in \Sigma$ if and only if $\exists p', q' \cdot p \Longrightarrow p' \wedge p' \xrightarrow{a} q' \wedge q' \Longrightarrow q$, and $p \overset{\tau}{\Longrightarrow} q$ if and only if $p \Longrightarrow q$. This is the *observed transition relation.*

Definition 3.4. 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:

$$\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma^{\tau} \cdot n_1 \; R \; n_2 \wedge n_1 \overset{x}{\Longrightarrow} m_1 \quad \Rightarrow \quad \exists m_2 \in N \cdot n_2 \overset{x}{\Longrightarrow} m_2 \wedge m_1 \; R \; m_2$$
$$\forall\, n_1, n_2, m_2 \in N \cdot \forall\, x \in \Sigma^{\tau} \cdot n_1 \; R \; n_2 \wedge n_2 \overset{x}{\Longrightarrow} m_2 \quad \Rightarrow \quad \exists m_1 \in N \cdot n_1 \overset{x}{\Longrightarrow} m_1 \wedge m_1 \; R \; m_2$$
$$\forall\, n_1, n_2 \in N \cdot n_1 \; R \; n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$
$$\forall\, n_1, n_2 \in N \cdot n_1 \; R \; n_2 \quad \Rightarrow \quad n_1{\Uparrow} \Leftrightarrow n_2{\Uparrow}$$

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

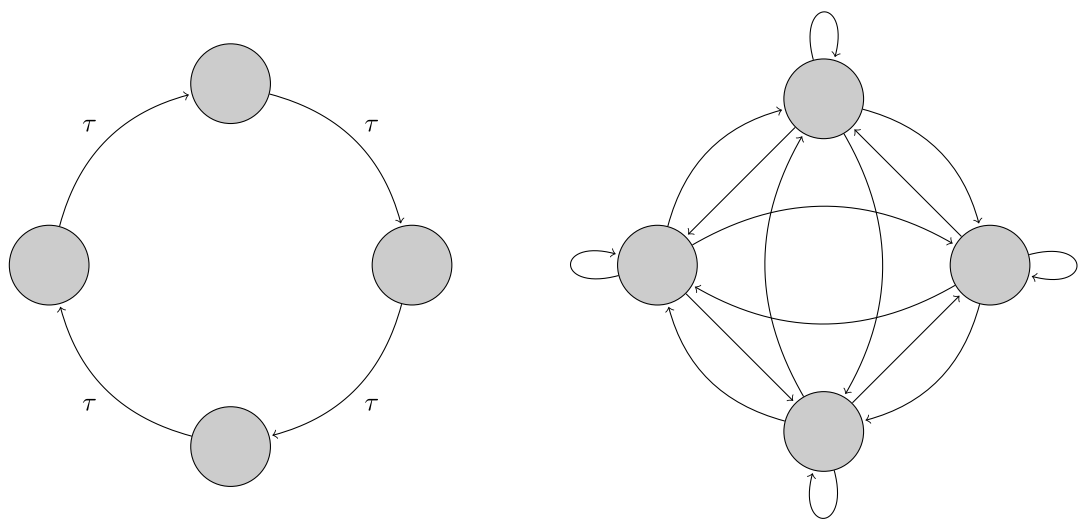
The FDR3 compression function `wbisim` computes the maximal DRWB on its input GLTS and returns a GLTS with a single node DRW-bisimilar to each equivalence class in the input. It is an important compression because, like `sbisim` and `dbisim` it preserves semantics in all CSP models, while potentially offering a higher amount of compression than `dbisim`. Weak bisimulation is also important because (at least in its non divergence-respecting form) it and strong bisimulation are the best known and most studied bisimulations in the literature. This compression is new to FDR3 and is the strongest implemented compression for CSP models richer than the failures model and the other cases where `diamond` is not semantically valid.

3.1. Reduction to Strong Bisimulation for DRDB and DRWB

The definition of DRDB suggests a reduction to strong bisimulation. 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 (respectively, DRWB) of S by construction, and this can be computed by any of the algorithms described in Section 2.1, as shown in Figure 6. FDR2 employs such a reduction to compute a maximal DRDB, and uses naïve iterative refinement for the strong bisimulation step.

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 7 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 [Flo62] requires $O(n^3)$ operations. The strong bisimulation step after this transformation will take a correspondingly large amount of time ($O(An^3 \log n)$, for naïve iterative refinement). Regardless of the strong bisimulation algorithm used, the memory usage is likely to be prohibitive.



(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. 7. The constructed LTS can be quadratically larger than the input.

Divergence-Respecting Weak Bisimulation. It is possible to compute a maximal DRWB with a similar reduction, creating $\widehat{S}$ with the observed transitions of S instead of its delayed transitions. Given our results from testing with DRDB, we do not believe that such an implementation would be useful, and we have therefore not created one.

3.2. Dynamic Programming Approach for DRDB

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. To the authors' knowledge, ours is the first algorithm that avoids calculating $\widehat{S}$ explicitly.

3.2.1. 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[4] (line 18). 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 [Tar72] 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 [Tar76], so that there are no upstream τ-transitions (line 19).

The topological sort allows us to obtain the transitions of the $\widehat{S}$ described in Section 3.1 using a dynamic programming approach. The most downstream node in this topological sort has no outgoing τ transitions, so its new *initials* and *coloured afters* are precisely those in S (lines 6-8) with the addition of itself after τ (line 5). We then proceed upstream and for each node compute the union of its own *coloured afters* (with the inclusion of a self-transition under τ) and the *coloured afters* of each of the nodes it can reach under a

[4] Even if the nodes on a τ loop have different different labels, their mutual reachability means that it is impossible to distinguish them in any of the CSP models, so the compression is sound as long as the resulting node's label captures all the possible behaviours (e.g., if the node labels are *minimal acceptances*, `tau_loop_factor` would output their minimised union).

```
1: global $toponodes$    ▷ Read by ComputeAllDelayedAfters; persisted for optimisation.

2: function ComputeAllDelayedAfters($((N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho)$)
3:     $cafters \leftarrow \langle (n, \{(a, \emptyset) \mid \exists m. n \overset{a}{\hookrightarrow} m\}) \mid n \in N \rangle$    ▷ Can be cached across iterations.
4:     for each $n \leftarrow toponodes$ do
5:         $cafters(n)(\tau) \leftarrow \{\rho(n)\}$    ▷ Implicit self-loop.
6:         for each $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m\}$ do    ▷ Visible afters.
7:             $cafters(n)(a) \leftarrow cafters(n)(a) \cup \{\rho(m)\}$
8:         end for
9:         for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do    ▷ Delayed afters.
10:            for each $(a, M) \in cafters(m)$ do
11:                $cafters(n)(a) \leftarrow cafters(n)(a) \cup M$
12:            end for
13:        end for
14:    end for
15:    return $cafters$
16: end function

17: function DBisimDP($(G)$)
18:    $(N, \Sigma, \longrightarrow, \Lambda, \lambda) \leftarrow$ TauLoopFactor($(G)$)
19:    $toponodes \leftarrow N$ sorted topologically according to $\xrightarrow{\tau}$
20:    return IterativeRefinement(
                   DBisimApproximation, ComputeAllDelayedAfters, RefineAll)
               $((N, \Sigma, \longrightarrow, \Lambda, \lambda))$
21: end function
```

Fig. 8. The dynamic programming algorithm for DRDB, with the straightforward DBisimApproximation omitted due to space constraints.

single τ transition (lines 9-13). 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 *coloured afters* of all τ-reachable nodes from the given node.

We can apply a modified Naïve Iterative Refinement (Section 2.2) to compute the maximal strong bisimulation of $\widehat{S}$, which is itself never constructed (line 20). We compute the *initials* and node labels for the initial approximation using dynamic programming on the topologically sorted nodes. For each refinement, we compute the *coloured afters* using the dynamic programming approach described above. For the construction step, we compute the equivalence classes of the *coloured afters* as above, but without inserting the τ self-transition.

