Compositional Verification of Priority Systems Using Sharp Bisimulation

Luca Di Stefano[1,2*] and Frédéric Lang[1*]

[1]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP[†], LIG, 38000 Grenoble, France.
[2]University of Gothenburg, Gothenburg, Sweden.

*Corresponding author(s). E-mail(s): Frederic.Lang@inria.fr; luca.di.stefano@gu.se;

Abstract

Sharp bisimulation is a refinement of branching bisimulation, parameterized by a subset of the system's actions, called strong actions. This parameterization allows the sharp bisimulation to be tailored by the property under verification, whichever property of the modal μ-calculus is considered, while potentially reducing more than strong bisimulation. Sharp bisimulation equivalence is a congruence for process algebraic operators such as parallel composition, hide, cut, and rename, and hence can be used in a compositional verification setting. In this paper, we prove that sharp bisimulation equivalence is also a congruence for action priority operators under some conditions on strong actions. We compare sharp bisimulation with orthogonal bisimulation, whose equivalence is also a congruence for action priority. We show that, if the internal action τ neither gives priority to nor takes priority over other actions, then the quotient of a system with respect to sharp bisimulation equivalence (called sharp minimization) cannot be larger than the quotient of the same system with respect to orthogonal bisimulation equivalence. We then describe a signature-based partition refinement algorithm for sharp minimization, implemented in the BCG_MIN and BCG_CMP tools of the CADP software toolbox. This algorithm can be adapted to implement orthogonal minimization. We show on a crafted example that using compositional sharp minimization may yield state space reductions that outperform compositional orthogonal minimization by

[†]Institute of Engineering Univ. Grenoble Alpes

several orders of magnitude. Finally, we illustrate the use of sharp minimization and priority to verify a bully leader election algorithm.

Keywords: Concurrency, Congruence, Enumerative verification, Finite state processes, State space reduction

1 Introduction

We consider the verification of systems consisting of parallel processes that execute asynchronously and that may synchronize by rendezvous. The toolbox CADP[1] [19] provides languages for the description of such systems (in particular the language LNT[2] [9], which inherits many features from process algebra), languages for the description of properties (in particular the temporal logic MCL[3] [33], which is an extension of the modal μ-calculus with regular-expressions on actions and data handling constructs), and various tools for generating and minimizing state spaces, composing state spaces, evaluating temporal logic properties on state spaces, generating tests, etc. CADP provides various verification methods, some of the most successful of which are based on a compositional minimization of the system using appropriate equivalence relations [34, 18, 31, 32].

As such, *sharp bisimulation* [32] was proposed recently as a relation parameterized by a subset of the system's actions, called *strong actions*. These actions are generally obtained by analysis of the property under verification, ensuring that this property is preserved by equivalence. *Strong bisimulation* [36] and *branching bisimulation* [21, 22] are instances of sharp bisimulation, all actions being strong in the former, and all actions being *weak* (i.e., not strong) in the latter. Keeping the set of strong actions as small as possible allows smaller quotients to be obtained. Sharp bisimulation is a congruence for many process algebraic operators, such as parallel composition, hide, cut, and rename, and can therefore be used compositionally. A *divergence-preserving* variant of sharp bisimulation (named *divsharp bisimulation* in the current paper) was applied successfully to the CTL verification problems of the RERS verification challenges[4] in 2019[5] and 2020,[6] where it allowed properties not preserved by divergence-preserving branching (a.k.a. *divbranching*) bisimulation to be verified on systems whose state space could not be generated using compositional strong bisimulation reduction.

In this paper, we present contributions that are complementary to the already published results on sharp bisimulation [32]:

[1] http://cadp.inria.fr
[2] LNT stands for *LOTOS New Technology*.
[3] MCL stands for *Model Checking Language*.
[4] http://rers-challenge.org
[5] http://cadp.inria.fr/news12.html
[6] http://cadp.inria.fr/news13.html#section-3

- We consider systems, some actions of which may be preempted by others using an action priority operator, similar in spirit to the one of ACP [2]. It is well-known that strong bisimulation equivalence is a congruence for such an operator, but also that branching is not. We give sufficient conditions relating the set of strong actions and the priority rules, so that sharp bisimulation equivalence is a congruence for priority. We provide a proof of congruence.
- We establish a precise relationship between sharp bisimulation and *orthogonal bisimulation* [3], the latter being also a congruence for priority. We prove that if the internal action τ does neither give priority to nor take priority over any other action, then sharp minimization can be applied compositionally and cannot reduce less than orthogonal minimization. We exhibit an example where sharp minimization produces an LTS that is several orders of magnitude smaller than the one obtained using orthogonal minimization.
- While previous work only described a partial reduction algorithm preserving sharp bisimulation, we provide an algorithm based on signatures, which computes the quotient of a system with respect to sharp bisimulation equivalence. We compare the performance of its implementation in the BCG_MIN tool with strong and divbranching minimizations.
- Finally, we describe an application of compositional sharp minimization to the verification of a collective adaptive system, which uses action priority.

This paper is not only a theoretical paper. It combines theory with implementation in software tools distributed in CADP, a widely used toolbox, which has been continuously maintained, improved, and extended over the past three decades.

Overview.

Processes and their compositions, sharp bisimulation, and the priority operator are presented in Section 2. Congruence of sharp bisimulation equivalence for the priority operator, which depends on strong actions, is proven in Section 3. A comparison between sharp bisimulation and orthogonal bisimulation is made in Section 4. The signature-based sharp minimization algorithm is presented in Section 5, where we also describe how it can be adapted to implement orthogonal minimization. A toy example illustrating the potential benefit of using sharp bisimulation rather than orthogonal bisimulation is presented in Section 6. The case study is presented in Section 7. We discuss related work in Section 8. Finally, we conclude in Section 9.

Data availability statement.

The datasets generated and analysed during the current study, as well as the scripts used to define the experiments, are available as a replication package at https://doi.org/10.5281/zenodo.7602897.

2 Background

2.1 Processes

We consider systems of processes whose behavioural semantics can be represented using an LTS (*Labelled Transition System*).

Definition 1 (LTS)**.** *Let $\mathcal{A}$ be an infinite set of actions including the invisible action τ and visible actions $\mathcal{A} \setminus \{\tau\}$. An LTS P is a tuple $(\Sigma, A, \longrightarrow, p_{init})$, where:*

- *Σ is a set of states,*
- *$A \subseteq \mathcal{A}$ is a set of actions,*
- *$\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the (labelled) transition relation, and*
- *$p_{init} \in \Sigma$ is the initial state.*

We may write $\Sigma_P, A_P, \longrightarrow_P$ for the sets of states, actions, and transitions of an LTS P, and $init(P)$ for its initial state. We write $p \xrightarrow{a} p'$ for $(p, a, p') \in \longrightarrow$ and $p \xrightarrow{A}$ for $(\exists p' \in \Sigma_P, a \in A)\ p \xrightarrow{a} p'$.

To build complex behaviours, LTSs can be composed and abstracted using standard process algebraic operators, such as parallel composition, hide, cut, and rename (see e.g., [32]). In this paper, we will use the following parallel composition operator.

Definition 2 (Parallel composition of LTSs)**.** *Let P and Q be LTSs and let $A \subseteq \mathcal{A} \setminus \{\tau\}$ be a set of visible actions. The parallel composition of P and Q with synchronization on A, written "$P \,|[A]|\, Q$", is defined as the LTS $(\Sigma_P \times \Sigma_Q, A_P \cup A_Q, \longrightarrow, (init(P), init(Q)))$, where $(p, q) \xrightarrow{a} (p', q')$ iff:*

1. *$p \xrightarrow{a}_P p'$, $q' = q$, and $a \notin A$, or*
2. *$p' = p$, $q \xrightarrow{a}_Q q'$, and $a \notin A$, or*
3. *$p \xrightarrow{a}_P p'$, $q \xrightarrow{a}_Q q'$, and $a \in A$.*

2.2 Sharp Bisimulation

LTSs can be compared and reduced modulo bisimulation relations between states. We consider the family of sharp bisimulations as well as their divergence-preserving variants [32] defined hereafter in a general form subsuming strong [36], branching, and divbranching [21, 22] bisimulations.

Definition 3 (Sharp bisimulation [32])**.** *A sharp bisimulation[7] w.r.t. a set of strong actions A_s is a symmetric relation $R \subseteq \Sigma \times \Sigma$ such that if $(p, q) \in R$*

[7] Note that sharp bisimulation as defined in [32] corresponds to divsharp bisimulation in the current paper. We prefer this naming, which is more in line with the distinction between branching and divbranching bisimulations.

then for all $p \xrightarrow{a} p'$*, there exists* q' *such that* $(p', q') \in R$ *and either of the following holds:*

1. $q \xrightarrow{a} q'$*, or*
2. $a = \tau$*,* $\tau \notin A_s$*, and* $q' = q$*, or*
3. $a \notin A_s$*, and there exist* $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \xrightarrow{a} q'$ $(n \geq 0)$ *such that* $q_0 = q$*, and for all* $i \in 1..n$*,* $(p, q_i) \in R$*.*

A divergence-preserving sharp bisimulation (called divsharp bisimulation in the sequel) R *additionally satisfies the following divergence-preservation condition: for all* $(p_0, q_0) \in R$ *such that* $p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \ldots$ *with* $(p_i, q_0) \in R$ *for all* $i \geq 0$*, there is also an infinite sequence* $q_0 \xrightarrow{\tau} q_1 \xrightarrow{\tau} q_2 \xrightarrow{\tau} \ldots$ *such that* $(p_i, q_j) \in R$ *for all* $i, j \geq 0$*.*

States p, q *are sharp (resp. divsharp) bisimilar w.r.t.* A_s*, written* $p \sim_{\sharp A_s} q$ *(resp.* $p \sim^{@}_{\sharp A_s} q$*), iff there exists a sharp (resp. divsharp) bisimulation* R *w.r.t.* A_s *such that* $(p, q) \in R$*.*

A sharp bisimulation is a hybrid between a strong and a branching bisimulation, characterized by the set of strong actions. If a state has a transition labelled by a strong action, every sharp bisimilar state must also have a similar transition labelled by the same strong action. This transition cannot be delayed by a transition labelled by τ (or τ-transition). By contrast, a transition labelled by a weak action can be delayed under the same condition as branching bisimulation.

Sharp bisimulation generalizes results that are well-known in the framework of strong and (div)branching bisimulations [32]. For all sets of strong actions A_s and A'_s:

- $\sim_{\sharp A_s}$ (resp. $\sim^{@}_{\sharp A_s}$) is an equivalence relation. The quotient of an LTS P w.r.t. this equivalence relation (written $P/\sim_{\sharp A_s}$) is unique and minimal both in number of states and number of transitions. Computing this quotient is called minimization.
- If $A_s = \mathcal{A}$ (i.e., all actions are strong, including τ), then sharp bisimulation and divsharp bisimulation coincide with strong bisimulation, whose equivalence is written $\sim$.
- If $A_s = \emptyset$ (i.e., all actions are weak), then sharp (resp. divsharp) bisimulation coincides with branching (resp. divbranching) bisimulation, whose equivalence is written $\sim_{br}$ (resp. $\sim_{dbr}$).
- The set of sharp (resp. divsharp) bisimulation equivalences equipped with set inclusion actually forms a complete lattice whose supremum is branching (resp. divbranching) bisimulation equivalence and whose infimum is strong bisimulation equivalence.
- If $A'_s \subset A_s$ then $\sim_{\sharp A_s}$ (resp. $\sim^{@}_{\sharp A_s}$) is strictly stronger than $\sim_{\sharp A'_s}$ (resp. $\sim^{@}_{\sharp A'_s}$), hence generalizing that strong bisimulation equivalence is strictly stronger than (div)branching. As a consequence, an LTS minimized for $\sim_{\sharp A'_s}$ (resp. $\sim^{@}_{\sharp A'_s}$) cannot be larger than the same LTS minimized for $\sim_{\sharp A_s}$ (resp. $\sim^{@}_{\sharp A_s}$), i.e., the fewer strong actions, the greater the reduction.

- $\sim_{\sharp A_s}$ (resp. $\sim^{@}_{\sharp A_s}$) is a congruence for all abstraction and composition operators encodable as *admissible networks of LTSs* [38, 18, 30], such as synchronization vectors and the parallel composition, hide, cut, and rename operators of CCS [35], CSP [7], mCRL [24], LOTOS [26], E-LOTOS [27], and LNT [9], including the operator of Definition 2. Minimization can thus be applied compositionally through these operators. Note that, just like (div)branching, $\sim_{\sharp A_s}$ and $\sim^{@}_{\sharp A_s}$ are not congruences for the nondeterministic choice operator (written "**+**" in CCS and mCRL, "**[ ]**" in CSP and LOTOS, and **select** in LNT), unless all initial actions of the nondeterministic choice are strong actions. In particular, strong bisimulation equivalence is a congruence for the nondeterministic choice operator, because all actions are strong.
- Finally, $\sim^{@}_{\sharp A_s}$ is adequate with a well-characterized fragment $L_{\mu}^{strong}(A_s)$ of the modal μ-calculus,[8] also parameterized by A_s. The union indexed by A_s of all such fragments is the modal μ-calculus itself, so that every formula belongs to some of these fragments. Thus, identifying a small enough A_s to which a temporal logic formula belongs[9] provides more chances for the system to be reduced efficiently. See [31, 32] for details.

2.3 Action Priority

A classification of priority operators in process algebra is given in [10]. With respect to this classification, the operator presented in this section is static prioritized choice with global preemption. This priority operator is implemented in the tool EXP.OPEN [30] distributed in the CADP toolbox. Strong and orthogonal bisimulation equivalences [3] are congruences for this operator, which has lots of similarities with the priority operator of ACP [2].

Definition 4. *We consider action formulas α, whose syntax and semantics w.r.t. a set of actions A are defined as follows:*

$$\begin{array}{rll} \alpha ::= & a & [\![a]\!]_A = \{a\} \\ | & \mathsf{false} & [\![\mathsf{false}]\!]_A = \emptyset \\ | & \alpha_1 \vee \alpha_2 & [\![\alpha_1 \vee \alpha_2]\!]_A = [\![\alpha_1]\!]_A \cup [\![\alpha_2]\!]_A \\ | & \neg\alpha_0 & [\![\neg\alpha_0]\!]_A = A \setminus [\![\alpha_0]\!]_A \end{array}$$

Operators true *and* $\wedge$ *are derived from the above using equations* $\mathsf{true} = \neg\mathsf{false}$ *and* $\alpha_1 \wedge \alpha_2 = \neg(\neg\alpha_1 \vee \neg\alpha_2)$.

