On Symmetries and Spotlights – Verifying Parameterised Systems

Nils Timm and Heike Wehrheim

Department of Computer Science, University of Paderborn
D-33098 Paderborn, Germany
{timm84,wehrheim}@uni-paderborn.de

Abstract. Parameterised model checking is concerned with verifying properties of arbitrary numbers of homogeneous processes composed in parallel. The problem is known to be undecidable in general. Nevertheless, a number of approaches have developed verification techniques for certain classes of parameterised systems. Here, we present an approach combining *symmetry* arguments with *spotlight* abstractions. The technique determines (the size of) a particular *instantiation* of the parameterised system from the given temporal logic formula, and feds this into an abstracting model checker. The *degree* of abstraction with respect to processes occurring during model checking determines whether the obtained result is also valid for all other instantiations. This enables us to prove *safety* as well as *liveness* properties (specified in full CTL) of parameterised systems on very small instantiations.

Keywords: symmetry reduction, spotlight abstraction, parameterised verification, model checking.

1 Introduction

Parameterised systems consist of an arbitrary number of homogeneous processes usually composed in parallel. The objective in verifying parameterised systems is to show certain correctness properties regardless of the number of processes involved. Examples can be found in all sorts of distributed algorithms, like mutual exclusion, leader election or cache coherence. Verifying parameterised systems is undecidable in general [3]. Model checking can of course prove properties for particular instantiations by fixing the number of processes, however, this is limited to reasonably sized numbers.

Nevertheless, a lot of approaches have been developed which verify particular classes of parameterised systems, or which devise methods that are sound but not complete. These include techniques which apply *regular model checking* [2], *induction* [15] or decision procedures for *second order logics* [4] to the verification of parameterised systems. Regular model checking represents sets of states by regular expressions and performs a reachability analysis by means of transducers. This technique is quite costly as it involves a number of automata theoretic constructions. In [1] a more efficient method has been proposed which however

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 534–548, 2010.

can only treat safety properties. The approach in [4] models parameterised systems in the logic WS1S and computes an abstraction of it on which properties can be shown. This technique requires to manually define abstraction relations. The invisible invariants technique of [15] computes inductive invariants on small instantiations by model checking and uses theorem proving for showing these to be inductive on the parameterised system as well. The method of [7] also uses model checking on small instances, where the number of instances (the *cutoff*) is computed from the description of the parameterised system, given this satisfies a particular format.

In this paper, we base our method on symmetry arguments. Symmetry and symmetry reductions [8,10] have long been proposed to reduce the state space in model checking. The general idea is to consider symmetric states, which only differ in permutations of values, as equivalent. Instead of constructing the whole state space, only equivalence classes (orbits) are built. These symmetries may refer to data values as well as process names. Parametric systems are inherently symmetric: usually either all processes are similar, or they can at least be divided into (a finite number of) classes of similar processes. The symmetry idea is applied in [7] to compute cutoffs, and in a sense also in [16], which counts the number of (symmetric) processes being in particular states.

Similar to [7] we will compute the size of an instance of the parameterised system. However, this size is not determined from the system description but straightforwardly from the temporal logic formula used to specify the correctness property. The size is simply 1 plus the number of different process *variables* occurring in the formula. If we for instance want to show a mutual exclusion property over a parameterised system, specified as

$$\forall\, i, j, i \neq j : AG\neg(i@crit \wedge j@crit)$$

(no processes i and j can ever be at their critical sections at the same time), the size is three (one process in addition to those mentioned). The parameterised system is instantiated with this number and the instantiation is fed into the model checker 3Spot [17]. 3Spot is our three-valued model checker [17] which employs both predicate abstraction and *spotlight abstraction* [18]. The third value "don't know" is used to represent unknowns which may arise due to the abstraction. Since we use a three-valued logic both true and false results can be transferred to the unabstracted system, only an unknown result necessitates abstraction refinement.

It is however the principle of spotlight abstraction which helps us towards our goal of proving (or disproving) the property not only for the particular instance but for the parameterised system as a whole. Spotlight abstractions completely abstract away the processes whose behaviour is irrelevant for the property to be checked. If the additional process in the instantiation is not drawn into the spotlight during the model checking run (i.e. completely abstracted away), the obtained result is valid for the parameterised system as well. Thus, it is the *degree of abstraction* occurring during the model checking run which tells us whether we can transfer our result to any number of processes. Since we employ

a three-valued model checker this holds for "true" and "false" results. Only in case that the additional process is moved into the spotlight, the result tells us nothing about the parameterised system.