3.2.2. 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 *coloured afters* we compute for each node, which are sets of equivalence class identifiers, 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 *coloured 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 7, the initial classification will identify all nodes, and the first *coloured 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 *coloured 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 *coloured 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 $O(An^3c)$.

```
global toponodes            ▷ Read by ComputeAllObservedAfters; persisted for optimisation.

function ComputeAllObservedAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
    $cafters \leftarrow \langle (n, \{(a, \emptyset) \mid \exists m. n \overset{a}{\hookrightarrow} m\}) \mid n \in N \rangle$    ▷ Can be cached across iterations.
    for each $n \leftarrow toponodes$ do    ▷ Compute nodes reachable in 0 or more $\tau$s:
        $cafters(n)(\tau) \leftarrow \{\rho(n)\}$
        for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do
            $cafters(n)(\tau) \leftarrow cafters(n)(\tau) \cup cafters(m)(\tau)$
        end for
    end for
                ▷ Compute nodes reachable in 0 or more $\tau$s, 1 visible event, 0 or more $\tau$s:
    for each $n \leftarrow toponodes$ do
        for each $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m \wedge a \neq \tau\}$ do
            $cafters(n)(a) \leftarrow cafters(n)(a) \cup cafters(m)(\tau)$
        end for
        for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do
            for each $(a, M) \in cafters(m), a \neq \tau$ do
                $cafters(n)(a) \leftarrow cafters(n)(a) \cup M$
            end for
        end for
    end for
    return cafters
end function
```

Fig. 9. The dynamic programming algorithm for DRWB, with the straightforward WBisimApproximation and entry-point WBisimDP omitted due to space constraints.

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 in systems with divergence, which eliminates many worst cases. 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 computing the unions of duplicate *coloured afters* sets (though we do not currently employ such optimisations). Section 4.3 demonstrates that the dynamic programming approach is faster on many examples with a large number of τs than the explicit reduction approach.

3.3. Dynamic Programming Approach for DRWB

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 (recall Definition 3) using the two-pass dynamic programming approach in Figure 9. One pass, as in delay bisimulation, is not sufficient since we need to determine the *coloured* τ^* *afters* of the visible *afters* of each node, and these visible *coloured afters* might not have been previously explored. In the first pass, we compute the *coloured* τ^* *afters* of each node (lines 4-9). 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 *coloured* τ *afters* (with the inclusion of its own equivalence class) and the previously computed *coloured* 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 *coloured* τ^* *afters* of its visible *afters* (lines 12-14) and the visible observed transitions of its τ *afters* (lines 15-19). 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 described in Section 3.2, removing the τ self-transition from each node in the construction step.

3.3.1. Complexity

In the typical case this algorithm will require more space to store the *coloured 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.

In practice we have found that `wbisim` is nearly as fast as `dbisim`, and produces identical results on nearly all inputs.

3.4. Change-Tracking with Dynamic Programming for DRDB and DRWB

FDR3 improves this dynamic programming approach to computing *coloured afters* with an adaptation of the change tracking introduced in [BO05] and discussed here in Section 2.3.

3.4.1. Two-Pass Change-Tracking DRDB

The idea behind the two-pass approach is to separate the change-tracking and the dynamic programming aspects into two separate passes at each iteration.

As is necessary for the dynamic programming component, we factor the input by the relation that identifies nodes on a τ loop using `tau_loop_factor` and topologically sort the nodes by the τ-transition relation, recording a bidirectional map from nodes to their indices in the sort. As in change-tracking iterative refinement for strong bisimulation, we will maintain *changed* and *affected* bit vectors, the former initialised to contain all the nodes in the input. So that we can efficiently traverse *affected* in topological order, we will also maintain an *affected_topo* bit vector containing the indices of affected nodes under the topological sort. We do not want to create a map from nodes to their predecessors under the delayed transition relation due to space considerations: such a map could be as large as the transition set of the derived GLTS $\widehat{S}$, which could be significantly larger than the input GLTS. Instead, we will create a map from nodes to their predecessors under a single visible event and another map from nodes to their predecessors under a single τ. We will then use these two maps to compute the affected nodes.

Initial Approximation. The initial approximation can be computed by dynamic programming as in Section 3.2. To start, we record that the label for the most downstream node in the topological sort according to $\widehat{S}$ is as per S and its *initials* are as per S with the addition of τ. We then proceed upstream and for each node record its *initials* under $\widehat{S}$ as the union of $\{\tau\}$, its *initials* in S, and the *initials* of each of its τ-successors in $\widehat{S}$ (which have already been computed owing to the topological order of our computation); its label is computed similarly.

Iteration. As outlined in Figure 10, we first need to determine which nodes need their *coloured afters* recomputed, that is, the set of nodes potentially affected by the reclassification of nodes in *changed*, $\{n \mid m \in \mathit{changed} \wedge x \in \Sigma^\tau \wedge n \overset{x}{\hookrightarrow} m\}$ (lines 8-12). From the definition of $\hookrightarrow$, we can decompose this into $P \cup \{n \mid m \in P \wedge n \Longrightarrow m\}$ where P contains the reclassified nodes and their visible predecessors, $\mathit{changed} \cup \{n \mid m \in \mathit{changed} \wedge x \in \Sigma \wedge n \xrightarrow{x} m\}$. P is a subset of the nodes affected in the strong bisimulation sense, and can be computed using the map from nodes to their visible predecessors with the worst case run time in $O(t)$. We then compute *affected* (simultaneously updating *affected_topo*) using the map from nodes to their τ predecessors and any of the well-known graph exploration algorithms. Our implementation uses a depth-first search; to facilitate this, P is computed directly into a stack. The search takes $O(t)$ time at worst.

Next, we need to recompute the *coloured afters* for each of the nodes indexed by *affected_topo* (lines 14-27). To do this, we will use dynamic programming as described in Section 3.2. Finally, we proceed to sort and reclassify the nodes as in Section 2.3, recording which nodes have been reclassified in *changed*. We must also clear *affected* and *affected_topo* for the next iteration.

Figure 11 illustrates the important points of this algorithm on a small example.

```
**global** *toponodes*      ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.
**global** *changed*        ▷ Output from RefineChanged to ComputeChangedDelayedAfters.
**global** *affected*       ▷ Output from ComputeChangedDelayedAfters to RefineChanged.
**global** *cafters*        ▷ Read, written by ComputeChangedDelayedAfters; persisted for optimisation.
**global** $T^{-1}$         ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.
**global** $T_\tau^{-1}$    ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.

**function** ComputeChangedDelayedAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
    $affected \leftarrow \emptyset$
    **for each** $n \in changed$ **do**
        $affected \leftarrow affected \cup T^{-1}(n) \cup \{n\}$
    **end for**
    Add all nodes reachable from $affected$ via $T_\tau^{-1}$ to $affected$ using, e.g., a DFS.

    **for each** $n \in affected\_topo$ **do**
        **for each** $(a, M) \in cafters(n)$ **do**
            $cafters(n)(a) \leftarrow \emptyset$
        **end for**
        $cafters(n)(\tau) \leftarrow \{\rho(n)\}$                      ▷ Implicit self-loop.
        **for each** $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m\}$ **do**            ▷ Visible afters.
            $cafters(n)(a) \leftarrow cafters(n)(a) \cup \{\rho(m)\}$
        **end for**
        **for each** $m \in \{m \mid n \xrightarrow{\tau} m\}$ **do**              ▷ Delayed afters.
            **for each** $(a, M) \in cafters(m)$ **do**
                $cafters(n)(a) \leftarrow cafters(n)(a) \cup M$
            **end for**
        **end for**
    **end for**
    **return** *cafters*
**end function**

**function** DBisimCTDP($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
    $(N, \Sigma, \longrightarrow, \Lambda, \lambda) \leftarrow$ TauLoopFactor($G$)
    $toponodes \leftarrow N$ sorted topologically according to $\xrightarrow{\tau}$
    $changed \leftarrow N$
    $affected \leftarrow \emptyset$
    $cafters \leftarrow \langle (n, \{(a, \emptyset) \mid \exists m.n \overset{a}{\hookrightarrow} m\}) \mid n \in N \rangle$
    $T^{-1} \leftarrow \{(n, \{m \mid \exists a \in \Sigma.n \xrightarrow{a} m\}) \mid n \in N\}$
    $T_\tau^{-1} \leftarrow \{(n, \{m \mid n \xrightarrow{\tau} m\}) \mid n \in N\}$
    **return** IterativeRefinement(
                DBisimApproximation, ComputeChangedDelayedAfters, RefineChanged)
            ($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
**end function**
```

Fig. 10. The dynamic programming algorithm for DRDB with change tracking. We do not explicitly mention *affected_topo* for brevity, since it is always updated together with *affected*.

