1-2016

Regular symmetry patterns

Anthony W. LIN

Truong Khanh NGUYEN

Philipp RÜMMER

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Regular Symmetry Patterns

Anthony W. Lin[1], Truong Khanh Nguyen[2], Philipp Rümmer[3(✉)], and Jun Sun[4]

[1] Yale-NUS College, Singapore, Singapore
[2] Autodesk, Singapore, Singapore
[3] Uppsala University, Uppsala, Sweden
philipp.ruemmer@it.uu.se
[4] Singapore University of Design and Technology, Singapore, Singapore

Abstract. Symmetry reduction is a well-known approach for alleviating the state explosion problem in model checking. Automatically identifying symmetries in concurrent systems, however, is computationally expensive. We propose a symbolic framework for capturing symmetry patterns in parameterised systems (i.e. an infinite family of finite-state systems): two regular word transducers to represent, respectively, parameterised systems and symmetry patterns. The framework subsumes various types of "symmetry relations" ranging from weaker notions (e.g. simulation preorders) to the strongest notion (i.e. isomorphisms). Our framework enjoys two algorithmic properties: (1) symmetry verification: given a transducer, we can automatically check whether it is a symmetry pattern of a given system, and (2) symmetry synthesis: we can automatically generate a symmetry pattern for a given system in the form of a transducer. Furthermore, our symbolic language allows additional constraints that the symmetry patterns need to satisfy to be easily incorporated in the verification/synthesis. We show how these properties can help identify symmetry patterns in examples like dining philosopher protocols, self-stabilising protocols, and prioritised resource-allocator protocol. In some cases (e.g. Gries's coffee can problem), our technique automatically synthesises a safety-preserving finite approximant, which can then be verified for safety solely using a finite-state model checker.

1 Introduction

Symmetry reduction [12,19,22] is a well-known approach for alleviating the state explosion problem in automatic verification of concurrent systems. The essence of symmetry reduction is to identify symmetries in the system and avoid exploring states that are "similar" (under these symmetries) to previously explored states.

One main challenge with symmetry reduction methods is the difficulty in identifying symmetries in a given system in general. One approach is to provide dedicated language instructions for specifying symmetries (e.g. see [22,29,30]) or specific languages (e.g. see [13,24,25]) so that users can provide insight on what symmetries are there in the system. For instance, Murφ provides a special data type with a list of syntactic restrictions and all values that belong

B. Jobstmann and K.R.M. Leino (Eds.): VMCAI 2016, LNCS 9583, pp. 455–475, 2016.
DOI: 10.1007/978-3-662-49122-5_22

to this type are symmetric. Another approach is to detect symmetry automatically without requiring expert insights. Automatic detection of symmetries is an extremely difficult computational problem. A number of approaches have been proposed in this direction (e.g. [15,16,33]). For example, Donaldson and Miller [15,16] designed an automatic approach to detecting process symmetries for channel-based communication systems, based on constructing a graph called *static channel diagram* from a Promela model whose automorphisms correspond to symmetries in the model. Nonetheless, it is clear from their experiments that existing approaches work only for small numbers of processes.

In practice, concurrent systems are often obtained by replicating a generic behavioral description [32]. For example, a prioritised resource-allocator protocol [14], [Sect. 4.4] provides a description of an allocator program and a client program in a network with a star topology (allocator in the center), from which a concurrent system with 1 allocator and m clients (for any given $m \in \mathbb{Z}_{>0}$) can be generated. This is in fact the standard setting of parameterised systems (e.g. see [4,31]), which are symbolic descriptions of infinite families $\{\mathfrak{S}_i\}_{i=1}^{\infty}$ of transition systems $\mathfrak{S}_i$ that can be generated by instantiating some parameters (e.g. the number of processes).

Adopting this setting of parameterised systems, we consider the problem of formulating and generating symbolic *symmetry patterns*, abstract descriptions of symmetries that can be instantiated to obtain concrete symmetries for every instance of a parameterised system. A formal language to specify symmetry patterns should be able to capture interesting symmetry patterns, e.g., that each instance $\mathfrak{S}_i$ of the parameterised system $\mathfrak{S} = \{\mathfrak{S}_i\}_{i=1}^{\infty}$ exhibits the full symmetry S_n (i.e. invariant under permuting the locations of the processes). Ideally, such a language $\mathcal{L}$ should also enjoy the following algorithmic properties: (1) *symmetry verification*, i.e., given a symmetry pattern $P \in \mathcal{L}$, we can automatically check whether P is a symmetry pattern of a given parameterised system, and (2) *symmetry synthesis*: given a parameterised system, we can automatically generate symmetry patterns $P \in \mathcal{L}$ that the system exhibits. In particular, if $\mathcal{L}$ is sufficiently expressive to specify commonly occuring symmetry patterns, Property (1) would allow us to automatically compute which common symmetry patterns hold for a given parameterised system. In the case when symmetry patterns might be less obvious, Property (2) would allow us to identify further symmetries that are satisfied by the given parameterised systems. To the best of our knowledge, to date no such languages have been proposed.

Contribution: We propose a general symbolic framework for capturing symmetry patterns for parameterised systems. The framework uses *finite-state letter-to-letter word transducers* to represent *both* parameterised systems and symmetry patterns. In the sequel, symmetry patterns that are recognised by transducers are called *regular symmetry patterns*. Based on extensive studies in regular model checking (e.g. see [1,4,27,31]), finite-state word transducers are now well-known to be good symbolic representations of parameterised systems. Moreover, equivalent logic-based (instead of automata-based) formalisms are also available, e.g.,

LTL(MSO) [3] which can be used to specify parameterised systems and properties (e.g. safety and liveness) in a convenient way. In this paper, we show that transducers are not only also sufficiently expressive for representing many common symmetry patterns, but they enjoy the two aforementioned desirable algorithmic properties: automatic symmetry verification and synthesis.

There is a broad spectrum of notions of "symmetries" for transition systems that are of interest to model checking. These include simulation preorders (a weak variant) and isomorphisms (the strongest), e.g., see [6]. We suggest that transducers are not only sufficiently powerful in expressing many such notions of symmetries, but they are also a flexible symbolic language in that constraints (e.g. the symmetry pattern is a bijection) can be easily added to or relaxed from the specification. In this paper, we shall illustrate this point by handling simulation preorders and isomorphisms (i.e. bijective simulation preorders) within the same framework. Another notable point of our symbolic language is its ability to specify that the simulation preorder gives rise to an *abstracted system* that is finite-state and preserves non-safety (i.e. if the original system is not safe, then so is the abstracted system). In other words, *we can specify that the symmetry pattern reduces the infinite-state parameterised system to a finite-state system.* Safety of finite-state systems can then be checked using standard finite-state model checkers.

We next show how to specialise our framework to *process symmetries* [12,19,22]. Roughly speaking, a process symmetry for a concurrent system $\mathfrak{S}$ with n processes is a permutation $\pi : [n] \to [n]$ (where $[n] := \{1, \ldots, n\}$) such that the behavior of $\mathfrak{S}$ is invariant under permuting the process indices by π (i.e. the resulting system is isomorphic to the original one under the natural bijection induced by π). For example, if the process indices of clients in the aforementioned resource-allocator protocol with 1 allocator and m clients are $1, \ldots, m+1$, then any permutation $\pi : [m+1] \to [m+1]$ that fixes 1 is a process symmetry for the protocol. The set of such process symmetries is a permutation group on $[m+1]$ (under functional composition) generated by the following two permutations specified in standard cyclic notations: $(2, 3)$ and $(2, 3, \ldots, m+1)$. This is true for *every value of* $m \geq 2$. In addition, finite-state model checkers represent symmetry permutation groups by their (often exponentially more succinct) finite set of generators. Thus, if $\mathfrak{S} = \{\mathfrak{S}_n\}_{n=1}^{\infty}$ is a parameterised system where $\mathfrak{S}_n$ is the instance with n processes, we represent the *parameterised symmetry groups* $\mathcal{G} = \{G_n\}_{n=1}^{\infty}$ (where G_n is the process symmetry group for $\mathfrak{S}_n$) by a finite list of regular symmetry patterns that generate $\mathcal{G}$. We postulate that commonly occuring parameterised process symmetry groups (e.g. full symmetry groups and rotations groups) can be captured in this framework, e.g., parameterised symmetry groups for the aforementioned resource-allocator protocol can be generated by the symmetry patterns $\{(2, 3)(4) \cdots (m+1)\}_{m \geq 3}$ and $\{(2, 3, \ldots, m+1)\}_{m \geq 3}$, which can be easily expressed using transducers. Thus, using our symmetry verification algorithm, commonly occuring process symmetries for a given parameterised system could be automatically identified.