The method currently works for completely symmetric systems (all processes the same, statements do not depend on process identifiers) as well as systems in which the processes can be divided into classes of similar programs. It allows for communication via shared variables. We exemplify the technique on a simple mutual exclusion and a readers/writers algorithm. The technique can be classified as being completely automatic, usually carrying out checks on very small instances (since the properties usually refer to a very limited number of different processes, most often just two) and thus being fast, and being sound but not complete.

2 Definitions

We look at systems of the following form (called *fully symmetric systems*):

$$\texttt{global } u_1 : \texttt{type}, \ldots, u_l : \texttt{type where } \varphi_{ginit}$$

$$\Big|\Big|_{i=1}^{n} P_i :: \left[\begin{array}{c}\texttt{local } v_1 : \texttt{type}, \ldots, v_h : \texttt{type where } \varphi_{linit} \\ \textit{some program text}\end{array}\right]$$

where $u_1, u_2 \ldots$ are *global* variables from some set V_g, $v_1, v_2 \ldots$ are local variables from a set V_l and P_1 to P_n are identical processes. For both local and global variables an initialisation is given in terms of a formula φ ($\varphi_{linit}, \varphi_{ginit}$, respectively). We assume this to give us a unique initial value for all variables (initialisation predicates are deterministic). We furthermore implicitly assume the set of local variables to contain a dedicated variable *pc*, a program counter. The numbers 1 to n act as *process identifiers*, from a set $PID = [1..n]$. Within a process P_i, its process id cannot be referred to, and the local state of P_i is not accessible by P_j, $i \neq j$. The processes are completely symmetric, i.e. each process P_i executes exactly the same code. For convenience we sometimes just write $P = ||_{i=1}^{n} P_i$.

The following example of a mutual exclusion algorithm by means of a semaphore serves for illustrating our technique for fully symmetric systems (written in a language similar to SPL [13]):

$$\texttt{global } y : \texttt{ sem where } y = 1$$

$$\Big|\Big|_{i=1}^{n} P_i :: \left[\begin{array}{l}\texttt{loop forever do} \\ \left[\begin{array}{ll}0: & \texttt{Non-Critical} \\ 1: & \texttt{request } y; \\ 2: & \texttt{Critical} \\ 3: & \texttt{release } y;\end{array}\right]\end{array}\right]$$

A *state* of such a system consists of a valuation of the global variables V_g and - for every process - valuations of the local variables, with values taken from some

domain D. We define $V = V_g \cup (V_l \times PID)$ to be the overall set of variables and hence a state is a mapping $s : V \to D$. We refrain from explicitly defining types and type-preserving assignments. We assume to have - amongst others - a domain for locations called *Loc*, and the variable *pc* is assigned to values of *Loc* only. The valuation of a global variable $v \in V_g$ in a state s is denoted by $s(v)$, the valuation of a local variable $v \in V_l$ of a process P_i in s is denoted by $s(v, i)$. The set $V_i = V_g \cup V_l$ is the set of variables of a single process P_i, and we write $s[i]$ to describe the local view of P_i on a state s: for $v \in V_g$, $s[i](v) = s(v)$, and for $v \in V_l$, $s[i](v) = s(v, i)$.

Transitions in such systems are caused by some local process P_i executing its next statement (if enabled). The transitions of process P_i are described by a predicate R_i on primed and unprimed global and local variables. The predicate R_i is derived from the program text, and consequently in our setting of fully symmetric systems predicates R_i are the same for all $i \in PID$. The predicate R_i for the mutual exclusion example is

$$\begin{aligned}&(pc = 0 \wedge pc' = 1 \wedge y' = y)\\ &\vee (pc = 1 \wedge y = 1 \wedge pc' = 2 \wedge y' = 0)\\ &\vee (pc = 2 \wedge pc' = 3 \wedge y' = y)\\ &\vee (pc = 3 \wedge y = 0 \wedge y' = 1 \wedge pc' = 0)\end{aligned}$$

We write $R_i(s[i], s'[i])$ to describe the case when the predicate R_i is true for the local views: $(s[i], s'[i]) \models R_i$.

As a computational model for our systems we use Kripke structures.

Definition 1. *A* Kripke structure *over a set of atomic propositions AP is a 4-tuple $K = (S, s_0, R, L)$ where*

- *S is a set of states,*
- *$s_0 \in S$ is the initial state,*
- *$R : S \times S \to \{true, false\}$ is a transition function,*
- *$L : S \times AP \to \{true, false\}$ is a function labelling states with atomic propositions.*

A path τ of a Kripke structure K is an infinite sequence of states $s_0 s_1 s_2 ...$ with $R(s_i, s_{i+1}) = true$; τ_i denotes the i-th state of τ and T_s denotes the set of all paths starting in $s \in S$.