Construction. The output GLTS is constructed as usual.

Complexity. Factoring the cost of change tracking into the analysis for the version without change tracking, we note that the performance is still dominated by the iterated *coloured afters* computation, and is in the worst case $O(An^3c)$. In practice, change tracking should reduce the number of nodes that need their *coloured afters* recomputed at each iteration and improve the typical case.

The additional data structures required do not take up a significant amount of space. The topological

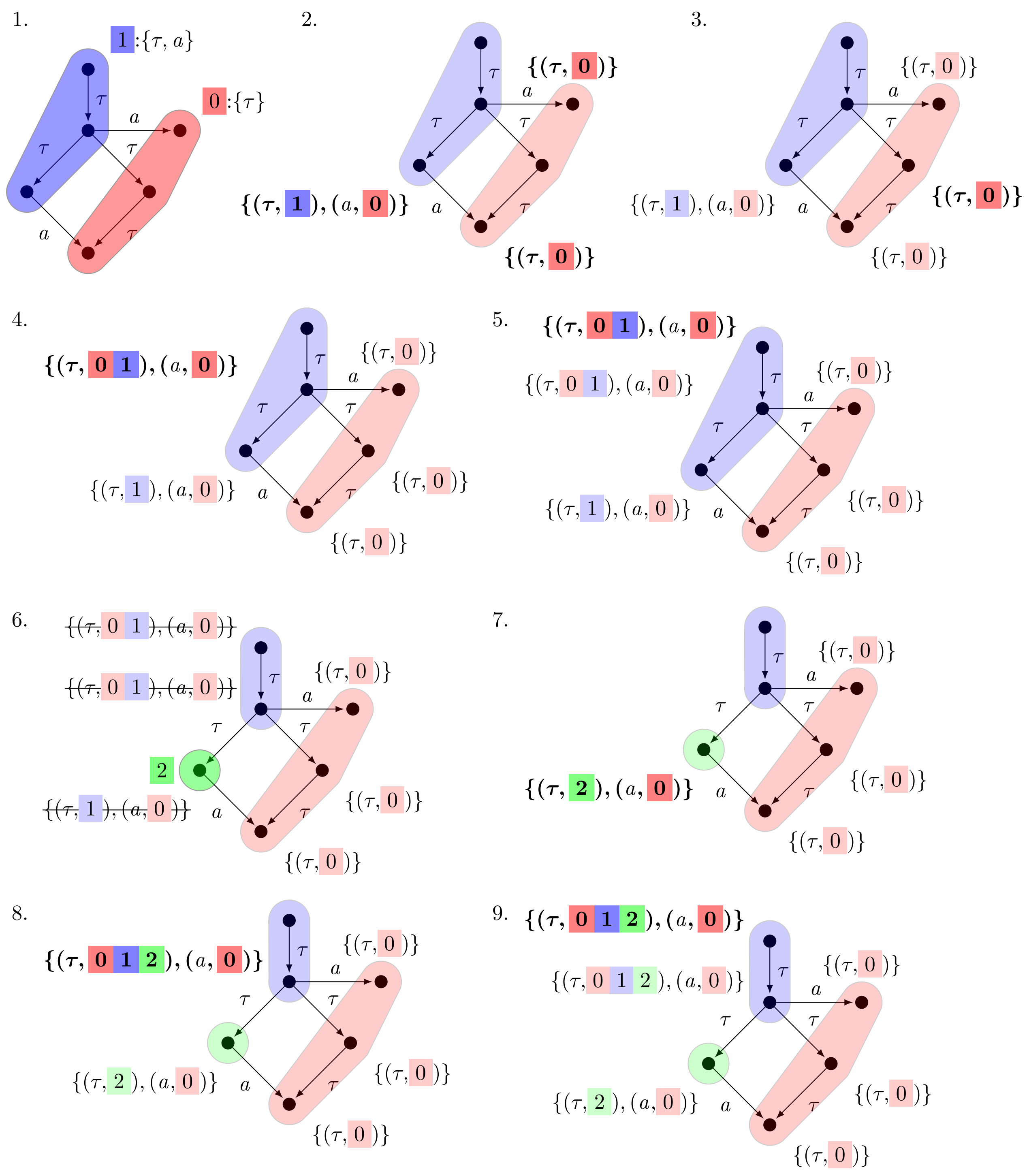


Fig. 11. An illustration of CTIR for delay bisimulation with dynamic programming. The coloured regions indicate equivalence classes. Bold face and brighter colours emphasise newly computed *coloured afters* and equivalence classes. First, an initial partition is made based on the *initials* (step 1). Then, the *coloured afters* are computed in topological order (steps 2-5). A reclassification splits class 2 from 1, invalidating some *coloured afters* (step 6); these are struck through. The invalidated *coloured afters* are recomputed in steps 7-9. The resulting partition is stable.

```
 1: global $toponodes$, $changed$, $affected$, $cafters$, $T^{-1}$, $T_\tau^{-1}$
 2:
 3: function ComputeChangedObservedAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
 4:     $affected \leftarrow \emptyset$
 5:     Add all nodes reachable from $changed$ via $T_\tau^{-1}$ to $affected$ using, e.g., a DFS.
 6:     $P_2 \leftarrow \emptyset$
 7:     for each $n \in affected$ do
 8:         $P_2 \leftarrow P_2 \cup T^{-1}(n)$
 9:     end for
10:     Add all nodes reachable from $P_2$ via $T_\tau^{-1}$ to $affected$ using, e.g., a DFS.
11:
12:     for each $n \in affected\_topo$ do                ▷ Compute nodes reachable in 0 or more $\tau$s:
13:         for each $(a, M) \in cafters(n)$ do
14:             $cafters(n)(a) \leftarrow \emptyset$
15:         end for
16:         $cafters(n)(\tau) \leftarrow \{\rho(n)\}$
17:         for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do
18:             $cafters(n)(\tau) \leftarrow cafters(n)(\tau) \cup cafters(m)(\tau)$
19:         end for
20:     end for
21:                         ▷ Compute nodes reachable in 0 or more $\tau$s, 1 visible event, 0 or more $\tau$s:
22:     for each $n \in affected\_topo$ do
23:         for each $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m \wedge a \neq \tau\}$ do
24:             $cafters(n)(a) \leftarrow cafters(n)(a) \cup cafters(m)(\tau)$
25:         end for
26:         for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do
27:             for each $(a, M) \in cafters(m), a \neq \tau$ do
28:                 $cafters(n)(a) \leftarrow cafters(n)(a) \cup M$
29:             end for
30:         end for
31:     end for
32:     return $cafters$
33: end function
```

Fig. 12. The dynamic programming algorithm for DRWB with change tracking. The WBisimCTDP function itself is not listed here due to space limitations; DBisimCTDP from Figure 10 can be used with the obvious modifications.

sort and the bit vectors take $\Theta(n)$ space. The predecessor maps combined have no more than t entries, and will often have fewer since transitions differing only in their visible events only require one entry.

3.4.2. Change-Tracking DRWB

The above algorithm can be adapted to compute the maximal DRWB instead of the maximal DRDB, as shown in Figure 12. The iterations are adapted as follows, while everything else remains unchanged.