The aforementioned approach of checking a given parameterised system against a "library" of common regular symmetry patterns has two problems.

Firstly, some common symmetry patterns are not regular, e.g., reflections. To address this, we equip our transducers with an unbounded pushdown stack. Since pushdown transducers in general cannot be synchronised [5] (a crucial property to obtain our symmetry verification algorithm), we propose a restriction of pushdown transducers for which we can recover automatic symmetry verification. Secondly, there are many useful but subtle symmetry patterns in practice. To address this, we propose the use of our symmetry synthesis algorithm. Since a naive enumeration of all transducers with $k = 1, \ldots, n$ states does not scale, we devise a CEGAR loop for our algorithm in which a SAT-solver provides a candidate symmetry pattern (perhaps satisfying some extra constraints) and an automata-based algorithm either verifies the correctness of the guess, or returns a counterexample that can be further incorporated into the guess of the SAT-solver.

We have implemented our symmetry verification/synthesis algorithms and demonstrated its usefulness in identifying regular symmetry patterns for examples like dining philosopher protocols, self-stabilising protocols, resource-allocator protocol, and Gries's coffee can problem. In the case of the coffee can problem, we managed to obtain a reduction from the infinite system to a finite-state system.

Related Work: Our work is inspired by regular model checking (e.g. [1,3,4,31]), which focuses on symbolically computing the sets of reachable configurations of parameterised systems as regular languages. Such methods are generic, but are not guaranteed to terminate in general. As in regular model checking, our framework uses transducers to represent parameterised systems. However, instead of computing their sets of reachable configurations, our work finds symmetry patterns of the parameterised systems, which can be exploited by an explicit-state finite-state model checker to verify the desired property over finite instances of the system (see [32] for more details). Although our verification algorithm is guaranteed to terminate in general (in fact, in polynomial-time assuming the parameterised system is given as a DFA), our synthesis algorithm only terminates when we fix the number of states for the transducers. Finding process symmetry patterns is often easier since there are available tools for finding symmetries for finite (albeit small) instances of the systems (e.g. [15,16,33]).

Another related line of works is "cutoff techniques" (e.g. see [17,18] and the survey [31]), which allows one to reduce verification of parameterised systems into verification of finitely many instances (in some cases, ≤ 10 processes). These works usually assume verification of $\mathsf{LTL}\backslash\mathsf{X}$ properties. Although such techniques are extremely powerful, the systems that can be handled using the techniques are often quite specific (e.g. see [31]).

Organisation: Section 2 contains preliminaries. In Sect. 3, we present our framework of regular symmetry patterns. In Sect. 4 (resp. Section 5), we present our symmetry verification algorithm (resp. synthesis) algorithms. Section 6 discusses our implementation and experiment results. Section 7 concludes with

future work. Due to space constraints, some details are relegated into the full version [28].

2 Preliminaries

General Notations. For two given natural numbers $i \leq j$, we define $[i,j] = \{i, i+1, \ldots, j\}$. Define $[k] = [1,k]$. Given a set S, we use S^* to denote the set of all finite sequences of elements from S. The set S^* always includes the empty sequence which we denote by ϵ. Given two sets of words S_1, S_2, we use $S_1 \cdot S_2$ to denote the set $\{v \cdot w \mid v \in S_1, w \in S_2\}$ of words formed by concatenating words from S_1 with words from S_2. Given two relations $R_1, R_2 \subseteq S \times S$, we define their composition as $R_1 \circ R_2 = \{(s_1, s_3) \mid \exists s_2.\ (s_1, s_2) \in R_1 \wedge (s_2, s_3) \in R_2\}$. Given a subset $X \subseteq S$, we define the image $R(X)$ (resp. preimage $R^{-1}(X)$) of X under R as the set $\{s \in S \mid \exists s'.\ (s', s) \in R\}$ (resp. $\{s' \in S \mid \exists s.\ (s', s) \in R\}$). Given a finite set $S = \{s_1, \ldots, s_n\}$, the *Parikh vector* $\mathbb{P}(v)$ of a word $v \in S^*$ is the vector $(|v|_{s_1}, \ldots, |v|_{s_n})$ of the number of occurrences of the elements $s_1, \ldots, s_n$, respectively, in v.

Transition Systems. Let ACT be a finite set of *action symbols*. A *transition system* over ACT is a tuple $\mathfrak{S} = \langle S; \{\rightarrow\}_{a \in \mathsf{ACT}} \rangle$, where S is a set of *configurations*, and $\rightarrow_a \subseteq S \times S$ is a binary relation over S. We use $\rightarrow$ to denote the relation $\left(\bigcup_{a \in \mathsf{ACT}} \rightarrow_a\right)$. In the sequel, we will often only consider the case when $|\mathsf{ACT}| = 1$ for simplicity. The notation $\rightarrow^+$ (resp. $\rightarrow^*$) is used to denote the transitive (resp. transitive-reflexive) closure of $\rightarrow$. We say that a sequence $s_1 \rightarrow \cdots \rightarrow s_n$ is a *path* (or *run*) in $\mathfrak{S}$ (or in $\rightarrow$). Given two paths $\pi_1 : s_1 \rightarrow^* s_2$ and $\pi_2 : s_2 \rightarrow^* s_3$ in $\rightarrow$, we may concatenate them to obtain $\pi_1 \odot \pi_2$ (by gluing together s_2). In the sequel, for each $S' \subseteq S$ we use the notation $post^*_{\rightarrow}(S')$ to denote the set of configurations $s \in S$ reachable in $\mathfrak{S}$ from some $s \in S$.

Words, Automata, and Transducers. We assume basic familiarity with word automata. Fix a finite alphabet Σ. For each finite word $w = w_1 \ldots w_n \in \Sigma^*$, we write $w[i,j]$, where $1 \leq i \leq j \leq n$, to denote the segment $w_i \ldots w_j$. Given a (nondeterministic finite) automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$, a run of $\mathcal{A}$ on w is a function $\rho : \{0, \ldots, n\} \rightarrow Q$ with $\rho(0) = q_0$ that obeys the transition relation δ. We may also denote the run ρ by the word $\rho(0) \cdots \rho(n)$ over the alphabet Q. The run ρ is said to be *accepting* if $\rho(n) \in F$, in which case we say that the word w is *accepted* by $\mathcal{A}$. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of words in Σ^* accepted by $\mathcal{A}$. In the sequel, we will use the standard abbreviations DFA/NFA (Deterministic/Nondeterministic Finite Automaton).

Transducers are automata that accept binary relations over words [8,9] (a.k.a. "letter-to-letter" automata, or synchronised transducers). Given two words $w = w_1 \ldots w_n$ and $w' = w'_1 \ldots w'_m$ over the alphabet Σ, let $k = \max\{n, m\}$ and $\Sigma_\# := \Sigma \cup \{\#\}$, where $\#$ is a special padding symbol not in Σ. We define a word $w \otimes w'$ of length k over alphabet $\Sigma_\# \times \Sigma_\#$ as follows:

$$w \otimes w' \;=\; (a_1, b_1) \ldots (a_k, b_k), \text{ where } a_i \;=\; \begin{cases} w_i & i \leq n \\ \# & i > n, \end{cases} \text{ and } b_i \;=\; \begin{cases} w'_i & i \leq m \\ \# & i > m. \end{cases}$$

In other words, the shorter word is padded with #'s, and the ith letter of $w \otimes w'$ is then the pair of the ith letters of padded w and w'. A *transducer* (a.k.a. letter-to-letter automaton) is simply a finite-state automaton over $\Sigma_\# \times \Sigma_\#$, and a binary relation $R \subseteq \Sigma^* \times \Sigma^*$ is *regular* if the set $\{w \otimes w' : (w, w') \in R\}$ is accepted by a letter-to-letter automaton. The relation R is said to be *length-preserving* if R only relates words of the same length [4], i.e., that any automaton recognising R consumes no padded letters of the form $(a, \#)$ or $(\#, a)$. In the sequel, for notation simplicity, we will confuse a transducer and the binary relation that it recognises (i.e. R is used to mean both).

