

Bisimulation Learning

Alessandro Abate¹, Mirco Giacobbe², and Yannik Schnitzer¹

¹ University of Oxford, UK

{alessandro.abate,yannik.schnitzer}@cs.ox.ac.uk

² University of Birmingham, UK

m.giacobbe@bham.ac.uk

Abstract. We introduce a data-driven approach to computing finite bisimulations for state transition systems with very large, possibly infinite state space. Our novel technique computes stutter-insensitive bisimulations of deterministic systems, which we characterize as the problem of learning a state classifier together with a ranking function for each class. Our procedure learns a candidate state classifier and candidate ranking functions from a finite dataset of sample states; then, it checks whether these generalise to the entire state space using satisfiability modulo theory solving. Upon the affirmative answer, the procedure concludes that the classifier constitutes a valid stutter-insensitive bisimulation of the system. Upon a negative answer, the solver produces a counterexample state for which the classifier violates the claim, adds it to the dataset, and repeats learning and checking in a counterexample-guided inductive synthesis loop until a valid bisimulation is found. We demonstrate on a range of benchmarks from reactive verification and software model checking that our method yields faster verification results than alternative state-of-the-art tools in practice. Our method produces succinct abstractions that enable an effective verification of linear temporal logic without next operator, and are interpretable for system diagnostics.

Keywords: Data-driven verification · Stutter-insensitive bisimulation · Reactive verification · Software model checking · Abstraction

1 Introduction

Abstraction of state transition systems is the process for which a system under analysis—the concrete system—is reduced to another system—the abstract system—that is simpler to analyze and preserves certain temporal properties of the former [20,25,38,48]. It is a fundamental approach to state space reduction in the verification of finite-state systems and an essential element for the verification of infinite-state systems. Bisimulations are the abstractions that preserve linear and branching behaviour with respect to propositional observations, for which the model checking question for both linear- and branching-time logics have the same answer on the abstract and the concrete system [14,29].

Computing a bisimulation amounts to computing an equivalence relation on the state space that is stable with respect to a notion of state change, and preserves propositional observations. An equivalence relation defines a partition of

the concrete state space and induces an abstract system where every abstract state corresponds to an equivalence class. The problem of computing bisimulations over an explicit representation of the state graph has been widely studied in the past [5,32], since Hopcroft’s graph minimisation algorithm and the Paige-Tarjan algorithm for iterative partition refinement [31,46]. Partition refinement was improved with on-the-fly partition refinement of the reachable state space as well as parallelisation [21,35,36,37]. Yet, explicit-state algorithms fall short on systems with very large or infinite state space, for which one must resort to procedures that represent regions of state space symbolically [10].

Partition refinement relies on computing exact pre- and post-images through the transition function of the system [10,26]. This entails quantifier elimination, which is computationally costly. Counterexample-guided abstraction refinement (CEGAR) provides an approach to avoid pre- and post-image computation; it computes *simulations* of state transition systems incrementally, from infeasibility proofs of spurious counterexamples [17,30]. The resulting abstract system is tight enough to verify a specific property of interest, but cannot generally provide concrete counterexamples when a property is false and, for this purpose, methods based on CEGAR are usually coupled with bounded model checking [8]. Similarly, methods for temporal logic verification based on proof rules (i.e., certificates) provide sufficient conditions to verify whether a property holds but do not provide a counterexample when this is false [27,52,18,42,2]. By contrast, bisimulations provide a tight abstraction where abstract counterexamples correspond to concrete counterexamples and, as such, these are directly interpretable for system debugging and diagnostics.

We present a data-driven approach to computing finite bisimulations from sample states and transitions of the system, which skips partition refinement entirely. We adapt the notion of *well-founded bisimulations*, where the condition of stability of the equivalence relation with respect to stuttering is characterised as the existence of ranking functions over well-founded sets [43]. While originally introduced solely as a proof rule, we leverage well-founded bisimulations for the first time to directly compute finite bisimulations. We instantiate well-founded bisimulations with ranking functions that, for every state transition to a different state in the abstract system, map states to natural numbers that decrease strictly as the system stutters. This characterises *stutter-insensitive bisimulations* for deterministic transition systems and also applies to strong bisimulations, which is the special case of our method where ranking functions are constant.

Stutter-insensitive bisimulations are stable bisimulations with respect to observation change in the system, and is closed with respect to all state transitions between these changes. A system stutters when it changes concrete state without changing observation [33], and stutter-insensitive bisimulations abstract stuttering away. In contrast to strong bisimulations, stutter-insensitive bisimulations result in much more succinct abstractions, while being sufficiently strong to preserve the validity of any linear temporal logic specification without next operator. While our approach also applies to strong bisimulations, we generalise

our method to stutter-insensitive bisimulations, because they more effectively yield finite abstractions on infinite-state systems in practice.

We build on the observation that a finite partition can be characterized as a state classifier mapping the (possibly infinite) state space into a finite set of classes. This reduces the problem of computing a stutter-insensitive bisimulation to training a classifier and a ranking function for each class [51,45,22]. For the partition classifier, we employ binary a decision tree (BDT) with parametric linear predicates at each decision node, and we associate each leaf node with a parametric linear ranking function. This structure forms our template.

Our approach is underpinned by a learner and a verifier interacting with each other, both using a satisfiability modulo theory (SMT) solver. The learner proposes a candidate bisimulation by computing parameters of the classifier and ranking function templates to satisfy conditions over sampled transitions. The verifier then checks if these conditions hold over the *entire state space*. If affirmed, the classifier induces a stutter-insensitive bisimulation. If not, the verifier provides a counterexample, a state where stutter-insensitive bisimulation conditions are violated. This counterexample is fed back to the learner, which updates the classifier and ranking functions. The process repeats in a counterexample-guided inductive synthesis (CEGIS) loop until the verifier confirms the bisimulation’s validity [50]. If the template cannot fit the finite set of samples, for instance, due to an insufficient number of partitions, our procedure automatically enlarges the BDT with an additional layer and resumes the CEGIS loop.

We demonstrate the experimental efficacy of our approach on numerical programs and reactive software systems with integer state spaces. We consider benchmarks from reactive verification and software model checking, in particular discrete-time synchronisation protocols and conditional termination analysis problems. We benchmark the former set against the nuXmv model checker for reactive verification and the latter against the Ultimate and the CPAChecker tools for software verification [7,16,28]. The results are two-fold. For the reactive verification benchmarks, our approach has faster verification times than nuXmv on systems with long stuttering intervals. For the conditional termination benchmarks, our approach is able to generate exact preconditions for which the program terminates, unlike the baselines that return negative answers when the program does not terminate for at least one input. In summary, we demonstrate that, on these problems, our approach yields both faster and more informative results than the alternative state-of-the-art tools.

We summarise our contributions in the following three points: (1) we introduce the first data-driven approach to construct bisimulations, as an alternative approach to partition refinement; (2) we implement the theory of well-founded bisimulations which we synthesise in a CEGIS loop, as a means to compute stutter-insensitive bisimulations; (3) we demonstrate the efficacy of our novel approach on reactive verification and software model checking benchmarks. Our approach is fully automatic and requires no user input beyond the system itself. It produces succinct abstractions of infinite-state systems, which effectively enables their LTL (without next) verification using finite-state model checkers.

