Compositional Verification of Priority Systems using Sharp Bisimulation

Luca Di Stefano, Frédéric Lang

Compositional Verification of Priority Systems using Sharp Bisimulation

Luca Di Stefano and Frédéric Lang

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP**, LIG, 38000 Grenoble, France

Abstract. Sharp bisimulation is a refinement of divergence-preserving branching (a.k.a. divbranching) 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 parallel composition and other process algebraic operators such as 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 yield priority to nor take 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 tool 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 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 systems

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] [14] provides languages for the description of such systems (in particular the language LNT [7], which inherits many features from process algebra), languages for the description of properties (in particular the temporal logic

** Institute of Engineering Univ. Grenoble Alpes

[1] http://cadp.inria.fr

MCL [28], 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 [13].

As such, sharp bisimulation [27] 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 [30] and divergence-preserving branching (a.k.a. divbranching) bisimulation [16, 17] are instances of sharp bisimulation, all actions being strong in the former, and all actions being not strong (i.e., weak) 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 parallel composition and many process algebraic operators, such as hide, cut, and rename, and can therefore be used compositionally. Sharp minimization was applied successfully to the verification problems of the RERS verification challenges[2] in 2019[3] and 2020, where it allowed properties not preserved by divbranching bisimulation to be verified on systems whose state graph 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 [27]:

- We consider systems, some actions of which may be preempted by others using an action priority operator, similar to the one of ACP [1]. It is well-known that strong bisimulation equivalence is a congruence for such an operator, but also that divbranching is not. We give sufficient conditions relating the set of strong actions and the priority rules, so that sharp bisimulation equivalence be a congruence for priority. We provide such a proof of congruence.
- We establish a precise relationship between sharp bisimulation and orthogonal bisimulation [2], orthogonal bisimulation equivalence being also a congruence for priority. We prove that if the internal action τ does not yield nor take priority over any other action, then sharp minimization cannot reduce less than orthogonal minimization. This improvement is another advantage of the parameterization on strong actions offered by sharp 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

[2] http://rers-challenge.org
[3] http://cadp.inria.fr/news12.html

has been continuously maintained, improved, and extended during the last three decades.

Overview. Processes and their compositions, sharp bisimulation, and our 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. Our 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.

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 $(\exists p' \in \Sigma_P, a \in A)\ p \xrightarrow{a} p'$.

LTS can be composed in parallel and their actions may be abstracted away using the parallel composition and action mapping defined below, of which hide, cut (also known as restriction), and rename are particular cases.

Definition 2 (Parallel composition of LTS). *Let P and Q be LTS 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 $a \in A$.*

Definition 3 (Action mapping). *Let $[\mathcal{A}]^{<\omega}$ denote the set of finite subsets of the set of actions $\mathcal{A}$. Let P be an LTS. An action mapping is a total function $\rho : A_P \to [\mathcal{A}]^{<\omega}$. We write $\rho(A_P)$ for the image of ρ, defined by $\bigcup_{a \in A_P} \rho(a)$. The application of action mapping ρ to the LTS P is written $\rho(P)$ and defined as the LTS $(\Sigma_P, \rho(A_P), \longrightarrow, init(P))$ where $\longrightarrow$ is defined as follows:*

$$\longrightarrow = \{(p, a', p') \mid (\exists a \in A_P)\ p \xrightarrow{a}_P p' \wedge a' \in \rho(a)\}$$

An action mapping ρ is admissible if $\tau \in A_P$ implies $\rho(\tau) = \{\tau\}$. We distinguish the following admissible action mappings:

- *The mapping $\rho(P)$ is called action hiding, written "**hide** A **in** P", if: $(\exists A \subseteq \mathcal{A} \setminus \{\tau\})\ (\forall a \in A \cap A_P)\ \rho(a) = \{\tau\} \wedge (\forall a \in A_P \setminus A)\ \rho(a) = \{a\}$.*
- *The mapping $\rho(P)$ is called action cut, written "**cut** A **in** P", if: $(\exists A \subseteq \mathcal{A} \setminus \{\tau\})\ (\forall a \in A \cap A_P)\ \rho(a) = \emptyset \wedge (\forall a \in A_P \setminus A)\ \rho(a) = \{a\}$.*
- *The mapping $\rho(P)$ is called action renaming, written "**rename** f **in** P", if: $(\exists f : A_P \to \mathcal{A})\ ((\forall a \in A_P)\ \rho(a) = \{f(a)\}) \wedge (\tau \in A_P \Rightarrow f(\tau) = \tau)$.*

Parallel composition and action mapping subsume all abstraction and composition operators encodable as *networks of LTS* [32, 13, 25], such as synchronization vectors and the parallel composition, hide, cut, and rename operators of CCS [29], CSP [5], mCRL [19], LOTOS [21], E-LOTOS [22], and LNT [7].

2.2 Sharp Bisimulation

LTS can be compared and reduced modulo bisimulation relations between states. We consider the family of sharp bisimulations [27] defined hereafter, which is a general definition subsuming strong [30], branching, and divergence-preserving branching [16, 17] (a.k.a. divbranching) bisimulations.