Finally, notice that the notion of regular relations can be easily extended to r-ary relations R for each positive integer r (e.g. see [8,9]). To this end, the input alphabet of the transducer will be $\Sigma_\#^r$. Similarly, for R to be regular, the set $\{w_1 \otimes \cdots \otimes w_r : (w_1, \ldots, w_r) \in R\}$ of words over the alphabet Σ^r must be regular.

Permutation Groups. We assume familiarity with basic group theory (e.g. see [11]). A *permutation* on $[n]$ is any bijection $\pi : [n] \to [n]$. The set of all permutations on $[n]$ forms the *(nth) full symmetry group* $\mathcal{S}_n$ under functional composition. A *permutation group* on $[n]$ is any set of permutations on $[n]$ that is a subgroup of $\mathcal{S}_n$ (i.e. closed under composition). A *generating set* for a permutation group G on $[n]$ is a finite set X of permutations (called *generators*) such that each permutation in G can be expressed by taking compositions of elements in X. In this case, we say that G can be generated by X. A word $w = a_0 \ldots a_{k-1} \in [n]^*$ containing distinct elements of $[n]$ (i.e. $a_i \neq a_j$ if $i \neq j$) can be used to denote the permutation that maps $a_i \mapsto a_{i+1 \bmod k}$ for each $i \in [0, k)$ and fixes other elements of $[n]$. In this case, w is called a *cycle* (more precisely, k-cycle or *transposition* in the case when $k = 2$), which we will often write in the standard notation $(a_0, \ldots, a_{k-1})$ so as to avoid confusion. Any permutation can be written as a composition of disjoint cycles [11]. In addition, it is known that $\mathcal{S}_n$ can be generated by the set $\{(1, 2), (1, 2, \ldots, n)\}$. Each subgroup G of $\mathcal{S}_n$ acts on the set Σ^n (over any finite alphabet Σ) under the group action of permuting indices, i.e., for each $\pi \in G$ and $\mathbf{v} = (a_1, \ldots, a_n) \in \Sigma^n$, we define $\pi\mathbf{v} := (a_{\pi^{-1}(1)}, \ldots, a_{\pi^{-1}(n)})$. That way, each π induces the bijection $f_\pi : \Sigma^n \to \Sigma^n$ such that $f_\pi(\mathbf{v}) = \pi\mathbf{v}$.

Given a permutation group G on $[n]$ and a transition system $\mathfrak{S} = \langle S; \to \rangle$ with state space $S = \Sigma^n$, we say that $\mathfrak{S}$ is *G-invariant* if the bijection $f_\pi : \Sigma^n \to \Sigma^n$ induced by each $\pi \in G_n$ is an automorphism on $\mathfrak{S}$, i.e., $\forall v, w \in S$: $v \to w$ implies $f_\pi(v) \to f_\pi(w)$.

3 The Formal Framework

This section describes our symbolic framework regular symmetry patterns.

3.1 Representing Parameterised Systems

As is standard in regular model checking [1,4,31], we use length-preserving transducers to represent parameterised systems. As we shall see below, we will use non-length-preserving transducers to represent symmetry patterns.

Definition 1 (Automatic transition systems[1]). *A transition system* $\mathfrak{S} = \langle S; \{\rightarrow\}_{a \in \mathsf{ACT}} \rangle$ *is said to be (length-preserving) automatic if* S *is a regular set over a finite alphabet* Σ *and each relation* $\rightarrow_a$ *is given by a transducer over* Σ.

More precisely, the parameterised system defined by $\mathfrak{S}$ is the family $\{\mathfrak{S}_n\}_{n \geq 0}$ with $\mathfrak{S}_n = \langle S_n; \rightarrow_{a,n} \rangle$, where $S_n := S \cap \Sigma^n$ is the set of all words in S of length n and $\rightarrow_{a,n}$ is the transition relation $\rightarrow_a$ restricted to S_n. In the sequel, for simplicity we will mostly consider examples when $|\mathsf{ACT}| = 1$. When the meaning is understood, we shall confuse the notation $\rightarrow_a$ for the transition relation of $\mathfrak{S}$ and the transducer that recognises it. To illustrate our framework and methods, we shall give three examples of automatic transition systems (see [3,31] for numerous other examples).

Example 1. We describe a prioritised resource-allocator protocol [14], [Sect. 4.4], which is a simple mutual exclusion protocol in network with a star topology. The protocol has one allocator and m clients. Initially, each process is in an *idle* state. However, clients might from time to time *request* for an access to a resource (*critical section*), which can only be used by one process at a time. For simplicity, we will assume that there is only one resource shared by all the clients. The allocator manages the use of the resource. When a request is lodged by a client, the allocator can allow the client to use the resource. When the client has finished using the resource, it will send a message to the allocator, which can then allow other clients to use the resource.

To model the protocol as a transducer, we let $\Sigma = \{i, r, c\}$, where i stands for "idle", r for "request", and c for "critical". Allocator can be in either the state i or the state c, while a client can be in one of the three states in Σ. A valid configuration is a word aw, where $a \in \{i, c\}$ represents the state of the allocator and $w \in \Sigma^*$ represents the states of the $|w|$ clients (i.e. each position in w represents a state of a client). Letting $I = \{(a, a) : a \in \Sigma\}$ (representing idle local transitions), the transducer can be described by a union of the following regular expressions:

- $I^+(i, r)I^*$ — a client requesting for a resource.
- $(i, c)I^*(r, c)I^*$ — a client request granted by the allocator.
- $(c, i)I^*(c, i)I^*$ — the client has finished using the resource. □

Example 2. We describe Israeli-Jalfon self-stabilising protocol [23]. The original protocol is probabilistic, but since we are only interested in reachability, we may use nondeterminism to model randomness. The protocol has a ring topology, and each process either holds a token (denoted by $\top$) or does not hold a token (denoted by $\bot$). Dynamics is given by the following rules:

- A process P holding a token can pass the token to either the left or the right neighbouring process P', provided that P' does not hold a token.

[1] Length-preserving automatic transition systems are instances of automatic structures [8,9].

– If two neighbouring processes P_1 and P_2 hold tokens, the tokens can be merged and kept at process P_1.

We now provide a transducer that formalises this parameterised system. Our relation is on words over the alphabet $\Sigma = \{\bot, \top\}$, and thus a transducer is an automaton that runs over $\Sigma \times \Sigma$. In the following, we use $I := \{(\top, \top), (\bot, \bot)\}$. The automaton is given by a union of the following regular expressions:

– $I^*(\top, \bot)(\bot, \top)I^*$
– $I^*(\bot, \top)(\top, \bot)I^*$
– $I^*(\top, \top)(\top, \bot)I^*$
– $(\bot, \top)I^*(\top, \bot)$
– $(\top, \bot)I^*(\bot, \top)$
– $(\top, \bot)I^*(\top, \top)$ □

Example 3. Our next example is the classical David Gries's coffee can problem, which uses two (nonnegative) integer variables x and y to store the number of black and white coffee beans, respectively. At any given step, if $x + y \geq 2$ (i.e. there are at least two coffee beans), then two coffee beans are nondeterministically chosen. First, if both are of the same colour, then they are both discarded and a new black bean is put in the can. Second, if they are of a different colour, the white bean is kept and the black one is discarded. We are usually interested in the colour of the last bean in the can. We formally model Gries's coffee can problem as a transition system with domain $\mathbb{N} \times \mathbb{N}$ and transitions:

(a) if $x \geq 2$, then $x := x - 1$ and $y := y$.
(b) if $y \geq 2$, then $x := x + 1$ and $y := y - 2$.
(c) if $x \geq 1$ and $y \geq 1$, then $x := x - 1$ and $y := y$.