2 Illustrative Example

We motivate our procedure with an example from software model checking. Consider the code snippet in Figure 1a. The program takes two arbitrary integers as input and subtracts the smaller from the larger until the two values coincide. We ask the question of whether the program terminates for every initial condition, which is not straightforward to answer for this example. Given two positive inputs, the program runs the Euclidean algorithm for the greatest common divisor and terminates once it is found. However, for any two unequal non-positive inputs, this implementation will never exit the loop and run forever.

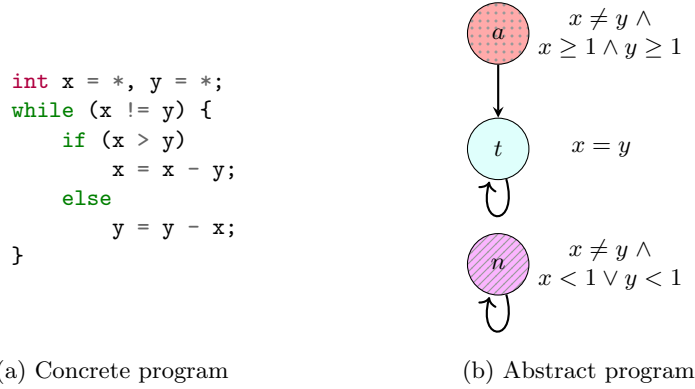


Fig. 1. Learned stutter-insensitive bisimulation of the Euclidean algorithm.

Our procedure solves the termination problem by iteratively learning parameters for a given state classifier template, such that its induced partition of the state space satisfies the stutter-insensitive bisimulation conditions over a finite set of sample transitions of the program. We ensure this by simultaneously computing parameters for given ranking function templates, which, together with the partition induced by the classifier, satisfy the equivalent conditions of a well-founded bisimulation. We leverage an SMT solver to check for counterexamples, i.e., states that are not equivalent to other states with the same class assigned by the classifier. These counterexample states are passed back to the learning procedure to update the classifier and the ranking function parameters until the SMT solver cannot generate a counterexample anymore and, thus, certifies that the learned classifier generalises to the entire infinite state space and induces a valid stutter-insensitive bisimulation.

Figure 2 illustrates the iterative update of the classifier with respect to the sampled program behaviour, given an initial partitioning of the state space into the class of *terminated* states violating the loop condition $x \neq y$ and the disjoint class of *not terminated* states. Upon termination the learned classifier correctly

separates the states into those for which both variables are positive and which will eventually reach a terminated state after stuttering for a finite number of steps and the states that infinitely stutter in the class of not-terminated states.

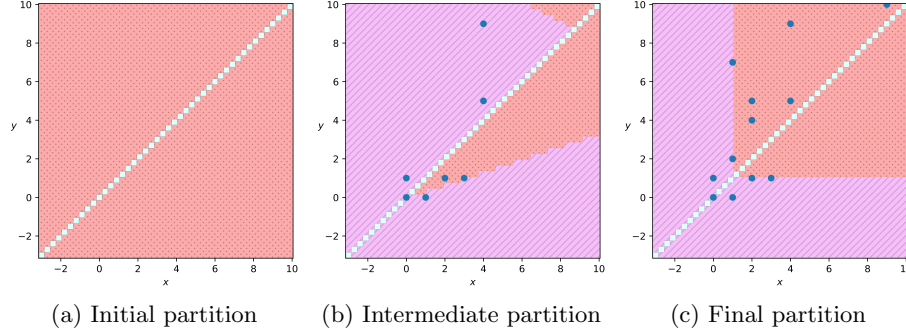


Fig. 2. Iterative process of bisimulation learning. Starting from the initial label-preserving partition (a), our procedure generates counterexamples (blue dots) until it attains a valid stutter-insensitive bisimulation (c).

In addition to the stutter-insensitive bisimulation, our procedure generates the corresponding abstract system by computing the behavior of the abstract states (i.e., the classes of the partition) alongside the classifier. Figure 1b shows the synthesized abstract system for the Euclidean algorithm, where each abstract state corresponds to an infinite subset of the concrete state space. The stutter-insensitive bisimulation ensures that the termination question has the same answer for all concrete states within the same class. A key advantage of our approach over methods providing a single counterexample is that it produces interpretable representations of the abstract system, aiding in system diagnostics. Specifically, our approach yields interpretable classifiers as binary decision trees. Figure 1b shows the abstract system and the automatically generated predicates defining the partition. Even for high-dimensional state spaces and complex partitions, this approach provides accessible means to interpret and diagnose the system for potential faults and undesired behavior [3,13].

3 Stutter-insensitive Bisimulations of Deterministic Transition Systems

We introduce the fundamental concepts underpinning our approach.

Definition 1 (Transition Systems). *A transition system $\mathcal{M}$ consists of*

- a state space S ,
- an initial region $I \subseteq S$, and

- a non-blocking transition function $T: S \rightarrow (2^S \setminus \emptyset)$.

We say that $\mathcal{M}$ is deterministic if $|T(s)| = 1$ for all $s \in S$. It is labelled when it additionally comprises

- a set of atomic propositions AP (the observables), and
- a labelling (or observation) function $\langle\!\langle \cdot \!\rangle\!\rangle: S \rightarrow 2^{AP}$.

A trajectory of $\mathcal{M}$ is any sequence of states $\tau = s_0, s_1, s_2, \dots$ such that $s_{i+1} \in T(s_i)$ for all consecutive s_i, s_{i+1} in τ . We say that τ is initialised if $s_0 \in I$.

Definition 2 (Partitions). A partition on $\mathcal{M}$ is an equivalence relation $\simeq \subseteq S \times S$ on S , which defines the quotient space $S/\simeq$ (i.e., the set of equivalence classes of $\simeq$) of pairwise-disjoint regions of S whose union is S .

Since we are interested in a notion of state equivalence insensitive to behaviour that does not change the observation of a state, the concept of divergence will be essential to distinguish between states that progress while not changing observation and those that do not progress at all [4,53].

Definition 3 (Divergence Sensitivity). Let $\simeq$ be a partition on $\mathcal{M}$. A state $s \in S$ is $\simeq$ -divergent if there exists an infinite trajectory $s_0, s_1, \dots$ such that $s_0 = s$ and $s_i \simeq s$ for all $i > 0$. Partition $\simeq$ is divergence-sensitive when $s \simeq t$ and s is $\simeq$ -divergent implies that t is $\simeq$ -divergent.

A partition of the state space induces a reduced transition system — the corresponding abstract system or *quotient*.

Definition 4 (Quotient). The quotient of $\mathcal{M}$ under the partition $\simeq$ is the transition system $\mathcal{M}/\simeq$ with

- state space $S/\simeq$,
- initial region $I/\simeq$ where $R \in I/\simeq$ iff $R \cap I \neq \emptyset$, and
- transition function $T/\simeq$ where
 1. $R \neq Q \in T/\simeq(R)$ iff $T(s) \in Q$ for some $s \in R$,
 2. $R \in T/\simeq(R)$ iff some $s \in R$ is $\simeq$ -divergent.

The quotient is the aggregation of equivalent states and their behaviours. The specifications preserved by the quotient, i.e., the statements that carry over from the abstract to the concrete system, depend on the properties of the underlying partition [46]. The most important property to preserve sensible specifications is that equivalent states must have equal observations.

Definition 5 (Label-preserving Partitions). A partition $\simeq$ on a labelled transition system is label-preserving when $s \simeq t$ implies $\langle\!\langle s \!\rangle\!\rangle = \langle\!\langle t \!\rangle\!\rangle$. The quotient $\mathcal{M}/\simeq$ of a labelled transition system $\mathcal{M}$ under a label-preserving partition $\simeq$ is labelled with the extended labelling function $\langle\!\langle \cdot \!\rangle\!\rangle: S \cup S/\simeq \rightarrow 2^{AP}$ where, for every region $R \in S/\simeq$, $\langle\!\langle R \!\rangle\!\rangle = \langle\!\langle s \!\rangle\!\rangle$ for any representative $s \in R$.