Iteration. We first compute the set $\{n \mid m \in changed \wedge x \in \Sigma^\tau \wedge n \overset{x}{\Longrightarrow} m\}$ of nodes whose *coloured afters* need to be recomputed. This can be decomposed into $P_1 \cup P_2 \cup P_3$, where

$$P_1 = \{n \mid m \in changed \wedge n \Longrightarrow m\}$$
$$P_2 = \{n \mid m \in P_1 \wedge x \in \Sigma \wedge n \xrightarrow{x} m\}$$
$$P_3 = \{n \mid m \in P_2 \wedge n \Longrightarrow m\}.$$

P_1 can be computed using the map from nodes to their τ predecessors and any of the well-known graph exploration algorithms, such as a depth-first search (line 5). P_2 can be computed by iterating through P_1 and using the map from nodes to their visible predecessors (lines 6-9). Alternatively, computation of P_1 and P_2 can be interleaved by adding the visible predecessors of each node visited during the exploration

```
 1: global $toponodes$, $changed$, $affected$, $cafters$, $T^{-1}$, $T_\tau^{-1}$

 2: function ComputeChangedDelayedAfters'$((N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho)$
 3:     $affected\_queue \leftarrow$ empty min-priority queue
 4:     for each $n \in changed$ do
 5:         $affected \leftarrow affected \cup T^{-1}(n) \cup \{n\}$
 6:     end for
 7:     $affected\_queue \leftarrow$ min-priority queue from $affected$
 8:
 9:     while $affected\_queue$ is not empty do
10:         $n \leftarrow pull(affected\_queue)$
11:         for each $(a, M) \in cafters(n)$ do
12:             $cafters(n)(a) \leftarrow \emptyset$
13:         end for
14:         $cafters(n)(\tau) \leftarrow \{\rho(n)\}$                          ▷ Implicit self-loop.
15:         for each $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m\}$ do                 ▷ Visible afters.
16:             $cafters(n)(a) \leftarrow cafters(n)(a) \cup \{\rho(m)\}$
17:         end for
18:         for each $m \in \{m \mid n \xrightarrow{\tau} m\}$ do                        ▷ Delayed afters.
19:             for each $(a, M) \in cafters(m)$ do
20:                 $cafters(n)(a) \leftarrow cafters(n)(a) \cup M$
21:             end for
22:         end for
23:
24:         $insert\_all(affected\_queue, T_\tau^{-1}(n) \setminus affected)$
25:         $affected \leftarrow affected \cup T_\tau^{-1}(n)$
26:     end while
27:     return $cafters$
28: end function
```

Fig. 13. The single-pass dynamic programming algorithm for DRDB with change tracking. The DBisimCTDP' function itself is not listed here due to space limitations; DBisimCTDP from Figure 10 can be used with the obvious modifications.

of P_1 to P_2 immediately. P_3 can be computed with another depth-first search starting from P_2 (line 10). While computing these sets, *affected_topo* and *affected* should be kept up to date. We can recompute the *coloured afters* for each of the affected nodes by iterating through *affected_topo* and using the dynamic programming approach outlined in Section 3.3 (lines 12-31). Finally, we proceed to sort and reclassify the nodes as in Section 2.3, recording which nodes have been reclassified in *changed*. We must also clear *affected* and *affected_topo* for the next iteration.

Performance. Each of the P_i can be computed in $O(t)$ time, so the worst-case time complexity remains in $O(An^3c)$.

3.4.3. Single-Pass Change-Tracking DRDB

Instead of separating change tracking and the dynamic *coloured afters* computation into two passes, they can be performed in the same pass, as shown in Figure 13. Since the *coloured afters* need to be computed in topological order, but the affected nodes will in general not be discovered in this order, we will store the affected nodes in a min-priority queue, *affected_queue*, with each node's priority being its position in the topological order (so more downstream nodes will be pulled first). All insertions into (but not pulls from) *affected_queue* will be mirrored to the *affected* bit vector, so that the latter can be used for checking whether a node has already been seen and during the reclassification phase for determining which nodes' *coloured afters* might have changed. We do not use a *changed* bit vector in this algorithm, we will still use it in the analysis to represent the set of nodes that changed class in the previous iteration. For change tracking, we will need a map from nodes to their predecessors under a single visible event and another map from nodes to their predecessors under a single τ. We also need to topologically sort the nodes and record a bidirectional map from nodes to their indices in the sort.

Initial Approximation. The initial approximation is computed as in Section 3.2. All nodes are marked changed by inserting them into *affected_queue* (and correspondingly *affected*).

Iteration. As a precondition to each iteration, we expect *affected_queue* to contain all the reclassified nodes and their visible predecessors, $P = \mathit{changed} \cup \{n \mid m \in \mathit{changed} \wedge x \in \Sigma \wedge n \xrightarrow{x} m\}$ (lines 3-7) We then repeat the following steps until *affected_queue* is empty:

Pull a node n from *affected_queue* (line 10). We will only update *coloured afters* for nodes pulled from *affected_queue* and we will only insert nodes with higher topological indices during this iteration of *affected_queue*, so we know all the nodes downstream from n have had their *coloured afters* computed for this iteration. Therefore, compute and record n's *coloured afters* according to $\widehat{S}$ as the union of itself after τ (line 14) its *coloured afters* according to S (lines 15-17) and the *coloured afters* according to $\widehat{S}$ of all nodes reachable from n in exactly one τ (lines 18-22). Finally, insert each of the predecessors of n under τ that is not already in *affected* into *affected_queue* and *affected* (lines 24-25); this is the step that ensures that all nodes in $\{n \mid m \in P \wedge n \Longrightarrow m\} = \{n \mid m \in \mathit{changed} \wedge x \in \Sigma^\tau \wedge n \overset{x}{\hookrightarrow} m\}$ eventually get inserted into *affected_queue*.

By this point, *affected_queue* is empty, so all the affected nodes have had their *coloured afters* updated and have been marked in *affected*. We can therefore proceed with the reclassification stage described in Section 2.3. However, instead of inserting changed nodes into *changed*, we will insert them and their visible predecessors into *affected_queue*, to satisfy the precondition of the next iteration.

Construction. The output GLTS is constructed as usual.

Complexity. The *coloured afters* computation and reclassification are the same as in our other dynamic programming approaches. It is only the change tracking that is different, now that in addition to the graph traversal we have $O(n)$ pulls from and insertions into *affected_queue*, which can be done in $O(n \log n)$ time. This does not increase the overall asymptotic time complexity from $O(An^3c)$, but can in practice impact performance (see Section 4.3). Because of this and the existence of the simpler two-pass algorithm, we have not developed a similarly interleaved version for DRWB.

3.5. DRDB with the Paige-Tarjan Algorithm

We can adapt the multilabel version of the Paige-Tarjan algorithm presented in [Fer90] to compute a maximal DRDB. This will use a similar form of change tracking as that described in Section 3.4.1.

If the input GLTS does not have node labels, we initially assign all the nodes to the same class. Otherwise, we topologically sort the nodes and compute their node labels according to $\widehat{S}$ using dynamic programming. We then create an initial partition based on the node labels.

We leave the core of the algorithm unmodified, but instead of using the inverted transition relation of $\widehat{S}$ (which would suffer from the potentially quadratic explosion discussed in Section 3.1), we compute it dynamically each time it is requested. To facilitate this, we pre-compute the inverted transition relation of S (this can be done in $\Theta(t)$ time). When we need to visit the nodes in $\{m \mid m \overset{a}{\hookrightarrow} n\}$, we compute a starting set P such that $\{m \mid m \overset{a}{\hookrightarrow} n\} = \{m \mid p \in P \wedge m \Longrightarrow p\}$. If $a = \tau$, $P = \{n\}$; otherwise, $P = \{m \mid m \xrightarrow{a} n\}$. We then visit all nodes reachable from P in the inverted transition relation of S under τ using any of the known algorithms, such a depth-first search. This exploration can take time in $O(t)$.

Complexity. The added computation can increase the cost of exploring the predecessors of a node from $O(|\{m \mid m \xrightarrow{a} n\}|)$ as shown in [PT87] to $O(t)$. The cost of a single refinement by block B therefore takes $O(At|B|)$ time. Recalling from [PT87] that each node can be in at most $\log_2 n + 1$ blocks used for refinement and summing over all such blocks, we get a total run time in $O(Atn \log n)$.

3.6. DRWB with the Paige-Tarjan Algorithm

We can similarly adapt the multilabel version of the Paige-Tarjan algorithm presented in [Fer90] to compute a maximal DRWB. This will use a similar form of change tracking to that described in Section 3.4.2. We can create the initial partition by dynamically computing the labels of each node according to $\widehat{S}$ as described in Section 3.5.

To dynamically compute the inverted transition relation of $\widehat{S}$, we pre-compute the inverted transition relation of S. When we need to visit the nodes in $\{m \mid m \overset{a}{\Longrightarrow} n\}$, we note that this is equivalent to P_3, where

$$\begin{aligned}
&P_1 = \{m \mid m \Longrightarrow n\}\\
&P_2 = \{m \mid p \in P_1 \wedge m \xrightarrow{a} p\}\\
&P_3 = \{m \mid p \in P_2 \wedge m \Longrightarrow p\}.
\end{aligned}$$

So, we compute P_1 using the precomputed inverted transition relation and any of the known algorithms, such a depth-first search. P_2 can be computed by iterating through P_1 and again using the inverted transition relation. Finally, P_3 can be explored using another depth-first search. The entire process can take time in $O(t)$.