To distinguish the colour of the last bean, we shall add self-loops to all configurations in $\mathbb{N} \times \mathbb{N}$, except for the configuration $(1, 0)$. We can model the system as a length-preserving transducer as follows. The alphabet is $\Sigma := \Omega_{\mathtt{x}} \cup \Omega_{\mathtt{y}}$, where $\Omega_{\mathtt{x}} := \{1_{\mathtt{x}}, \bot_{\mathtt{x}}\}$ and $\Omega_{\mathtt{y}} := \{1_{\mathtt{y}}, \bot_{\mathtt{y}}\}$. A configuration is a word in the regular language $1^*_{\mathtt{x}}\bot^*_{\mathtt{x}}1^*_{\mathtt{y}}\bot^*_{\mathtt{y}}$. For example, the configuration with $x = 5$ and $y = 3$, where the maximum size of the integer buffers $\mathtt{x}$ and $\mathtt{y}$ is 10, is represented as the word $(1_{\mathtt{x}})^5(\bot_{\mathtt{x}})^5(1_{\mathtt{y}})^3(\bot_{\mathtt{y}})^7$. The transducer for the coffee can problem can be easily constructed. □

3.2 Representing Symmetry Patterns

Definition 2. *Let $\mathfrak{S} = \langle S; \rightarrow \rangle$ be a transition system with $S \subseteq \Sigma^*$. A symmetry pattern for $\mathfrak{S} = \langle S; \rightarrow \rangle$ is a simulation preorder $R \subseteq S \times S$ for $\mathfrak{S}$, i.e., satisfying:*

(S1) *R respects each $\rightarrow_a$, i.e., for all $v_1, v_2, w_1 \in S$, if $v_1 \rightarrow_a w_1$, and $(v_1, v_2) \in R$, then there exists $w_2 \in S$ such that $(w_1, w_2) \in R$ and $v_2 \rightarrow_a w_2$;*
(S2) *R is* length-decreasing*, i.e., for all $v_1, v_2 \in S$, if $(v_1, v_2) \in R$, then $|v_1| \geq |v_2|$.*

The symmetry pattern is said to be complete if additionally the relation is length-preserving and a bijective function.

Complete symmetry patterns will also be denoted by functional notation f. In the case of complete symmetry pattern f, it can be observed that Condition (S1) also entails that $f(v) \rightarrow_a f(w)$ implies $v \rightarrow_a w$. This condition does not hold in general for simulation preorders. We shall also remark that, owing to the well-known property of simulation preorders, symmetry patterns preserve non-safety. To make this notion more precise, we define the image of a transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ (with $S \subseteq \Sigma^*$) under the symmetry pattern R as the transition system $\mathfrak{S}_1 = \langle S_1; \rightarrow_1 \rangle$ such that $S_1 = R(S)$ and that $\rightarrow_1$ is the restriction of $\rightarrow$ to S_1.

Proposition 1. *Given two sets $I, F \subseteq \Sigma^*$, if $post^*_{\rightarrow_1}(R(I)) \cap R(F) = \emptyset$, then $post^*_{\rightarrow}(I) \cap F = \emptyset$.*

In other words, if $\mathfrak{S}_1$ is safe, then so is $\mathfrak{S}$. In the case when $\mathfrak{S}_1$ is finite-state, this check can be performed using a standard finite-state model checker. We shall define now a class of symmetry patterns under which the image $\mathfrak{S}_1$ of the input transition system can be automatically computed.

Definition 3 (Regular Symmetry Pattern). *A symmetry pattern $R \subseteq S \times S$ for an automatic transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ is said to be regular if the relation R is regular.*

Proposition 2. *Given an automatic transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ (with $S \subseteq \Sigma^*$) and a regular symmetry pattern $R \subseteq S \times S$, the image of $\mathfrak{S}$ under R is an automatic transition system and can be constructed in polynomial-time.*

In particular, whether the image of $\mathfrak{S}$ under R is a finite system can be automatically checked since checking whether the language of an NFA is finite can be done in polynomial-time. The proof of this proposition (in the full version) is a simple automata construction that relies on the fact that regular relations are closed under projections. We shall next illustrate the concept of regular symmetry patterns in action, especially for Israeli-Jalfon self-stabilising protocol and Gries's coffee can problem.

We start with Gries's coffee can problem (cf. Example 3). Consider the function $f : (\mathbb{N} \times \mathbb{N}) \rightarrow (\mathbb{N} \times \mathbb{N})$ where $f(x, y)$ is defined to be (i) $(0, 1)$ if y is odd, (ii) $(2, 0)$ if y is even and $(x, y) \neq (1, 0)$, and (iii) $(1, 0)$ if $(x, y) = (1, 0)$. This is a symmetry pattern since the last bean for the coffee can problem is white iff y is odd. Also, that a configuration (x, y) with $y \equiv 0 \pmod 2$ and $x > 1$ is mapped to $(2, 0)$ is because $(2, 0)$ has a self-loop, while $(1, 0)$ is a dead end. It is easy to show that f is a regular symmetry pattern. To this end, we construct a transducer for each of the cases (i)–(iii). For example, the transducer handling the case (x, y) when $y \equiv 1 \pmod 2$ works as follows: simultaneously read the pair (v, w) of words and ensure that $w = \bot_{\mathtt{x}} \bot_{\mathtt{x}} 1_y$ and $v \in 1^*_{\mathtt{x}} \bot^*_{\mathtt{x}} 1_{\mathtt{y}} (1_{\mathtt{y}} 1_{\mathtt{y}})^* \bot^*_{\mathtt{y}}$. As an NFA, the final transducer has ~ 10 states.

Process Symmetry Patterns. We now apply the idea of regular symmetry patterns to capture process symmetries in parameterised systems. We shall show how this applies to Israeli-Jalfon self-stabilising protocol. A *parameterised permutation* is a family $\bar{\pi} = \{\pi_n\}_{n \geq 1}$ of permutations π_n on $[n]$. We say that $\bar{\pi}$ is *regular* if, for each alphabet Σ, the bijection $f_{\bar{\pi}} : \Sigma^* \rightarrow \Sigma^*$ defined by $f_{\bar{\pi}}(\mathbf{v}) := \pi_n \mathbf{v}$,

where $\mathbf{v} \in \Sigma^n$, is a regular relation. We say that $\bar{\pi}$ is *effectively regular* if $\bar{\pi}$ is regular and if there is an algorithm which, on input Σ, constructs a transducer for the bijection $f_{\bar{\pi}}$. As we shall only deal with effectively regular permutations, when understood we will omit mention of the word "effectively". As we shall see below, examples of effectively regular parameterised permutations include transpositions (e.g. $\{(1,2)(3)\cdots(n)\}_{n\geq 2}$) and rotations $\{(1,2,\ldots,n)\}_{n\geq 1}$.

We now extend the notion of parameterised permutations to *parameterised symmetry groups* $\mathcal{G} := \{G_n\}_{n\geq 1}$ for parameterised systems, i.e., each G_n is a permutation group on $[n]$. A finite set $F = \{\bar{\pi}^1, \ldots, \bar{\pi}^r\}$ of parameterised permutations (with $\bar{\pi}^j = \{\pi_n^j\}_{n\geq 1}$) *generates* the parameterised symmetry groups $\mathcal{G}$ if each group $G_n \in \mathcal{G}$ can be generated by the set $\{\pi_n^j : j \in [r]\}$, i.e., the nth instances of parameterised permutations in F. We say that $\mathcal{G}$ is *regular* if each $\bar{\pi}^j$ in F is regular.

We will single out three commonly occuring process symmetry groups for concurrent systems with n processes: full symmetry group $\mathcal{S}_n$ (i.e. generated by $(1,2)$ and $(1,2,\ldots,n)$), rotation group $\mathcal{R}_n$ (i.e. generated by $(1,2,\ldots,n)$), and the dihedral group $\mathcal{D}_n$ (i.e. generated by $(1,2,\ldots,n)$ and the "reflection" permutation $(1,n)(2,n-1)\cdots(\lfloor n/2\rfloor, \lceil n/2\rceil)$). The parameterised versions of them are: (1) $\mathcal{S} := \{\mathcal{S}_n\}_{n\geq 1}$, (2) $\mathcal{R} := \{\mathcal{R}_n\}_{n\geq 1}$, and (3) $\mathcal{D} := \{\mathcal{D}_n\}_{n\geq 1}$.

Theorem 1. *Parameterised full symmetry groups $\mathcal{S}$ and parameterised rotation symmetry groups $\mathcal{R}$ are effectively regular.*

As we will see in Proposition 3 below, parameterised dihedral groups are not regular. We will say how to deal with this in the next section. As we will see in Theorem 4, Theorem 1 can be used to construct a fully-automatic method for checking whether *each* instance $\mathfrak{S}_n$ of a parameterised system $\mathfrak{S} = \{\mathfrak{S}_n\}_{n\geq 0}$ represented by a given transducer $\mathcal{A}$ has a full/rotation process symmetry group.