A standard notion of state equivalence on labelled transition systems is *bisimilarity* [39]. Bisimilarity preserves both linear- and branching-time behaviour by co-inductively requiring that every pair of related states can match each others' transitions with equivalent transitions. However, this stability with respect to stepwise behaviour often results in large quotients, thus limiting its suitability to facilitate reasoning over the system [46]. Therefore, we focus on *stutter-insensitive bisimulations* [14]. By abstracting from stepwise behaviour that does not change the observation of a state, stutter-insensitive bisimulations yield smaller quotients while preserving important specifications, as we will see in the following section.

Definition 6 (Stutter-insensitive Bisimulation). *A label-preserving partition $\simeq$ is a stutter-insensitive bisimulation if, for all states $s, s', t \in S$ such that $s \simeq t$ and $s \not\simeq s' \in T(s)$, there exists a finite trajectory $t_0, t_1, \dots, t_k$ such that $t_0 = t$, $t_i \simeq s$ for all $i = 1, \dots, k-1$, and $t_k = t'$ for some $t' \simeq s'$.*

Figure 3 illustrates the stability condition of stutter-insensitive bisimulations. This condition requires that for related states, transitions to unrelated states can be matched by finite trajectories that pass through the same equivalence class.

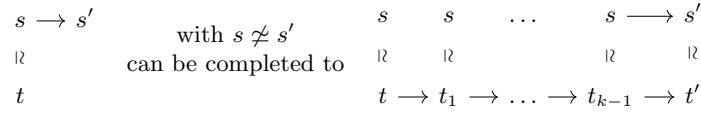


Fig. 3. Trajectory-based representation of the stutter-insensitive stability condition.

Lemma 1. *Every stutter-insensitive bisimulation on any deterministic labelled transition system admits a deterministic quotient.*

Proof. Let $\mathcal{M}$ be a deterministic transition system and $\simeq$ be a stutter-insensitive bisimulation on $\mathcal{M}$. Assume $\mathcal{M}/\simeq$ is nondeterministic, this implies that there exists pairwise distinct $R, Q, V \in S/\simeq$ such that $\{Q, V\} \subseteq T/\simeq(R)$. It follows that there exist $s, t \in R$ with $T(s) \in Q$ and $T(t) \in V$. Since $s, t \in R$ it holds that $s \simeq t$ and as $\mathcal{M}$ is deterministic and $Q \neq V$, $\simeq$ cannot satisfy Def. 6. $\square$ $\square$

3.1 Model Checking

We introduce Linear Temporal Logic without *next*-operator ($\text{LTL}_{\setminus \bigcirc}$) as a formal specification language for the temporal behaviour of a system and its states [4, 49]. $\text{LTL}_{\setminus \bigcirc}$ formulas are constructed according to the following grammar:

$$\varphi ::= \text{true} \mid p \mid \varphi \wedge \varphi \mid \neg \varphi \mid \varphi U \varphi$$

The model checking problem for $LTL_{\setminus \bigcirc}$ is to decide whether transition system $\mathcal{M}$ satisfies a given $LTL_{\setminus \bigcirc}$ formula φ , where the satisfaction relation $\models$ for trajectories of $\mathcal{M}$ is defined as

$$\begin{aligned} \tau, i &\models \text{true} \\ \tau, i &\models p && \text{iff } p \in \langle\langle s_i \rangle\rangle \text{ where } \tau = s_0, s_1, s_2, \dots \\ \tau, i &\models \varphi_1 \wedge \varphi_2 && \text{iff } \tau, i \models \varphi_1 \text{ and } \tau, i \models \varphi_2 \\ \tau, i &\models \neg \varphi && \text{iff } \tau, i \not\models \varphi \\ \tau, i &\models \varphi_1 U \varphi_2 && \text{iff for some finite } k \geq i, \tau, k \models \varphi_2 \text{ and} \\ &&& \tau, j \models \varphi_1 \text{ for all } j = i, \dots, k-1 \end{aligned}$$

and is lifted to the entire transition system by requiring that every initialised trajectory satisfies φ :

$$\mathcal{M} \models \varphi \text{ iff } \tau, 0 \models \varphi \text{ for all infinite initialised trajectories } \tau \text{ of } \mathcal{M}.$$

We also introduce the derived operators “eventually” $\Diamond$ and “globally” $\Box$. The formula $\Diamond \varphi := \text{true} U \varphi$ states that φ must be true in some state on the trajectory. The formula $\Box \varphi := \neg(\Diamond \neg \varphi)$ requires that φ holds true in all states of the trajectory. We do not include the “next” operator $\bigcirc$ from full LTL since we are interested in stutter-insensitive bisimulations, which do not preserve a system’s stepwise behavior as expressed by the *next*-operator. It is a well-known fact that divergence-sensitive stutter-insensitive bisimulations preserve specifications expressable in $LTL_{\setminus \bigcirc}$ [4]. Divergence-sensitivity is crucial to properly treat stutter-trajectories, i.e., trajectories that forever stutter inside the same equivalence class [44]. However, for deterministic transition systems, each state has only one outgoing trajectory that either eventually leaves its equivalence class or stutters indefinitely. Therefore, any stutter-insensitive bisimulation on a deterministic system must be divergence-sensitive, as stated in Lemma 2.

Lemma 2. *Every stutter-insensitive bisimulation on any deterministic labelled transition system is divergence-sensitive.*

Proof. Let $\mathcal{M}$ be a deterministic transition system and $\simeq$ be a stutter-insensitive bisimulation on $\mathcal{M}$. Let $s \simeq t$ and assume $s \simeq$ -divergent but t not $\simeq$ -divergent. As t not $\simeq$ -divergent, there exists a finite trajectory $\tau = t, t_1, \dots, t_n, t'$ with $t \simeq t_i, \forall i \leq n$ and $t \not\simeq t'$, for some $n \geq 0$. This implies that there exists a state $u \simeq s$ with $s \not\simeq t' \in T(u)$. However, since $\mathcal{M}$ is deterministic and s is $\simeq$ -divergent, the unique trajectory $\tau = s, s_1, \dots$ initialised in s satisfies $s \simeq s_i, \forall i \geq 0$, which is a contradiction. $\square$ $\square$

Theorem 1. *Let $\mathcal{M}$ be a deterministic labelled transition system. If $\simeq$ is a stutter-insensitive bisimulation on $\mathcal{M}$, then $\mathcal{M} \models \varphi$ if and only if $\mathcal{M}/\simeq \models \varphi$ for any $LTL_{\setminus \bigcirc}$ formula φ .*

Proof. Any divergence-sensitive stutter-insensitive bisimulation $\simeq$ on any (possibly non-deterministic) transition system $\mathcal{M}$ implies, for every $LTL_{\setminus \bigcirc}$ formula

φ , the model checking problems $\mathcal{M} \models \varphi$ and $\mathcal{M}/\simeq \models \varphi$ have the same answer. Lemma 2 establishes that, since $\mathcal{M}$ is deterministic and $\simeq$ is stutter-insensitive on $\mathcal{M}$, then $\simeq$ is also divergence-sensitive. Therefore, the statement follows. $\square$ $\square$

Remark 1. Theorem 1 in general does not hold for nondeterministic transition systems, as can be seen by the counterexample in Figure 4.