Definition 5. *An action priority rule π is an ordered pair of the form $\alpha \succ \alpha'$, where α and α' are action formulas such that $[\![\alpha]\!]_A \neq \emptyset$, $[\![\alpha']\!]_A \neq \emptyset$, and $[\![\alpha]\!]_A \cap [\![\alpha']\!]_A = \emptyset$. To an action priority rule $\pi = \alpha \succ \alpha'$, we associate the*

[8] Adequacy of an equivalence and a logic means that two LTSs are equivalent iff they verify exactly the same formulas of the logic.

[9] It was shown that such a smallest fragment is not unique [31]. Also, given A_s, deciding whether a formula belongs to $L_{\mu}^{strong}(A_s)$ is still an open problem.

following antisymmetric relation $\succ_{\pi,A}$*:*

$$\succ_{\pi,A} = \{(a, a') \mid a \in [\![\alpha]\!]_A \wedge a' \in [\![\alpha']\!]_A\}$$

Given $\pi = \alpha \succ \alpha'$*, we write* $\mathrm{greater}_A(\pi)$ *for the set* $[\![\alpha]\!]_A$ *of labels of* A *that take priority over some other labels, and* $\mathrm{lesser}_A(\pi)$ *for the set* $[\![\alpha']\!]_A$ *of labels of* A *that give priority to some other label, following rule* π*.*

A priority set Ω *is a set* $\{\pi_1, \ldots, \pi_n\}$ *of priority rules. To a priority set* Ω*, we associate the following relation* $\succ_{\Omega,A}$*:*

$$\succ_{\Omega,A} = \succ_{\pi_1,A} \cup \ldots \cup \succ_{\pi_n,A}$$

We write $>_{\Omega,A}$ *for the transitive closure of* $\succ_{\Omega,A}$*. A priority set* Ω *is valid if* $>_{\Omega,A}$ *is a strict (partial) order relation, i.e., antireflexive and antisymmetric (and of course transitive, which is granted by definition).*

If Ω *is a valid priority set and* P *is the LTS* $(\Sigma, A, \longrightarrow, p_{init})$*, then the expression "****prio*** Ω ***in*** P*" is defined as the LTS* $(\Sigma, A, \longrightarrow', p_{init})$ *such that* $\longrightarrow'$ *is the following set of transitions:*

$$\longrightarrow' = \{(p, a, p') \mid p \xrightarrow{a} p' \wedge (\forall p'' \in \Sigma, a' \in A)\ p \xrightarrow{a'} p'' \Rightarrow \neg(a' >_{\Omega,A} a)\}$$

Finally, we extend the operations $\mathrm{greater}_A$ *and* lesser_A *to priority sets as follows:*

$$\mathrm{greater}_A(\Omega) = \bigcup_{\pi_i \in \Omega} \mathrm{greater}_A(\pi_i)$$
$$\mathrm{lesser}_A(\Omega) = \bigcup_{\pi_i \in \Omega} \mathrm{lesser}_A(\pi_i)$$

Example 1. *Let* A *be the set of actions* $\{a, b, c, d, e, f\}$ *and* Ω *be the priority set* $\{a \succ b, b \succ c \vee d, d \succ \neg(a \vee b \vee c \vee d \vee e)\}$*, denoting that (using transitivity):* a *takes priority over* b*,* c*,* d*, and* f*;* b *takes priority over* c*,* d*, and* f*; and* d *takes priority over* f*. We have* $\mathrm{greater}_A(\Omega) = \{a, b, d\}$*, and* $\mathrm{lesser}_A(\Omega) = \{b, c, d, f\}$*. This example illustrates that* $\mathrm{greater}_A(\Omega)$ *and* $\mathrm{lesser}_A(\Omega)$ *are not necessarily disjoint.*

Lemma 1. *The following propositions hold:*

- $a \in \mathrm{greater}_A(\Omega)$ *iff* $a \in A$ *and there exists* $a' \in A$ *such that* $a >_{\Omega,A} a'$*.*
- $a \in \mathrm{lesser}_A(\Omega)$ *iff* $a \in A$ *and there exists* $a' \in A$ *such that* $a' >_{\Omega,A} a$*.*

Proof Immediate from the definitions of $\mathrm{greater}_A(\Omega)$ and $\mathrm{lesser}_A(\Omega)$. □

3 Sharp Congruences for Action Priority

It is well-known that, although strong bisimulation equivalence is a congruence for priority, (div)branching bisimulation equivalence is not. This latter fact can be illustrated by the following example.

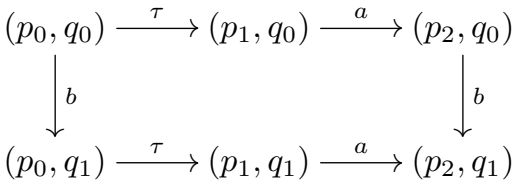


Fig. 1 LTS of Example 2

Example 2. *Consider the network "**prio** $a \succ b$ **in** $P \,|[\emptyset]|\, Q$", where a and b are visible actions, P is the sequence $p_0 \xrightarrow{\tau} p_1 \xrightarrow{a} p_2$ and Q consists of the single transition $q_0 \xrightarrow{b} q_1$. The LTS corresponding to this network is depicted in Figure 1. Note that there is no transition labelled by b from (p_1, q_0) to (p_1, q_1) because the outgoing transition labelled by a from (p_1, q_0) has priority. P is branching equivalent to P', which consists of the single transition $p'_0 \xrightarrow{a} p'_1$. The LTS corresponding to "**prio** $a \succ b$ in $P' \,|[\emptyset]|\, Q$" is the sequence of transitions $(p'_0, q_0) \xrightarrow{a} (p'_1, q_0) \xrightarrow{b} (p'_1, q_1)$. It is clear that (p_0, q_0), from which actions a and b can be fired in any order, is not branching equivalent to (p'_0, q_0), from which a can only be fired before b.*

Theorem 1 below expresses however that (div)sharp bisimulation equivalence is a congruence for priority, provided (1) all actions that may take priority over some other action are strong and (2) τ does not give priority to any other action.

Theorem 1. *If* $\text{greater}_A(\Omega) \subseteq A_s$, $\tau \notin \text{lesser}_A(\Omega)$, *and* $P \sim_{\sharp A_s} P'$, *then* ***prio*** Ω ***in*** $P \sim_{\sharp A_s}$ ***prio*** Ω ***in*** P'. *This also holds when replacing* $\sim_{\sharp A_s}$ *by* $\sim^{@}_{\sharp A_s}$.

Proof The key of the proof is that a state has a prioritized transition iff every $\sim_{\sharp A_s}$ bisimilar state also has a similar transition with same label, as this label belongs to A_s, and that τ-transitions cannot be cut by transitions with a higher priority.

Formally, let R be a (div)sharp bisimulation w.r.t. A_s between P and P'. We show that R is also a (div)sharp bisimulation w.r.t. A_s between "**prio** Ω **in** P" and "**prio** Ω **in** P'". Let $(p_0, p'_0) \in R$ and assume that there exists p_1 such that "**prio** Ω **in** P" has a transition $p_0 \xrightarrow{a} p_1$. By definition of **prio**, P also has this transition.

For all labels a' such that $a' >_{\Omega,A} a$, we have $a' \in \mathit{greater}_A(\Omega)$, which implies $a' \in A_s$ since $\text{greater}_A(\Omega) \subseteq A_s$. By definition of **prio**, since "**prio** Ω **in** P" has a transition $p_0 \xrightarrow{a} p_1$, then P has no transition of the form $p_0 \xrightarrow{a'} p_2$ (otherwise this transition would take priority over $p_0 \xrightarrow{a} p_1$, which would then not occur in "**prio** Ω **in** P"). By definition of R, since $(p_0, p'_0) \in R$ and $a' \in A_s$, P' thus has no transition of the form $p'_0 \xrightarrow{a'} p'_2$ either.

By definition of (div)sharp bisimulation, since P has a transition $p_0 \xrightarrow{a} p_1$, and $(p_0, p'_0) \in R$, then P' has a state p'_1 such that one of the following three conditions is satisfied in P'; in each case, we show that the same condition is satisfied in "**prio** Ω **in** P'":

1. P' has a transition $p'_0 \xrightarrow{a} p'_1$. Since P' has no transition of the form $p'_0 \xrightarrow{a'} p'_2$ with $a' >_{\Omega,A} a$, then "**prio** Ω **in** P'" does also have the same transition $p'_0 \xrightarrow{a} p'_1$.
2. $a = \tau$, $\tau \notin A_s$, and $p'_1 = p'_0$. This condition obviously keeps holding true in "**prio** Ω **in** P'".
3. $a \notin A_s$ and P' has a sequence $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \xrightarrow{a} p'_1$ $(n \geq 0)$ such that $q_0 = p'_0$ and for all $i \in 1..n$, $(p_0, q_i) \in R$. Since $\tau \notin \text{lesser}_A(\Omega)$, no label can take priority over τ-transitions, hence "**prio** Ω **in** P'" has the same sequence of τ-transitions $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n$ with $q_0 = p'_0$ and for all $i \in 1..n$, $(p_0, q_i) \in R$. In particular, $(p_0, q_n) \in R$, which implies that, since P has no transition of the form $p_0 \xrightarrow{a'} p_2$ with $a' >_{\Omega,A} a$, then P' has no transition of the form $q_n \xrightarrow{a'} p'_2$ with $a' >_{\Omega,A} a$ either. "**prio** Ω **in** P'" thus has the transition $q_n \xrightarrow{a} p'_1$.

Finally, if R is a divsharp bisimulation and if p_0 has a divergence in P, then this divergence is also present in "**prio** Ω **in** P", because $\tau \notin \text{lesser}_A(\Omega)$. Therefore, p'_0 also has a divergence in P' because $(p_0, p'_0) \in R$, and also in "**prio** Ω **in** P'", again because $\tau \notin \text{lesser}_A(\Omega)$. □

Example 3. *Back to Example 2, we know from Theorem 1 that if we replace respectively P and Q by any sharp equivalent LTSs, then we will get as the result an LTS that is sharp equivalent —hence branching equivalent— to the network "**prio** $a \succ b$ in $P \,|[\emptyset]|\, Q$", provided a is a strong action. Note however that both P and Q are minimal for $\sim_{\sharp\{a\}}$. They are therefore also minimal for any sharp bisimulation equivalence where a is a strong action.*

Note that the proof of congruence of strong bisimulation for priority can be derived from the above proof by replacing the condition $\text{greater}_A(\Omega) \subseteq A_s$ by $A_s = \mathcal{A}$ (the condition under which (div)sharp bisimulation coincides with strong bisimulation), which makes items 2 and 3 inapplicable and relaxes the (now useless) assumption $\tau \notin \text{lesser}_A(\Omega)$.

4 Sharp vs. Orthogonal Bisimulation

Sharp bisimulation has resemblances with orthogonal bisimulation [3], whose equivalence is also a congruence for many operators including choice and action priority, even if τ is subject to priority. We show the precise relationship between orthogonal and sharp bisimulations.

Definition 6 (Orthogonal bisimulation [3])**.** *An orthogonal bisimulation is a symmetric relation $R \subseteq \Sigma \times \Sigma$ such that if $(p, q) \in R$ then for all $p \xrightarrow{a} p'$, there exists q' such that $(p', q') \in R$ and either of the following holds:*

1. $a \neq \tau$ and $q \xrightarrow{a} q'$, or

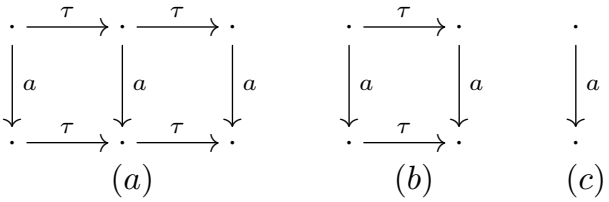


Fig. 2 Three LTSs of Example 4

2. $a = \tau$ *and* $q \xrightarrow{\tau}$ *and there exists a sequence of transitions* $q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n$ $(n \geq 0)$ *such that* $q_0 = q$, $q_n = q'$, *and for all* $i \in 1..n-1$, $(p, q_i) \in R$.

Two states p *and* q *are orthogonally bisimilar, written* $p \sim_{\perp} q$, *iff there exists an orthogonal bisimulation* R *such that* $(p, q) \in R$.

Divergence-preserving orthogonal bisimulation, written $\sim_{\perp}^{@}$, *is also defined in [3], by adding the same divergence-preserving condition as divbranching and divsharp bisimulations (see Definition 3).*

The above definition shows that orthogonal bisimulation is stronger than branching bisimulation and weaker than strong bisimulation, with which it coincides if all actions are visible. As regards τ-transitions, orthogonal bisimulation may compress their sequences, but always preserves at least one. It is not a sharp bisimulation, as shown in the following example.

Example 4. *Consider the three LTSs displayed in Figure 2. Orthogonal minimization of LTS* (a) *results in LTS* (b). *However, sharp minimization of LTS* (a) *results in LTS* (c) *when* $\tau \notin A_s$, *and in LTS* (a) *itself when* $\tau \in A_s$. *Interestingly, in our framework, LTS* (a) *can be reduced to LTS* (c) *if* τ *does neither give nor take priority to any other label. By contrast, orthogonal bisimulation enables only reduction to LTS* (b) *in all cases, which has twice as many states and four times more transitions than LTS* (c).

Theorem 2 gives the precise relationship between orthogonal and sharp bisimulations.

Theorem 2. *A relation* $R \subseteq \Sigma \times \Sigma$ *is an orthogonal bisimulation (resp. divergence-preserving orthogonal bisimulation) iff (1)* $R \in \sim_{\sharp\mathcal{A}\setminus\{\tau\}}$ *(resp.* $\sim_{\sharp\mathcal{A}\setminus\{\tau\}}^{@}$*) and (2) for all* $(p, q) \in R$, $p \xrightarrow{\tau}$ *implies* $q \xrightarrow{\tau}$.