Proof (*Sketch of Theorem 1*). To show this, it suffices to show that $\mathcal{F} = \{(1,2)(3)\cdots(n)\}_{n\geq 2}$ and $\mathcal{F}' = \{(1,2,\ldots,n)\}_{n\geq 2}$ are effectively regular. [The degenerate case when $n = 1$ can be handled easily if necessary.] For, if this is the case, then the parameterised full symmetry $\mathcal{S}$ and the parameterised rotation symmetry groups can be generated by (respectively) $\{\mathcal{F}, \mathcal{F}'\}$ and $\mathcal{F}'$. Given an input Σ, the transducers for both $\mathcal{F}$ and $\mathcal{F}'$ are easy. For example, the transducer for $\mathcal{F}$ simply swaps the first two letters in the input, i.e., accepts pairs of words of the form (abw, baw) where $a, b \in \Sigma$ and $w \in \Sigma^*$. These transducers can be constructed in polynomial time (details in the full version). □

The above proof shows that $\{(1,2)(3)\cdots(n)\}_{n\geq 0}$ and $\{(1,2,\ldots,n)\}_{n\geq 0}$ are regular parameterised permutations. Using the same proof techniques, we can also show that the following simple variants of these parameterised permutations are also regular for each $i \in \mathbb{Z}_{>0}$: (a) $\{(i,i+1)(i+2)\cdots(n)\}_{n\geq 1}$, and (b) $\{(i,i+1,\ldots,n)\}_{n\geq 1}$. As we saw from Introduction, the prioritised resource-allocator protocol has a star topology and so both $\{(2,3)(4)\cdots\}_{n\geq 1}$ and $\{(2,3,\ldots,n)\}_{n\geq 1}$ generate complete symmetry patterns for the protocol (i.e. invariant under permuting the clients). Therefore, our library $\mathcal{L}$ of regular

symmetry patterns could store all of these regular parameterised permutations (up to some fixed i).

Parameterised dihedral groups $\mathcal{D}$ are generated by rotations $\bar{\pi} = \{(1, 2, \ldots, n)\}_{n\geq 2}$ and reflections $\bar{\sigma} = \{(1, n)(2, n-1)\cdots(\lfloor n/2\rfloor, \lceil n/2\rceil)\}_{n\geq 2}$. Reflections $\bar{\sigma}$ are, however, not regular for the same reason that the language of palindromes (i.e. words that are the same read backward as forward). In fact, it is not possible to find a different list of generating parameterised permutations that are regular (proof in the full version):

Proposition 3. *Parameterised dihedral groups $\mathcal{D}$ are not regular.*

4 Symmetry Verification

In this section, we will present our symmetry verification algorithm for regular symmetry patterns. We then show how to extend the algorithm to a more general framework of symmetry patterns that subsumes parameterised dihedral groups.

4.1 The Algorithm

Theorem 2. *Given an automatic transition system $\mathfrak{S} = \langle S; \rightarrow\rangle$ and a regular relation $R \subseteq S \times S$, we can automatically check if R is a symmetry pattern of $\mathfrak{S}$.*

Proof. Let D be the set of words over the alphabet Σ^3 of the form $v_1 \otimes v_2 \otimes w_1$, for some words $v_1, v_2, w_2 \in \Sigma^*$ satisfying: (1) $v_1 \rightarrow w_1$, (2) $(v_1, v_2) \in R$, and (3) there does *not* exist $w_2 \in \Sigma^*$ such that $v_2 \rightarrow w_2$ and $(w_1, w_2) \in R$. Observe that R is a symmetry pattern for $\mathfrak{S}$ iff D is empty. An automaton $\mathcal{A} = (\Sigma^3, Q, \Delta, q_0, F)$ for D can be constructed via a classical automata construction.

As before, for simplicity of presentation, we will assume that $S = \Sigma^*$; for, otherwise, we can perform a simple product automata construction with the automaton for S. Let $\mathcal{A}_1 = (\Sigma^2, Q_1, \Delta_1, q_0^1, F_1)$ be an automaton for $\rightarrow$, and $\mathcal{A}_2 = (\Sigma^2_{\#}, Q_2, \Delta_2, q_0^2, F_2)$ an automaton for R.

We first construct an NFA $\mathcal{A}_3 = (\Sigma^2_{\#}, Q_3, \Delta_3, q_0^3, F_3)$ for the set $Y \subseteq S \times S$ consisting of pairs (v_2, w_1) such that the condition (3) above is *false*. This can be done by a simple product/projection automata construction that takes into account the fact that R might not be length-preserving: That is, define $Q_3 := Q_1 \times Q_2$, $q_0^3 := (q_0^1, q_0^2)$, and $F_3 := F_1 \times F_2$. The transition relation Δ consists of transitions $((q_1, q_2), (a, b), (q_1', q_2'))$ such that, for some $c \in \Sigma_{\#}$, it is the case that $(q_2, (b, c), q_2') \in \Delta_2$ and one of the following is true: (i) $(q_1, (a, c), q_1') \in \Delta_1$, (ii) $q_1 = q_1'$, $b \neq \#$, and $a = c = \#$. Observe that the construction for $\mathcal{A}_3$ runs in polynomial-time.

In order to construct $\mathcal{A}$, we will have to perform a complementation operation on $\mathcal{A}_3$ (to compute the complement of Y) and apply a similar product automata construction. The former takes exponential time (since $\mathcal{A}_3$ is nondeterministic), while the latter costs an additional polynomial-time overhead. □

The above algorithm runs in exponential-time even if R and $\mathfrak{S}$ are presented as DFA, since an automata projection operation in general yields an NFA. The situation improves dramatically when R is *functional* (i.e. for all $x \in S$, there exists a unique $y \in S$ such that $R(x, y)$).

Theorem 3. *There exists a polynomial-time algorithm which, given an automatic transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ presented as a DFA and a functional regular relation $R \subseteq S \times S$ presented as an NFA, decides whether R is a symmetry pattern for $\mathfrak{S}$.*

Proof. Let D be the set of words over the alphabet Σ^4 of the form $v_1 \otimes v_2 \otimes w_1 \otimes w_2$, for some words $v_1, v_2, w_1, w_2 \in \Sigma^*$ satisfying: (1) $v_1 \rightarrow w_1$, (2) $(v_1, v_2) \in R$, (2') $(w_1, w_2) \in R$, and (3) $v_2 \not\rightarrow w_2$ Observe that R is a symmetry pattern for $\mathfrak{S}$ iff D is empty. The reasoning is similar to the proof of Theorem 2, but the difference now is that given any $w_1 \in \Sigma^*$, there is a *unique* w_2 such that $(w_1, w_2) \in R$ since R is functional. For this reason, we need only to make sure that $v_2 \not\rightarrow w_2$. An automaton $\mathcal{A}$ for D can be constructed by first complementing the automaton for $\rightarrow$ and then a standard product automata construction as before. The latter takes polynomial-time if $\rightarrow$ is presented as a DFA, while the latter costs an additional polynomial-time computation overhead (even if R is presented as an NFA). □

Proposition 4. *The following two problems are solvable in polynomial-space: given a regular relation $R \subseteq S \times S$, check whether (1) R is functional, and (2) R is a bijective function. Furthermore, the problems are polynomial-time reducible to language inclusion for NFA.*

Observe that there are fast heuristics for checking language inclusion for NFA using antichain and simulation techniques (e.g. see [2,10]). The proof of the above proposition uses standard automata construction, which is relegated to the full version.

4.2 Process Symmetries for Concurrent Systems

We say that an automatic transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ (with $S \subseteq \Sigma^*$) is *$\mathcal{G}$-invariant* if each instance $\mathfrak{S}_n = \langle S \cap \Gamma^n; \rightarrow \rangle$ of $\mathfrak{S}$ is G_n-invariant. If $\mathcal{G}$ is generated by regular parameterised permutations $\bar{\pi}^1, \ldots, \bar{\pi}^r$, then $\mathcal{G}$-invariance is equivalent to the condition that, for each $j \in [r]$, the bijection $f_{\pi^j} : \Sigma^* \rightarrow \Sigma^*$ is a regular symmetry pattern for $\mathfrak{S}$. The following theorem is an immediate corollary of Theorem 3.