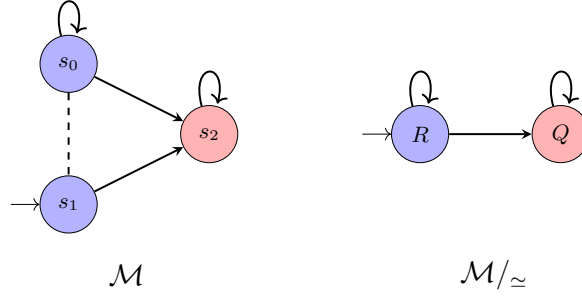


Fig. 4. A stutter-insensitive but not divergence-sensitive bisimulation $\simeq$ is indicated by the dashed lines. It holds that $\mathcal{M} \models \Diamond(\text{red})$ but $\mathcal{M}/\simeq \not\models \Diamond(\text{red})$.

Remark 2. It may seem counterintuitive to the reader to relate stutter-insensitive bisimulation and $\text{LTL}_{\setminus \bigcirc}$ -equivalence, as it is usually associated with the more expressive $\text{CTL}^*_{\setminus \bigcirc}$ [4]. However, recall that we focus on deterministic transition systems for which both logics coincide in expressivity.

4 Counterexample-guided Bisimulation Learning

This section introduces our main contributions. We present our adaption of Namjoshi’s well-founded bisimulations to deterministic transition systems [43] and describe a counterexample guided learning algorithm for simultaneous computation of a stutter-insensitive bisimulation and its corresponding quotient from a finite sampling of the state space. Well-founded bisimulation implements the stability conditions of stutter-insensitive bisimulation (see Def. 6) in the form of ranking functions that map states to a well-founded set, ensuring that a finite trajectory matches every transition of equivalent states. We present an adaption to deterministic systems that only requires the ranking functions to map single states of certain classes to a well-founded set and show that this characterizes stutter-insensitive bisimulation. Furthermore, we show that the problem of computing a stutter-insensitive bisimulation can be rephrased to finding a classifier on states and ranking functions for the corresponding classes.

Theorem 2. *Let $\mathcal{M}$ be a deterministic labelled transition system with state space S and transition function T . Let $\simeq$ be a label-preserving partition on $\mathcal{M}$.*

Suppose that for every region $R \in S/\simeq$ there exists a function $h_R: S \rightarrow \mathbb{N}$ such that, for every $R \neq Q \in T/\simeq(R)$, the following condition holds:

$$\forall s \in R: T(s) \in Q \vee [T(s) \in R \wedge h_R(s) > h_R(T(s))]. \quad (1)$$

Then, $\simeq$ is a stutter-insensitive bisimulation on $\mathcal{M}$.

Proof. Let $s \simeq t$ such that $s \not\simeq s' \in T(s)$. This implies that $\exists R \neq Q \in S/\simeq: s, t \in R$ and $s' \in Q$, and $Q \in T/\simeq(R)$. Assume there exists no finite trajectory $\tau = t, t_1, \dots, t_n, t', n \geq 0$ with $t \simeq t_i, \forall i \leq n$ and $s' \simeq t'$. As $\mathcal{M}$ is deterministic, we only need to distinguish the two cases:

- The infinite trajectory $\tau = t, t_1, t_2, \dots$ stutters infinitely in R , i.e., $t_i \in R, \forall i \geq 1$. This contradicts the ranking property 1.
- There exists a finite trajectory $\tau = t, t_1, \dots, t_n, t', n \geq 0$ with $t' \in V \neq Q$ and $t_i \in R, \forall i \leq n$. However $t_n \in R$ and $t' \in V \neq Q$ is a contradiction to the ranking property 1 only allowing for an exit to Q . $\square$

The ranking functions h_R in Theorem 2 ensure that if class R has an outgoing transition to Q , for all states in R either (1) their successor is in Q or (2) their successor is in R and h_R decreases when transitioning. As the value of h_R is bounded from below and must strictly decrease along any trajectory, no trajectory can stutter in R indefinitely and must eventually enter Q (see Fig. 5).

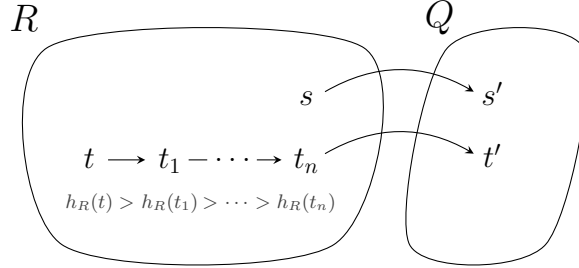


Fig. 5. Intuitive representation of Theorem 2. Since a state $s \in R$ has a successor in Q , the value of h_R must strictly decrease along any trajectory through R . Therefore, all states in R eventually transition to Q after possible stuttering.

Remark 3. For deterministic systems, strong bisimulations form a special case of stutter-insensitive bisimulations, not allowing for any stuttering. In our formulation, they only admit constant ranking functions h_R . Strong bisimulations preserve a system's stepwise behavior, hence, full LTL including the *next*-operator [4]. However, they may induce much larger quotients, less suitable for verifying large systems with long stuttering intervals. Furthermore, there exist infinite state systems that do not admit a finite strong bisimulation, but do admit a finite stutter-insensitive bisimulation quotient.

□

We aim to phrase the problem of finding a suitable partition and ranking functions that satisfy the conditions in Theorem 2 as a learning problem. For that, we introduce the notion of state classifiers.

Definition 7 (State Classifier). *A state classifier on a labelled transition system with state space S is any function $f: S \rightarrow C$ that maps states to a finite set of classes C . It is label-preserving if $f(s) = f(t)$ implies $\langle s \rangle = \langle t \rangle$.*

We can now state Theorem 2 for a state classifier f and give sufficient conditions for f to induce a valid stutter-insensitive bisimulation.

Theorem 3. *Let $\mathcal{M}$ be a deterministic labelled transition system with state space S and transition function T . Suppose that there exists a label-preserving state classifier $f: S \rightarrow C$, a function $g: C \rightarrow C$ and functions $h_c: S \rightarrow \mathbb{N}$ for each $c \in C$ such that, for every $c \neq d \in C$ and $s \in S$, the following two conditions hold:*

$$f(s) = c \wedge g(c) = d \implies f(T(s)) = d \vee [f(T(s)) = c \wedge h_c(s) > h_c(T(s))], \quad (2)$$

$$f(s) = c \wedge f(T(s)) = d \implies g(c) = d. \quad (3)$$

Then, $\simeq_f$ defined as $\simeq_f = \{(s, t) \mid f(s) = f(t)\}$ is a stutter-insensitive bisimulation on $\mathcal{M}$ and $T_{\simeq_f}(f^{-1}[c]) = \{f^{-1}[g(c)]\}$.