Definition 4 (Sharp bisimulation [27]). *A divergence-unpreserving sharp bisimulation w.r.t. a set of actions A_s 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. *$q \xrightarrow{a} q'$, 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 (simply called sharp 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. divergence-unpreserving sharp) bisimilar w.r.t. A_s, written $p \sim_{\sharp A_s} q$ (resp. $p \sim^{\dashv}_{\sharp A_s} q$), if and only if there exists a sharp (resp. divergence-unpreserving sharp) bisimulation R w.r.t. A_s such that $(p, q) \in R$.

A sharp bisimulation is a hybrid between a strong and a divbranching bisimulation, characterized by the set of actions A_s called 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 τ-transition. By contrast, a transition labelled by a non-strong (called weak) action can be delayed under the same condition as divbranching bisimulation.

Sharp bisimulation generalizes results that are well-known in the framework of strong and divbranching bisimulations [27]. For all sets of strong actions A_s and A'_s:

- $\sim_{\sharp A_s}$ (resp. $\sim^{\dashv}_{\sharp A_s}$) is an equivalence relation. The quotient of an LTS P w.r.t. this equivalence relation 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 divergence-unpreserving sharp bisimulation coincide with strong bisimulation, whose equivalence is written $\sim$.
- If $A_s = \emptyset$ (i.e., all actions are weak), then sharp bisimulation (resp. divergence-unpreserving sharp bisimulation) coincides with divbranching (resp. branching) bisimulation, whose equivalence is written $\sim_{dbr}$ (resp. $\sim_{br}$).
- The set of sharp (resp. divergence-unpreserving sharp) bisimulation equivalences equipped with set inclusion actually forms a complete lattice whose supremum is divbranching (resp. branching) bisimulation equivalence and whose infimum is strong bisimulation equivalence.
- If $A'_s \subset A_s$ then $\sim_{\sharp A_s}$ (resp. $\sim^{\dashv}_{\sharp A_s}$) is strictly stronger than $\sim_{\sharp A'_s}$ (resp. $\sim^{\dashv}_{\sharp A'_s}$), hence generalizing that strong bisimulation equivalence is strictly stronger than (div)branching. As a consequence, an LTS minimized for $\sim_{\sharp A'_s}$ (resp. $\sim^{\dashv}_{\sharp A'_s}$) cannot be larger than the same LTS minimized for $\sim_{\sharp A_s}$ (resp. $\sim^{\dashv}_{\sharp A_s}$), i.e., the less strong actions, the greater the reduction.
- $\sim_{\sharp A_s}$ (resp. $\sim^{\dashv}_{\sharp A_s}$) is a congruence for parallel composition and admissible action mapping. It follows that it is also a congruence for hide, cut, and rename. Minimization can thus be applied compositionally through these operators. Note that, as (div)branching, $\sim_{\sharp A_s}$ and $\sim^{\dashv}_{\sharp A_s}$ are not congruences for the choice operator, unless all initial actions of the choice are strong actions. In particular, strong bisimulation equivalence is a congruence for the 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[4], 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[5] provides more chances to reduce the system efficiently. See [26, 27] for details.

[4] Adequacy of an equivalence and a logic means that two LTS are equivalent if and only if they verify exactly the same formulas of the logic.

[5] It was shown that such a smallest fragment is not unique [26]. Also, deciding whether a formula belongs to some fragment $L_\mu^{strong}(A_s)$ is still an open problem.

2.3 Action Priority

A classification of priority operators in process algebra is given in [8]. 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 [25] distributed in the CADP toolbox. Strong and orthogonal bisimulation equivalences [2] are congruences for this operator, which has lots of similarities with the priority operator of ACP [1].

Definition 5. *We consider action formulas α, whose syntax and semantics w.r.t. a set of actions A are defined as follows:*

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

Definition 6. *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 π, we associate the 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 which take priority over some other labels, and* $\mathrm{lesser}_A(\pi)$ *for the set $[\![\alpha']\!]_A$ of labels of A which yield 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:*

$$\begin{array}{c} \mathrm{greater}_A(\Omega) = \bigcup_{\pi_i \in \Omega} \mathrm{greater}_A(\pi_i) \\ \mathrm{lesser}_A(\Omega) = \bigcup_{\pi_i \in \Omega} \mathrm{lesser}_A(\pi_i) \end{array}$$

Example 1. Let A be the set of actions $\{a, b, c, d, e, f\}$ and Ω be the priority set $\{a \succ b, b \succ c \vee d, d \succ \neg(a \vee b \vee c \vee d \vee e)\}$. We have $\mathrm{greater}_A(\Omega) = \{a, b, d\}$, and $\mathrm{lesser}_A(\Omega) = \{b, c, d, f\}$. Having $\mathrm{greater}_A(\Omega) \cap \mathrm{lesser}_A(\Omega) \neq \emptyset$ is allowed, as in this example.

Lemma 1. *The following propositions hold:*

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

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

3 Sharp Congruences for Action Priority

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

Example 2. Consider the network "**prio** $a \succ b$ **in** $P \,|[\emptyset]|\, Q$", where a and b are visible actions, P consists of 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 the following:

$$\begin{array}{ccccc} (p_0,q_0) & \xrightarrow{\tau} & (p_1,q_0) & \xrightarrow{a} & (p_2,q_0) \\ \downarrow{\scriptstyle b} & & & & \downarrow{\scriptstyle b} \\ (p_0,q_1) & \xrightarrow{\tau} & (p_1,q_1) & \xrightarrow{a} & (p_2,q_1) \end{array}$$

P is divbranching equivalent to P', which consists of the single transition $p'_0 \xrightarrow{a} p'_1$. The LTS corresponding to "**prio** $a \succ b$ *in* $P' \,|[\emptyset]|\, Q$" consists of 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 divbranching equivalent to (p'_0,q_0), from which a can only be fired before b.