Theorem 4. *Given an automatic transition system $\mathfrak{S} = \langle S; \rightarrow \rangle$ (with $S \subseteq \Sigma^*$) and a regular parameterised symmetry group $\mathcal{G}$ presented by regular parameterised permutations $\bar{\pi}^1, \ldots, \bar{\pi}^k$, we can check that $\mathfrak{S}$ is $\mathcal{G}$-invariant in polynomial-time assuming that $\mathfrak{S}$ is presented as DFA.*

In fact, to check whether $\mathfrak{S}$ is $\mathcal{G}$-invariant, it suffices to sequentially go through *each* $\bar{\pi}^j$ and ensure that it is a symmetry pattern for $\mathfrak{S}$, which by Theorem 3 can be done in polynomial-time.

4.3 Beyond Regular Symmetry Patterns

Proposition 3 tells us that regular symmetry patterns do not suffice to capture parameterised reflection permutation. This leads us to our inability to check whether a parameterised system is invariant under parameterised dihedral symmetry groups, e.g., Israeli-Jalfon's self-stabilising protocol and other randomised protocols including Lehmann-Rabin's protocol (e.g. [26]). To deal with this problem, we extend the notion of regular length-preserving symmetry patterns to a subclass of "context-free" symmetry patterns that preserves some nice algorithmic properties. *Proviso: All relations considered in this subsection are length-preserving.*

Recall that a *pushdown automaton (PDA)* is a tuple $\mathcal{P} = (\Sigma, \Gamma, Q, \Delta, q_0, F)$, where Σ is the input alphabet, Γ is the stack alphabet (containing a special bottom-stack symbol, denoted by $\perp$, that cannot be popped), Q is the finite set of control states, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is a set of final states, and $\Delta \subseteq (Q \times \Gamma) \times \Sigma \times (Q \times \Gamma^{\leq 2})$ is a set of transitions, where $\Gamma^{\leq 2}$ denotes the set of all words of length at most 2. A configuration of $\mathcal{P}$ is a pair $(q, w) \in Q \times \Gamma^*$ with *stack-height* $|w|$. For each $a \in \Sigma$, we define the binary relation $\rightarrow_a$ on configurations of $\mathcal{P}$ as follows: $(q_1, w_1) \rightarrow_a (q_2, w_2)$ if there exists a transition $((q_1, o), a, (q_2, v)) \in \Delta$ such that $w_1 = wo$ and $w_2 = wv$ for some $w \in \Gamma^*$. A *computation path* π of $\mathcal{P}$ on input $a_1 \dots a_n$ is any sequence

$$(q_0, \perp) \rightarrow_{a_1} (q_1, w_1) \rightarrow_{a_2} \cdots \rightarrow_{a_n} (q_n, w_n)$$

of configurations from the initial state q_0. In the following, the *stack-height sequence* of π is the sequence $|\perp|, |w_1|, \dots, |w_n|$ of stack-heights. We say that a computation path π is *accepting* if $q_n \in F$.

We now extend Theorem 4 to a class of transducers that allows us to capture the reflection symmetry. This class consists of "height-unambiguous" pushdown transducers, which is a subclass of pushdown transducers that is amenable to synchronisation. We say that a pushdown automaton is *height-unambiguous (h.u.)* if it satisfies the restriction that the stack-height sequence in an *accepting* computation path on an input word w is uniquely determined by the length $|w|$ of w. That is, given an accepting computation path π on w and an accepting computation path π' of w' with $|w| = |w'|$, the stack-height sequences of π and π' coincide. Observe that the definition allows the stack-height sequence of a non-accepting path to differ. A language $L \subseteq \Sigma^*$ is said to be *height-unambiguous context-free (huCF)* if it is recognised by a height-unambiguous PDA. A simple example of a huCF language is the language of palindromes (i.e. the input word is the same backward as forward). A simple non-example of a huCF language is the language of well-formed nested parentheses. This can be proved by a standard pumping argument.

We extend the definitions of regularity of length-preserving relations, symmetry patterns, etc. from Sects. 2 and 3 to height-unambiguous pushdown automata in the obvious way, e.g., a length-preserving relation $R \subseteq S \times S$ is *huCF* if $\{v \otimes w : (v, w) \in R\}$ is a huCF language. We saw in Proposition 3 that

parameterised dihedral symmetry groups $\mathcal{D}$ are not regular. We shall show now that they are huCF.

Theorem 5. *Parameterised dihedral symmetry groups $\mathcal{D}$ are effectively height-unambiguous context-free.*

Proof. To show this, it suffices to show that the parameterised reflection permutation $\bar{\sigma} = \{\sigma_n\}_{n\geq 2}$, where $\sigma_n := (1,n)(2,n-1)\cdots(\lfloor n/2\rfloor, \lceil n/2\rceil)$, is huCF. To this end, given an input alphabet Σ, we construct a PDA $\mathcal{P} = (\Sigma^2, \Gamma, Q, \Delta, q_0, F)$ that recognises $f_{\bar{\sigma}} : \Sigma^* \to \Sigma^*$ such that $f_{\bar{\sigma}}(\mathbf{v}) = \sigma_n \mathbf{v}$ whenever $\mathbf{v} \in \Sigma^n$. The PDA $\mathcal{P}$ works just like the PDA recognising the language of palindromes. We shall first give the intuition. Given a word w of the form $v_1 \otimes v_2 \in (\Sigma^2)^*$, we write w^{-1} to denote the word $v_2 \otimes v_1$. On an input word $w_1 w_2 w_3 \in (\Sigma^2)^*$, where $|w_1| = |w_3|$ and $|w_2| \in \{0,1\}$, the PDA will save w_1 in the stack and compares it with w_3 ensuring that w_3 is the reverse of w_1^{-1}. It will also make sure that $w_2 = (a,a)$ for some $a \in \Sigma$ in the case when $|w_2| = 1$. The formal definition of $\mathcal{P}$ is given in the full version. □

Theorem 6. *There exists a polynomial-time algorithm which, given an automatic transition system $\mathfrak{S} = \langle S; \to\rangle$ presented as a DFA and a functional h.u. context-free relation $R \subseteq S \times S$ presented as an NFA, decides whether R is a symmetry pattern for $\mathfrak{S}$.*

To prove this theorem, let us revisit the automata construction from the proof of Theorem 3. The problematic part of the construction is that we need to show that, given an huCF relation R, the 4-ary relation

$$\mathcal{R} := (R \times R) \cap \{(w_1, w_2, w_3, w_4) \in (\Sigma^*)^4 : |w_1| = |w_2| = |w_3| = |w_4|\} \qquad (*)$$

is also huCF. The rest of the construction requires only taking product with regular relations (i.e. $\to$ or its complement), which works for unrestricted pushdown automata since context-free languages are closed under taking product with regular languages via the usual product automata construction for regular languages.

Lemma 1. *Given an huCF relation R, we can construct in polynomial-time an h.u. PDA recognising the 4-ary relation $\mathcal{R}$.*

Proof. Given a h.u. PDA $\mathcal{P} = (\Sigma^2, \Gamma, Q, \Delta, q_0, F)$ recognising R, we will construct a PDA $\mathcal{P}' = (\Sigma^4, \Gamma', Q', \Delta', q_0', F')$ recognising $\mathcal{R}$. Intuitively, given an input $(v,w) \in \mathcal{R}$, the PDA $\mathcal{P}'$ is required to run two copies of $\mathcal{P}$ at the same time, one on the input v (to check that $v \in R$) and the other on input w (to check that $w \in R$). Since $\mathcal{P}$ is height-unambiguous and $|v| = |w|$, we can assume that the stack-height sequences of accepting runs of $\mathcal{P}$ on v and w coincide. That is, in an accepting run π_1 of $\mathcal{P}$ on v and an accepting run of π_2 of $\mathcal{P}$ on w, when a symbol is pushed onto (resp. popped from) the stack at a certain position in π_1, then a symbol is also pushed onto (resp. popped from) the stack in the same position in π_2. The converse is also true. These two stacks can, therefore, be

simultaneously simulated using only one stack of $\mathcal{P}'$ with $\Gamma' = \Gamma \times \Gamma$. For this reason, the rest of the details is a standard product automata construction for finite-state automata. Therefore, the automaton $\mathcal{P}'$ is of size quadratic in the size of $\mathcal{P}$. The detailed definition of $\mathcal{P}'$ is given in the full version. □