Complexity Applying the same logic as in Section 3.5, we get a total run time in $O(Atn \log n)$.

4. Performance

4.1. Benchmark Descriptions

In the following performance tests we will compare the performances of the algorithms described throughout the paper to each other, as well as to implementations provided by the BCG_MIN tool included in the CADP Toolbox [GLMS13].

Our primary source for example LTSs is the Very Large Transition Systems (VLTS) Benchmark Suite[5] that is provided alongside the CADP Toolbox; some information about the LTSs it contains is given in Table 1. Note that some of the included LTSs do not contain any τ transitions, meaning that `dbisim` and `wbisim` cannot offer more compression than `sbisim`. We will still include them in all the tests for completeness. We have excluded those examples where the run time for all of the tested algorithms did not exceed 1 second.

To ensure proper operation of FDR3, we have developed a suite of regression and feature tests containing tests generated randomly at runtime, examples from [Ros98] and [Ros10], and assorted test files. CSP processes are frequently written as compositions of smaller component processes, which in turn can be such compositions. To help mitigate the state explosion that can result from such combinations, FDR3 can automatically apply compressions to leaf processes (those components that are not composed of other processes, and will be represented as explicit GLTSs). By default, this compression is `sbisim`, so many of the tests exercise `sbisim`. There are about 90,000 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. Due to the nature of these tests, results are not available for CADP.

4.2. Strong Bisimulation Performance

We will now compare the performance of the `sbisim` algorithms on several examples. The test system used contains a server-grade CPU[6] and 256 GiB of RAM, running 64-bit Debian GNU/Linux 7.8. We will use the

[5] At the time of writing, the VLTS Benchmark Suite is available from `http://cadp.inria.fr/resources/vlts/`.
[6] The system has two 8-core 2 GHz Intel® Xeon® E5-2650 CPUs with 20 MB of cache each. Our bisimulation algorithms are single-threaded, so the number of CPUs and cores is not significant.

Table 1. Parameters of the VLTS Benchmark Suite [GLMS13] used in our performance tests.

Name	States	Transitions	τ-transitions	Alphabet size	Branching avg	[min-max]
`cwi_3_14`	3996	14552	14551	2	3.64	[0 - 6]
`vasy_18_73`	18746	73043	39217	17	3.90	[1 - 6]
`vasy_25_25`	25217	25216	0	25216	1.00	[0 - 1]
`vasy_40_60`	40006	60007	20003	3	1.50	[1 - 2]
`vasy_52_318`	52268	318126	130752	17	6.09	[1 - 17]
`vasy_65_2621`	65537	2621480	0	72	40.00	[40 - 40]
`vasy_66_1302`	66929	1302664	117866	81	19.46	[2 - 42]
`vasy_69_520`	69754	520633	1	135	7.46	[0 - 35]
`vasy_83_325`	83436	325584	45696	211	3.90	[0 - 96]
`vasy_116_368`	116456	368569	263296	21	3.16	[1 - 8]
`cwi_142_925`	142472	925429	862298	7	6.50	[0 - 9]
`vasy_157_297`	157604	297000	31798	235	1.88	[0 - 48]
`vasy_164_1619`	164865	1619204	109910	37	9.82	[1 - 16]
`vasy_166_651`	166464	651168	91392	211	3.91	[0 - 96]
`cwi_214_684`	214202	684419	550611	5	3.20	[0 - 7]
`cwi_371_641`	371804	641565	445600	61	1.73	[1 - 25]
`vasy_386_1171`	386496	1171872	122976	73	3.03	[1 - 38]
`cwi_566_3984`	566640	3984157	3666614	11	7.03	[0 - 10]
`vasy_574_13561`	574057	13561040	0	141	23.62	[1 - 64]
`vasy_720_390`	720247	390999	1	49	0.54	[0 - 39]
`vasy_1112_5290`	1112490	5290860	0	23	4.76	[3 - 6]
`cwi_2165_8723`	2165446	8723465	3830225	26	4.03	[1 - 14]
`cwi_2416_17605`	2416632	17605592	17490904	15	7.29	[0 - 14]
`vasy_2581_11442`	2581374	11442382	2508518	223	4.43	[0 - 97]
`vasy_4220_13944`	4220790	13944372	2546649	223	3.30	[0 - 97]
`vasy_4338_15666`	4338672	15666588	3127116	223	3.61	[0 - 97]
`vasy_6020_19353`	6020550	19353474	17526144	511	3.21	[2 - 260]
`vasy_6120_11031`	6120718	11031292	3152976	125	1.80	[0 - 16]
`cwi_7838_59101`	7838608	59101007	22842122	20	7.54	[3 - 13]
`vasy_8082_42933`	8082905	42933110	2535944	211	5.31	[0 - 48]
`vasy_11026_24660`	11026932	24660513	2748559	119	2.24	[0 - 13]
`vasy_12323_27667`	12323703	27667803	3153502	119	2.25	[0 - 13]
`cwi_33949_165318`	33949609	165318222	74133306	31	4.87	[1 - 17]

following abbreviations to refer to the algorithms in column headings:

Shorthand	Description
NIRa	Naïve Iterative Refinement using a tree-based set representation
CTIRa	Change-Tracking Iterative Refinement using a tree-based set representation
NIRb	Naïve Iterative Refinement using a sorted array set representation
CTIRb	Change-Tracking Iterative Refinement using a sorted array set representation (FDR3)
PT(LL)	Multilabel Paige-Tarjan algorithm using linked lists
PT(A)	Multilabel Paige-Tarjan algorithm using arrays
CADP	`BCG_MIN -strong` from the CADP Toolbox [GLMS13]

We can see from Table 2 that for most of our experiments, change-tracking iterative refinement outperforms naïve iterative refinement and the Paige-Tarjan algorithm: this is particularly noticeable for larger checks. For smaller inputs, naïve iterative refinement is sometimes slightly faster due to a smaller bookkeeping overhead, but not significantly so. The Paige-Tarjan algorithm was particularly slow on `vasy_25_25` due to its large alphabet; our iterative refinement algorithms were designed for labelled transition systems and do not suffer from the large alphabet. CADP's implementation of strong bisimulation was comparable to our change-tracking iterative refinement with sorted vectors, performing better on some examples and worse on others. We have omitted NIRa and CTIRa from the table in order to avoid cluttering it with implementation details. On average, NIRa took 2.3 times as long as NIRb and CTIRa took 1.6 times as long as CTIRb on these VLTS examples.

In the FDR3 test suite, the move from naïve to change-tracking iterative refinement affords a noticeable speedup, as evidenced by Table 3. Much of this speedup can be attributed to a number of outliers that take a

Table 2. Run times of various implementations of strong bisimulation on the VLTS benchmarks in seconds.