Proof. We first show that $\simeq_f$ is a stutter-insensitive bisimulation on $\mathcal{M}$. Since f is label-preserving, $\simeq_f$ is label-preserving by definition. The quotient space is the set of non-empty pre-images of the classes C under f , i.e., $S/\simeq_f = \{f^{-1}[c] \mid c \in C\} \setminus \emptyset$. By definition of $T/\simeq_f$ (see Def. 4) it holds that $f^{-1}[c] \neq f^{-1}[d] \in T/\simeq_f(f^{-1}[c])$ implies that there exists an $s \in f^{-1}[c]$ with $T(s) \in f^{-1}[d]$. With Condition 3 this implies that $g(c) = d$. The claim follows by Condition 2 and Theorem 2. We now show that $T_{\simeq_f}(f^{-1}[c]) = \{f^{-1}[g(c)]\}$. Since $\simeq_f$ is a stutter-insensitive bisimulation Lemma 1 implies that $T/\simeq_f(f^{-1}[c])$ can only be a singleton for any $c \in C$. We distinguish the two cases:

- $f^{-1}[c] \neq f^{-1}[d] \in T/\simeq_f(f^{-1}[c])$, then $f^{-1}[c] \neq f^{-1}[d]$ implies that $c \neq d$. By Def. 4 there must exist a $s \in f^{-1}[c]$ with $T(s) \in f^{-1}[d]$, which by Condition 3 implies that $g(c) = d$.
- $f^{-1}[c] \in T/\simeq_f(f^{-1}[c])$, then some state in $s \in f^{-1}[c]$ must be $\simeq_f$ -divergent by Def. 4. The only possibility for g to be a total function and not to violate Condition 2 is $g(c) = c$, as $g(c) = d \neq c$ would contradict the $\simeq_f$ -divergency of s due to Condition 2. □

□

Remark 4. Note that Theorem 3 requires g to be well-defined, i.e., represent a deterministic transition function. However, this is not a restriction as per Lemma 1 any stutter-insensitive bisimulation on a deterministic transition system has a deterministic quotient. The fact that g has to be total additionally requires it to correctly account for the self-loops of the divergent classes.

In Theorem 3 function g takes on the role of the deterministic transition function of the quotient induced by f . Thus, f and g together provide a complete description of a stutter-insensitive bisimulation quotient of the underlying transition system. In the following, we introduce our counterexample-guided learning approach for generating appropriate functions on a given transition system.

4.1 Learner-Verifier Framework for Bisimulation Learning

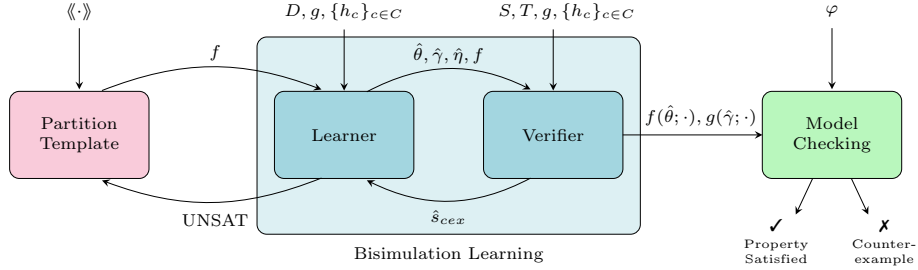


Fig. 6. Architecture of our learner-verifier framework for bisimulation learning.

Our procedure involves two communicating components, the *learner* and the *verifier*, implementing a CEGIS loop. The learner proposes candidate functions that satisfy the stutter-insensitive bisimulation conditions over a finite set of sample states. The verifier checks if a counterexample state exists for which the functions proposed by the learner violate the conditions, which are then passed back to the learner to update the functions (see Figure 6).

Learner We consider parametric function templates whose maps solely depend on the provided parameters. Therefore, the learner seeks suitable parameters for a label-preserving state classifier template $f: \Theta \times S \rightarrow C$, a transition function template $g: \Gamma \times C \rightarrow C$ and ranking function templates $h_c: H \times S \rightarrow \mathbb{N}$ for each $c \in C$, i.e., attempts to solve:

$$\exists \theta \in \Theta, \gamma \in \Gamma, \eta \in H: \bigwedge_{\hat{s} \in D} \Phi_1(\theta, \gamma, \eta; \hat{s}, T(\hat{s})) \wedge \Phi_2(\theta, \gamma, \eta; \hat{s}, T(\hat{s})), \quad (4)$$

where Φ_1 encodes Condition 2 of Theorem 3:

$$\begin{aligned} \Phi_1(\theta, \gamma, \eta; s, s') = \bigwedge_{c \neq d \in C} f(\theta; s) = c \wedge g(\gamma; c) = d \implies \\ f(\theta; s') = d \vee [f(\theta; s') = c \wedge h_c(\eta; s) > h_c(\eta; s')], \end{aligned} \quad (5)$$

and Φ_2 represents Condition 3:

$$\Phi_2(\theta, \gamma, \eta; s, s') = \bigwedge_{c \neq d \in C} f(\theta; s) = c \wedge f(\theta; s') = d \implies g(\gamma; c) = d, \quad (6)$$

for a deterministic transition system $\mathcal{M}$ and finite set of sample states $D \subseteq S$. In our instantiation, we use an SMT-solver to seek a satisfying assignment for the parameters θ , γ , and η in the quantifier-free inner formula of 4.

Verifier The verifier checks the functions induced by the proposed candidate parameters $\hat{\theta}$, $\hat{\gamma}$ and $\hat{\eta}$ for generalisation to the entire state space, i.e., attempts to solve:

$$\exists s \in S: \neg \Phi_1(\hat{\theta}, \hat{\gamma}, \hat{\eta}; s, T(s)) \vee \neg \Phi_2(\hat{\theta}, \hat{\gamma}, \hat{\eta}; s, T(s)). \quad (7)$$

Similar to the learner, the verifier is an SMT-solver to which we hand the quantifier-free inner formula of 7. A found satisfying assignment for a counterexample state s is returned to the learner. If the formula is unsatisfiable, the procedure terminates and has successfully synthesised a valid stutter-insensitive bisimulation and its corresponding quotient.

4.2 Binary Decision Tree Partition Templates

From here on, our focus is on transition systems with discrete state spaces $S \subseteq \mathbb{Z}^n$ defined over the integers. We present the parametric function templates used in our instantiation of the framework. For the state classifier templates, we employ binary decision trees with real-valued decision functions in the inner nodes. We construct binary decision trees preserving the system's labelling function and automatically enlarge them when the template is not expressive enough to fit the finite set of sample states (see Figure 6).

Definition 8 (Binary Decision Tree Templates). *The set of binary decision tree templates $\mathbb{T}$ over a finite set of classes C and parameters Θ consists of trees t , where t is either*

- a leaf node $\text{LEAF}(c)$ with $c \in C$, or
- a decision node $\text{NODE}(p, t_1, t_2)$, where $t_1, t_2 \in \mathbb{T}$ are the left and right subtrees, and $p: \Theta \times S \rightarrow \mathbb{R}$ is a parametrised real-valued function of the states.

A parametric tree template $t \in \mathbb{T}$ over classes C and parameters Θ defines the parametric state classifier $f_t: \Theta \times S \rightarrow C$ given as

$$f_t(\theta, s) = \begin{cases} c & \text{if } t = \text{LEAF}(c) \\ f_{t_1}(\theta, s) & \text{if } t = \text{NODE}(p, t_1, t_2) \text{ and } p(\theta; s) \geq 0 \\ f_{t_2}(\theta, s) & \text{if } t = \text{NODE}(p, t_1, t_2) \text{ and } p(\theta; s) < 0. \end{cases}$$

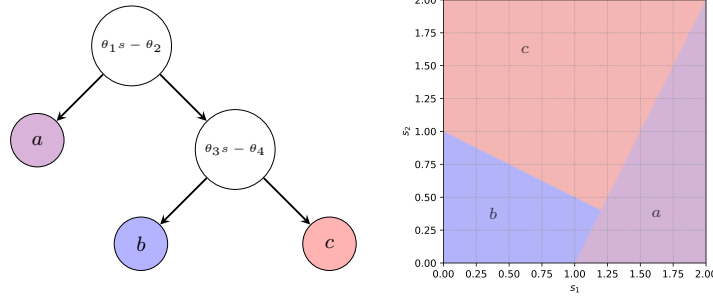


Fig. 7. Binary decision tree with parameters $\theta_1 = [-2 \ 1]$, $\theta_2 = -2$, $\theta_3 = [\frac{1}{2} \ 1]$, and $\theta_4 = 1$ for the parametrised functions, and its corresponding state classifier.