We shall finally pinpoint a limitation of huCF symmetry patterns, and discuss how we can address the problem in practice. It can be proved by a simple reduction from Post Correspondence Problem that it is undecidable to check whether a given PDA is height-unambiguous. In practice, however, this is not a major obstacle since it is possible to *manually* (or *semi-automatically*) add a selection of huCF symmetry patterns to our library $\mathcal{L}$ of regular symmetry patterns from Sect. 3. Observe that this effort is *independent* of any parameterised system that one needs to check for symmetry. Checking whether any huCF symmetry pattern in $\mathcal{C}$ is a symmetry pattern for a given automatic transition system $\mathfrak{S}$ can then be done automatically and efficiently (cf. Theorem 6). For example, Theorems 5 and 6 imply that we can automatically check whether an automatic transition system is invariant under the parameterised dihedral groups:

Theorem 7. *Given an automatic transition system $\mathfrak{S} = \langle S; \to \rangle$ (with $S \subseteq \Sigma^*$) presented as DFA, checking whether $\mathfrak{S}$ is $\mathcal{D}$-invariant can be done in polynomial-time.*

Among others, this allows us to automatically confirm that Israeli-Jalfon self-stabilising protocol is $\mathcal{D}$-invariant.

5 Automatic Synthesis of Regular Symmetry Patterns

Some regular symmetry patterns for a given automatic system might not be obvious, e.g., Gries's coffee can example. Even in the case of process symmetries, the user might choose different representations for the same protocol. For example, the allocator process in Example 1 could be represented by the last (instead of the first) letter in the word, which would mean that $\{(1, 2, \ldots, n-1)\}_{n\geq 3}$ and $\{(1,2)(3)\cdots(n)\}_{n\geq 3}$ are symmetry patterns for the system (instead of $\{(2, 3, \ldots, n)\}_{n\geq 2}$ and $\{(2,3)(4)\cdots(n)\}_{n\geq 3}$). Although we can put reasonable variations of common symmetry patterns in our library $\mathcal{L}$, we would benefit from a systematic way of synthesising regular symmetry patterns for a given automatic transition system $\mathfrak{S}$. In this section, we will describe our automatic technique for achieving this. We focus on the case of symmetry patterns that are *total functions* (i.e. homomorphisms), but the approach can be generalised to other patterns.

Every transducer $\mathcal{A} = (\Sigma_\# \times \Sigma_\#, Q, \delta, q_0, F)$ over $\Sigma_\#^*$ represents a regular binary relation R over Σ^*. We have shown in Sect. 4 that we can automatically check whether R represents a symmetry pattern, perhaps satisfying further constraints like functionality or bijectivity as desired by the user. Furthermore, we can also automatically check that it is a symmetry pattern for a given automatic transition system $\mathfrak{S}$. Our overall approach for computing such transducers makes use of two main components, which are performed iteratively within a refinement loop:

Synthesise. A candidate transducer $\mathcal{A}$ with n states is computed with the help of a SAT-solver, enforcing a relaxed set of conditions encoded as a Boolean constraint ψ (Sect. 5.1).

Verify. As described in Sect. 4, it is checked whether the binary relation R represented by $\mathcal{A}$ is a symmetry pattern for $\mathfrak{S}$ (satisfying further constraints like completeness, as desired by the user). If this check is negative, ψ is strengthened to eliminate counterexamples, and SYNTHESISE is invoked (Sect. 5.2).

This refinement loop is enclosed by an outer loop that increments the parameter n (initially set to some small number n_0) when SYNTHESISE determines that no transducers satisfying ψ exist anymore. The next sections describe the SYNTHESISE step, and the generation of counterexamples in case VERIFY fails, in more detail.

5.1 Synthesise: Computation of a Candidate Transducer $\mathcal{A}$

Our general encoding of transducers $\mathcal{A} = (\Sigma_\# \times \Sigma_\#, Q, \delta, q_0, F)$ uses a representation as a deterministic automaton (DFA), which is suitable for our refinement loop since counterexamples (in particular, words that should not be accepted) can be eliminated using succinct additional constraints. We assume that the states of the transducer $\mathcal{A}$ to be computed are $Q = \{1, \ldots, n\}$, and that $q_0 = 1$ is the initial state. We use the following variables to encode transducers with n states:

- x_t (of type Boolean), for each tuple $t = (q, a, b, q') \in Q \times \Sigma_\# \times \Sigma_\# \times Q$;
- z_q (of type Boolean), for each $q \in Q$.

The assignment $x_t = 1$ is interpreted as the existence of the transition t in $\mathcal{A}$. Likewise, we use $z_q = 1$ to represent that q is an accepting state in the automaton; since we use DFA, it is in general necessary to have more than one accepting state.

The set of considered transducers in step SYNTHESISE is restricted by imposing a number of conditions, selected depending on the kind of symmetry to be synthesised: for general symmetry homomorphisms, conditions **(C1)**–**(C8)** are used, for *complete* symmetry patterns **(C1)**–**(C10)**, and for *process symmetries* **(C1)**–**(C11)**.

(C1) The transducer $\mathcal{A}$ is deterministic.
(C2) For every transition $q \xrightarrow{(a,b)} q'$ in $\mathcal{A}$ it is the case that $a \neq \#$.[2]
(C3) Every state of the transducer is reachable from the initial state.
(C4) From every state of the transducer an accepting state can be reached.
(C5) The initial state q_0 is accepting.
(C6) The language accepted by the transducer is infinite.
(C7) There are no two transitions $q \xrightarrow{(a,b)} q'$ and $q \xrightarrow{(a,b')} q'$ with $b \neq b'$.

[2] Note that all occurrences of # are in the end of words.

(C8) If an accepting state q has self-transitions $q \xrightarrow{(a,a)} q$ for every letter $a \in \Sigma_{\#}$, then q has no outgoing edges.

(C9) For every transition $q \xrightarrow{(a,b)} q'$ in $\mathcal{A}$ it is the case that $b \neq \#$.

(C10) There are no two transitions $q \xrightarrow{(a,b)} q'$ and $q \xrightarrow{(a',b)} q'$ with $a \neq a'$.

Condition **(C2)** implies that computed transducers are length-decreasing, while **(C3)** and **(C4)** rule out transducers with redundant states. **(C5)** and **(C6)** follow from the simplifying assumption that only homomorphic symmetries patterns are computed, since a transducer representing a total function $\Sigma^* \to \Sigma^*$ has to accept the empty word and words of unbounded length. Note that **(C5)** and **(C6)** are necessary, but not sufficient conditions for total functions, so further checks are needed in Verify. **(C7)** and **(C8)** are necessary (but again not sufficient) conditions for transducers representing total functions, given the additional properties **(C3)** and **(C4)**; it can be shown that a transducer violating **(C7)** or **(C8)** cannot be a total function. Condition **(C9)** implies that padding # does not occur in any accepted word, and is a sufficient condition for length-preservation; as a result, the symbol # can be eliminated altogether from the transducer construction.

Finally, for *process symmetries* the assumption can be made that the transducer preserves not only word length, but also the number of occurrences of each symbol:

(C11) The relation R represented by the transducer only relates words with the same Parikh vector, i.e., $R(v, w)$ implies $\mathbb{P}(v) = \mathbb{P}(w)$.

The encoding of the conditions **(C1)**–**(C11)** as Boolean constraints is mostly straightforward. Further Boolean constraints can be useful in special cases, in particular for Example 3 the restriction can be made that only *image-finite* transducers are computed. We can also constrain the search in the Synthesise stage to those transducers that accept manually defined words $W = \{v_1 \otimes w_1, \ldots, v_k \otimes w_k\}$, using a similar encoding as the one for counterexamples in Sect. 5.2. This technique can be used, among others, to systematically search for symmetry patterns that generalise some known finite symmetry.