Theorem 1 below expresses however that 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 yield 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'. *The same holds when replacing* $\sim_{\sharp A_s}$ *by* $\sim^{\neg}_{\sharp A_s}$.

Proof. The key of the proof is that a state has a prioritized transition if and only if every $\sim_{\sharp A_s}$ bisimilar state also has a similar transition with same label, as this label belongs to A_s.

Formally, let R be a sharp bisimulation (resp. divergence-unpreserving sharp bisimulation) w.r.t. A_s between P and P'. We show that R is also a sharp bisimulation (resp. divergence-unpreserving 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 $a' \in greater_A(\Omega)$, 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 $p'_0 \xrightarrow{a'} p'_2$ neither.

By definition of 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 $a' >_{\Omega,A} a$ neither. "**prio** Ω **in** P'" thus has the transition $q_n \xrightarrow{a} p'_1$.

Finally, if R is divergence-preserving 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 (or divergence-unpreserving sharp) equivalent LTS, then we will get as result an LTS that is sharp (or divergence-unpreserving sharp) equivalent (hence divbranching or 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 $\sim_{\sharp\{a\}}$ and $\sim^{\dashv}_{\sharp\{a\}}$. They are therefore also minimal for any sharp bisimulation equivalence such that 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 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 [2], 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.

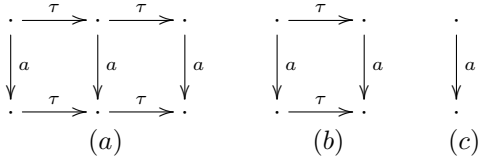
Definition 7 (Orthogonal bisimulation [2]). *An orthogonal bisimulation is a symmetric relation $R \subseteq \Sigma \times \Sigma$ such that if $(p, q) \in R$ then for all $p \xrightarrow{a} p'$, there exists q' such that $(p', q') \in R$ and either of the following holds:*

1. *$a \neq \tau$ and $q \xrightarrow{a} q'$, or*
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 $p \sim_{\perp}^{\dashv} q$, if and only if there exists an orthogonal bisimulation R such that $(p, q) \in R$.

Divergence-preserving orthogonal bisimulation, written $\sim_{\perp}$, is also defined in [2], by adding the same divergence-preserving condition as divbranching and sharp bisimulations (see Definition 4).

The above definition shows that orthogonal bisimulation is 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 preserving at least one. It is not comparable with sharp bisimulation as can be shown by considering the three LTS below:



Orthogonal minimization of LTS (a) results in LTS (b). However, (divergence-preserving and divergence-unpreserving) 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 yield 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) if and only if (1) $R \in \sim_{\sharp \mathcal{A} \setminus \{\tau\}}^{\dashv}$ (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. More formally (we skip the divergence-preserving case, which is an obvious consequence of the proof below):

$$
\begin{aligned}
& R \in \sim^{\dashv}_{\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}) \\
& \qquad \text{by definition of } \sim^{\dashv}_{\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 \\
& (a = \tau \wedge (\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}) \\
& \qquad \text{because } \tau \notin \mathcal{A}\setminus\{\tau\} \text{ holds and } a \notin \mathcal{A}\setminus\{\tau\} \text{ implies } 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 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}) \\
& \qquad \text{by factoring both cases where } 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 n > 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}) \\
& \qquad \text{by reworking index } 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 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}) \\
& \qquad \text{by combining } q' = q \text{ with } (\exists n > 0)\ \ldots \\
\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 ((\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}) \\
& \qquad \text{because } q \xrightarrow{\tau} q' \text{ is covered by } n = 1 \text{ in } (\exists n \geq 0)\ \ldots \\
\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 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 \\
& \qquad \text{by moving } p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau} \text{ under the case } a = \tau \\
\Leftrightarrow\ & R \in \sim^{\dashv}_{\perp} \\
& \qquad \text{by definition of } \sim^{\dashv}_{\perp}
\end{aligned}
$$

As orthogonal bisimulation equivalence requires every visible action to be strong, a consequence is that it is strictly stronger than any divergence-un-

preserving 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.

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 the LTS $P_0, \ldots, P_n$ can be reduced with respect to $\sim_{\sharp A_s \cup A_>}$ beforehand, where $A_> = \text{greater}_A(\Omega)$ and $A = A_{P_1} \cup \ldots \cup A_{P_n}$. Note however that this requires $\tau \notin \text{lesser}_A(\Omega)$. Otherwise, one has to use either strong bisimulation $\sim$ or divergence-preserving 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 also hold when replacing $\sim_{\sharp A_s}$, $\sim_{\sharp A_s \cup A_>}$, and $\sim_\perp$ by their divergence-unpreserving variants $\sim^\dashv_{\sharp A_s}$, $\sim^\dashv_{\sharp A_s \cup A_>}$, and $\sim^\dashv_\perp$, respectively. They are summarized in Figure 1. 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 [12] of CADP.