Binary decision trees appeal as state classifier templates as they are interpretable, expressive, and simple to translate into quantifier-free expressions to instantiate the formulas of the learner and the verifier, see Figure 7. The parametric transition function template $g: \Gamma \times C \rightarrow C$ is simply a vector or list over classes C , indexed by C , and for the parametric ranking function templates $h_c: H \times S \rightarrow \mathbb{N}$ we consider linear functions of the form $h_c(\eta, s) = \eta_1 \cdot s + \eta_2$.

An important requirement for our procedure is that the synthesised state classifier is label-preserving. We guarantee this by constructing label-preserving templates which have this property by design for any parameter instantiation. We assume that any atomic proposition $a \in AP$ is associated with a real-valued function $p_a: S \rightarrow \mathbb{R}$, such that

$$\llbracket s \rrbracket = \{a \in AP \mid p_a(s) \geq 0\}. \quad (8)$$

We construct label-preserving templates by encoding the functions corresponding to the atomic propositions into the top nodes of the binary decision tree, i.e., fixing the functions for the top nodes to represent the observation partition (see for example [47] for a canonical construction). This resembles the prerequisite of classical partition-refinement algorithms, which are initialised from label-preserving partitions. Fixing the labelling with the top nodes ensures that any instantiated state classifier is label-preserving, and the subsequent nodes further refine the label-preserving partition. Figure 8 shows the top nodes with fixed functions for a label-preserving binary tree template for the Euclidean algorithm from Figures 1 and 2.

When the binary decision tree used is too small to fit the minimum number of regions in a quotient or if it requires more decision boundaries, the learner will return UNSAT as it cannot fit a partition with the given template on the finite set of samples. In such cases, our procedure automatically increases the size of the employed BDT template and resumes bisimulation learning with the more expressive template (see Figure 6). Our template construction is entirely automatic and requires no user input other than the labeling function. Bisim-

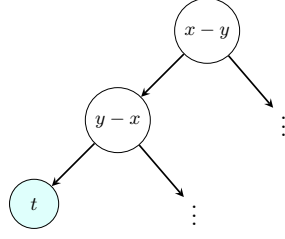


Fig. 8. Label-preserving binary decision tree template for the Euclidian algorithm from Figure 1. The functions are assigned with respect to the loop condition $x \neq y$. The *terminated* proposition, i.e., class t , is assigned to the states satisfying $x \leq y$ and $y \leq x$. Any state satisfying either of $x > y$ or $y > x$ is labelled with *not-terminated*.

ulation learning starts from a small, automatically generated BDT template encoding the labeling function and successively enlarges the partition template as required. We enlarge the partition template by adding an additional layer to the BDT, doubling the number of available partitions.

5 Experimental Evaluation

We implemented our approach in a software prototype and evaluated bisimulation learning on a range of benchmark systems representing two common classes of problems: verification of reactive systems and software model checking. We compare our procedure to established state-of-the-art tools: the nuXmv model checker [16,15,12] for the reactive system problems and the Ultimate [28] and CPAChecker [7] tools for software model checking benchmarks. All benchmarks, our implementation, and the used templates are publicly available. We employ the Z3 SMT-solver [41] in both learner and verifier, and the nuXmv model checker to verify the properties of interest on the obtained abstractions.

5.1 Discrete-Time Clock Synchronization

Setup For reactive systems, we consider two distributed synchronization protocols for potentially drifted discrete clocks of distributed agents. First, the TTethernet protocol, where all agents send their current clock value to a central synchronization master. This synchronization master computes the median clock value and sends it back to the agents, which use the received value to update their internal clocks [9]. Second, we consider an interactive convergence algorithm where the agents directly exchange clock values and compute the average to update their internal clocks while excluding received values that differ more than a given threshold from their own [34]. We check the systems for two kinds of properties: a safety invariant, which specifies that all clock valuations remain within a predefined maximum distance ($G(\text{safe})$); and whether all clocks infinitely often synchronize on the same valuation ($GF(\text{sync})$). Note that while the baseline

procedures verify the systems regarding the given specification, our abstraction procedure is agnostic to the specification, i.e., the obtained abstraction can be used to verify arbitrary $LTL_{\setminus \bigcirc}$ formulas over the atomic propositions.

To render verification with BDDs feasible, we leverage that all clock valuations remain within an interval that depends on the time discretization, as they are either continuously reset or enter a dead-lock state when violating the safety requirement. We explicitly pass this invariant to the BDD toolchain in the form of finite variable domains to allow for the construction of BDDs, whereas IC3 and our abstraction approach operate over unbounded integer variables, which would not be possible for BDDs. For both benchmarks, we consider a safe variant, where the agents use the received values to update their internal clock correctly, and an unsafe version, where they stick with their internal values and drift further. In all instances, we assess multiple instances of time discretization, i.e., the sampling frequency (number of discrete time steps) for a unit second.

Results Table 1 presents the runtime results. Our approach depends on generating candidate parameters and counterexamples through an SMT solver. These can vary across runs of the procedure, even under identical initial conditions (i.e., provided initial samples). Since this can impact the convergence speed and overall runtime of the algorithm, we conduct each experiment 10 times and report the average runtimes and standard deviations. We only report a single outcome for our approach, as we check both properties of interest on the same abstraction and the differences in verification time are negligible on the obtained small abstractions.

Table 1. Results for reactive clock-synchronization benchmarks. All times are measured in seconds with “oot” denoting a timeout at 500 [sec]. The benchmark names include the used parameters, e.g., “tte-sf-1k” describes a safe TTEthernet instance with a time discretization of 1000 steps per second.

Benchmark	No. States	nuXmv (IC3)		nuXmv (BDDs)		Bisimulation Learning
		G(safe)	GF(sync)	G(safe)	GF(sync)	
tte-sf-10	250	0.1	0.7	0.1	0.1	0.3±0.4
tte-sf-100	2500	13.3	423	0.3	0.3	0.7±0.6
tte-sf-1k	2.5×10^6	oot	oot	1.8	31	1.2±0.4
tte-sf-2k	1×10^7	oot	oot	6.4	162	1.5±0.1
tte-sf-5k	6.25×10^7	oot	oot	34	417	1.6±0.4
tte-sf-10k	2.5×10^8	oot	oot	193	oot	1.6±0.2
tte-usf-10	250	0.1	0.5	0.1	0.1	0.2±0.1
tte-usf-100	2500	15.2	9.2	0.1	0.2	0.2±0.1
tte-usf-1k	2.5×10^6	421	405	1.5	10	0.3±0.1
tte-usf-2k	1×10^7	oot	oot	7.1	41	0.4±0.2
tte-usf-5k	6.25×10^7	oot	oot	32	242	0.5±0.5
tte-usf-10k	2.5×10^8	oot	oot	130	oot	0.6±0.5
con-sf-10	250	0.6	0.5	0.1	0.1	0.4±0.2
con-sf-100	2500	22	oot	0.2	0.3	0.4±0.3
con-sf-1k	2.5×10^6	oot	oot	4.6	45	0.7±0.4
con-sf-2k	1×10^7	oot	oot	17.6	210	0.7±0.2
con-sf-5k	6.25×10^7	oot	oot	95	oot	0.8±0.5
con-sf-10k	2.5×10^8	oot	oot	oot	oot	1.4±1.4
con-usf-10	250	0.2	0.3	0.1	0.1	0.2±0.2
con-usf-100	2500	31	33	0.3	0.2	0.2±0.1
con-usf-1k	2.5×10^6	oot	oot	2.8	24	0.3±0.2
con-usf-2k	1×10^7	oot	oot	8.2	154	0.4±0.3
con-usf-5k	6.25×10^7	oot	oot	36	oot	0.7±0.4
con-usf-10k	2.5×10^8	oot	oot	156	oot	0.8±0.3