Proof This is quite intuitive by looking closely at the definitions of sharp bisimulation and orthogonal bisimulation. A formal equivalence proof in eight steps follows (we skip divergence-preservation, which is obvious). To help reading, we decorate each step of the proof by putting in shaded background the parts that will change (i.e., that will be either modified, moved, or removed) in the next step, and by putting in boxes the parts that changed in the current step.

$$R \in \sim_{\sharp\mathcal{A}\setminus\{\tau\}} \wedge (\forall (p, q) \in R)\ (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge (q \xrightarrow{a} q' \vee (a = \tau \wedge \tau \notin \mathcal{A} \setminus \{\tau\} \wedge q' = q) \vee (a \notin \mathcal{A} \setminus \{\tau\} \wedge (\exists n \geq 0, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \xrightarrow{a} q' \wedge q_0 = q \wedge (\forall i \in 1..n)\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

by definition of $\sim_{\sharp \mathcal{A} \setminus \{\tau\}}$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge (q \xrightarrow{a} q' \vee (a = \tau \wedge q' = q) \vee (\boxed{a = \tau} \wedge (\exists n \geq 0, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \xrightarrow{\boxed{\tau}} q' \wedge q_0 = q \wedge (\forall i \in 1..n)\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

because $\tau \notin \mathcal{A} \setminus \{\tau\}$ is trivial and $a \notin \mathcal{A} \setminus \{\tau\}$ iff $a = \tau$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge (q \xrightarrow{a} q' \vee (\boxed{a = \tau} \wedge (q' = q \vee ((\exists\ n \geq 0, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \xrightarrow{\tau} q' \wedge q_0 = q \wedge (\forall\ i \in 1..n)\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

by factoring $a = \tau$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge (q \xrightarrow{a} q' \vee (a = \tau \wedge (\ q' = q \vee ((\exists \boxed{\ n > 0\ }, q_0, \ldots, q_n \in \Sigma)\ \boxed{q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n} \wedge q_0 = q \wedge \boxed{q_n = q'} \wedge (\forall \boxed{i \in 1..n-1}\ )\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

by reworking indices (q_n becomes q_{n-1}, q' becomes q_n)

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge (\ q \xrightarrow{a} q'\ \vee (a = \tau \wedge ((\exists \boxed{\ n \geq 0\ }, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \wedge q_0 = q \wedge q_n = q' \wedge (\forall i \in 1..n-1)\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

because $q' = q$ is subsumed by case $n = 0$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge ((\boxed{a \neq \tau \wedge q \xrightarrow{a} q'}) \vee (a = \tau \wedge ((\exists n \geq 0, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \wedge q_0 = q \wedge q_n = q' \wedge (\forall i \in 1..n-1)\ (p, q_i) \in R))) \wedge (q,p) \in R \wedge (p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau})$

because $q \xrightarrow{\tau} q'$ is subsumed by case $n = 1$

$\Leftrightarrow$ $(\forall (p,q) \in R, a \in A, p' \in \Sigma)\ (p \xrightarrow{a} p' \Rightarrow (\exists q' \in \Sigma)\ (p',q') \in R \wedge ((a \neq \tau \wedge q \xrightarrow{a} q') \vee (a = \tau \wedge \boxed{q \xrightarrow{\tau}} \wedge\ ((\exists n \geq 0, q_0, \ldots, q_n \in \Sigma)\ q_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} q_n \wedge q_0 = q \wedge q_n = q' \wedge (\forall i \in 1..n-1)\ (p, q_i) \in R))) \wedge (q,p) \in R$

by moving $q \xrightarrow{\tau}$ under the assumption $a = \tau$

$\Leftrightarrow$ $R \in \sim_{\perp}$

by definition of $\sim_{\perp}$

□

As orthogonal bisimulation equivalence requires every visible action to be strong (only τ being weak), a consequence is that it is strictly stronger than any sharp bisimulation equivalence where τ is weak. In this case, the quotient of the system for sharp bisimulation equivalence cannot be larger than the one for orthogonal bisimulation equivalence. The fewer actions are declared as strong (i.e., the fewer actions being prioritized), the more sharp bisimulation can improve the reduction as compared to orthogonal bisimulation.

Equiv. R for system	$\tau \in A_<$?	$\tau \in A_>$?	$\tau \in A_s$?	Equiv. R' for P_i $(i \in 0..n)$
$\sim_{\sharp A_s}$	yes	–	yes	$\sim$
	yes	–	no	$\sim_\perp$
	no	yes	yes	$\sim_{\sharp A_s \cup A_>}$
	no	yes	no	$\sim_{\sharp A_s \cup A_>}, \sim_\perp$
	no	no	–	$\sim_{\sharp A_s \cup A_>}$
$\sim_\perp$	–	–	–	$\sim_\perp$

Table 1 Weakest equivalence relations R' (among the relations addressed in this paper), which can be used on the subsystems $P_0, \ldots, P_n$ for "**prio** Ω **in** $P_0 \,|[A_1]|\, \ldots \,|[A_n]|\, P_n$" to preserve the equivalence relation R ($A_< = \text{lesser}_A(\Omega)$ and $A_> = \text{greater}_A(\Omega)$).

A practical consequence of Theorems 1 and 2 is that if one needs to generate the LTS of a network of the form "**prio** Ω **in** $(P_0 \,|[A_1]|\, \ldots \,|[A_n]|\, P_n)$" in a way that preserves $\sim_{\sharp A_s}$ (e.g., because the property to be verified on the network is in the modal μ-calculus fragment $L_\mu^{strong}(A_s)$), then $P_0, \ldots, P_n$ can be reduced with respect to $\sim_{\sharp A_s \cup A_>}$ beforehand, where $A_> = \text{greater}_A(\Omega)$ and $A = A_{P_1} \cup \ldots \cup A_{P_n}$. Here, the set $A_s \cup A_>$ is optimal, in the sense that if we take any set $A_\subset$ strictly included in $A_s \cup A_>$ instead, we can always find instances of $\varphi \in L_\mu^{strong}(A_s)$, Ω, $P_0, \ldots, P_n$, and $A_1, \ldots, A_n$ such that "**prio** Ω **in** $P_0 \,|[A_1]|\, \ldots \,|[A_n]|\, P_n$" satisfies φ but "**prio** Ω **in** $P_0 / \sim_{\sharp A_\subset} |[A_1]|\, \ldots \,|[A_n]|\, P_n / \sim_{\sharp A_\subset}$" does not. However, for specific instances, it may happen that minimizing $P_0, \ldots, P_n$ with respect to $\sim_{\sharp A_\subset}$ will preserve φ; in particular, this is the case if $A_> \subseteq A_\subset$ and $\varphi \in L_\mu^{strong}(A_\subset \cap A_s)$ (which is a subset of $L_\mu^{strong}(A_s)$), or if minimizations with respectively $\sim_{\sharp A_s \cup A_>}$ and $\sim_{\sharp A_\subset}$ give the same result when applied to the LTS "$P_0 \,|[A_1]|\, \ldots \,|[A_n]|\, P_n$".

Note however that the principle exposed in the previous paragraph requires $\tau \notin \text{lesser}_A(\Omega)$. Otherwise, one has to use either strong bisimulation $\sim$ or orthogonal bisimulation $\sim_\perp$, for which the congruence property holds. Note however that $\sim_\perp$ can only be used in the case where $\tau \notin A_s$, because it cannot preserve $\sim_{\sharp\{\tau\}}$. Finally, if $\tau \notin \text{lesser}_A(\Omega)$ and $\tau \notin A_s$, then each P_i can be reduced with respect to both $\sim_\perp$ and $\sim_{\sharp A_s \cup A_>}$. If in addition $\tau \notin A_>$, then the former is subsumed by the latter (because it is stronger), but this is not the case if $\tau \in A_>$. In the latter case, it may be worth combining both equivalences.

These principles, which are summarized in Table 1, also hold when replacing $\sim_{\sharp A_s}$, $\sim_{\sharp A_s \cup A_>}$, and $\sim_\perp$ by their divergence-preserving variants $\sim^{@}_{\sharp A_s}$, $\sim^{@}_{\sharp A_s \cup A_>}$, and $\sim^{@}_\perp$, respectively. They could be implemented in a procedure that decides automatically which equivalence is to be used on the subsystems, e.g., in the SVL scripting language [16] of CADP.

Finally, as the congruence property for orthogonal bisimulation does not require the condition $\tau \notin \text{lesser}_A(\Omega)$ (unlike sharp bisimulation), one might wonder whether an equivalence relation $\sim_{\sharp A_s \perp}$ (and its divergence-preserving

variant), which combines sharp bisimulation and orthogonal bisimulation as given by the following definition:

$$R \in \sim_{\sharp A_s \perp} \Leftrightarrow \; R \in \sim_{\sharp A_s} \wedge \, (\forall (p,q) \in R) \; p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau}$$

would have the congruence property without requiring $\tau \notin \text{lesser}_A(\Omega)$. (Note that $\sim_\perp = \sim_{\sharp \mathcal{A} \setminus \{\tau\} \perp}$.) In other words, can we relax the definition of orthogonal bisimulation so as to consider a reduced set of strong actions instead of all visible actions, while preserving the congruence when τ may give priority to other actions? The answer is no, as shows the counter-example of Figure 3.

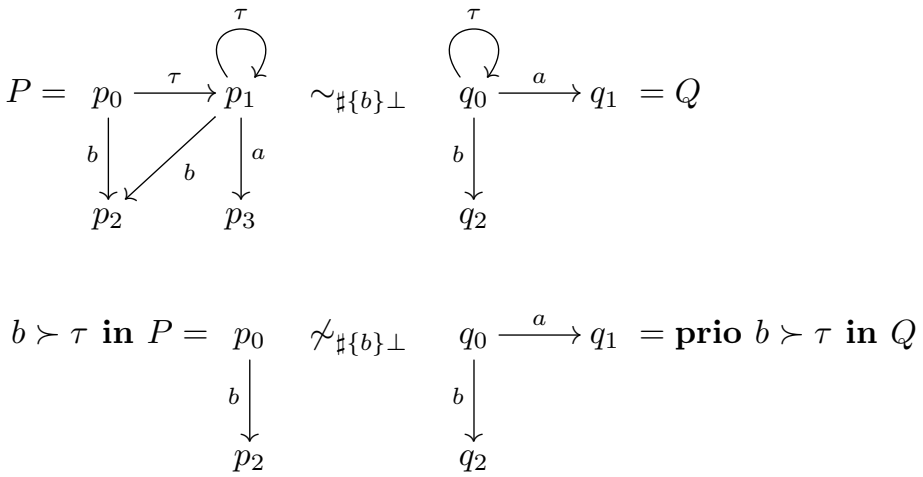


Fig. 3 Counter-example showing that combining sharp and orthogonal bisimulations does not allow the condition $\tau \notin \text{lesser}_A(\Omega)$ to be removed from Theorem 1.

5 Signature-Based Sharp Minimization

5.1 Partition-Refinement using Signatures

Efficient LTS minimization algorithms are generally based on partition refinement: starting from an initial partition[10] $\{\Sigma\}$ of the set of states Σ, refinement (i.e., block splitting) is applied until each block represents an equivalence class.

Signature-based partition refinement was first proposed by Blom & Orzan [4, 5, 6] as a conceptually simple way to implement partition refinement for strong and (div)branching bisimulations, among others. Although signature-based partition refinement algorithms are not optimal, with a worst-case time complexity of $O(mn^2)$ where m is the number of transitions and n is the number of states (to be compared with the best-known worst-case time complexity of $O(m \log n)$ for both strong [40] and (div)branching [28] bisimulations), they are quite efficient in practice and suitable for distributed

[10] A partition of a set S is a set of non-empty disjoint subsets of S (called blocks), whose union is S itself.

implementations[11]. In the CADP toolbox [19], the BCG_MIN tool implements sequential signature-based partition refinement for strong bisimulation, branching bisimulation, divbranching bisimulation, as well as stochastic and probabilistic extensions. Its companion tool BCG_CMP checks bisimulation equivalence between LTSs, by testing whether the initial states of both LTSs are in the same equivalence class. In this section, we present a signature-based partition refinement algorithm that implements sharp minimization and that is implemented in BCG_MIN and BCG_CMP.

We start from a generic signature-based partition refinement algorithm. For a bisimulation equivalence R, this algorithm uses a function $sig_R(s, P)$, which returns the signature of a state s with respect to the current partition P. The signature σ of a state s in the partition P is a set of ordered pairs $\sigma \in 2^{A \times P}$, each pair consisting of a label $a \in A$ and a block $B \in P$. Its precise definition for strong, (div)branching, and (div)sharp bisimulations will be detailed later. Function sig_R naturally extends to blocks B as follows:

$$sig_R(B, P) = \{sig_R(s, P) \mid s \in B\}$$

In a given partition P, a block B may be split into several blocks using the function $split_R(B, P)$ defined as follows:

$$split_R(B, P) = \{\{s \mid s \in B \wedge sig_R(s, P) = \sigma\} \mid \sigma \in sig_R(B, P)\}$$

When a block B cannot be split anymore, i.e., the signature is the same for all states of B, then $split_R(B, P)$ returns the singleton set $\{B\}$. The generic signature-based partition refinement algorithm can be defined as follows:

P := $\{\Sigma\}$;
loop P', P := P, $\bigcup_{B \in P} split_R(B, P)$ **until** $P = P'$ **end loop**;
return P

This algorithm can be refined to avoid unnecessary work. For instance, during an iteration, blocks can be partitioned into splitter and non-splitter blocks, so that signatures of blocks which are not predecessors of splitter blocks do not need to be split, as their signatures would remain unchanged. However, we do not detail such improvements further as they are not specific to sharp bisimulation.

[11] The difference between $O(m \log n)$ and $O(mn^2)$ may seem big, and it is, indeed. However, $O(mn^2)$ is the complexity upper bound, reached only in few corner cases (e.g., long sequences or large trees of transitions all labelled by the same action). Moreover, the bottleneck for state space reduction is usually memory rather than time. Even though they may be sensibly slower, signature-based partition refinement algorithms behave well in terms of memory consumption.

5.2 Computing Signatures

We first recall the signature definitions for strong and (div)branching bisimulations. We will then define signatures for (div)sharp bisimulation and explain how they can be computed in practice. To this aim, we introduce a few notations.