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	–	$\sim_{\sharp A_s \cup A_>}$
$\sim^\dashv_{\sharp A_s}$	yes	–	yes	$\sim$
	yes	–	no	$\sim^\dashv_\perp$
	no	yes	yes	$\sim^\dashv_{\sharp A_s \cup A_>}$
	no	yes	no	$\sim^\dashv_{\sharp A_s \cup A_>}, \sim^\dashv_\perp$
	no	no	–	$\sim^\dashv_{\sharp A_s \cup A_>}$
$\sim_\perp$	–	–	–	$\sim_\perp$
$\sim^\dashv_\perp$	–	–	–	$\sim^\dashv_\perp$

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 $A_> = \text{greater}_A(\Omega)$).

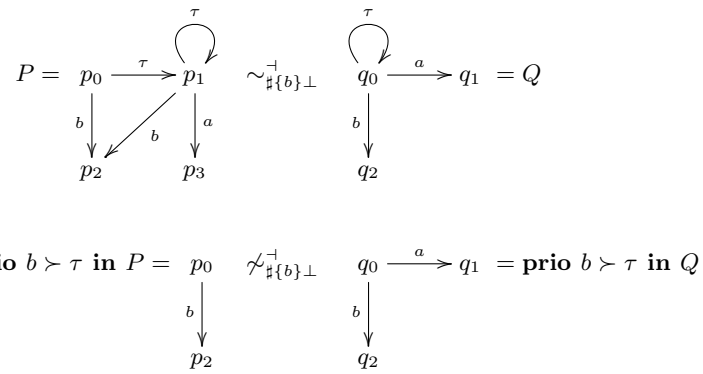


Fig. 1. Counter-example showing that combining sharp and orthogonal bisimulations does not allow the condition $\tau \notin \text{lesser}_A(\Omega)$ to be removed from Theorem 1.

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 $\sim^{\dashv}_{\sharp A_s \perp}$ (and its divergence-preserving variant), which combines sharp bisimulation and orthogonal bisimulation as given by the following definition:

$$R \in \sim^{\dashv}_{\sharp A_s \perp} \Leftrightarrow\ R \in \sim^{\dashv}_{\sharp A_s} \wedge (\forall (p,q) \in R)\ p \xrightarrow{\tau} \Rightarrow q \xrightarrow{\tau}$$

would have the congruence property without requiring $\tau \notin \text{lesser}_A(\Omega)$. (Note that $\sim^{\dashv}_{\perp} = \sim^{\dashv}_{\sharp \mathcal{A} \setminus \{\tau\} \perp}$.) 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 yield priority to other actions? The answer is no, as shows the counter-example of Figure 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[6] $\{\Sigma\}$ 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 [3, 4] 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 $\mathcal{O}(mn^2)$ 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 $\mathcal{O}(m\ log\ n)$ for

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

both strong [33] and (div)branching [23] bisimulations), they are quite efficient in practice and suitable for distributed implementations[7]. In the CADP toolbox [14], 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 LTS, by testing whether the initial states of both LTS 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.

Below, we present 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 of a state is a set of ordered pairs, each consisting of a label and a block belonging to P. Its precise definition for strong, branching, and divbranching 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 cannot be split anymore, i.e., all states have the same signature, $split_R(B, P)$ returns B itself. 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 they will remain unchanged. However, we do not detail such improvements further as they are not specific to sharp bisimulation.

[7] The difference between $\mathcal{O}(m \ log \ n)$ and $\mathcal{O}(mn^2)$ may seem big, and it is, indeed. However, $\mathcal{O}(mn^2)$ 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

Definition 8. *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:*

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

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 a-transition to some state that belongs to block B of partition P.

The following are the signatures of a state for respectively strong, branching, and divbranching bisimulations:

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

In the case of (div)branching bisimulation, computing the signature efficiently in practice (i.e., with a complexity linear in the number of states) 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 circuits of τ-transitions. However, one can observe that all states in the same circuit of τ-transitions are (div)branching bisimilar. Therefore, before starting to minimize an LTS, Tarjan's SCC algorithm is used for two purposes:

- All states in a circuit of τ-transitions are compressed into a single representative state. In the case of divbranching bisimulation, a self τ-loop is added to the representative state to reflect the existence of a circuit. This allows the condition $\tau^\omega_{B,P}(s)$ to be replaced by $(\exists s' \in \Sigma)\ \tau^*_{B,P}(s,s') \wedge \tau_{B,P}(s',s')$.

– States are then ordered within the initial block, so that whenever there is a transition $s \xrightarrow{\tau} s'$, then either $s = s'$ or s' occurs before s. When splitting a block, this order is maintained in the created subblocks. The signatures of states in a block are computed in 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 computed before starting to compute that of s. Therefore, the signature of s just has to be extended with the signatures of its immediate inert successors (such as s') without having to traverse the LTS further.

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

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

As for branching and divbranching bisimulations, we must take care how we compute the transitive closure of the relation τ_P. However, it is not possible to compress circuits of τ-transitions beforehand, because states on such circuits are not necessarily bisimilar, as illustrated below.

Example 4. The following LTS 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 circuit of τ-transitions but are not $\sim_{\sharp\{a\}}$ bisimilar because p'_0 is source of a transition labelled by strong action a, whereas p'_2 is not.