Discussion The results show that the learned bisimulations effectively and efficiently verify the specifications. Especially with decreased time discretization and, therefore, increased size of the state space, our approach clearly shows an advantageous performance. While a larger reachable state space renders verification harder for all considered approaches, a decreased time discretization is especially difficult for the IC3 toolchain based on bounded model checking, as it increases the completeness threshold and the depth of counterexamples. Since bisimulation learning generalizes from a finite set of samples, it is less susceptible to larger state spaces if the corresponding abstractions remain small. Generally, there is a trade-off between the number of provided initial samples and the number of CEGIS iterations needed to refine the initial partition. Although it may require more time to fit an initial partition on a more extensive set of uniform initial samples, it can reduce the counterexamples needed to obtain a valid stutter-insensitive bisimulation. As our instantiation leverages potentially expensive SMT solving in both the learner and the verifier, which scales in the number of considered samples, we aim at being sample-efficient: therefore, we decided to consider a fixed, small amount of uniform initial samples for all benchmarks of different sizes and leverage the generation of informative counterexamples in potentially more, but faster CEGIS cycles.

5.2 Conditional Termination

Setup For software model checking, we consider a range of benchmarks from program termination analysis, including a selection of programs sourced from the termination category of the SV-COMP competition for software verification [6]. As is the case for the Euclidean algorithm in Figure 1, these programs on unbounded integer variables may terminate for some inputs and enter a non-terminating loop for others. The two baseline tools determine whether a program terminates for all possible inputs. Our procedure instead goes a step further by providing an exact partition of the variable valuations, separating the inputs for which the algorithm eventually terminates from those for which it does not. As a distinguishing feature of the baseline benchmarks, we split each program into two versions: one that only allows for inputs for which the program terminates (denoted as “term”) and another that includes potentially non-terminating inputs (denoted as “¬term”).

Results Table 2 presents the runtime results. Note that we only report the analysis time for the baselines, without additional time spent on parsing or preprocessing the programs.

Discussion The results show that bisimulation learning, while computing more informative results and solving the more complex problem of conditional termination [11,19], operates in runtimes comparable to the state-of-the-art tools for the considered benchmarks. Especially for programs that involve disjunctions

Table 2. Results for software termination benchmarks. All times are measured in seconds, with “oot” denoting a timeout at 500 [sec]. A non-conclusive analysis outcome is denoted by “n/c” and “-” indicates that there is no such special case of the benchmark.

Benchmark	nuXmv (IC3)		CPAChecker		Ultimate		Bisimulation
	term	¬term	term	¬term	term	¬term	Learning
term-loop-1	oot	oot	0.6	1.8	1.5	2.9	0.2 ± 0.1
term-loop-2	oot	oot	0.6	0.3	0.7	0.2	0.4 ± 0.3
audio-compr	< 0.1	< 0.1	n/c	n/c	0.9	0.6	0.3 ± 0.3
euclid	oot	oot	n/c	0.3	1.6	0.4	0.6 ± 0.2
greater	oot	< 0.1	0.6	0.3	1.1	0.3	0.4 ± 0.2
smaller	oot	< 0.1	0.6	0.3	1.6	0.3	0.2 ± 0.1
conic	oot	< 0.1	0.7	0.4	n/c	0.4	4.2 ± 7.3
disjunction	oot	-	34	-	1.7	-	0.3 ± 0.3
parallel	oot	-	0.9	-	9.1	-	0.3 ± 0.3
quadratic	0.2	-	n/c	-	n/c	-	0.3 ± 0.1
cubic	0.2	< 0.1	n/c	n/c	n/c	0.2	0.4 ± 0.2
nlr-cond	< 0.1	< 0.1	n/c	n/c	n/c	0.2	0.2 ± 0.2

over variable valuations (cf. the *disjunction* and *parallel* benchmarks), our procedure is able to prove termination more efficiently. Additionally, our approach can handle non-linear operations if the employed templates are sufficiently expressive for the corresponding partition and ranking functions. As a further surplus, it yields interpretable binary decision trees representing the derived stutter-insensitive bisimulation. These trees are valuable for system diagnostics and fault analysis, providing further insight beyond single counterexamples. Once again, this experimental evaluation shows that, while not being complete in theory, our algorithm terminates in all of the considered experiments.

Limitations Bisimulation learning addresses a generally undecidable problem: finding finite bisimulations for systems with potentially infinite state spaces [40]. While our procedure is guaranteed to terminate on finite state systems, it must be inherently incomplete in general. Our experimental evaluation demonstrates that we can effectively and efficiently find finite bisimulations for infinite-state systems. However, there exist systems for which bisimulation learning can never successfully terminate. We give an example for such a system: Consider the infinite state space of natural numbers $S = \{0, 1, \dots\}$, where each state transitions by subtracting one, and zero loops on itself, i.e., $T = \{0 \mapsto 0\} \cup \{n \mapsto n - 1, n > 0\}$. The labelling function distinguishes *zero*, *even*, and *odd* numbers (see Figure 9).

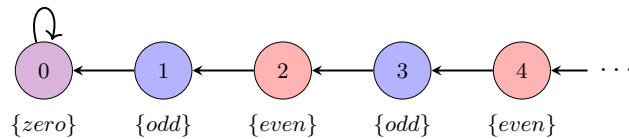


Fig. 9. A system for which bisimulation learning can never terminate, as no finite stutter-insensitive bisimulation exists.

Any infinite trajectory starting in some state i will eventually enter state zero. However, depending on the starting state, it will traverse a different sequence of even and odd states. Hence, we can construct $LTL_{\setminus \bigcirc}$ formulas that distinguish each state from *smaller* states. For instance, the formula $\Diamond(even \wedge \Diamond(odd \wedge \Diamond(zero)))$ can only be satisfied by states larger than one. Per Theorem 1, since $LTL_{\setminus \bigcirc}$ can distinguish any state from smaller states, every state must be its own equivalence class with respect to stutter-insensitive bisimulation. When applying bisimulation learning to the described system, the CEGIS loop can never terminate with a finite quotient. Our procedure will keep enlarging the partition template used to fit the growing set of counterexamples, but will never be able to generalize to the entire state space. We note that this is a limitation intrinsic to bisimulations, i.e., no bisimulation algorithm could successfully terminate when applied to the stated system.