Definition 7. *We use the following notations, where a is a label, B a block, P a partition, s, s' are states, and n is a natural number:*

$$
\begin{aligned}
\tau_{B,P}(s,s') &= B \in P \wedge s,s' \in B \wedge s \xrightarrow{\tau} s' \\
\tau^n_{B,P}(s,s') &= (\exists s_0,\ldots,s_n)\ s = s_0 \wedge (\forall i \in 0..n-1)\ \tau_{B,P}(s_i,s_{i+1}) \wedge s_n = s' \\
\tau^*_{B,P}(s,s') &= (\exists n \geq 0)\ \tau^n_{B,P}(s,s') \\
\tau^\omega_{B,P}(s) &= (\forall n \geq 0)\ (\exists s' \in B)\ \tau^n_{B,P}(s,s') \\
\tau_P(s,s') &= (\exists B \in P)\ \tau_{B,P}(s,s') \\
\tau^n_P(s,s') &= (\exists B \in P)\ \tau^n_{B,P}(s,s') \\
\tau^*_P(s,s') &= (\exists B \in P)\ \tau^*_{B,P}(s,s') \\
\tau^\omega_P(s) &= (\exists B \in P)\ \tau^\omega_{B,P}(s) \\
s \xrightarrow{a}_P B &= B \in P \wedge (\exists s' \in B)\ s \xrightarrow{a} s' \\
s \stackrel{a}{\Longrightarrow}_P B &= B \in P \wedge (\exists s' \in \Sigma, s'' \in B)\ \tau^*_P(s,s') \wedge s' \xrightarrow{a} s'' \wedge (a \neq \tau \vee s \notin B)
\end{aligned}
$$

The meaning of these notations can be phrased as follows. A τ-transition is called *inert* w.r.t. a state partition P (or simply *inert* if P is clear from the context) if its source and target states belong to the same block. States s and s' belonging to the same block B of partition P, the notation $\tau_{B,P}(s,s')$ indicates that there is an inert transition from s to s', $\tau^n_{B,P}(s,s')$ that there is a sequence of n inert transitions from s to s', $\tau^*_{B,P}(s,s')$ that there is a sequence of (zero or more) inert transitions from s to s', and $\tau^\omega_{B,P}(s)$ that s is the source of an infinite sequence of inert transitions. The variants of these notations where B does not appear in the subscript have a similar meaning, except that the block to which s and s' belong is unspecified. The notation $s \xrightarrow{a}_P B$ indicates that s has a transition labelled by a to some state that belongs to block B of partition P and $s \stackrel{a}{\Longrightarrow}_P B$ that there is an arbitrary long sequence of inert transitions starting in s, which leads to a state that has a non-inert transition labelled by a to some state that belongs to block B of partition P.

The signatures of a state for respectively strong, branching, and divbranching bisimulations are defined as follows:

$$
\begin{aligned}
sig_{\sim}(s,P) &= \{(a,B) \mid s \xrightarrow{a}_P B\} \\
sig_{\sim_{br}}(s,P) &= \{(a,B) \mid s \stackrel{a}{\Longrightarrow}_P B\} \\
sig_{\sim_{dbr}}(s,P) &= sig_{\sim_{br}}(s,P) \cup \{(\tau,B) \mid \tau^\omega_{B,P}(s)\}
\end{aligned}
$$

In the case of (div)branching bisimulation, computing signatures efficiently requires care, due to the necessity to compute the relation $\stackrel{a}{\Longrightarrow}_P$, built upon the transitive closure of the relation τ_P. In particular, one has to take care of

cycles of τ-transitions. However, one can exploit the fact that all states in the same cycle of τ-transitions are (div)branching bisimilar. To this aim, we use a procedure SCC_τ, which takes as input a block B and returns an ordered list $(C_1, \ldots, C_n)$ such that:

- $C_1, \ldots, C_n$ are the SCCs (Strongly Connected Components) of the τ-transition graph of B, i.e., a partition of B such that any two states s and s' are in the same SCC C_i iff $s \xrightarrow{\tau^*} s' \xrightarrow{\tau^*} s$. States that are not part of any cycle of τ-transitions are therefore singletons in $\{C_1, \ldots, C_n\}$.
- For all $s_i \in C_i, s_j \in C_j$, if $s_i \xrightarrow{\tau} s_j$ then $j \leq i$.

Such a procedure can be implemented in linear-time using, e.g., a famous algorithm due to Tarjan [39]. It is applied to the initial partition $\{\Sigma\}$, with a role that is twofold:

- All states in the same C_i are (div)branching bisimilar and can therefore be compressed into a single representative state. In the divergence-preserving case, a self τ-loop is added on the representative state if there is a τ-transition whose source and target are in C_i. This allows the condition $\tau^\omega_{B,P}(s)$ to be replaced by $(\exists s' \in \Sigma)\ \tau^*_{B,P}(s, s') \wedge \tau_{B,P}(s', s')$, which can be checked more efficiently. In the non divergence-preserving case, all such τ-transitions are eliminated.
- The ordering of SCCs $C_1, \ldots, C_n$ (or more precisely, of their representative states) is preserved when splitting blocks: representatives occur in the same relative order as long as they remain in the same block. The signatures of states in a block are computed following this order, so that if there is an inert transition $s \xrightarrow{\tau}_P s'$ with $s' \neq s$, then the signature of s' is fully computed before starting to compute that of s. The signature of s can therefore be computed just by inspecting its immediate transitions and the signatures of the target states for those transitions which are inert.

For (div)sharp bisimulation, the signature of a state is a combination of the signatures for strong and (div)branching bisimulation, defined as follows:

$$sig_{\sim_{\sharp A_s}}(s, P) = \{(a, B) \mid (a \in A_s \wedge s \xrightarrow{a}_P B) \vee (a \notin A_s \wedge s \xRightarrow{a}_P B)\}$$
$$sig_{\sim^{@}_{\sharp A_s}}(s, P) = sig_{\sim_{\sharp A_s}}(s, P) \cup \{(\tau, B) \mid \tau^\omega_{B,P}(s)\}$$

As for branching and divbranching bisimulations, we must be careful when computing the transitive closure of the relation τ_P. However, it is not possible to compress cycles of τ-transitions beforehand, because states on such cycles are not necessarily bisimilar, as illustrated by the following example.

Example 5. *The LTSs depicted in Figure 4 are $\sim_{\sharp\{a\}}$ bisimilar, while the rightmost one is minimal with respect to $\sim_{\sharp\{a\}}$. Observe that states p'_0 and p'_2 belong to the same cycle of τ-transitions but are not $\sim_{\sharp\{a\}}$ bisimilar because p'_0 is source of a transition labelled by the strong action a, whereas p'_2 is not.*

Fig. 4 LTSs of Example 5

Instead, the procedure SCC_τ is used each time the signatures of a block need to be updated, i.e., just before splitting the block. Details are presented in Algorithm 1, where we use the following notations:

- for all signatures σ, we write $\sigma \setminus\setminus A_s$ for $\{(a, B') \mid (a, B') \in \sigma \wedge a \notin A_s\}$, i.e., the elements of σ whose label is not in A_s
- for all states s, we write $succ_{B,P}(s)$ for $\{(a, B') \mid s \xrightarrow{a}_P B' \wedge (\tau \in A_s \vee a \neq \tau \vee B' \neq B)\}$, i.e., the signature corresponding to the non-inert immediate successors of s

The function call $signatures_sharp(B, P)$ returns a function sig that implements $sig_{\sim^{@}_{\sharp A_s}}(s, P)$ for each state s of block B. The algorithm performs the following steps:

1. The signatures corresponding to the non-inert successors of all states s of block B (line 2) and the (ordered) SCCs of the τ-transition graph internal to block B are computed (line 3).
2. Within each SCC, the signature elements that are common to all states of the SCC are accumulated in variable σ. For each state s of the SCC, σ is extended with all ordered pairs (a, B') such that $a \notin A_s$ and $s \xrightarrow{a} B'$ (line 8), all ordered pairs (a, B') part of the signature of some state s' such that $a \notin A_s, \tau_P(s, s')$, and s' is in a distinct SCC from s (lines 9 to 11), and a divergence (τ, B) if s has an inert transition internal to the SCC (lines 12 and 13). To implement $\sim_{\sharp A_s}$ minimization (i.e., without preservation of divergences), lines 12 and 13 just have to be removed.
3. Finally, once σ contains the signature elements accumulated in all states of the SCC, the signatures of all states of the SCC are extended with σ (line 17).

5.3 Termination and Correctness

Termination and correctness of our algorithm rely on the termination and correctness of $signatures_sharp(B, P)$. For the whole algorithm, the argument is the same as for Blom & Orzan's algorithm.

Termination is immediate, because all five **foreach** loops enumerate values belonging to finite sets.

A correctness proof consists in showing that the equivalent assertions at lines 20 and 21, which are direct consequences of the invariant at line 5, hold. This invariant is obviously valid initially. We thus have to show that, under the assumption that the invariant holds in the current iteration and the fact

Algorithm 1 Divsharp signature computation

```
1   function signatures_sharp(B, P) is
2       foreach s ∈ B loop sig(s) := succ_{B,P}(s) end loop;
3       (C_1, ..., C_n) := SCC_τ(B);
4       foreach i ∈ 1..n loop
5           – invariant: (∀j ∈ 1..i − 1) (∀s ∈ C_j) sig(s) = sig_{∼@ ♯A_s}(s, P)
6           σ := ∅;
7           foreach s ∈ C_i loop
8               σ := σ ∪ (sig(s) \\ A_s);
9               foreach s' such that τ_P(s, s') loop
10                  if s' ∉ C_i then
11                      σ := σ ∪ (sig(s') \\ A_s)
12                  else
13                      σ := σ ∪ {(τ, B)}
14                  end if
15              end loop
16          end loop;
17          foreach s ∈ C_i loop sig(s) := sig(s) ∪ σ end loop
18          – assert: (∀s ∈ C_i) sig(s) = sig_{∼@ ♯A_s}(s, P)
19      end loop;
20      – assert: (∀j ∈ 1..n) (∀s ∈ C_j) sig(s) = sig_{∼@ ♯A_s}(s, P)
21      – assert: (∀s ∈ B) sig(s) = sig_{∼@ ♯A_s}(s, P)
22      return sig
23  end function
```

that the SCCs C_i are ordered, the initialization of *sig* at line 2 combined with the code from line 6 to 17 ensure the assertion expressed at line 18, which ensures that the invariant still holds in the next iteration.

It is easy to check that all pairs (a, B) added to $sig(s)$ at line 2 actually belong to $sig_{\sim^{@}_{\sharp A_s}}(s, P)$ and that all pairs added to σ at lines 8, 11, 13, and finally to $sig(s)$ at line 17 belong to $sig_{\sim^{@}_{\sharp A_s}}(t, P)$ for all states t in the same SCC as s. From the definition of $sig_{\sim^{@}_{\sharp A_s}}(s, P)$, it remains to show that, for all state $s \in C_i$:

- if $(\exists a, B')\ a \in A_s \wedge s \xrightarrow{a}_P B'$ then $(a, B') \in sig(s)$. This is ensured by the initialization of *sig* at line 2, because then $(a, B') \in succ_{B,P}(s)$.
- if $(\exists a, B')\ a \notin A_s \wedge s \stackrel{a}{\Longrightarrow}_P B'$ then $(a, B') \in sig(s)$. If $s \xrightarrow{a}_P B'$, then this is ensured by the initialization of *sig* at line 2 again. Otherwise, there exists $n > 0$ and s' such that $\tau_P^n(s, s')$ and $s' \xrightarrow{a}_P B'$. Without loss of generality, the n inert transitions from s to s' can be decomposed into k transitions internal to the SCC C_i followed by l transitions outside C_i. Formally, we

consider the largest natural number k such that there exist a state $t \in C_i$ and a natural number l satisfying $n = k + l$, $\tau_P^k(s,t)$, and $\tau_P^l(t,s')$.
If $l = 0$, this means that $t = s'$, i.e., $s' \in C_i$; then (a, B') is added to σ in some iteration of line 8, since $a \notin A_s$.
Otherwise, $l > 0$. There is a first inert transition outside the SCC C_i, i.e., there exists a state $u \notin C_i$ such that $\tau_P(t,u)$ and $\tau_P^*(u,s')$. Due to the ordering of SCCs, we know that $u \in C_j$ for some $j < i$. Therefore, due to the invariant, $(a, B') \in sig(u)$. Since s and t are both in C_i, (a, B') is added to σ at line 11. Thereafter, σ is added to $sig(s)$ at line 17.

- if $\tau_{B,P}^{\omega}(s)$, then the condition at line 10 is necessarily falsified at least once for all states belonging to the SCC of s, and (τ, B) is added to σ at line 13. Thereafter, σ is added to $sig(s)$ at line 17.

5.4 Time and Space Complexity

Computing the signatures of the states of a given block in Blom & Orzan's branching bisimulation reduction algorithm [4] requires to enumerate both the states of each block, as well as their outgoing transitions, similarly to our Algorithm 1. The most notable difference is the call to SCC_τ at line 3, which is known to be feasible using Tarjan's algorithm in time linear in both the number of states in B and the number of transitions going out from states belonging to B and thus does not change the complexity of signature computation. Thus, the complexity of our algorithm is unchanged with respect to Blom & Orzan's, namely $O(mn^2)$. The argument is similar for space complexity, which is $O(mn)$.

5.5 Performance

Our minimization algorithm was implemented in the tools BCG_MIN and BCG_CMP and released since CADP version 2021-f "Saarbruecken" (May 2021). To assess the performance of this implementation, we made several measurements on a computer with Intel Core i5 quad-core processor at 2.5 GHz using 16 GB of RAM, running GNU/Linux.

First, we applied divsharp minimization to the 180 CTL problems[12] of the RERS 2019 challenge[13]. We compared the obtained LTS sizes to those obtained initially using the partial reduction algorithm presented in [32], which was applied compositionally. The detailed results can be found in Appendix A. On average, the final LTS obtained using divsharp minimization has 7.69 times fewer states and 11.99 times fewer transitions than using partial divsharp reduction. The maximum is for problem 106#16, where the final LTS obtained using divsharp minimization has 3 states and 4 transitions instead of 527 states and 1314 transitions, that is 175 times fewer states and 328 times fewer transitions (i.e., a reduction rate of more than 99 %). In this experiment, the size of the largest LTS drops from 3363 states and 12427 transitions using partial

[12] We only considered the "parallel CTL" category, because it turned out that the LTL formulas proposed in the "parallel LTL" category in 2019 were all preserved by divbranching bisimulation, for which we already had a minimization algorithm.
[13] http://www.rers-challenge.org/2019

sharp reduction down to 813 states and 2256 transitions using minimization. We also checked (without surprise) that the CTL properties were preserved by minimization, as theoretically expected.

Second, we compared the memory and time consumed by divsharp minimization in the two particular cases where the set of strong actions is either empty ($\emptyset$) or maximal ($\mathcal{A}$) with the (equivalent in terms of resulting LTS) respective minimization algorithms specialized for strong and divbranching already implemented in BCG_MIN since 2010. This comparison allows us both to validate our implementation in those two extreme cases, and to assess the overhead induced by checking action strongness/weakness, as well as the use of the Tarjan SCC algorithm implementing SCC_τ, which is called to update the state signatures each time a block has been split.