Instead, the Tarjan's SCC algorithm is used whenever the signatures of a block need to be updated (when computing the *split* operation), to traverse inert transitions. Details are presented in Figure 2, auxiliary functions *extend_sig_immediate*, *extend_sig_rep*, and *extend_sig_scc* being defined in Figure 3. The function call *signatures_sharp*(B,P) computes the function *sig* that returns the signature $sig(s)$ of each state s in block B, corresponding to $sig_{\sim_{\sharp A_s}}(s,P)$. Underlined statements are specific to the computation of signatures, whereas the remainder of the code is the standard Tarjan's SCC algorithm. The algorithm performs the following steps:

1. When a new SCC is found, its states are on the so-called SCC stack of the Tarjan's SCC algorithm. One of these states (the bottommost one s_0) is defined as its SCC representative.

2. Then, the states of the SCC are popped one after the other from the SCC stack. When popping a state s: (a) all ordered pairs (a, B') such that $s \xrightarrow{a} B'$ (not inertly if $\tau \notin A_s$) are added to the signature of the current state s, (b) all ordered pairs (a, B') such that $a \notin A_s$ and $s \xrightarrow{a} B'$ are added to the signature of the SCC representative s_0, (c) 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 as s are added to the signature of the SCC representative s_0, and (d) if s has an inert transition internal to the SCC, then a divergence is added to the signature of s_0. Step (a) is achieved by a call to function *extend_sig_immediate*(...), step (b) is achieved by a call to function *extend_sig_inert*(...), and steps (c) and (d) are achieved by a call to function *extend_sig_rep*(...), which may call functions *extend_sig_inert*(...) and *extend_sig_div*(...). To implement divergence-unpreserving sharp minimization, the call to *extend_sig_div*(...) just has to be skipped.
3. Finally, once all states of the SCC have been popped from the SCC stack, then the signature of the SCC representative s_0 has accumulated all its successors. This is not the case for the other states of the SCC, if any. Therefore, if the SCC has more than one state, then the components (a, B') of s_0 satisfying $a \notin A_s$ are added to the signatures of the other states of the SCC. This step is achieved by the function call *extend_sig_scc*(...).

5.3 Performance

This sharp minimization algorithm was implemented in the tool BCG_MIN and released since CADP version 2021-f "Saarbruecken" (May 2021). Note that we implemented only the divergence-preserving variant of sharp bisimulation so far, because we did not feel the need for divergence-unpreserving sharp minimization.

To assess the performance of this implementation, we made several measurements on a computer with Intel Core i5 quadri-processor running at 2.5 GHz using 16 GB of RAM, running GNU/Linux.

First, we applied sharp minimization to the 180 parallel problems of the RERS 2019 challenge[8]. We compared the obtained LTS sizes to those obtained initially using the partial reduction algorithm presented in [27], which was applied compositionally. On average, the final LTS obtained using sharp minimization has 7.69 times fewer states and 11.99 times fewer transitions than using partial sharp reduction. The maximum is for problem 106#16, where the final LTS obtained using sharp 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. We also checked (without surprise) that the RERS 2019 properties were preserved by minimization, as theoretically expected.

Second, we compared the memory and time consumed by sharp minimization in the two particular cases where the set of strong actions is either empty ($\emptyset$) or maximal ($\mathcal{A}$) with the (respectively equivalent in terms of resulting LTS)

[8] http://www.rers-challenge.org/2019/

```
function signatures_sharp(B, P) is
  scc_stack, dfs_stack, index_count := ∅, ∅, 0;
  foreach s ∈ B loop index(s), rep(s), sig(s) := −1, s, ∅ end loop;
  foreach s ∈ B loop
    if index(s) < 0 then
      start:
        index(s), lowlink(s) := index_count, index_count;
        index_count := index_count + 1;
        push(s, scc_stack);
        succ := {s′ | τ_P(s, s′)};
      continue:
        if succ ≠ ∅ then
          s′ := any state where s′ ∈ succ;
          succ := succ \ {s′};
          if index(s′) < 0 then
            push((s, s′, succ), dfs_stack);
            s := s′;
            goto start
          else if s ∈ scc_stack and index(s′) < lowlink(s) then
            lowlink(s) := index(s′)
          end if;
          goto continue
        end if;
        if lowlink(s) = index(s) then
          scc, s_0 := ∅, s;
          loop -- states above s_0 on scc_stack constitute s_0's SCC
            s := pop(scc_stack);
            rep(s), scc := s_0, scc ∪ {s};
            sig := extend_sig_immediate(sig, s, B, P);
            sig := extend_sig_inert(sig, s_0, s);
            sig := extend_sig_rep(sig, s_0, s, rep, scc_stack, B, P);
          until s = s_0 end loop;
          sig := extend_sig_scc(sig, s_0, scc);
        end if;
        if dfs_stack ≠ ∅ then
          (s, s′, succ) := pop(dfs_stack);
          if lowlink(s′) < lowlink(s) then lowlink(s) := lowlink(s′) end if;
          goto continue
        end if
    end if
  end loop;
  return sig
end function
```