5.2 Counterexample Generation

Once a transducer $\mathcal{A}$ representing a candidate relation $R \subseteq \Sigma^* \times \Sigma^*$ has been computed, Theorem 2 can be used to implement the Verify step of the algorithm. When using the construction from the proof of Theorem 2, one of three possible kinds of counterexample can be detected, corresponding to three different formulae to be added to the constraint ψ used in the Synthesise stage:

1. A word v has to be included in the domain $R^{-1}(\Sigma^*_{\#})$: $\exists w.\ R(v, w)$
2. A word w has to be included in the range $R(\Sigma^*_{\#})$: $\exists v.\ R(v, w)$
3. One of two contradictory pairs has to be eliminated: $\neg R(v_1, w_1) \vee \neg R(v_2, w_2)$

Case 1 indicates relations R that are not total; case 2 relations that are not surjective; and case 3 relations that are not functions, not injective, or not simulations.[3] Each of the formulae can be directly translated to a Boolean constraint over the vocabulary introduced in Sect. 5.1. We illustrate how the first kind of counterexample is handled, assuming $v = a_1 \cdots a_m \in \Sigma_{\#}^*$ is the word in question; the two other cases are similar. We introduce Boolean variables $e_{i,q}$ for each $i \in \{0, \ldots, m\}$ and state $q \in Q$, which will be used to identify an accepting path in the transducer with input letters corresponding to the word v. We add constraints that ensure that exactly one $e_{i,q}$ is set for each state $q \in Q$, and that the path starts at the initial state $q_0 = 1$ and ends in an accepting state:

$$\left\{ \bigvee_{q \in Q} e_{i,q} \right\}_{i \in \{0,\ldots,m\}}, \quad \left\{ \neg e_{i,q} \vee \neg e_{i,q'} \right\}_{\substack{i \in \{0,\ldots,m\} \\ q \neq q' \in Q}}, \quad e_{0,1}, \quad \left\{ e_{m,q} \rightarrow z_q \right\}_{q \in Q} .$$

For each $i \in \{1, \ldots, m\}$ a transition on the path, with input letter a_i has to be enabled:

$$\left\{ e_{i-1,q} \wedge e_{i,q'} \rightarrow \bigvee_{b \in \Sigma} x_{(q,a_i,b,q')} \right\}_{\substack{i \in \{1,\ldots,m\} \\ q,q' \in Q}} .$$

Table 1. Experimental results on verifying and generating symmetry patterns

Symmetry Systems (#letters)	# Transducer states	Verif. time	Synth. time
Herman Protocol (2)	5	0.0 s	4 s
Israeli-Jalfon Protocol (2)	5	0.0 s	5 s
Gries's Coffee Can (4)	8	0.1 s	3 m19 s
Resource Allocator (3)	11	0.0 s	4 m56 s
Dining Philosopher (4)	17	0.4 s	26 m

6 Implementation and Evaluation

We have implemented a prototype tool based on the aforementioned approach for verifying and synthesising regular symmetry patterns. The programming language is Java and we use SAT4J [7] as the SAT solver. The source code and the benchmarking examples can be found at https://bitbucket.org/truongkhanh/parasymmetry. The input of our tool includes a model (i.e. a textual representation of transducers), and optionally a set of finite instance symmetries (to speed up synthesis of regular symmetry patterns), which can be generated using existing tools like [33].

[3] Note that this is for the special case of homomorphisms. Simulation counterexamples are more complicated than case 3 when considering simulations relations that are not total functions.

We apply our tool to 5 models: the Herman self-stabilising protocol [21], Israeli-Jalfon self-stabilising protocol [23], the Gries' coffee can example [20], Resource Allocator, and Dining Philosopher. For the coffee can example, the tool generates the functional symmetry pattern described in Sect. 3, whereas the tool generates rotational process symmetries for the other models (see the full version for state diagrams). Finite instance symmetries were added as constraints in the last three examples.

Table 1 presents the experimental results: the number of states of the synthesised symmetry transducer, the time needed to verify that the transducer indeed represents a symmetry pattern (using the method from Sect. 4), and the total time needed to compute the transducer (using the procedure from Sect. 5). The data are obtained using a MacBook Pro (Retina, 13-inch, Mid 2014) with 3 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory. In almost all cases, it takes less than 5 min (primarily SAT-solving) to find the regular symmetry patterns for all these models. As expected, the verification step is quite fast (<1 second).

7 Future Work

Describe the expressivity and nice algorithmic properties that regular symmetry patterns enjoy, we have pinpointed a limitation of regular symmetry patterns in expressing certain process symmetry patterns (i.e. reflections) and showed how to circumvent it by extending the framework to include symmetry patterns that can be recognised by height-unambiguous pushdown automata. One possible future research direction is to generalise our symmetry synthesis algorithm to this more general class of symmetry patterns. Among others, this would require coming up with a syntactic restriction of this "semantic" class of pushdown automata.

Acknowledgment. Lin is supported by Yale-NUS Grants, Rümmer by the Swedish Research Council. We thank Marty Weissman for a fruitful discussion.

References

1. Abdulla, P.A.: Regular model checking. STTT **14**(2), 109–118 (2012)
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). STTT **14**(2), 223–241 (2012)
4. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
5. Arenas, M., Barceló, P., Libkin, L.: Regular languages of nested words: fixed points, automata, and synchronization. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 888–900. Springer, Heidelberg (2007)

6. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. JSAT **7**(2–3), 56–59 (2010)
8. Blumensath, A.: Automatic structures. Master's thesis, RWTH Aachen (1999)
9. Blumensath, A., Grädel, E.: Finite presentations of infinite structures: Automata and interpretations. Theor. Comput. Syst. **37**(6), 641–674 (2004)
10. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, Rome, Italy - January 23–25, 2013, pp. 457–468 (2013)
11. Cameron, P.J.: Permutation Groups. London Mathematical Society Student Texts. Cambridge University Press, Cambridge (1999)
12. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods Syst. Des. **9**(1/2), 77–104 (1996)
13. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: Proceedings 1991 IEEE International Conference on Computer Design: VLSI in Computer and Processors, ICCD 1992, Cambridge, MA, USA, October 11–14, 1992, pp. 522–525 (1992)
14. Donaldson, A.F.: Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking. Ph.D. thesis, University of Glasgow (2007)
15. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: FM, pp. 631–631 (2005)
16. Donaldson, A.F., Miller, A.: Automatic symmetry detection for promela. J. Autom. Reasoning **41**, 251–293 (2008)
17. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Proceedings of the 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17–20, 2000, Automated Deduction - CADE-17, pp. 236–254 (2000)
18. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL, 85–94, (1995)
19. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods Syst. Des. **9**(1/2), 105–131 (1996)
20. Gries, D.: The Science of Programming. Springer-Verlag (1981)
21. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
22. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods Syst. Des. **9**, 41–75 (1996)
23. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: PODC, pp. 119–131 (1990)
24. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking rebeca. Acta Inf. **47**, 33–66 (2010)
25. Jaghoori, M.M., Sirjani, M., Mousavi, M.R.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: Chakraborty, G. (ed.) ICDCIT 2005. LNCS, vol. 3816, pp. 494–507. Springer, Heidelberg (2005)
26. Lehmann, D., Rabin, M.: On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In: Proceedings 8th Annual ACM Symposium on Principles of Programming Languages (POPL 1981), pp. 133–138 (1981)
27. Lin, A.W.: Accelerating tree-automatic relations. In: FSTTCS, pp. 313–324 (2012)
28. Lin, A.W., Nguyen, T.K., Rümmer, P., Sun, J.: Regular symmetry patterns (technical report). http://arxiv.org/abs/1510.08506 (cited in 2015)

29. Sistla, A.P., Gyuris, V., Emerson, E.A.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Trans. Softw. Eng. Method. **9**, 133–166 (2000)
30. Spermann, C., Leuschel, M.: ProB gets nauty: effective symmetry reduction for B and Z models. In: TASE, pp. 15–22 (2008)
31. Vojnar, T.: Cut-offs and automata in formal verification of infinite-state systems: Habilitation Thesis. Brno University of Technology, Faculty of Information Technology (2007)
32. Wahl, T., Donaldson, A.F.: Replication and abstraction: Symmetry in automated formal verification. Symmetry **2**, 799–847 (2010)
33. Zhang, S.J., Sun, J., Sun, C., Liu, Y., Ma, J., Dong, J.S.: Constraint-based automatic symmetry detection. In: ASE, pp. 15–25 (2013)