We applied this comparison to four different packages of LTSs accumulated over time, represented in the BCG format of CADP: (1) a package BCG1 comprising 8995 "small" LTSs having less than $500,000$ transitions; (2) a package BCG2 comprising 542 "medium-size" LTSs having between $500,000$ and $5,000,000$ transitions; (3) a package BCG3 comprising 265 "large" LTSs having between $5,000,000$ and $50,000,000$ transitions; and (4) the public benchmark VLTS[14] comprising 40 LTSs ranging from 289 states and 1224 transitions up to $33,949,609$ states and $165,318,222$ transitions. The results are given in Tables 2 and 3, where for each package, we give the average memory, time consumption, and number of LTSs whose minimization requires more than the 16 GB of RAM available on the computer to complete (column fails). Each overhead represents the value (memory or time) obtained on a package using the general divsharp minimization algorithm (using either $\mathcal{A}$ or $\emptyset$ as set of strong actions) divided by the value obtained on the same package using the corresponding specialized minimization algorithm (i.e., strong or divbranching minimization), once the entries concerning LTSs on which at least one minimization failed have been removed. Time is expressed in seconds and memory is expressed in megabytes. This comparison shows that the overhead is not negligible, but acceptable.

Our implementation of (div)sharp minimization is most useful in cases where the set of strong actions is somewhere between the two extremes $\mathcal{A}$ and $\emptyset$, where it cannot be replaced by the implementation of strong or (div)branching minimization. From these experiments, we extrapolate that in such cases, the overhead of (div)sharp minimization should be compensated enough by the gain on LTS sizes, compared to strong minimization. This is at least what happened in our experiments concerning the RERS 2019 challenge.

Theorem 2 implies that orthogonal minimization can be implemented as a minor variation of $\sim_{\sharp\mathcal{A}\setminus\{\tau\}}$ minimization, by starting from an initial partition having at most two blocks, one block containing those states which are the source of a τ-transition (if any), and the other containing those which are not (if any). Finally, states of the quotient which were originally in the block containing states which are source of a τ-transition but are not anymore must

[14] http://cadp.inria.fr/resources/vlts

	$\sim$ minimization			$\sim^{@}_{\sharp\mathcal{A}}$ minimization			overhead $\sim^{@}_{\sharp\mathcal{A}}$ / $\sim$	
Package	memory	time	fails	memory	time	fails	memory	time
BCG1	2.6	0.14	0	2.7	0.21	0	1.04	1.82
BCG2	112	38	0	186	46	0	1.65	1.22
BCG3	646	52	16	1032	71	17	1.41	1.37
VLTS	242	15	0	363	22	0	1.34	1.46

Table 2 Performance of $\sim^{@}_{\sharp\mathcal{A}}$ minimization vs. dedicated $\sim$ minimization

	$\sim_{dbr}$ minimization			$\sim^{@}_{\sharp\emptyset}$ minimization			overhead $\sim^{@}_{\sharp\emptyset}$ / $\sim_{dbr}$	
Package	memory	time	fails	memory	time	fails	memory	time
BCG1	2.7	0.26	0	2.9	0.28	0	1.02	1.21
BCG2	208	34	0	332	40	0	1.60	1.17
BCG3	1255	64	16	1468	73	18	1.16	1.15
VLTS	305	13	0	407	16	0	1.30	1.09

Table 3 Performance of $\sim^{@}_{\sharp\emptyset}$ minimization vs. dedicated $\sim_{dbr}$ minimization

be decorated with a self τ-loop. We implemented this algorithm in BCG_MIN, based on the implementation of sharp minimization.

6 Compositional Verification: A Toy example

In Section 4, we showed that sharp minimization cannot reduce less than orthogonal minimization, as long as τ is a weak action. In this section, we provide a toy example to illustrate how much compositional verification can be improved, using sharp minimization rather than orthogonal minimization. This example is crafted on-purpose to be favorable to sharp bisimulation.

Our example is defined using two series of LTSs, namely P_m $(m \geq 1)$ and $Q_{n,m}$ $(n \geq 0, m \geq 1)$ defined as follows:

$$\begin{aligned}
(\forall m \geq 1) \quad P_m &= p_0 \xrightarrow{\tau} p_1 \xrightarrow{b} p_2 \ \ldots \ p_{2m-2} \xrightarrow{\tau} p_{2m-1} \xrightarrow{b} p_{2m} \\
(\forall m \geq 1) \ Q_{0,m} &= q_0 \xrightarrow{a} q_1 \\
(\forall m, n \geq 1) \ Q_{n,m} &= \textbf{prio } a \succ b \textbf{ in } (Q_{n-1,m} \,|[\emptyset]|\, P_m)
\end{aligned}$$

Informally, P_m is a sequence of $2m$ transitions in which the labels τ and b alternate, whereas $Q_{n,m}$ is the interleaving of n instances of P_m with an LTS consisting of a single transition labelled by a, to which the priority rule $a \succ b$ is applied.

We propose to generate the quotient of $Q_{n,m}$ for branching bisimulation equivalence, which is a sequence consisting of one transition labelled by a followed by nm transitions labelled by b. To this aim, we consider the compositional minimization strategy that consists in minimizing P_m and the LTS corresponding to $Q_{i,m}$ $(i \in 0..n)$ in sequence. In principle, branching minimization cannot be used for that purpose, as branching bisimulation equivalence is

not a congruence for "**prio** $a \succ b$". Instead, either minimization with respect to orthogonal bisimulation equivalence $\sim_\perp$ or sharp bisimulation equivalence $\sim_{\sharp\{a\}}$ can be used, because they both preserve branching bisimulation equivalence and are congruences for parallel composition and "**prio** $a \succ b$". A key difference is that P_m is minimal for $\sim_\perp$, whereas all its τ-transitions are inert with respect to $\sim_{\sharp\{a\}}$ and can thus be eliminated. Since none of the LTSs has divergences, the choice among sharp and divsharp bisimulation does not matter.

We compare the effectiveness of this compositional verification strategy using both equivalence relations, for m and n ranging in the interval 1..9. The results are given in Tables 4 and 5, in terms of the largest intermediate LTS size, i.e., number of states of the LTS corresponding to $Q_{n,m}$ before the final minimization. The tables show that while the largest intermediate LTS explodes using orthogonal minimization, it grows almost linearly in function of both n and m using sharp bisimulation. Note that the final LTS generated using sharp minimization is minimal for branching bisimulation equivalence in this example.

To understand the difference of effectiveness, consider the case $Q_{2,1}$, whose unreduced LTS is depicted in Figure 5 (top). Using compositional sharp minimization, we obtain the LTS depicted in Figure 5 (bottom). Using compositional orthogonal bisimulation, we obtain the LTS depicted in Figure 5 (middle), which is branching equivalent but has a stairs shape. Similar stairs shapes can be observed in all LTSs obtained using orthogonal bisimulation, for larger values of m and n.

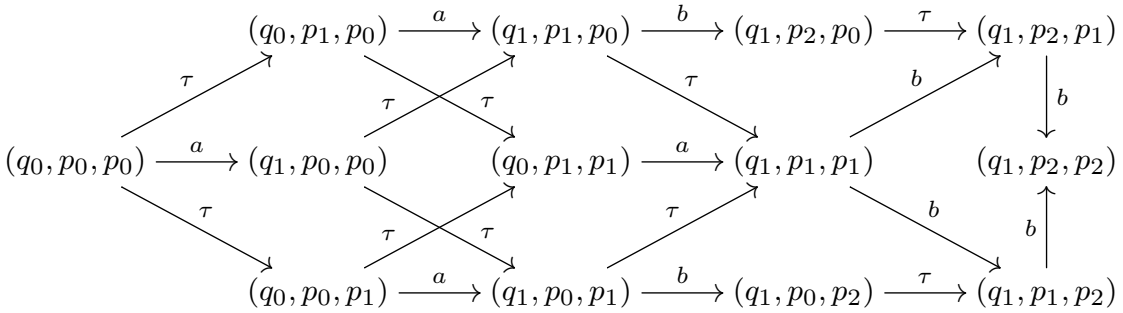
This illustrates that orthogonal bisimulation actually keeps most often more than one τ in every sequence of branching inert τ-transitions. Indeed, this can be seen in the above LTS, which is minimal for orthogonal bisimulation equivalence as none of its τ-transitions is inert with respect to orthogonal bisimulation:

- States t_2, t_3, and t_4 are not orthogonally equivalent to respectively t_5, t_6, and t_7, as each of the former is the source of a τ-transition, whereas each of the latter is not.
- State t_1 is not orthogonally equivalent to t_3, as t_3 is the source of a transition labelled by b, whereas t_1 is not.
- Finally, state t_0 is not orthogonally equivalent to t_2 as the transition labelled by a going out of t_0 goes to state t_1, which we have just shown to be not orthogonally equivalent to t_3, the target of the transition labelled by a going out of t_2.

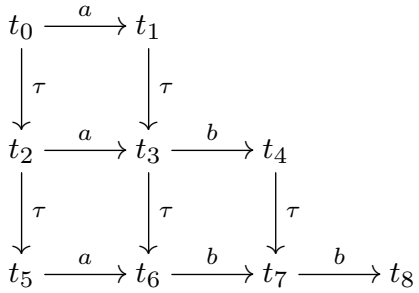
In terms of verification time and memory usage, the compositional verification scenario to generate $Q_{9,9}$ takes:

- 18 minutes and 4.5 GB of memory, yielding an LTS with $4,686,835$ states and $28,120,969$ transitions when orthogonal minimization is used
- 18 seconds and 4 MB of memory, yielding an LTS with 83 states and 82 transitions when $\sim_{\sharp\{a\}}$ minimization is used

$Q_{2,1} =$

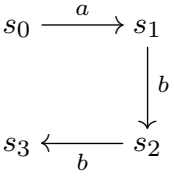


$Q_{2,1}/\sim_{\perp} =$



state	equivalent state(s) in $Q_{2,1}$
t_0	(q_0,p_0,p_0)
t_1	(q_1,p_0,p_0)
t_2	$(q_0,p_1,p_0), (q_0,p_0,p_1)$
t_3	$(q_1,p_0,p_1), (q_1,p_1,p_0)$
t_4	$(q_1,p_2,p_0), (q_1,p_0,p_2)$
t_5	(q_0,p_1,p_1)
t_6	(q_1,p_1,p_1)
t_7	$(q_1,p_1,p_2), (q_1,p_2,p_1)$
t_8	(q_1,p_2,p_2)

$Q_{2,1}/\sim_{\sharp\{a\}} =$



state	equivalent state(s) in $Q_{2,1}$
s_0	$(q_0,p_0,p_0), (q_0,p_1,p_0), (q_0,p_0,p_1), (q_0,p_1,p_1)$
s_1	$(q_1,p_0,p_0), (q_1,p_0,p_1), (q_1,p_1,p_0), (q_1,p_1,p_1)$
s_2	$(q_1,p_2,p_0), (q_1,p_0,p_2), (q_1,p_2,p_1), (q_1,p_1,p_2)$
s_3	(q_1,p_2,p_2)

Fig. 5 LTSs $Q_{2,1}$ (top), $Q_{2,1}/\sim_{\perp}$ (middle), and $Q_{2,1}/\sim_{\sharp\{a\}}$ (bottom).

As expected, both LTSs are branching equivalent. Using $\sim_{\sharp\{a\}}$ minimization, we were able to generate $Q_{40,40}$ in 189 seconds using 6.4 MB of memory.

Following Theorem 2, we know that the quotient of an LTS w.r.t. orthogonal bisimulation should not have more than twice the number of states as the quotient of the same LTS w.r.t. sharp bisimulation when all visible actions are strong. This shows that the gains observed here are mostly due to the possibility offered by sharp bisimulation to consider some actions as weak, such as b in this example.

7 Case Study: Bully Leader Election

As an example of a priority system, we consider a slight variation on the classic bully leader election algorithm [20]. We assume a system of n nodes, or agents, each with a unique numeric identifier, or id. In the typical bully election, each agent continuously broadcasts advertisement messages with the id of their current leader. Initially, every agent regards itself as the leader;

		n								
		1	2	3	4	5	6	7	8	9
m	1	5	13	24	38	55	75	98	124	153
	2	7	29	81	183	360	642	1064	1666	2493
	3	9	53	202	596	1480	3246	6482	12,028	21,039
	4	11	85	411	1493	4465	11,595	27,041	57,931	115,848
	5	13	125	732	3154	11,021	33,045	88,102	213,944	481,356
	6	15	173	1189	5923	23,670	80,456	241,346	655,060	1,637,628
	7	17	229	1806	10,208	45,910	174,432	581,414	1,744,216	4,796,568
	8	19	293	2607	16,481	82,375	345,945	1,268,435	4,167,685	12,503,025
	9	21	365	3616	25,278	138,995	639,343	2,557,338	9,133,316	29,683,243

Table 4 Largest intermediate LTS size (in number of states) during compositional $\sim^{@}_{\perp}$ minimization of $Q_{n,m}$

		n								
		1	2	3	4	5	6	7	8	9
m	1	3	5	7	9	11	13	15	17	19
	2	4	10	16	22	28	34	40	46	52
	3	5	17	29	41	53	65	77	89	101
	4	6	26	46	66	86	106	126	146	166
	5	7	37	67	97	127	157	187	217	247
	6	8	50	92	134	176	218	260	302	344
	7	9	65	121	177	233	289	345	401	457
	8	10	82	154	226	298	370	442	514	586
	9	11	101	191	281	371	461	551	641	731

Table 5 Largest intermediate LTS size (in number of states) during compositional $\sim_{\sharp\{a\}}$ minimization of $Q_{n,m}$

Listing 1 Bully leader election in LAbS.

```
system {
  spawn = Node: n
}
stigmergy Election {
  link = true
  ℓ: n
}

agent Node {
  stigmergies = Election
  Behavior =
    ℓ > id ->
      ℓ <~ id;
      Behavior
}
```

however, whenever an agent receives a message with an id i lower than its current leader, it appoints i as its new leader. Therefore, the whole system eventually elects the agent with the lowest identifier.

Our variation is written in LAbS [11], a simple language where agents cannot perform explicit message passing, but instead exploit an indirect communication mechanism based on *stigmergy variables* [37]. Just like an ordinary variable, a stigmergy variable may be assigned to (denoted by `x <~ e`, where `e` is an arithmetic expression) or referenced within an expression. When an agent assigns a value to one such variable, say `x`, the value is timestamped with the time of assignment.[15] Asynchronously, this will trigger a stigmergic *propagation* where the agent sends its (timestamped) value of `x` to all its neighbours. Neighbourhood is defined in the system specifications as a predicate over the

[15] For the sake of simplicity, we assume that timestamps are provided by a global clock, so that there cannot be two different values with the same timestamp.

state of a message sender and its potential receiver. This is called a *link predicate*, and LAbS allows different variables to have different predicates. Receivers will replace their own value with the received one if the latter is *newer* (i.e., it has a higher timestamp). Otherwise, they ignore the message. The semantics of LAbS assume that, whenever a propagation takes place, every receiver gets the message and possibly update their state atomically, similarly to a multi-party rendezvous.