Name	NIRb	CTIRb	PT(LL)	PT(A)	CADP
`cwi_3_14`	0.051	0.013	**0.010**	0.011	0.256
`vasy_18_73`	0.118	**0.062**	0.191	0.197	0.283
`vasy_25_25`	**0.022**	0.029	59.035	54.115	1.533
`vasy_40_60`	203.763	35.430	**2.284**	7.031	70.081
`vasy_52_318`	0.500	**0.392**	1.210	0.919	0.569
`vasy_65_2621`	**1.177**	1.436	21.275	26.563	2.642
`vasy_66_1302`	**0.640**	0.728	8.130	9.598	1.360
`vasy_69_520`	0.475	**0.410**	3.815	4.814	0.791
`vasy_83_325`	**0.331**	0.387	4.068	5.163	0.695
`vasy_116_368`	0.811	**0.752**	2.422	3.298	0.955
`cwi_142_925`	1.134	**0.817**	1.785	1.583	1.355
`vasy_157_297`	1.527	**0.307**	3.157	1.817	0.496
`vasy_164_1619`	1.762	1.369	5.515	3.987	**1.280**
`vasy_166_651`	0.894	**0.815**	7.907	11.389	0.974
`cwi_214_684`	6.290	**1.636**	2.915	3.529	2.440
`cwi_371_641`	9.112	**1.342**	4.178	4.162	2.642
`vasy_386_1171`	2.159	1.376	4.458	3.256	**1.193**
`cwi_566_3984`	6.491	**4.217**	10.234	7.834	5.588
`vasy_574_13561`	9.327	9.469	51.622	32.324	**8.171**
`vasy_720_390`	0.993	**0.413**	1.766	1.039	0.653
`vasy_1112_5290`	6.925	6.046	26.715	20.254	**4.093**
`cwi_2165_8723`	100.482	**14.547**	37.711	37.682	16.606
`cwi_2416_17605`	57.061	**15.115**	42.796	35.995	28.379
`vasy_2581_11442`	29.123	**20.982**	234.762	2438.673	28.410
`vasy_4220_13944`	85.000	**21.910**	267.716	734.701	26.866
`vasy_4338_15666`	51.796	**28.358**	332.553	1948.656	32.280
`vasy_6020_19353`	70.039	36.194	897.281	438.441	**22.998**
`vasy_6120_11031`	119.617	23.526	196.442	205.547	**15.888**
`cwi_7838_59101`	749.059	**115.385**	490.897	1063.371	135.400
`vasy_8082_42933`	130.746	73.546	416.726	280.090	**39.251**
`vasy_11026_24660`	361.281	50.874	444.343	847.334	**41.733**
`vasy_12323_27667`	303.017	56.320	509.774	1010.133	**49.498**
`cwi_33949_165318`	3256.559	**334.751**	962.506	1151.239	364.443
Geometric Mean	7.353	**3.236**	17.785	21.275	4.993

Table 3. `sbisim` timings for the FDR test suite. Total runtime and the 5 longest invocations for each algorithm (not necessarily corresponding to the same inputs).

	NIRa	CTIRa	NIRb	CTIRb	PT(LL)	PT(A)
Total	62.044	29.159	30.184	**20.313**	45.326	51.961
1	5.068	0.529	2.004	0.32	0.289	0.405
2	4.984	0.443	1.989	0.297	0.287	0.341
3	4.773	0.397	1.964	0.285	0.267	0.321
4	4.581	0.388	1.934	0.262	0.256	0.32
5	4.316	0.378	1.901	0.235	0.254	0.312

particularly long time with naïve iterative refinement. The Paige-Tarjan algorithm is noticeably slower than both naïve and 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. It should be noted, however, that the worst-case run times for the Paige-Tarjan algorithm are similar to those for change-tracking iterative refinement.

Table 4. Run times of various implementations of delay bisimulation on the VLTS benchmarks in seconds. A — indicates that the test failed to complete within 4 hours.

Name	FDR2	DYN	CT-DYN1	CT-DYN	PT
`cwi_3_14`	4.640	**0.010**	0.011	0.011	0.347
`vasy_18_73`	1.018	0.334	0.222	**0.204**	1.470
`vasy_25_25`	**0.061**	0.107	0.096	0.096	663.200
`vasy_40_60`	169.843	160.661	15.689	15.620	**7.097**
`vasy_52_318`	6.722	0.378	0.383	**0.353**	3.517
`vasy_65_2621`	2.960	**1.585**	1.723	1.732	63.035
`vasy_66_1302`	2.720	1.283	1.517	**1.154**	31.171
`vasy_69_520`	1.589	0.643	0.567	**0.530**	17.922
`vasy_83_325`	1.030	**0.433**	0.479	0.443	23.468
`vasy_116_368`	70.398	4.993	2.631	**2.391**	74.627
`cwi_142_925`	525.796	0.840	0.771	**0.717**	101.659
`vasy_157_297`	3.214	2.238	0.630	**0.564**	18.707
`vasy_164_1619`	4.060	2.903	2.197	**2.018**	14.724
`vasy_166_651`	3.302	0.985	1.038	**0.938**	50.885
`cwi_214_684`	113.378	5.116	1.956	**1.779**	28.247
`cwi_371_641`	96.592	2.608	2.251	**1.985**	95.357
`vasy_386_1171`	14.717	2.336	2.060	**1.963**	25.967
`cwi_566_3984`	—	4.714	4.486	**4.133**	826.634
`vasy_574_13561`	52.534	**13.522**	14.301	13.921	178.554
`vasy_720_390`	44.242	2.464	1.520	**1.412**	13.268
`vasy_1112_5290`	132.888	9.715	9.261	**8.667**	192.439
`cwi_2165_8723`	—	45.885	28.791	**26.823**	7262.560
`cwi_2416_17605`	—	**18.461**	20.858	19.548	—
`vasy_2581_11442`	730.577	29.734	23.908	**19.694**	6695.113
`vasy_4220_13944`	2128.400	83.883	34.915	**29.632**	4359.442
`vasy_4338_15666`	2257.923	51.196	35.149	**29.612**	6638.925
`vasy_6020_19353`	7448.062	**8.199**	9.143	8.739	811.669
`vasy_6120_11031`	6305.384	131.807	34.635	**30.236**	6980.648
`cwi_7838_59101`	—	9905.934	5214.302	**5017.498**	—
`vasy_8082_42933`	—	117.119	96.233	**89.629**	6268.539
`vasy_11026_24660`	—	442.642	94.820	**80.174**	—
`vasy_12323_27667`	—	434.968	108.348	**89.650**	—
`cwi_33949_165318`	—	1198.680	663.610	**592.355**	—
Geometric Mean	—	7.937	5.211	**4.759**	—

4.3. Divergence-Respecting Delay Bisimulation Performance

We will use the following abbreviations to refer to the various DRDB algorithms compared in column headings, with all variants of iterative refinement using sorted arrays to represent the *coloured afters*:

Shorthand	Description
FDR2	Reduction to strong bisimulation
DYN	Naïve IR with dynamic computation of *afters*
CT-DYN	Two-pass Change-Tracking IR with dynamic computation of *afters* (FDR3)
CT-DYN1	Single-pass Change-Tracking IR with dynamic computation of *afters*
PT	Paige-Tarjan algorithm using arrays

From Table 4 we can see that two-pass change-tracking iterative refinement with dynamic computation of *coloured afters* is significantly faster than the alternatives on nearly all of the examples. We also note that as for strong bisimulation, the Paige-Tarjan algorithm is particularly ineffective when dealing with large alphabets, as in `vasy_25_25` since it must repeat each refinement for every event in the alphabet.

4.4. Divergence-Respecting Weak Bisimulation Performance

We will use the following abbreviations to refer to the various DRWB algorithms compared in column headings, with both variants of iterative refinement using sorted arrays to represent the *coloured afters*:

Table 5. Run times of various implementations of weak bisimulation on the VLTS benchmarks in seconds. A — indicates that the test failed to complete within 4 hours.

Name	DYN	CT-DYN	PT	CADP-nobr	CADP-br
`cwi_3_14`	**0.010**	0.011	0.542	2.632	0.244
`vasy_18_73`	0.448	**0.279**	5.125	3.615	0.742
`vasy_25_25`	**0.092**	0.094	1558.280	1.134	1.468
`vasy_40_60`	170.634	15.734	**7.222**	109.133	115.415
`vasy_52_318`	0.424	**0.383**	9.032	7.152	0.806
`vasy_65_2621`	**1.605**	2.069	64.023	2.543	5.196
`vasy_66_1302`	1.442	**1.364**	65.053	9.512	8.370
`vasy_69_520`	0.783	**0.597**	21.125	6.573	2.001
`vasy_83_325`	0.492	**0.471**	69.043	12.762	0.997
`vasy_116_368`	5.990	**3.070**	410.344	221.241	48.121
`cwi_142_925`	**0.736**	0.767	306.684	290.736	1.098
`vasy_157_297`	2.513	**0.649**	37.907	45.589	0.694
`vasy_164_1619`	3.309	2.265	40.964	11.796	**1.947**
`vasy_166_651`	1.131	**1.041**	202.888	47.152	1.369
`cwi_214_684`	5.119	**2.177**	120.182	500.183	2.715
`cwi_371_641`	3.448	2.731	1913.489	413.884	**1.602**
`vasy_386_1171`	2.503	2.049	125.959	85.951	**1.452**
`cwi_566_3984`	5.358	4.531	3914.916	—	**4.085**
`vasy_574_13561`	14.624	14.749	210.813	**8.404**	8.618
`vasy_720_390`	2.541	**1.458**	14.330	195.721	1.468
`vasy_1112_5290`	10.895	9.523	238.637	**4.240**	4.375
`cwi_2165_8723`	80.916	41.494	—	7424.895	**13.869**
`cwi_2416_17605`	18.730	20.600	—	—	**16.691**
`vasy_2581_11442`	39.954	**23.773**	—	—	31.457
`vasy_4220_13944`	111.596	**37.217**	—	—	4148.586
`vasy_4338_15666`	64.766	**34.958**	—	—	38.171
`vasy_6020_19353`	**8.588**	9.167	—	—	12.011
`vasy_6120_11031`	149.394	33.287	—	—	**21.602**
`cwi_7838_59101`	12422.744	**6499.632**	—	—	9164.557
`vasy_8082_42933`	126.767	98.128	—	—	**42.469**
`vasy_11026_24660`	538.958	**91.665**	—	—	607.453
`vasy_12323_27667`	533.449	**103.169**	—	—	778.553
`cwi_33949_165318`	2665.535	1099.680	—	—	**259.913**
Geometric Mean	9.232	**5.518**	—	—	10.451