As atomic propositions we use the following: $(v = d)$ for global variables $v \in V_g$ and data values $d \in D$, $(i@l)$ for a process id i and a location l, and $(v@i = d)$ for a local variable $v \in V_l$, a process id i and a data value d. For a symmetric system $P = ||_{i=1}^{n} P_i$, we define its Kripke structure $K = (S, s_0, R, L)$ as follows:

- States: $S := V \to D$ (the set of type-preserving valuations to variables),
- Initial state: $s_0 := s \in S$ with $s \restriction V_g \models \varphi_{ginit} \wedge \forall\, i \in PID : s[i] \models \varphi_{linit}$ (due to the initialisation predicates being deterministic, this gives us the same initial values for the local variables in all processes),

– Transition relation:

$$R(s,s') := \exists\, i \in [1..n] : R_i(s[i], s'[i]) \\ \wedge\ \forall j \neq i, \forall\, v \in V_l : s[j](v) = s'[j](v),$$

– Labelling function:

$$L(s,p) := \begin{cases} s(v) = d & \text{for } p \mathrel{\widehat{=}} (v = d) \\ s(pc,i) = l & \text{for } p \mathrel{\widehat{=}} (i@l) \\ s(v,i) = d & \text{for } p \mathrel{\widehat{=}} (v@i = d). \end{cases}$$

For specifying properties of Kripke structures we use the computational tree logic (CTL). In the next section we will see that CTL is the base logic only, the properties we actually like to show about parameterised systems are a little more complicated since we wish to quantify over processes.

Definition 2. *Let AP be a set of atomic propositions and $p \in AP$. The* syntax of CTL *is given by*

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid EX\psi \mid AX\psi \mid EF\psi \mid \\ AF\psi \mid EG\psi \mid AG\psi \mid E[\psi\, U\, \psi] \mid A[\psi\, U\, \psi].$$

The validity of a CTL formula ψ on a Kripke structure K is denoted by $K, s_0 \models \psi$.

Definition 3. *Let $K = (S, s_0, R, L)$ be a Kripke structure over AP, $p \in AP$ and $\psi \in CTL$. Then the evaluation of ψ in a state s of K, $K, s \models \psi$, is inductively defined as follows*

$$\begin{array}{ll} K, s \models p & := L(s,p) \\ K, s \models \neg\psi & := \neg(K, s \models \psi) \\ K, s \models \psi \vee \psi' & := K, s \models \psi \ \vee\ K, s \models \psi' \\ K, s \models EX\psi & := \bigvee_{s' \in S} R(s,s') \wedge K, s' \models \psi \\ K, s \models EG\psi & := \bigvee_{\tau \in T_s} \bigwedge_{i \in \mathbb{N}} (K, \tau_i \models \psi) \\ K, s \models E[\psi\, U\, \psi'] & := \bigvee_{\tau \in T_s} \bigvee_{i \in \mathbb{N}} ((K, \tau_i \models \psi') \wedge \bigwedge_{0 \leq j < i} (K, \tau_j \models \psi)) \end{array}$$

(The remaining CTL operators can be derived by the usual dualities.)

3 Symmetries

Here, we look at Kripke structures representing parameterised systems $P = ||_{i=1}^{n} P_i$. We like to show properties of such systems for all the processes in it, thus our property formulae take the following form:

$$\forall\, i_1, \ldots, i_d, (i_l \neq i_j)_{1 \leq l,j \leq d, l \neq j} : \psi(i_1, \ldots, i_d)$$

where i_1 to i_d are variables for process identifiers and $\psi \in$ CTL. The number d of process variables appearing in a formula F is called the *process diameter*

of F. The following two formulae express properties we would like to show for our mutual exclusion algorithm: $F_1 : \forall\, i, j, i \neq j : AG\neg(i@2 \wedge j@2)$ (a safety property about mutual exclusion, process diameter 2) and $F_2 : \forall\, i, j, i \neq j : AG((i@2 \wedge j@1) \Rightarrow AF(j@2))$ (a liveness property about starvation freedom, process diameter 2).

The ultimate goal is to show such properties $F(i_1, \ldots, i_d)$ for any number of processes running in parallel, i.e. show that $\forall\, N > d : K, s_0 \models F(i_1, \ldots, i_d)$ where K is the Kripke structure representing $||_{i=1}^{N}\, P_i$.