Correspondingly, whenever an agent accesses a stigmergy variable `x` to evaluate an expression, it will later emit a *confirmation* message to ask neighbours whether its value for `x` is sufficiently up-to-date. Neighbours will still receive a timestamped value of `x`, and update their value if the received one is newer. However, in the case their local value is newer than the received one, they will (asynchronously) respond by propagating it in turn. These mechanisms allow knowledge to spread through the system and for new values to eventually replace old ones, which ensures eventual consistency.

In this example (Listing 1), each agent stores the id of its current leader in a stigmergy variable ℓ, defined (and initialized to n) at line 6. The definition is preceded by a line that defines the link predicate for the variable (line 5). In this case, the predicate is `true`, meaning that stigmergic messages will be broadcasted to every agent in the system. We assume identifiers of agents to be in the range $[0, n-1]$, and that ℓ is initially set to n for every agent. As the system evolves, each agent repeatedly assigns its own *id* to ℓ, but only as long as $\ell > id$ (lines 12–13). The repetitiveness of the agents' behaviour is implemented by the recursive call to `Behavior` at line 14.

Due to the stigmergic interaction mechanisms, it may happen that agents with lower ids choose one with a *higher* id as their leader, simply because that value is newer. However, high-id agents will eventually receive a value for ℓ that makes them stop, while the one with the lowest id will be able to perform one last assignment and one last broadcast to win the election.

A system of autonomous agents such as this one may be seen as a network of LTSs, namely one per agent plus an additional one that stores information about the timestamps. For the scope of this work, we developed a workflow [13] in the SLiVER tool[16] [14, 12] that turns a LAbS specification into an LNT program with this network structure, shown in Listing 2. Intuitively, agents in this LNT program may perform `refresh` actions to obtain a fresh timestamp for the stigmergy variable ℓ; `stig` actions to signal that they have set some stigmergic variable to a new value;[17] and `request` actions to compare timestamps with the sender of a stigmergy message. These actions are always decorated with the identifier of the agent performing them. For instance, a transition whose label starts with "`request !0`" denotes that the agent with id 0 is performing a `request` action. To perform a `request` or `refresh` action, an agent must synchronize with the timestamp process, whereas `stig` actions may be

[16] Available at https://github.com/labs-lang/sliver

[17] Our case study features only one such variable, namely ℓ. However, we always rely on a single gate `stig` to signal all updates, and put the name of the variable as data sent over the gate. This way, our encoding procedure is highly decoupled from the specific system being encoded.

Listing 2 Structure of an LNT program encoding a LAbS system.

```
process Main[refresh, request, debug, tick, ... : any] is
   par refresh, request in
      Timestamps [refresh, request, debug]
   ||
      hide put, qry: any in
      par tick, put, qry in
         Agent [tick, put, qry, refresh, request, stig] (ID(0))
      ||
         Agent [tick, put, qry, refresh, request, stig] (ID(1))
      || ··· ||
         Agent [tick, put, qry, refresh, request, stig] (ID(n − 1))
      end par
      end hide
   end par
end process
```

performed freely. The timestamp server performs an action `debug(...)` before every synchronization, to display its current internal state. This action does not affect the behaviour of the system in any way and is just a convenience feature to show more detailed information to the user. Additionally, agents may synchronize to exchange stigmergy messages (over the `put`, `qry` gates), or to decide which agent should act next (over the `tick` gate). We hide the `put`, `qry` gates, and leave `tick` visible.

A potential drawback of this encoding is that agents may react to a new message in any order: thus, each message-passing operation (which, as stated earlier, is atomic in the LAbS semantics) is split into a *diamond* with a large number of intermediate states and transitions which all lead to a single final state. What is worse, the size of these diamonds increases exponentially with the number of agents, as well as the number of transitions that each agent must perform to carry out the message reception. Similar diamonds can appear in the initialization phase of the system, where each agent initializes its state (in our example, setting ℓ to n) in any order.

By prioritizing some actions over the others, we can collapse these diamonds into much smaller ones. In the best case, i.e., when every intermediate state has a single outgoing transition that takes priority over all the others, the diamond effectively collapses into a single, representative path. Consider, for instance, the diamond in Figure 6: this is what we would obtain when three agents with ids 0, 1, 2 independently react to the same message (sent by a fourth agent, not shown). For the sake of simplicity, we assume that each agent of id n only has to perform one transition, which is labelled by a_n. If we assume the priority relation $a_0 \succ a_1 \succ a_2$ during LTS generation, all the dashed transitions in the diamond will be cut on the fly, leaving only the sequence $\cdot \xrightarrow{a_0} \cdot \xrightarrow{a_1} \cdot \xrightarrow{a_2} \cdot$.

To demonstrate the effect of priorities on these systems more concretely, we consider two leader election systems `leader3` and `leader4`, containing

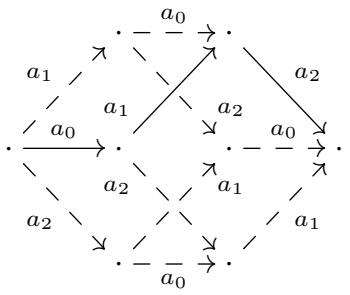


Fig. 6 Example of a diamond when 3 agents perform independent actions a_0, a_1, and a_2. Dotted transitions are cut by applying the priority relation $a_0 \succ a_1 \succ a_2$.

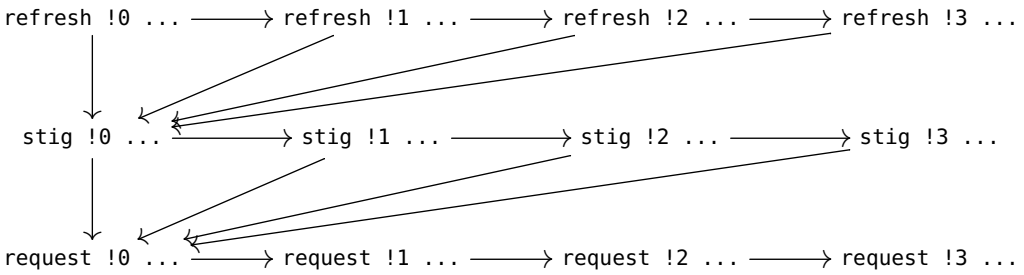


Fig. 7 A representation of the priority relation Ω used for the experiments in Table 6. An arrow $x \rightarrow y$ denotes that $x \succ y$.

respectively 3 and 4 agents, plus the aforementioned timestamp server. After translating each system to an LNT program, we use CADP to generate the LTS S of its `Main` process, describing the whole system (as shown in Listing 2). We then reduce S modulo strong, orthogonal and divbranching bisimulation, obtaining three additional LTSs $S_{\sim}$, $S_{\perp}$, and S_{dbr}. We also generate an LTS S^{Ω} = `prio` Ω `in` S, and minimize it modulo the aforementioned bisimulations to obtain $S^{\Omega}_{\sim}$, $S^{\Omega}_{\perp}$, and S^{Ω}_{dbr}. The priority relation Ω used in this phase is visualized in Figure 7, namely, `refresh !.*` $\succ$ `stig !.*` $\succ$ `request !.*`, regardless of which agent is performing it; furthermore, whenever two or more agents may perform the same action, the agent with the lowest identifier has the highest priority. Having set these priorities, we can now find a corresponding set of strong actions A_s that satisfies the requirements of Theorem 1, namely the set comprising all actions of the form "`refresh !.*`", "`stig !.*`", and "`request !.*`". Thus, we also reduce both S and S^{Ω} modulo divsharp bisimulation w.r.t A_s, and denote by $S_{\sharp}$ and $S^{\Omega}_{\sharp}$ the resulting LTSs. Notice that we merely use divbranching reduction as a theoretical optimum that divsharp minimization may achieve, since in general $\sim_{dbr}$ is not a congruence for priorities.

Table 6 shows the number of states and transitions for each LTS, and the overall resources (time and memory) needed to generate the LTS with priorities. Reported times only refer to the actual LTS generation process, i.e., excluding the time required to compile the LNT sources. On the

		Main		prio Ω in Main			
		states	transitions	states	tr.	time (s)	memory (kB)
leader3	LTS	1372	2375	884	1453	1.93	
	$\sim$	633	1115	427	733	2.17	
	$\sim_{\perp}$	621	1097	415	715	2.19	12940
	$\sim_{dbr}$	582	1037	379	661	2.01	
	$\sim^{@}_{\sharp A}$	582	1037	379	661	2.00	
leader4	LTS	16912	33368	6334	10427	110.02	
	$\sim$	5450	9895	2572	4458	110.35	
	$\sim_{\perp}$	5402	9823	2524	4386	110.25	14628
	$\sim_{dbr}$	5138	9427	2324	4086	110.17	
	$\sim^{@}_{\sharp A}$	5138	9427	2324	4086	110.14	

Table 6 State space generation of leader election systems (with and without priorities) using different bisimulations.

one hand, we can observe that orthogonal bisimulation is slightly more effective than strong bisimulation. On the other hand, divsharp minimization proves even more effective, producing LTSs as small as the ones obtained by divbranching minimization (i.e., the optimum). Notice that, since A_s satisfies the premises of Theorem 1 with respect to the priority rules Ω, we could have obtained the same LTS as $S^{\Omega}_{\sharp}$ by minimising S modulo $\sim^{@}_{\sharp A_s}$ before applying the priority rules to the resulting LTS. https://www.overleaf.com/project/621e0ce6c0f594d6d5119a21 Another insight from Table 6 is that adding one agent induces a 10-fold increase in the size of the LTS. This state space explosion makes "naive" state space generation impractical for larger systems: in fact, we tried generating the LTS for a `leader5` system but hit a 1-hour timeout limit. Therefore, we performed additional experiments using a compositional strategy, namely *root leaf reduction* [16], to see whether it would bring significant improvements to state space generation over the *sequential* approach used so far. This strategy consists in generating and minimizing the individual LTSs in the network (the *leaves*), then composing them to generate the LTS of the network, and finally performing one last minimization of the composite LTS (the *root*). If any priority rules are given, they are applied on the fly during composition. In our experiments, we carry out minimizations using either orthogonal or divsharp bisimulation. Results are shown in Table 7. In each sub-table, we first report the size of the `Timestamp` and `Agent` processes, before and after minimization. Once the individual LTSs are ready, we compose them to obtain the LTS for both the unprioritized and prioritized systems.

These two composite LTS are denoted `Main` and `prio Ω in Main` in our table, respectively. Notice that in both cases we report the size of the composite LTS both before and after performing the "root" minimization. We also report the time and memory needed to generate each LTS with divsharp reduction. We omit measurements related to orthogonal reduction, as they do not significantly differ from the reported ones. Again, all reported times ignore the overhead introduced by LNT compilation.

Notice that, in a system of n agents, the procedure will generate n `Agent` LTSs. These have the same size, but different ids and therefore a potentially different behaviour. For these LTSs, we report the average time and memory consumption. To allow easier comparisons with the data in Table 6, the `Main` and `prio Ω in Main` lines, before and after reduction, report *aggregate* measures. Namely, the "time" column shows the time needed to generate each individual LTS, plus the time required to compose them (and optionally to reduce the composite LTS); the "memory" column reports the maximum amount of memory consumed at any moment during the whole process.

From the table, one can still appreciate some form of explosive state space growth as the number of agents in the system increases. This is most noticeable in the `Timestamps` process, which displays the same 10-fold size increase seen in the LTSs of Table 6. In any case, the compositional procedure appears to handle larger systems more gracefully than the sequential one, at the cost of being slightly less effective on smaller instances. In fact, when the system contains only 3 agents, the sequential approach is faster, requiring 2 seconds to generate the LTS of the priority system compared to the 5.80 s of the compositional procedure. The most likely explanation for this is that the intermediate steps needed by the compositional approach introduce some amount of overhead, which at this small scale is significant. Still, we observe that the procedure is slightly more memory-efficient, consuming around 300 kB less than the sequential approach. However, with 4 agents we can already see that the sequential approach takes 110.14 s, while the compositional procedure finishes in 7.05 s. Memory savings are also significant, as the compositional approach uses 6.3 MB less than the sequential one to complete its task. These observations suggest that the latter may scale better with bigger systems. These gains become even more evident on `leader5`, where the sequential procedure times out after 1 hour, but the compositional one is able to complete the task in 9.79 s. Even when considering the LNT compilation overhead, the whole procedure finishes in less than a minute (namely, 55.43 s).

8 Related Work

The current work takes its place in the history of bisimulation tools, an extended survey of which can be found in [17].

Several approaches for priority in process languages have been proposed since the early 1980s. The interested reader may have a look at the introduction of [10] for a classification. Our aim in this paper is not to promote one approach over the other, but more pragmatically to provide a theoretical framework that suits well with the priority operator that was implemented in the EXP.OPEN tool of CADP more than 15 years ago.

Another congruence for parallel composition and priority in a setting close to ours is prioritized weak bisimulation equivalence [10], which is based on weak (a.k.a. observational) bisimulation [35] rather than (div)branching. Weak bisimulation equivalence is known to have less efficient algorithms than

		LTS		$\sim_\perp$		$\sim^{@}_{\sharp A_s}$			
		states	tr.	states	tr.	states	tr.	time (s)	mem. (kB)
leader3	Timestamps	27	131	26	130	26	130	0.83	7796
	Agent	317	2661	117	453	113	449	1.33	12680
	Main (before min.)	—	—	946	1556	892	1475	5.22	12680
	Main	—	—	621	1097	582	1037	5.52	12680
	prio Ω in Main (before min.)	—	—	632	979	584	907	5.18	12680
	prio Ω in Main	—	—	415	715	379	661	5.80	12680
leader4	Timestamps	151	1276	150	1275	150	1275	0.83	7812
	Agent	585	6960	250	1030	245	1025	1.37	7812
	Main (before min.)	—	—	9520	17068	9208	16600	6.83	10888
	Main	—	—	5402	9823	5138	9427	7.11	10888
	prio Ω in Main (before min.)	—	—	4334	6527	4086	6155	6.78	8168
	prio Ω in Main	—	—	2524	4386	2324	4086	7.05	8168
leader5	Timestamps	542	13550	541	13525	541	13525	1.02	8276
	Agent	977	14291	461	1949	455	1943	1.45	12624
	Main (before min.)	—	—	55627	140120	54647	139140	10.95	12624
	Main	—	—	22924	51856	22339	51271	11.55	12624
	prio Ω in Main (before min.)	—	—	15806	20470	15101	19765	9.41	12632
	prio Ω in Main	—	—	7832	12496	7247	11911	9.79	12632

Table 7 Compositional state space generation of distributed leader election systems.

(div)branching. As sharp bisimulation, prioritized weak bisimulation takes into account the set of prioritized actions in its definition, which is however much more involved than sharp bisimulation. We are not aware of any available implementation of prioritized weak bisimulation. Also, there are differences in the definition of the priority operator (based on CCS), which makes it difficult to precisely compare both approaches in practice.