Shorthand	Description
DYN	Naïve IR with dynamic computation of *coloured afters*
CT-DYN	Two-pass Change-Tracking IR with dynamic computation of *afters* (FDR3)
PT	Paige-Tarjan algorithm using arrays
CADP-nobr	`BCG_MIN -observational -class` from the CADP Toolbox [GLMS13]
CADP-br	`BCG_MIN -observational` from the CADP Toolbox [GLMS13]

`BCG_MIN -observational` by default applies branching bisimulation before applying weak bisimulation; this is what CADP-br measures. This behaviour can be disabled with the `-class` option, allowing us to measure the performance of the weak bisimulation directly; we denote this CADP-nobr.

From Table 5 we can see that change-tracking iterative refinement with dynamic computation of *coloured afters* is significantly faster than the alternatives that rely solely on weak bisimulation on nearly all of the examples. CADP's branching bisimulation followed by weak bisimulation was found to be slower or faster depending on the example.

4.5. Other Compressions

It is interesting to compare `dbisim` and `wbisim` with alternative compressions. Prior to their introduction in FDR, the most widely used compression was `sbisim`(`diamond`(P)), which we will call `sbdia`. In all the

Table 6. Timings with no compression, `dbisim`, and `sbdia`.

Problem	Compilation Time (s)			Exploration Time (s)		
	Raw	`dbisim`	`sbdia`	Raw	`dbisim`	`sbdia`
`bully`	0.06	28.12	25.17	1.76	0.36	0.88
`bakery.3`	0.37	0.38	0.36	137.52	0.93	1.07
`bakery.4`	—	12.28	9.54	—	3.63	1.64

Table 7. State and transition counts with no compression, `dbisim`, and `sbdia`. "Raw" indicates the size before compression.

Problem	States			Transitions		
	Raw	`dbisim`	`sbdia`	Raw	`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

following examples `sbdia` is valid. Other tools also use divergence-respecting *branching* bisimulation due to the existence of a fast algorithm for computing it.

We examined the performance and effectiveness of `dbisim` and `sbdia` on the *bully* algorithm (the FDR implementation is outlined in Section 14.4 of [Ros10]) with 5 processors and an implementation of Lamport's bakery algorithm (Section 18.5 of [Ros10]) with either 3 or 4 threads and integers drawn from the fixed range 0 to 7.[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 compressed these processes *inductively*[8] (as described in Section 8.8 of [Ros10]); that is, we added 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 6 shows that `sbdia` runs faster than `dbisim` and Table 7 shows that it is more effective at reducing state counts, but can add transitions, whereas `dbisim` cannot by design.

Table 8 compares the effectiveness of `wbisim`, `dbisim`, branching bisimulation, and `sbdia` on the VLTS benchmarks. Table 9 compares their run times, as well as `sbisim`'s. We have found that `sbisim` is frequently slower than the other compressions despite being less effective.

5. Conclusions

We have presented a number of GLTS compression algorithms, including novel algorithms for computing the maximal delay and weak bisimulation. 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 that is particularly effective when used in conjunction with change-tracking iterative refinement.

CADP's attempt to reduce this potential explosion by applying branching bisimulation reduction first, thus reducing the size of the input to weak bisimulation, was effective in some cases, but not others. Since our approach is able to cope with large transition systems without requiring them to be pre-compressed by a more time-efficient compression we believe it to be a useful contribution.

Change-tracking iterative refinement algorithm for `sbisim` offered a significant improvement over the naïve iterative refinement used by previous versions of FDR, supporting the conclusions of [BO05]. It outperformed the Paige-Tarjan algorithm as well; in particular, we found the multilabel Paige-Tarjan algorithm of [Fer90] to not be tractable for systems with large alphabets.

Comparing `dbisim` and `wbisim`, we have noticed that they produce identical output on nearly all of the examples we have tested and differ by only a few states when they differ. They also tend to exhibit similar run times. Divergence-respecting branching bisimulation frequently produced identical output to `dbisim`,

[7] The example files are available from the authors website.
[8] 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.

Table 8. A comparison of the efficiencies of different compressions.

Problem	States				Transitions		
	Raw	`w/dbisim`	`branch`	`sbdia`	Raw	`wbisim`	`sbdia`
`cwi_3_14`	3996	2	2	2	14552	1	1
`vasy_18_73`	18746	2326	2326	**954**	73043	9751	**5727**
`vasy_25_25`	25217	25217	25217	25217	25216	25216	25216
`vasy_40_60`	40006	20003	20003	20003	60007	40004	40004
`vasy_52_318`	52268	66	4593	**28**	318126	333	**120**
`vasy_65_2621`	65537	65536	65536	65536	2621480	2621440	2621440
`vasy_66_1302`	66929	51128	51128	51128	1302664	**1018692**	1505446
`vasy_69_520`	69754	69753	69753	69753	520633	520632	520632
`vasy_83_325`	83436	42195	42195	42195	325584	197200	197200
`vasy_116_368`	116456	17641	22398	**616**	368569	72955	**1987**
`cwi_142_925`	142472	19	23	**10**	925429	37	**16**
`vasy_157_297`	157604	3038	3038	3038	297000	12095	12095
`vasy_164_1619`	164865	992	992	**512**	1619204	3456	**1408**
`vasy_166_651`	166464	42195	42195	42195	651168	197200	197200
`cwi_214_684`	214202	450	603	**222**	684419	1546	**1372**
`cwi_371_641`	371804	2134	6033	**1496**	641565	5634	**5100**
`vasy_386_1171`	386496	71	71	71	1171872	108	108
`cwi_566_3984`	566640	128	198	**21**	3984157	523	**50**
`vasy_574_13561`	574057	3577	3577	3577	13561040	16168	16168
`vasy_720_390`	720247	3292	3292	**3277**	390999	116910	**116498**
`vasy_1112_5290`	1112490	265	265	265	5290860	1300	1300
`cwi_2165_8723`	2165446	**4256**	**4256**	4701	8723465	**20880**	87575
`cwi_2416_17605`	2416632	730	730	**2**	17605592	2899	**14**
`vasy_2581_11442`	2581374	704737	704737	704737	11442382	3972600	3972600
`vasy_4220_13944`	4220790	1185975	1186266	**483404**	13944372	6862722	**3420840**
`vasy_4338_15666`	4338672	704737	704737	704737	15666588	3972600	3972600
`vasy_6020_19353`	6020550	256	256	256	19353474	510	510
`vasy_6120_11031`	6120718	2505	2505	2505	11031292	5358	5358
`cwi_7838_59101`	7838608	61233	62031	**36972**	59101007	**464102**	1369417
`vasy_8082_42933`	8082905	290	290	290	42933110	680	680
`vasy_11026_24660`	11026932	775578/775618	775618	**637639**	24660513	2454736	**1993745**
`vasy_12323_27667`	12323703	876944	876944	**719324**	27667803	2780022	**2251773**
`cwi_33949_165318`	33949609	**12463**	**12463**	15121	165318222	**71466**	500580

but was on some examples much less effective (for instance, compressing `vasy_52_318` from 52268 to 4593 states versus 66 states for `dbisim`). While branching bisimulation was usually faster than `dbisim`, with the exception of one VLTS example, the difference was not as striking as we had expected – roughly a factor of 2. The comparison between `dbisim` and `sbdia` was rather more varied: while the difference is not nearly as large as in previous versions of FDR and there are a number of examples where `dbisim` is significantly faster than `sbdia` (most notably, 10 minutes against 150 on the largest VLTS example we tried), there are still many examples where `sbdia` is the better choice.