6 Conclusion

We have presented the first data-driven method to compute bisimulations. We have demonstrated that our method effectively computes finite abstractions for model checking and diagnostics. We instantiated our method to stutter-insensitive bisimulations, showcased its efficacy on $LTL_{\setminus \bigcirc}$ model checking of discrete-time synchronization protocols as well as on conditional termination analysis benchmarks from the SV-COMP. On these benchmarks, our method yielded faster results than alternative model checking algorithms based on BDDs and IC3 (nuXmv), and state-of-the-art software model checking procedures (Ultimate and CPAChecker). Our benchmark sets are systems with long completeness thresholds and deep counterexamples, for which stutter-insensitive bisimulations provide succinct abstract quotients.

Our technique builds upon an existing proof rule for well-founded bisimulations. This allows us to characterise stutter-insensitive bisimulations as classifiers from infinite concrete states to finite abstract states, with an attached ranking function on each abstract state that strictly decreases as the concrete system stutters. This has enabled implementing a learner-verifier framework to compute bisimulations for deterministic systems with discrete state space. Our approach readily extends to *strong* bisimulations for deterministic systems, even though in practice these produce too large abstractions for effective model checking. Stutter-insensitive bisimulations instead are coarser and, therefore, generate smaller quotients. Not only this enables an effective verification of $LTL_{\setminus \bigcirc}$ properties, but also provides succinct and interpretable abstractions.

Our result is the basis for several extensions. First, we envision extensions towards stutter-insensitive bisimulations for non-deterministic systems, which are harder because they require more general conditions on learner and verifier. Second, we target extensions towards continuous-state systems, which are harder because they offer much less flexibility in terms of numerical representation [23,24,54]. Lastly, we envision extensions towards using neural architectures for further flexibility and scalability in state classifier representation [1].

References

1. Abate, A., Edwards, A., Giacobbe, M.: Neural abstractions. In: NeurIPS (2022)
2. Abate, A., Giacobbe, M., Roy, D.: Stochastic omega-regular verification and control with supermartingales. In: CAV. Lecture Notes in Computer Science, Springer (2024)
3. Ashok, P., Jackermeier, M., Jagtap, P., Kretínský, J., Weininger, M., Zamani, M.: dtcontrol: decision tree learning algorithms for controller representation. In: HSCC. pp. 17:1–17:7. ACM (2020)
4. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
5. Balcázar, J.L., Gabarró, J., Santha, M.: Deciding bisimilarity is P-Complete. Formal Aspects Comput. **4**(6A), 638–648 (1992)
6. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: TACAS (2). Lecture Notes in Computer Science, vol. 13994, pp. 495–522. Springer (2023)
7. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
9. Bogomolov, S., Herrera, C., Steiner, W.: Verification of fault-tolerant clock synchronization algorithms. In: ARCH@CPSWeek. EPiC Series in Computing, vol. 43, pp. 36–41. EasyChair (2016)
10. Bouajjani, A., Fernandez, J., Halbwachs, N.: Minimal model generation. In: CAV. Lecture Notes in Computer Science, vol. 531, pp. 197–203. Springer (1990)
11. Bozga, M., Iosif, R., Konečný, F.: Deciding conditional termination. In: TACAS. Lecture Notes in Computer Science, vol. 7214, pp. 252–266. Springer (2012)
12. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
13. Brázdil, T., Chatterjee, K., Kretínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 385–407. Springer (2018)
14. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**, 115–131 (1988)
15. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10²⁰ states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
16. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)
18. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL. pp. 265–276. ACM (2007)
19. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: CAV. Lecture Notes in Computer Science, vol. 5123, pp. 328–340. Springer (2008)
20. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)

21. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. *Int. J. Softw. Tools Technol. Transf.* **20**(2), 157–177 (2018)
22. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: *ES-EC/SIGSOFT FSE*. pp. 633–645. ACM (2022)
23. Girard, A.: Approximately bisimilar finite abstractions of stable linear systems. In: *HSCC. Lecture Notes in Computer Science*, vol. 4416, pp. 231–244. Springer (2007)
24. Girard, A., Pappas, G.J.: Approximate bisimulation: A bridge between computer science and control theory. *Eur. J. Control* **17**(5-6), 568–578 (2011)
25. van Glabbeek, R.J.: The linear time - branching time spectrum II. In: *CONCUR. Lecture Notes in Computer Science*, vol. 715, pp. 66–81. Springer (1993)
26. 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)
27. Grumberg, O., Francez, N., Makowsky, J.A., de Roever, W.P.: A proof rule for fair termination of guarded commands. *Inf. Control.* **66**(1/2), 83–102 (1985)
28. Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate automizer and the commuhash normal form - (competition contribution). In: *TACAS (2). Lecture Notes in Computer Science*, vol. 13994, pp. 577–581. Springer (2023)
29. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. *J. ACM* **32**(1), 137–161 (1985)
30. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: *POPL*. pp. 232–244. ACM (2004)
31. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) *Theory of Machines and Computations*, pp. 189–196. Academic Press (1971)
32. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.* **86**(1), 43–68 (1990)
33. Lamport, L.: What good is temporal logic? In: *IFIP Congress*. pp. 657–668. North-Holland/IFIP (1983)
34. Lamport, L., Melliar-Smith, P.M.: Byzantine clock synchronization. In: *PODC*. pp. 68–74. ACM (1984)
35. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: *STOC*. pp. 264–274. ACM (1992)
36. Lee, I., Rajasekaran, S.: A parallel algorithm for relational coarsest partition problems and its implementation. In: *CAV. Lecture Notes in Computer Science*, vol. 818, pp. 404–414. Springer (1994)
37. Martens, J., Groote, J.F., van den Haak, L.B., Hijma, P., Wijs, A.: A linear parallel algorithm to compute bisimulation and relational coarsest partitions. In: *FACS. Lecture Notes in Computer Science*, vol. 13077, pp. 115–133. Springer (2021)
38. Milner, R.: *A Calculus of Communicating Systems*, *Lecture Notes in Computer Science*, vol. 92. Springer (1980)
39. Milner, R.: *Communication and concurrency*. PHI Series in computer science, Prentice Hall (1989)
40. Moller, F.: Infinite results. In: *CONCUR. Lecture Notes in Computer Science*, vol. 1119, pp. 195–216. Springer (1996)
41. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: *TACAS. Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
42. Murali, V., Trivedi, A., Zamani, M.: Closure certificates. In: *HSCC*. pp. 10:1–10:11. ACM (2024)

43. Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: FSTTCS. Lecture Notes in Computer Science, vol. 1346, pp. 284–296. Springer (1997)
44. Nicola, R.D., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
45. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE. pp. 246–256. ACM (2013)
46. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987)
47. Pappas, G.J.: Bisimilar linear systems. Autom. **39**(12), 2035–2047 (2003)
48. Park, D.M.R.: Concurrency and automata on infinite sequences. In: Theoretical Computer Science. Lecture Notes in Computer Science, vol. 104, pp. 167–183. Springer (1981)
49. Pnueli, A.: The temporal logic of programs. In: FOCS. pp. 46–57. IEEE Computer Society (1977)
50. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. pp. 404–415. ACM (2006)
51. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: SAS. Lecture Notes in Computer Science, vol. 8723, pp. 302–318. Springer (2014)
52. Vardi, M.Y.: Verification of concurrent programs: The automata-theoretic framework. Ann. Pure Appl. Log. **51**(1-2), 79–98 (1991)
53. Walker, D.J.: Bisimulations and divergence. In: LICS. pp. 186–192. IEEE Computer Society (1988)
54. Zamani, M., Esfahani, P.M., Majumdar, R., Abate, A., Lygeros, J.: Symbolic control of stochastic systems via approximately bisimilar finite abstractions. IEEE Trans. Autom. Control. **59**(12), 3135–3150 (2014)