For the verification we want to exploit the symmetry in such systems and therefore use a technique inspired by symmetry reductions [8]. The symmetries concern process ids only, i.e. we permute processes, but no data variables.

Definition 4. *A* process permutation *is a bijective function* $\pi : PID \to PID$.

Process permutations can be lifted to states: for a state s we define $\pi(s)$ as follows: $\pi(s)(v) = s(v)$ in case of global variables $v \in V_g$, and $\pi(s)(v, i)$ is $s(v, \pi(i))$ in case of local variables including the program counters. Process permutations can also be applied to atomic propositions: $\pi(v = d) = (v = d), \pi(i@l) = (\pi(i)@l)$ and $\pi(v@i = d) = (v@\pi(i) = d)$.

For using process permutations for verification, they should in some sense preserve the semantics of systems:

Definition 5. *A process permutation* π *is a* symmetry *for a Kripke structure* $K = (S, s_0, R, L)$ *if the following conditions are met*

1. $R(s, s') \Leftrightarrow R(\pi(s), \pi(s'))$,
2. *for all atomic propositions* $p \in AP$*:*
   $L(s, p) \Leftrightarrow L(\pi(s), \pi(p))$,
3. $\pi(s_0) = s_0$.

Note that in contrary to permutations used in classical symmetry reduction, we apply the permutation on the atomic propositions as well, i.e. we do not require $L(s, p) \Leftrightarrow L(\pi(s), p)$.

Proposition 1. *On fully symmetric systems all process permutations are symmetries.*

Proof. First, we look at the transition relation. We only show the '⇒'-direction. The '⇐'-direction is proved analogously.

$$
\begin{aligned}
R(s, s') \Rightarrow \quad & \exists\, i \in [1..n] : R_i(s[i], s'[i]) \\
& \wedge\ \forall\, j \neq i, \forall\, v \in V_l : s[j](v) = s'[j](v) \\
\\
\Rightarrow \quad & \exists\, h = \pi(i) : R_{\pi(i)}(\pi(s[i]), \pi(s'[i])) \\
& \wedge\ \forall\, j \neq h, \forall\, v \in V_l : \pi(s[j])(v) = \pi(s'[j])(v) \\
\\
\Rightarrow\ & R(\pi(s), \pi(s'))
\end{aligned}
$$

Second, the labelling function. Here, we have to distinguish three cases:

1. $p \mathrel{\widehat{=}} (v = d)$:
   $L(s, (v = d))$
   $\Leftrightarrow s(v) = d$
   $\Leftrightarrow \pi(s)(v) = d$
   $\Leftrightarrow L(\pi(s), (v = d))$
   $\Leftrightarrow L(\pi(s), \pi(v = d))$
2. $p \mathrel{\widehat{=}} (i@l)$:
   $L(s, (i@l))$
   $\Leftrightarrow s(pc, i) = l$
   $\Leftrightarrow \pi(s)(pc, \pi(i)) = l$
   $\Leftrightarrow L(\pi(s), \pi(i)@l)$
   $\Leftrightarrow L(\pi(s), \pi(i@l))$
3. $p \mathrel{\widehat{=}} (v@i = d)$:
   $L(s, (v@i = d))$
   $\Leftrightarrow s(v, i) = d$
   $\Leftrightarrow \pi(s)(v, \pi(i)) = d$
   $\Leftrightarrow L(\pi(s), \pi(v@i = d))$ □

If a process permutation is a symmetry, then we can also permute paths in the Kripke structure, thereby again getting valid paths.

Proposition 2. *Let π be a symmetry for a Kripke structure $K = (S, s_0, R, L)$ and $\tau = s_0 s_1 s_2 \ldots$ a path in K. Then $\pi(\tau) = \pi(s_0)\pi(s_1)\pi(s_2)\ldots$ is a path in K as well with $L(s_i, p) = L(\pi(s_i), \pi(p))$ for all atomic propositions $p \in AP$ and $i \in \mathbb{N}$.*

This is a key property for our symmetry argument since it lets us prove properties about certain subsets of processes which will then also hold for other subsets gained by permutation. In general our approach to verifying properties about symmetric systems works as follows. For a formula $\forall\, i_1, \ldots, i_d : \psi(i_1, \ldots, i_d)$ we first of all only consider the temporal logic part, i.e. $\psi(i_1, \ldots, i_d)$. For determining its validity for a symmetric system, we instantiate the process variables i_j with concrete values $p_j \in PID$ (pairwise different) and look at the formula $\psi(p_1, \ldots, p_d)$. Our first theorem states that the application of a permutation does not change the validity of such properties.