We were most surprised to find that `dbisim` was occasionally faster than `sbisim`, despite offering more compression. In fact, the surprising speed is *because* of the higher compression, since the number of classes affects both the amount of work that needs to be done on each iteration and the number of iterations.

Related and 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.

On-the-fly τ-closure reduction [Mat05] is related to our *coloured afters* computation for DRDB. Mateescu's approach initiates a depth-first search from each of the nodes under consideration. Our method is able to avoid much of the overlapping work these DFSs might do by topologically sorting *all* the nodes and using dynamic programming. This is not possible for on-the-fly verification since the nodes are discovered

Table 9. A comparison of the run times of different compressions.

Name	`wbisim`	`dbisim`	`sbdia`	`branch`	`sbisim`
`cwi_3_14`	0.011	0.011	**0.003**	0.069	0.013
`vasy_18_73`	0.279	0.204	0.180	0.167	**0.062**
`vasy_25_25`	0.094	0.096	0.050	1.136	**0.029**
`vasy_40_60`	15.734	15.620	**11.723**	70.846	35.430
`vasy_52_318`	0.383	**0.353**	0.546	0.492	0.392
`vasy_65_2621`	2.069	1.732	2.271	2.541	**1.436**
`vasy_66_1302`	1.364	1.154	1.588	1.355	**0.728**
`vasy_69_520`	0.597	0.530	0.588	0.881	**0.410**
`vasy_83_325`	0.471	0.443	**0.308**	0.544	0.387
`vasy_116_368`	3.070	2.391	**0.233**	0.819	0.752
`cwi_142_925`	0.767	0.717	**0.332**	0.729	0.817
`vasy_157_297`	0.649	0.564	0.445	0.472	**0.307**
`vasy_164_1619`	2.265	2.018	**0.847**	1.443	1.369
`vasy_166_651`	1.041	0.938	**0.682**	0.922	0.815
`cwi_214_684`	2.177	1.779	**0.353**	1.566	1.636
`cwi_371_641`	2.731	1.985	6.465	**1.336**	1.342
`vasy_386_1171`	2.049	1.963	1.605	**1.258**	1.376
`cwi_566_3984`	4.531	4.133	**2.529**	3.665	4.217
`vasy_574_13561`	14.749	13.921	14.484	**8.325**	9.469
`vasy_720_390`	1.458	1.412	0.417	0.850	**0.413**
`vasy_1112_5290`	9.523	8.667	8.346	**4.163**	6.046
`cwi_2165_8723`	41.494	26.823	116.233	**12.907**	14.547
`cwi_2416_17605`	20.600	19.548	**3.480**	14.604	15.115
`vasy_2581_11442`	23.773	19.694	**10.129**	23.388	20.982
`vasy_4220_13944`	37.217	29.632	**16.541**	28.392	21.910
`vasy_4338_15666`	34.958	29.612	**18.876**	30.091	28.358
`vasy_6020_19353`	9.167	8.739	**7.892**	12.107	36.194
`vasy_6120_11031`	33.287	30.236	**16.981**	20.673	23.526
`cwi_7838_59101`	6499.632	5017.498	1801.493	145.481	**115.385**
`vasy_8082_42933`	98.128	89.629	78.149	**42.434**	73.546
`vasy_11026_24660`	91.665	80.174	54.986	**48.827**	50.874
`vasy_12323_27667`	103.169	89.650	60.724	**55.878**	56.320
`cwi_33949_165318`	1099.680	592.355	9102.699	**254.545**	334.751
Geometric Mean	5.518	4.759	3.483	4.183	**3.236**

as the algorithm is running, and in particular the most downstream nodes (which we need process first) are the last to be discovered. However, it would be interesting to consider whether either of the algorithms could incorporate some ideas from the other.

The strong bisimulation algorithm in [DPP04] is an improvement of the Paige-Tarjan algorithm that uses set-theoretic *rank functions* to select splitters more intelligently. By doing so they are able to achieve a two-fold speedup over the Paige-Tarjan algorithm, but more interestingly, their algorithm uses an initial partition that does not require the entire graph to be in memory at the same time. This enables scaling to larger systems, as well as opening up some potential for distributing the work, and it would be interesting to see if some of their ideas could be incorporated into Change-Tracking Iterative Refinement.

In [BvdP09], Blom and van de Pol introduce the concept of *inductive signatures* which allow the *afters* under certain events to be coloured with respect to the *current* partition instead of the *previous* one. This allows for faster convergence, but requires a *well-founded* subset of the alphabet (such that the transition set restricted to these events has no cycles); they achieve this for branching bisimulation by removing τ loops and using $\{\tau\}$ as the subset. Since this is also possible for DRDB and DRWB, it would be interesting to investigate, though it would require a data structure that allows states to be reclassified before their entire block's *coloured afters* have been recomputed (for example, a hash table).

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 *coloured afters* computations and parallel sorts of a number of *coloured afters* lists of arbitrary size. The sort phases can be sped up by sorting partitions in parallel or using multi-threaded sorting algorithms, and in the case of strong bisimulation the *coloured afters* computation phase can be parallelised as well. We have already developed parallel extensions of some of the algorithms detailed in this paper, though our implementations are still in the preliminary stages. The parallelisation of the *coloured afters* computation has the nice property that the transition set can be partitioned across workers and only the node to equivalence class map needs to be shared. This could allow for an efficient GPU implementation.

Acknowledgements. We would like to thank Michael Goldsmith, James Worrell, and the anonymous reviewers for their useful comments. Research into FDR3 has been partially sponsored by DARPA under agreement number FA8750-12-2-0247.

References

[AGL+12] Philip Armstrong, Michael Goldsmith, Gavin Lowe, Joël Ouaknine, Hristina Palikareva, A.W. Roscoe, and James Worrell. Recent developments in FDR. In *Computer Aided Verification*, pages 699–704. Springer, 2012.

[BGRR14] Alexandre Boulgakov, Thomas Gibson-Robinson, and AW Roscoe. Computing maximal bisimulations. In *Formal Methods and Software Engineering*, pages 11–26. Springer, 2014.

[BO02] Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Electronic Notes in Theoretical Computer Science*, 68(4):523–538, 2002.

[BO05] Stefan Blom and Simona Orzan. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[BvdP09] Stefan Blom and Jaco van de Pol. Distributed branching bisimulation minimization by inductive signatures. *arXiv preprint arXiv:0912.2550*, 2009.

[DPP04] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1):221–256, 2004.

[Fer90] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2):219–236, 1990.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[GRABR15] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3: A parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[GV90] Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In Michael S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Berlin Heidelberg, 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[KS83] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 228–240, New York, NY, USA, 1983. ACM.

[Mat05] Radu Mateescu. On-the-fly state space reductions for weak equivalences. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 80–89. ACM, 2005.

[Mil81] Robin Milner. A modal characterisation of observable machine-behaviour. In *CAAP'81*, pages 25–34. Springer, 1981.

[Par81] David Park. *Concurrency and automata on infinite sequences.* Springer, 1981.

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

[PU96] ICC Phillips and Irek Ulidowski. Ordered SOS rules and weak bisimulation. *Theory and Formal Methods*, 1996.

[RGG+95] A. W. Roscoe, P.H.B. Gardiner, M.H. Goldsmith, J.R. Hulance, D.M.Jackson, and J.B. Scattergood. Hierarchical compression for model-checking CSP, or How to check 10^{20} dining philosophers for deadlock. In *Proceedings of TACAS 1995.* BRICS, 1995.

[Ros94] A. W. Roscoe. Model-Checking CSP. *A Classical Mind: Essays in Honour of CAR Hoare*, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* 1998.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[San96] Davide Sangiorgi. A theory of bisimulation for the π-calculus. *Acta informatica*, 33(1):69–97, 1996.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[Tar76] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

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

[WHH+06] Ralf Wimmer, Marc Herbstritt, Holger Hermanns, Kelley Strampp, and Bernd Becker. Sigref–a symbolic bisimulation tool box. In *Automated Technology for Verification and Analysis*, pages 477–492. Springer, 2006.