Fig. 2. Sharp signature computation (underlined) based on Tarjan's algorithm

function *extend_sig_immediate*(sig, s, B, P) **is**
  -- *extend s's signature with (non-inert) immediate actions*
  $sig(s)$:= $sig(s) \cup \{(a, B') \mid s \xrightarrow{a}_P B' \wedge (\tau \in A_s \vee a \neq \tau \vee B' \neq B)\}$;
  return *sig*
end function

function *extend_sig_inert*(sig, s, s') **is**
  -- *extend s's signature with weak actions of s' 's signature,*
  -- *knowing* $\tau^*_P(s, s')$
  if $s \neq s'$ **then**
    $sig(s)$:= $sig(s) \cup \{(a, B) \mid (a, B) \in sig(s') \wedge a \notin A_s\}$
  end if;
  return *sig*
end function

function *extend_sig_div*(sig, s, B) **is**
  -- *extend s's signature with a divergence in block B*
  $sig(s)$:= $sig(s) \cup \{(\tau, B)\}$;
  return *sig*
end function

function *extend_sig_rep*$(sig, s_0, s, rep, scc\_stack, B, P)$ **is**
  -- *extend* s_0*'s signature with weak actions reachable by*
  -- *inert transitions from s, knowing* s_0 *and s in same SCC*
  foreach s' **such that** $\tau_P(s, s')$ **loop**
    -- $s_0 \xrightarrow{\tau^*} s'$ *inert*
    if $rep(s') = s_0 \vee s' \in scc\_stack$ **then**
      -- s_0 *and* s' *are in same SCC: divergence*
      sig := *extend_sig_div*(sig, s_0, B)
    else
      -- s_0 *and* s' *are not in same SCC*
      sig := *extend_sig_inert*(sig, s_0, s')
    end if
  end loop;
  return *sig*
end function

function *extend_sig_scc*(sig, s_0, scc) **is**
  -- *extend signature of states strongly connected to* s_0
  -- *with weak actions once* s_0*'s signature is up to date*
  foreach $s \in scc \setminus \{s_0\}$ **loop**
    -- $s \xrightarrow{\tau^*} s_0$ *inert*
    sig := *extend_sig_inert*(sig, s, s_0)
  end loop;
  return *sig*
end function

Fig. 3. Functions used in Figure 2

	$\sim$ 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 $\sim_{\sharp\mathcal{A}_f}$ minimization vs. dedicated $\sim$ minimization

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's SCC algorithm to compute state signatures each time a block has been split.

We applied this comparison to four different packages of LTS accumulated over time, represented in the BCG format of CADP: (1) a package BCG1 comprising 8995 "small" LTS having less than $500,000$ transitions; (2) a package BCG2 comprising 542 "medium-size" LTS having between $500,000$ and $5,000,000$ transitions; (3) a package BCG3 comprising 265 "large" LTS having between $5,000,000$ and $50,000,000$ transitions; and (4) the public benchmark VLTS[9] comprising 40 LTS 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 LTS 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 sharp 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 LTS 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 rather small.

Our implementation of 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 divbranching minimization. From these experiments, we extrapolate that in such cases, the overhead of sharp minimization should be compensated enough by the gain on LTS sizes, compared to strong minimization.

Theorem 2 implies that orthogonal minimization can be implemented as a minor variation of $\sim^{\dashv}_{\sharp\mathcal{A}\setminus\{\tau\}}$ 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).

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

	$\sim_{dbr}$ 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

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 be decorated with a self-loop labelled by τ. 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 LTS, namely P_m $(m \geq 1)$ and $Q_{n,m}$ $(n \geq 0, m \geq 1)$ defined as follows:

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

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 $\sim_{\sharp\{a\}}$

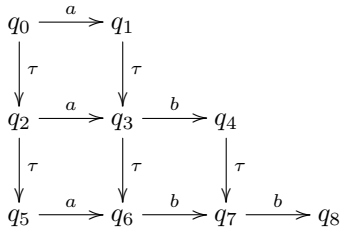
and can thus be eliminated. Since none of the LTS has divergences, taking the divergence-preserving or divergence-unpreserving variants of these equivalence relations does not matter.

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 $Q_{2,1}$. Using compositional sharp minimization, we obtain the following LTS:

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3$$

Using compositional orthogonal bisimulation, we obtain the following one, which is branching equivalent but has a stairs shape:



Similar stairs shapes can be observed in all LTS 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 transitions labelled by τ is inert with respect to orthogonal bisimulation:

- States q_2, q_3, and q_4 are not orthogonally equivalent to respectively q_5, q_6, and q_7, as each of the former is the source of a transition labelled by τ, whereas each of the latter is not.
- State q_1 is not orthogonally equivalent to q_3, as q_3 is the source of a transition labelled by b, whereas q_1 is not.
- Finally, state q_0 is not orthogonally equivalent to q_2 as the transition labelled by a going out of q_0 goes to state q_1, which we have just shown to be not orthogonally equivalent to q_3, the target of the transition labelled by a going out of q_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

As expected, both LTS 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.

		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 $\sim^{\dashv}_{\sharp\{a\}}$ minimization of $Q_{n,m}$

7 Case Study

As an example of a priority system, we consider a slight variation on the classic bully leader election algorithm [15]. 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; 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 [9], a simple language where agents cannot perform explicit message passing, but instead exploit an indirect communication mechanism based on *stigmergy variables* [31]. When an agent assigns a value to such a variable, the value is timestamped with the time of assignment.[10] Agents asynchronously broadcast values after performing an assignment, and receivers will replace their own value with the received one if the latter is *newer* (i.e., it has a higher timestamp). The semantics of LAbS assume that the broadcast and all potential value updates by the receivers happen atomically.