Orthogonal minimization has to face a similar problem as sharp minimization, namely states in the same cycle of τ transitions are not necessarily orthogonally bisimilar. This issue was addressed by Vu [41], whose algorithm, similarly to ours, also relies on a linear-time procedure for computing the SCCs of a directed graph applied inside blocks, initially and after each block splitting. This preserves the time complexity of the algorithm it was based on, namely Groote & Vaandrager's algorithm for branching minimization [25]. The worst-case time complexity $O(mn)$ of this algorithm (where n and m are respectively the numbers of states and transitions of the LTS) is lower than that of signature-based minimization $O(mn^2)$ on which our algorithm is based. However, an experimental evaluation would be interesting to gain more insight, as our experience indicates that signature-based minimization may be more efficient than Groote & Vaandrager's algorithm in many practical cases,[18] see e.g., [23] (Fig. 9, right). Unfortunately, we could not find any implementation of Vu's algorithm. Orthogonal bisimulation is also implemented in the tool Sigref [42], which uses a symbolic representation of state spaces (in the form of Binary Decision Diagrams) and implements the transitive closure of τ-inert transitions using the iterative squaring method of symbolic model checking [8].

Going further on complexity, there exists a recent algorithm for (div)branching minimization, with worst-case time complexity $O(m \log n)$ [28]. The question whether this algorithm could be adapted to implement (div)sharp minimization while keeping its complexity is open.

Inductive sequentialization [29] is a technique where an asynchronous program is analysed by constructing a *sequential reduction*, i.e., a sequential program that captures every behaviour of the original up to reordering of commutative actions. This is similar to our use of priorities to remove diamonds from LTSs that encode a concurrent system (e.g., the leader election of Section 7). Both approaches still require a certain amount of creative work. Inductive sequentialization requires one to come up with an idealized sequential execution, while our approach requires finding an adequate priority set. Applying priorities to a system does not generally preserve its behaviour. In other words, typically P and `prio` Ω `in` P are not bisimilar. However, if the altered behaviour does not affect the satisfaction of a given formula φ, then verifying φ against the prioritized system can be seen as a form of partial order reduction [1]. This is what happens in our case study: since we know that the prioritized system conforms to the semantics of LAbS, we are not interested in

[18] For the anecdote, branching minimization was based on Groote & Vaandrager's algorithm in version 1 of BCG_MIN. In 2010, we released version 2, whose implementation based on signatures was found 20 times faster on a benchmark of 3700 realistic examples systematically collected over time.

the portion of state space that gets cut thanks to the priority operator. These reductions are known to be effective in countering state-space explosion [1, 15]. In order to be generally applicable they usually require the system of interest to be equipped with a partial order semantics, whereas LAbS and LNT currently rely on interleaving semantics.

9 Conclusion

Sharp bisimulation was already known as an efficient way to tackle verification problems by taking a fine account of the temporal logic formula to be verified and by being suitable for compositional verification. In this paper, we extended the set of fundamental results on sharp bisimulation equivalence, by showing that it may be a congruence for action priority operators. Therefore, it is also appropriate to verify systems with priority compositionally. The relationship between sharp bisimulation and orthogonal bisimulation, another congruence for priority, was clarified.

We also solved the problem of minimizing a process with respect to sharp bisimulation equivalence in an efficient way, providing an extension of Blom & Orzan's signature-based partition-refinement algorithm for (div)branching minimization. The extension is not trivial, as sharp bisimulation equivalence does not allow cycles of τ-transitions to be eliminated beforehand, unlike branching bisimulation. Instead, blocks must be traversed using a linear-time algorithm for detecting strongly-connected components at each partition-refinement step. Our results are implemented in tools in CADP, namely BCG_MIN and BCG_CMP for (div)sharp minimization and comparison, and EXP.OPEN for action priority and other LTS composition operators.

We applied those tools successfully to a crafted toy example, showing that using sharp bisimulation may produce LTSs that are several orders of magnitude smaller than those obtained using orthogonal bisimulation, and to a case study in the domain of collective adaptive systems, where action priority is used to restrict the state space. Remarkably, minimization of the composed processes with respect to divsharp bisimulation equivalence offers more reduction than strong and orthogonal bisimulations, and even as much reduction as divbranching bisimulation. Yet, contrary to (div)branching bisimulation, (div)sharp bisimulation offers all theoretical guarantees for the resulting state space to preserve the semantics of the non-reduced system. The effect of (div)sharp minimization on state space size could be greatly amplified in systems whose network definition is hierarchical, the composed processes being themselves the result of compositions and minimizations.

Acknowledgments

A part of this work has been performed in the A-IQ Ready project, which receives funding within the Key Digital Technologies Joint Undertaking (KDT JU) — the Public-Private Partnership for research, development and innovation under Horizon Europe — and National Authorities under grant agreement

no. 101096658. It was also partly upported by ERC consolidator grant no. 772459 *D-SynMA* (Distributed Synthesis: from Single to Multiple Agents).

Some experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.[19]

The authors would like to warmly thank the anonymous referees, as well as David Jansen who inspired this work by pointing to the second author a possible relation between orthogonal and sharp bisimulations during a presentation of [32] at TACAS'2021.

Appendix A Detailed Results on RERS 2019

The following tables give, for each problem of the "parallel CTL" category of the RERS 2019 challenge, the size of the LTS obtained using the partial sharp reduction described in [32] and the size of the LTS obtained using sharp minimization. The last column indicates the reduction rate (in number of states) obtained using minimization with respect to partial reduction. Each problem consists of the number of a model (from 101 to 109) and the number of a property that depends on the model (from 01 to 20). In each case, the set of strong actions is determined by the property, so that its preservation by sharp bisimulation can be ensured.

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
101#01	10	23	6	11	40 %
101#02	5	8	5	8	0 %
101#03	10	23	10	23	0 %
101#04	5	9	5	9	0 %
101#05	3	6	3	6	0 %
101#06	9	10	8	18	11 %
101#07	26	63	26	63	0 %
101#08	9	19	9	19	0 %
101#09	7	12	7	12	0 %
101#10	5	8	4	6	25 %
101#11	6	10	6	10	0 %
101#12	17	33	17	33	0 %
101#13	13	37	4	7	54 %
101#14	5	9	5	9	0 %
101#15	6	11	6	11	0 %
101#16	8	13	8	13	0 %
101#17	7	12	7	12	0 %
101#18	10	24	10	24	0 %
101#19	17	49	5	9	71 %
101#20	12	26	12	26	0 %

[19] See https://www.grid5000.fr

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
102#01	9	20	6	11	33 %
102#02	16	40	9	18	78 %
102#03	5	9	5	9	0 %
102#04	12	24	7	12	42 %
102#05	16	33	10	17	38 %
102#06	4	7	4	7	0 %
102#07	4	9	4	9	0 %
102#08	7	11	7	11	0 %
102#09	7	12	7	12	0 %
102#10	6	15	6	15	0 %
102#11	3	5	3	5	0 %
102#12	5	9	5	9	0 %
102#13	5	8	5	8	0 %
102#14	8	16	6	11	25 %
102#15	16	29	9	14	44 %
102#16	3	4	3	4	0 %
102#17	9	15	9	15	0 %
102#18	4	5	4	5	0 %
102#19	8	16	8	16	0 %
102#20	7	11	7	11	0 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
103#01	8	13	5	7	38 %
103#02	6	12	6	12	0 %
103#03	3	6	3	6	0 %
103#04	4	6	4	6	0 %
103#05	6	12	6	12	0 %
103#06	6	11	5	9	17 %
103#07	6	12	6	12	0 %
103#08	5	8	5	8	0 %
103#09	51	154	21	55	59 %
103#10	5	10	5	10	0 %
103#11	10	16	7	10	30 %
103#12	10	24	8	16	20 %
103#13	7	13	7	13	0 %
103#14	5	9	5	9	0 %
103#15	6	10	6	10	0 %
103#16	4	7	4	7	0 %
103#17	6	12	6	12	0 %
103#18	8	19	8	19	0 %
103#19	4	6	4	6	0 %
103#20	3	6	3	6	0 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
104#01	25	48	5	6	80 %
104#02	6	8	6	8	0 %
104#03	9	12	9	12	0 %
104#04	7	9	7	9	0 %
104#05	34	74	6	8	82 %
104#06	16	31	6	8	63 %
104#07	13	21	8	10	38 %
104#08	16	34	4	6	75 %
104#09	10	16	6	8	40 %
104#10	4	6	4	6	0 %
104#11	55	110	15	24	73 %
104#12	14	27	5	7	64 %
104#13	28	58	24	51	14 %
104#14	14	30	3	4	79 %
104#15	16	29	5	7	69 %
104#16	47	104	5	7	89 %
104#17	15	30	10	18	33 %
104#18	15	23	13	18	13 %
104#19	25	55	15	29	40 %
104#20	7	9	7	9	0 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
105#01	19	37	11	17	42 %
105#02	12	26	12	26	0 %
105#03	24	66	20	56	17 %
105#04	7	12	7	12	0 %
105#05	20	60	10	29	50 %
105#06	5	7	5	7	0 %
105#07	5	8	5	8	0 %
105#08	6	8	6	8	0 %
105#09	4	7	4	7	0 %
105#10	7	10	7	10	0 %
105#11	14	31	8	14	43 %
105#12	5	12	5	12	0 %
105#13	2	4	2	4	0 %
105#14	10	17	10	17	0 %
105#15	8	25	8	25	0 %
105#16	30	91	16	39	47 %
105#17	4	7	4	7	0 %
105#18	7	12	7	12	0 %
105#19	6	13	6	13	0 %
105#20	10	23	7	14	30 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
106#01	18	48	6	12	67 %
106#02	18	51	10	25	44 %
106#03	8	21	4	7	50 %
106#04	18	48	6	11	67 %
106#05	22	54	8	14	64 %
106#06	18	48	6	12	67 %
106#07	7	15	3	5	57 %
106#08	5	11	5	11	0 %
106#09	15	35	9	19	40 %
106#10	16	41	6	11	63 %
106#11	7	15	3	5	57 %
106#12	5	10	5	10	0 %
106#13	4	9	2	3	50 %
106#14	23	86	13	42	43 %
106#15	14	33	14	33	0 %
106#16	527	1314	3	4	99 %
106#17	11	30	5	10	55 %
106#18	8	18	4	6	50 %
106#19	338	1085	5	11	99 %
106#20	14	36	6	12	57 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
107#01	1	2	1	2	0 %
107#02	99	247	8	13	92 %
107#03	71	192	6	10	92 %
107#04	86	221	8	13	91 %
107#05	86	279	7	16	92 %
107#06	29	69	3	4	90 %
107#07	1	2	1	2	0 %
107#08	269	829	12	25	96 %
107#09	30	70	4	5	87 %
107#10	72	223	12	31	83 %
107#11	71	264	6	17	92 %
107#12	46	100	7	9	85 %
107#13	29	70	3	5	90 %
107#14	102	287	16	40	84 %
107#15	99	233	8	12	84 %
107#16	43	124	4	7	91 %
107#17	44	99	5	8	89 %
107#18	29	70	3	5	90 %
107#19	32	81	5	11	84 %
107#20	85	234	7	13	92 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
108#01	1	2	1	2	0 %
108#02	6	15	6	15	0 %
108#03	11	27	8	16	27 %
108#04	11	21	9	15	18 %
108#05	3	4	3	4	0 %
108#06	16	35	8	13	50 %
108#07	7	15	7	15	0 %
108#08	15	41	11	31	27 %
108#09	22	68	22	68	0 %
108#10	22	61	10	18	55 %
108#11	4	5	4	5	0 %
108#12	3	5	3	5	0 %
108#13	4	8	4	8	0 %
108#14	8	22	5	12	38 %
108#15	4	5	4	5	0 %
108#16	5	8	5	8	0 %
108#17	6	13	6	13	0 %
108#18	15	30	11	19	27 %
108#19	4	11	4	11	0 %
108#20	9	19	8	15	11 %

problem	partial reduction		minimization		extra reduction
	#states	#trans	#states	#trans	(% states)
109#01	93	386	9	26	90 %
109#02	1	2	1	2	0 %
109#03	207	763	10	24	95 %
109#04	105	396	9	24	91 %
109#05	106	296	8	15	92 %
109#06	53	205	5	13	91 %
109#07	68	156	8	12	88 %
109#08	68	251	6	16	91 %
109#09	27	64	3	4	89 %
109#10	53	141	5	9	91 %
109#11	27	65	3	5	89 %
109#12	14	53	2	5	86 %
109#13	1	2	1	2	0 %
109#14	28	91	4	7	86 %
109#15	69	213	8	18	88 %
109#16	145	485	13	29	91 %
109#17	28	108	3	9	89 %
109#18	17	58	5	10	71 %
109#19	27	64	3	4	89 %
109#20	66	178	6	10	91 %

Listing 3 Timestamp server for the LAbS-to-LNT encoding

```
process Timestamps [refresh, request, debug: any] is
  var M: Matrix, i, j:ID, x, c:Comparison in
    M := Matrix(MatrixVar(SAME));
    loop
      debug(M);
      select
        refresh(?i, ?x) where (x < NVARS);
        eval refreshVar(!?M, x, i)
      []
        x := any Nat where (x < NVARS);
        i := any ID;
        j := any ID where (i != j);
        c := M^x_{i,j};
        request(j, x, i, c);
        if (c == GREATER) then
          eval sync(!?M, x, i, j)
        end if
      end select
    end loop
  end var
end process
```

Appendix B LNT encoding of timestamp server

In this section we give some details about the LNT process that stores information about the timestamps of stigmergy variables. Specifically, for every pair of agents i, j and every stigmergy variable x, the process records whether the timestamp that agent i binds to x is greater, equal, or smaller than the one bound by j to the same variable. Since the LAbS semantics only cares about these comparisons, there is no need to store actual timestamps. We encode these comparisons as an LNT enumeration `Comparison` with three values `GREATER`, `SAME`, and `LESS`. So, the process essentially has to maintain an $n \times n$ matrix of comparisons M^x for each stigmergy variable x, and provide responses when queried about its contents.

Listing 3 shows a sketch of our implementation of the timestamps server. Notice that `MatrixVar` is the type of a 2-dimensional matrix associated to a single variable, whereas `Matrix` is the type of a 3-dimensional matrix composed of one `MatrixVar` for each variable. We use `NVARS` to denote the number of variables.

At first, the server initializes every element of M to `SAME` (line 3). Then, it enters a loop where it repeatedly sends M over the `debug` gate and handles a message from the environment. The send action does not affect the server in any way, and is only there to provide a richer counterexample to the user when it tries to model-check a property that does not hold.