In this example, each agent stores the id of its current leader in a stigmergy variable ℓ. 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 $\ell > id$. Thus, 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 LTS, namely one per agent plus an additional one that stores information about the timestamps. For the scope of this work, we developed a new workflow in the SLiVER[11] tool [11, 10] that turns a LAbS specification into an LNT program with this network structure, shown in Listing 1.1. Intuitively, agents in this LNT program may perform `refresh` actions to obtain a fresh timestamp for the stigmergy variable ℓ; `l` actions to signal that they set ℓ to a new value; 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 `l` actions may be performed freely. The timestamp server performs an action `debug(...)` before every synchronization, to display its current internal state. Additionally, agents may synchronize to exchange stigmergy messages (over the

[10] 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.

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

Listing 1.1. 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 [...] (ID(0))
      ||
         Agent [...] (ID(1))
      ||
         Agent [...] (ID(n − 1))
      end par
      end hide
   end par
end process
```

`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.

With priorities, we can prevent these diamonds from occurring by only considering a single, representative sequence of events for each one of them. Since we are not interested in the intermediate states, this can be highly beneficial. Consider, for instance, the diamond in Fig. 4: 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 $\circ \xrightarrow{a_0} \circ \xrightarrow{a_1} \circ \xrightarrow{a_2} \circ$.

To demonstrate the effect of priorities on these systems more concretely, we consider two leader election systems `leader3` and `leader4`, containing 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 1.1). We then reduce S modulo strong, orthogonal, divbranching, and sharp bisimulation,

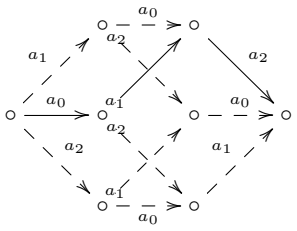


Fig. 4. Example of a diamond when 3 agents perform independent actions a_0, a_1, and a_2. Dotted transitions are cut by applying the priority relation $a_0 \succ a_1 \succ a_2$.

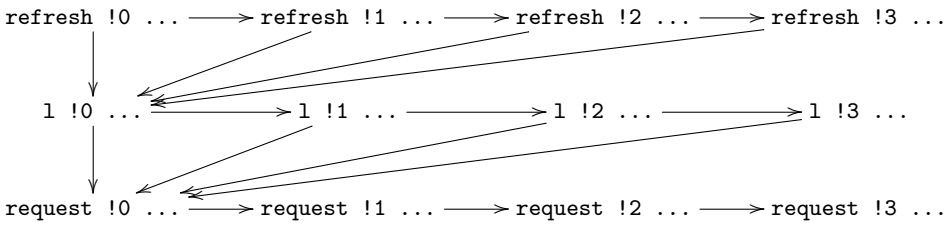


Fig. 5. A representation of the priority relation Ω used for the experiments in Table 6. An arrow $x \to y$ denotes that $x \succ y$.

obtaining four additional LTS $S_{\sim}$, $S_{\perp}$, S_{dbr}, and $S_{\sharp}$. For sharp bisimulation, we choose as set of strong actions A_s the set comprising all actions of the form "`refresh !.*`", "`l !.*`", and "`request !.*`".

We also generate an LTS $S^{\Omega} =$ `prio` Ω `in` S, and minimize it modulo the aforementioned bisimulations to obtain $S^{\Omega}_{\sim}$, $S^{\Omega}_{\perp}$, S^{Ω}_{dbr}, and $S^{\Omega}_{\sharp}$. The priority relation Ω used in this phase is visualized in Fig. 5, namely, `refresh !.*` $\succ$ `l !.*` $\succ$ `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. Notice that we merely use divbranching reduction as a theoretical optimum that sharp 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 one hand, we can observe that orthogonal bisimulation is slightly more effective than strong bisimulation. On the other hand, sharp minimization proves even more effective, producing LTS as small as the ones obtained by divbranching minimization (i.e., the optimum). Furthermore, since our choice of strong actions satisfies the premises of Theorem 1 with respect to the priority rules Ω, we also know $\sim_{\sharp A_s}$ to be a congru-

		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	
	$\sim_{\perp}^{\dashv}$	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	
	$\sim_{\perp}^{\dashv}$	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.

ence. Indeed, in our experiments CADP mechanically proves that $S^{\Omega} \sim_{\sharp A_s} S^{\Omega}_{\sharp}$ in both the 3- and the 4-agent systems. In both scenarios, memory consumption is dominated by the generation of the initial LTS S, which requires around 12.6 MB for `leader3` and 14.3 MB for `leader4`.

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* [12], 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 LTS in the network, then composing them to generate the LTS of the network (applying priority rules on the fly), and finally performing one last minimization of the composite LTS. In our experiments, we carry out minimizations using either orthogonal or sharp 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. Then, we measure the size of the composite LTS (`Main`), before and after the last minimization. We do the same for the system with priorities (`prio` Ω `in Main`). We also report the time and memory needed to generate each LTS with sharp 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` LTS. 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

Several approaches for priority in process languages have been proposed since the early 80s. The interested reader may have a look at the introduction of [8] for a classification. Our aim in this paper is not to promote one approach rather than 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 15 years ago.

Another congruence for parallel composition and priority in a setting close to ours is prioritized weak bisimulation equivalence [8], which is based upon weak (a.k.a. observational) bisimulation [29] rather than (div)branching. Weak bisimulation equivalence is known to have less efficient algorithms than (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 circuit of τ transitions are not necessarily or-

		LTS		$\sim_{\perp}^{\dashv}$		$\sim_{\sharp A_s}$			
		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.

thogonally bisimilar. This issue was addressed by Vu [34], whose algorithm, similarly to ours, also relies on a linear-time procedure for computing the strongly connected components 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 [20]. The worst-case time complexity $\mathcal{O}(m.n)$ 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 $\mathcal{O}(m.n^2)$ 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[12], see e.g., [18] (Fig. 9, right). Unfortunately, we could not find any implementation of Vu's algorithm. Orthogonal bisimulation is also implemented in the tool Sigref [35], 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 [6].