Listing 4 Agent process for the LAbS-to-LNT encoding.

```
1  process Agent [
2      tick,put,qry,refresh,request,stig:any,dead:none
3  ] (id: ID) is
4    var t: ID, I: Iface, pc: PC, L: Lstig, Zput, Zqry: Pending in
5    Zput := Pending (false); Zqry := Pending (false);
6    InitAgent[l, refresh] (id, ?I, ?pc, ?L);
7
8    loop select
9       -- Let another agent make a move (Listing 5)
10   [] -- Make a move (Listing 5)
11   [] -- send put (Listing 6)
12   [] -- receive put (Listing 6)
13   [] -- send qry
14   [] -- receive qry
15   end select end loop
16   end var
17 end process
```

As outlined in Section 7, the server accepts messages over two gates `refresh` and `request`. Gate `refresh` allows an agent `i` to ask for one of its own timestamps to be updated (lines 7–8). The server handles this request by calling a helper function `refreshVar(M, x, i)`: this, in turn, updates M so that $M^{\mathtt{x}}_{\mathtt{i},j} = \mathtt{GREATER}$ for every $j \neq \mathtt{i}$.

Gate `request`, instead, allows an agent to retrieve the value of $M^{\mathtt{x}}_{i,j}$ (lines 10–17). What actually happens, here, is that the server simultaneously offers *all* the contents of M over this gate. Since we force synchronization over `request`, the server will just wait for an agent to synchronize on one of these offers. After this happens, if the element $M^{\mathtt{x}}_{\mathtt{i},\mathtt{j}}$ that was offered was `GREATER`, we need to perform some bookkeeping on M. Namely, this means that the receiver `j` has gotten a stigmergy message from `i` with a newer value of `x` than its own local one. According to the semantics of LAbS, then, `j` will update its value *and timestamp* to the ones in the message.

While the value update is handled within the `Agent` process, the timestamp update must be implemented in the server. This essentially amounts to setting $M^{x}_{\mathtt{i},\mathtt{j}}$ to `SAME` and then copying the column $M^{x}_{\mathtt{i}}$ onto $M^{x}_{\mathtt{j}}$. These operations are implemented by a helper function `sync`, not shown here.

Appendix C LNT encoding of agents

The behaviour of LAbS agents is encoded into a parameterized LNT process `Agent(id)`, where the parameter `id` is a unique numeric identifier. The structure of this process is shown in Listing 4. Similarly to the timestamp server (Appendix B), `Agent` is composed of an initialization step (lines 5–6) followed by an event loop (lines 8–15).

Listing 5 Agent process: individual actions.

```
1  -- Let another agent make a move
2  tick(?t) where t != id
3  []
4  -- Make a move
5  only if (empty (Zput) and empty (Zqry)) then
6    tick(id);
7    if canProceed(id, I, pc, L) then
8      select
9        action_0 [...] (...)
10     []
11       action_1 [...] (...)
12     []
13       ...
14     end select
15   else loop dead end loop
16   end if
17 end if
```

The initialization creates two empty sets `Zput`, `Zqry` that will track pending stigmergy messages, and sets the agent's state to its (potentially nondeterministic) initial as described in the input specifications, implemented by the helper process `InitAgent`. The agent's state includes its local stigmergy variables `L` and a set of local variables `I` that are not subject to stigmergic interaction, but may influence the agents' link predicates.

The event loop allows the agent to perform an individual action, wait for another agent to make its move, and to send or react to a stigmergy message. Notice that this scheme also features an additional gate `dead`, which we omitted from the main text for sake of brevity. It will be used by the agent to signal whether it is in a deadlocked state.

Individual actions. Listing 5 shows the behaviours related to individual actions in more detail. These behaviours are guarded by a multi-party rendezvous on the `tick` gate. Every agent can always do nothing and allow a different one to act (line 2), or try and perform an action (lines 5–17). When multiple agents are all willing to make a move, the rendezvous will force the system to choose one.

Furthermore, an agent may only try to make a move if it has no pending messages (this is specified in the semantics of LAbS): the `only if` block at line 5 implements this rule by making this branch of the `select` deadlocked until the condition is met. Every individual action in the original behaviour is encoded into its own process `action_<x>` (lines 8–14). These processes are always guarded by a condition on the `pc` variable: this variable is the *program counter* of the agent and tracks its execution point. Since LAbS also has guarded processes (like the ℓ `> id ->`... one in Listing 1, lines 12–14), some of these actions may be further guarded by additional conditions. Thus,

Listing 6 Agent process: exchange of propagation messages.

```
1    -- send put
2    only if length (Zput) > 0 then
3      var key: Nat in
4          key := any Nat where (key < NVARS) and member(key, Zput);
5          put(id, key, L, I);
6          Zput := remove (key, Zput)
7      end var
8    end if
9  []
10   -- receive put
11   var s:ID, k:Nat, Ls:Lstig, Is:Iface, cmp:Comparison, lnk:Bool
12   in
13     put(?s, k, ?Ls, ?Is) where (s != id) and (k < NVARS);
14     request(id, k, s, ?cmp);
15     lnk := link(Is, I, Ls, L, s, id, k) and (cmp == GREATER)
16
17     if lnk then
18       L[key] := Ls[key];
19       stig (id, key, L[k]);
20       Zput := insert(key, Zput);
21       Zqry := remove(key, Zqry)
22     end if
23   end var
```

before getting to choose the next action to perform, we evaluate a function `canProceed` that tells us whether at least one of these guards can be satisfied (line 7). If this is not the case, we send the agent to a sink state where it endlessly performs a `dead` action to signal it is deadlocked (line 15).

Exchange of stigmergy messages. An exchange of a message is triggered by a rendezvous on either the `put` or `qry` gate. This, combined with the fact that actions over `tick` also require a multi-party rendezvous, guarantees that message exchange never overlap with each other or with individual actions. We only report the details about `put` behaviour, as the one for `qry` is quite similar. To propagate a variable, an agent must have at least one variable in its set of pending messages `Zput`: if this is the case, the agent may nondeterministically select one of these variables, offer it over `put` along with additional information, and remove the variable from `Zput` (lines 2–2). Each receiver, on the other hand, waits until a rendezvous on `put` becomes available (that is, until another agent `s` is willing to propagate: line 13). Then, it gets the appropriate timestamp comparison from the timestamp server, by means of a `request` transition (line 14). Finally, if the link predicate holds and the comparison value is `GREATER`, it overwrites its value for variable `x` with the one coming from the sender (lines 15–19). It also adds `x` to the pending propagation messages and removes it from that of pending confirmation messages,

since receiving a new value is enough to conclude that the previous one was not up-to-date (lines 20–21).

References

[1] Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state space exploration. In: Grumberg, O. (ed.) 9th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 1254, pp. 340–351. Springer, Haifa, Israel (1997). https://doi.org/10.1007/3-540-63166-6_34

[2] Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Syntax and defining equations for an interrupt mechanism in process algebra. Fundamenta Informaticae **IX**, 127–168 (1986)

[3] Bergstra, J.A., Ponse, A., van der Zwaag, M.B.: Branching time and orthogonal bisimulation equivalence. Theoretical Computer Science **309**(1–3) (2003)

[4] Blom, S., Orzan, S.: Distributed Branching Bisimulation Reduction of State Spaces. Electronic Notes in Theoretical Computer Science **89**(1), 99–113 (2003)

[5] Blom, S., Orzan, S.: A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. Software Tools for Technology Transfer **7**(1), 74–86 (2005)

[6] Blom, S., Orzan, S.: Distributed State Space Minimization. Software Tools for Technology Transfer **7**(3), 280–291 (2005)

[7] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. J. ACM **31**(3), 560–599 (Jul 1984)

[8] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential circuit verification using symbolic model checking. In: Smith, R.C. (ed.) Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990. pp. 46–51. IEEE Computer Society Press (1990). https://doi.org/10.1145/123186.123223

[9] Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LNT to LOTOS Translator (Version 7.0) (Sep 2021), INRIA, Grenoble, France

[10] Cleaveland, R., Lüttgen, G., Natarajan, V.: Priority in process algebras. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, chap. 12, pp. 711–765. North-Holland (2001)

[11] De Nicola, R., Di Stefano, L., Inverso, O.: Multi-agent systems with virtual stigmergy. Science of Computer Programming **187**, 102345 (2020). https://doi.org/10.1016/j.scico.2019.102345

[12] Di Stefano, L., Lang, F.: Verifying temporal properties of stigmergic collective systems using CADP. In: Margaria, T., Steffen, B. (eds.) 10th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Lecture Notes in Computer Science,

vol. 13036, pp. 473–489. Springer (2021). https://doi.org/10.1007/978-3-030-89159-6_29

[13] Di Stefano, L., Lang, F.: Compositional verification of stigmergic collective systems. In: Dragoi, C., Emmi, M., Wang, J. (eds.) 24th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). Lecture Notes in Computer Science, vol. 13881, pp. 155–176. Springer, Boston, MA, USA (2023). https://doi.org/10.1007/978-3-031-24950-1_8

[14] Di Stefano, L., Lang, F., Serwe, W.: Combining SLiVER with CADP to analyze multi-agent systems. In: Bliudze, S., Bocchi, L. (eds.) 22nd International Conference on Coordination Models and Languages (COORDINATION). Lecture Notes in Computer Science, vol. 12134, pp. 370–385. Springer, Valletta, Malta (2020). https://doi.org/10.1007/978-3-030-50029-0_23

[15] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) 32nd Symposium on Principles of Programming Languages (POPL). pp. 110–121. ACM, Long Beach, CA, USA (2005). https://doi.org/10.1145/1040305.1040315

[16] Garavel, H., Lang, F.: SVL: a Scripting Language for Compositional Verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01), Cheju Island, Korea. pp. 377–392. Kluwer Academic Publishers (Aug 2001), full version available as INRIA Research Report RR-4223

[17] Garavel, H., Lang, F.: Equivalence Checking 40 Years After: A Review of Bisimulation Tools. In: Jansen, N., Stoelinga, M., van den Bos, P. (eds.) A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 13560, pp. 213–265. Springer (2022). https://doi.org/10.1007/978-3-031-15629-8_13

[18] Garavel, H., Lang, F., Mateescu, R.: Compositional Verification of Asynchronous Concurrent Systems Using CADP. Acta Informatica **52**(4), 337–392 (Apr 2015)

[19] Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT) **15**(2), 89–107 (Apr 2013)

[20] Garcia-Molina, H.: Elections in a Distributed Computing System. IEEE Transactions on Computers **31**(1), 48–59 (1982). https://doi.org/10.1109/TC.1982.1675885

[21] van Glabbeek, R.J., Weijland, W.P.: Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam (1989), also in proc. IFIP 11th World Computer Congress, San Francisco, 1989

[22] van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in

Bisimulation Semantics. Journal of the ACM **43**(3), 555–600 (1996)

[23] Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.: An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. ACM Trans. Comput. Log. **18**(2), 13:1–13:34 (2017). https://doi.org/10.1145/3060140

[24] Groote, J.F., Ponse, A.: The Syntax and Semantics of μCRL. CS-R 9076, Centrum voor Wiskunde en Informatica, Amsterdam (1990)

[25] Groote, J.F., Vaandrager, F.: An efficient algorithm for branching bisimulation and stuttering equivalence. In: Patterson, M.S. (ed.) Proceedings of the 17th ICALP (Warwick). Lecture Notes in Computer Science, vol. 443, pp. 626–638. Springer (1990)

[26] ISO/IEC: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva (Sep 1989)

[27] ISO/IEC: Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva (Sep 2001)

[28] Jansen, D.N., Groote, J.F., Keiren, J.J.A., Wijs, A.: An $\mathcal{O}(m \ log \ n)$ algorithm for branching bisimilarity on labelled transition systems. In: Biere, A., Parker, D. (eds.) Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20), Dublin, Ireland. Lecture Notes in Computer Science, vol. 12079, pp. 3–20. Springer (2020)

[29] Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: Donaldson, A.F., Torlak, E. (eds.) 41st International Conference on Programming Language Design and Implementation (PLDI), London, UK. pp. 227–242. ACM (2020). https://doi.org/10.1145/3385412.3385980

[30] Lang, F.: EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In: Romijn, J., Smith, G., van de Pol, J. (eds.) Proceedings of the 5th International Conference on Integrated Formal Methods (IFM'05), Eindhoven, The Netherlands. Lecture Notes in Computer Science, vol. 3771, pp. 70–88. Springer (Nov 2005), full version available as INRIA Research Report RR-5673

[31] Lang, F., Mateescu, R., Mazzanti, F.: Compositional verification of concurrent systems by combining bisimulations. In: McIver, A., ter Beek, M. (eds.) Proceedings of the 23rd International Symposium on Formal Methods — 3rd World Congress on Formal Methods (FM'19), Porto, Portugal. Lecture Notes in Computer Science, vol. 11800, pp. 196–213. Springer (2019)

[32] Lang, F., Mateescu, R., Mazzanti, F.: Sharp congruences adequate with temporal logics combining weak and strong modalities. In: Biere, A., Parker, D. (eds.) Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems

(TACAS'20), Dublin, Ireland. Lecture Notes in Computer Science, vol. 12079, pp. 57–76. Springer (Apr 2020)

[33] Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer (May 2008)

[34] Mateescu, R., Wijs, A.: Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. Sci. Comput. Program. **96**(3), 354–376 (2014)

[35] Milner, R.: Communication and Concurrency. Prentice-Hall (1989)

[36] Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) Theoretical Computer Science. Lecture Notes in Computer Science, vol. 104, pp. 167–183. Springer (Mar 1981)

[37] Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3794–3800. IEEE, Daejeon, South Korea (2016). https://doi.org/10.1109/IROS.2016.7759558

[38] de Putter, S., Lang, F., Wijs, A.: Compositional model checking with divergence preserving branching bisimilarity is lively. Science of Computer Programming **196** (2020)

[39] Tarjan, R.E.: Depth First Search and Linear Graph Algorithms. SIAM Journal of Computing **1**(2), 146–160 (1972)

[40] Valmari, A.: Bisimilarity minimization in $\mathcal{O}(m\ log\ n)$ time. In: Franceschinis, G., Wolf, K. (eds.) Proceedings of Applications and theory of Petri nets (PETRI NETS) 2009. Lecture Notes in Computer Science, vol. 5606, pp. 123–142. Springer (2009)

[41] Vu, T.D.: Deciding orthogonal bisimulation. Formal Aspects of Computing **19**(4), 475–485 (2007)

[42] Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref - A symbolic bisimulation tool box. In: Graf, S., Zhang, W. (eds.) Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006. Lecture Notes in Computer Science, vol. 4218, pp. 477–492. Springer (2006). https://doi.org/10.1007/11901914_35