[12] 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.

Going further on complexity, there exists a recent algorithm for (div)branching minimization, with worst-case time complexity $\mathcal{O}(m \log n)$ [23]. The question whether this algorithm could be adapted to implement sharp minimization while keeping its complexity is open.

Inductive sequentialization (IS) [24] is a technique where an asynchronous program is analyzed 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 LTS that encode a concurrent system (e.g., the leader election of Section 7). Both approaches still require a certain amount of creative work. In IS one has to come up with an idealized sequential execution, while our approach requires finding an adequate priority set, and a corresponding set of strong actions for sharp reduction.

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 of non-prioritized systems. In this paper, we extended the set of fundamental results on sharp bisimulation equivalence, by showing that it is also a congruence for action priority operators. Therefore, it is also appropriate to verify compositionally systems with priority. 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 circuits of τ-transitions to be eliminated beforehand, unlike (div)branching. Instead, blocks must be traversed using a linear-time algorithm for detecting strongly-connected components (we used the famous one by Tarjan) at each partition-refinement step. Our results are implemented in tools in CADP, namely BCG_MIN and BCG_CMP for 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 that shows gains of several orders of magnitude on state space size that can be potentially obtained using sharp bisimulation rather than 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 sharp bisimulation equivalence offers more reduction than strong and orthogonal bisimulations, and even as much reduction as divbranching bisimulation, which is the maximal reduction that can be obtained in our setting. Yet, contrary to divbranching bisimulation, sharp bisimulation offers all theoretical guarantees for the resulting state space to preserve the semantics of the non-reduced system. The effect of 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

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.[13] The authors would like to warmly thank David Jansen who pointed to us a possible relation between orthogonal and sharp bisimulations during a presentation of [27] at TACAS'2021.

References

1. Baeten, J., Bergstra, J., Klop, J.: Syntax and defining equations for an interrupt mechanism in process algebra. Fundamenta Informaticae **IX**, 127–168 (1986)
2. Bergstra, J.A., Ponse, A., van der Zwaag, M.B.: Branching time and orthogonal bisimulation equivalence. Theoretical Computer Science **309**(1–3) (2003)
3. Blom, S., Orzan, S.: A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. Software Tools for Technology Transfer **7**(1), 74–86 (2005)
4. Blom, S., Orzan, S.: Distributed State Space Minimization. Software Tools for Technology Transfer **7**(3), 280–291 (2005)
5. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. J. ACM **31**(3), 560–599 (Jul 1984)
6. 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, https://doi.org/10.1145/123186.123223
7. 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
8. 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)
9. 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
10. 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). LNCS, vol. 13036, pp. 473–489. Springer (2021). https://doi.org/10.1007/978-3-030-89159-6_29

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

11. 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). LNCS, vol. 12134, pp. 370–385. Springer, Valletta, Malta (Jun 2020). https://doi.org/10.1007/978-3-030-50029-0_23
12. 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
13. Garavel, H., Lang, F., Mateescu, R.: Compositional Verification of Asynchronous Concurrent Systems Using CADP. Acta Informatica **52**(4), 337–392 (Apr 2015)
14. 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)
15. 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
16. 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
17. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM **43**(3), 555–600 (1996)
18. 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, https://doi.org/10.1145/3060140
19. Groote, J., Ponse, A.: The Syntax and Semantics of μCRL. CS-R 9076, Centrum voor Wiskunde en Informatica, Amsterdam (1990)
20. Groote, J., 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)
21. 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)
22. ISO/IEC: Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva (Sep 2001)
23. Jansen, D.N., Groote, J.F., Keirena, 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 Confrence on Tools and Algorithms for the Construction and Analysis of Systems TACAS 2020 (Dublin, Ireland), held online in 2021. Lecture Notes in Computer Science, vol. 12079, pp. 3–20. Springer (2020)
24. 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 De-

sign and Implementation (PLDI). pp. 227–242. ACM, London, UK (Jun 2020). https://doi.org/10.1145/3385412.3385980
25. 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
26. 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)
27. 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)
28. 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)
29. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
30. 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)
31. 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 (Oct 2016). https://doi.org/10.1109/IROS.2016.7759558
32. de Putter, S., Lang, F., Wijs, A.: Compositional model checking with divergence preserving branching bisimilarity is lively. Science of Computer Programming **196** (2020)
33. 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)
34. Vu, T.D.: Deciding orthogonal bisimulation. Formal Aspects of Computing **19**(4), 475–485 (2007)
35. 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, https://doi.org/10.1007/11901914_35