Theorem 1. *Let $||_{i=1}^{n}\ P_i$ be a fully symmetric system, $K = (S, s_0, R, L)$ the corresponding Kripke structure over AP and π a process permutation which is a symmetry for K. Moreover, let ψ be a CTL formula and $s \in S$. Then*

$$K, s \models \psi(p_1, \ldots, p_d) \;\Leftrightarrow\; K, \pi(s) \models \psi(\pi(p_1), \ldots, \pi(p_d)) \ .$$

Proof. Induction on the structure of ψ. The argumentation is based on Definition 5 and Proposition 2. For short we write ψ for $\psi(p_1, \ldots, p_k)$ and $\pi(\psi)$ for $\psi(\pi(p_1), \ldots, \pi(p_k))$.

– $\psi = p,\ p \in AP$:
$K, s \models p$
$\Leftrightarrow\ L(s, p)$
$\Leftrightarrow\ L(\pi(s), \pi(p))$
$\Leftrightarrow\ K, \pi(s) \models \pi(p)$

– $\psi = \neg\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ K, s \not\models \psi_1$
$\Leftrightarrow\ K, \pi(s) \not\models \pi(\psi_1)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = \psi_1 \vee \psi_2$:
$K, s \models \psi$
$\Leftrightarrow\ K, s \models \psi_1 \vee K, s \models \psi_2$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi_1) \vee K, \pi(s) \models \pi(\psi_2)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = EX\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{s' \in S} R(s, s') \wedge K, s' \models \psi_1$
$\Leftrightarrow\ \bigvee_{\pi(s') \in S} R(\pi(s), \pi(s')) \wedge K, \pi(s') \models \pi(\psi_1)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = EG\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{\tau \in T_s} \bigwedge_{i \in \mathbb{N}} (K, \tau_i \models \psi_1)$
$\Leftrightarrow\ \bigvee_{\pi(\tau) \in T_{\pi(s)}} \bigwedge_{i \in \mathbb{N}} (K, \pi(\tau_i) \models \pi(\psi_1))$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = E[\psi_1 U \psi_2]$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{\tau \in T_s} \bigvee_{i \in \mathbb{N}} ((K, \tau_i \models \psi_2) \wedge \bigwedge_{0 \leq j < i} (K, \tau_j \models \psi_1))$
$\Leftrightarrow\ \bigvee_{\pi(\tau) \in T_{\pi(s)}} \bigvee_{i \in \mathbb{N}} ((K, \pi(\tau_i) \models \pi(\psi_2)) \wedge \bigwedge_{0 \leq j < i} (K, \pi(\tau_j) \models \pi(\psi_1)))$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$ □

This is all we need with respect to symmetries: for symmetric systems a property is true if and only if its permutation is true. Up to here, we however have no means of dealing with the $\forall\, N > d$ in our formulae. For parameterisation we next combine this technique with spotlight abstractions.

4 Spotlights

By having instantiated our formula with concrete process ids, we have obtained a *local* property, referring to only some of the processes in *PID*. Instantiations of F_1 and F_2 could be $AG\neg(1@2 \wedge 2@2)$ and $AG((1@2 \wedge 2@1) \Rightarrow AF(2@2))$, respectively. Local properties of parallel processes can be checked using spotlight

abstractions and the tool 3Spot. 3Spot is a verification tool based on the concept of predicate abstraction and abstraction refinement [17], which - however - does not only apply predicate abstraction to the code of processes in a parallel composition but also abstracts away complete processes. Those processes that are referred to in the property to be checked are taken into the *spotlight* whereas all others are kept in the *shade*. On the processes in the spotlight we apply ordinary predicate abstraction. The processes in the shade are automatically abstracted into one approximative process $P_\perp$. This process only coarsely reflects the behaviour of the shade processes. In particular, $P_\perp$ neglects the original control flow of the processes in the shade. Instead it approximates operations on global variables occurring in shade processes by continuously modifying predicates over those variables. Due to the approximative character of $P_\perp$ and the inherent loss of information about the shade processes, predicates might be set to *unknown*. "Unknown" is in fact a valid value as we operate with three-valued logics (Kleene logic [9]), where predicates can be true, false or unknown.

Spotlight abstraction now works as follows:

1. We start with a spotlight which only contains those processes the property formula speaks about. For the verification, we construct a predicate abstraction of these processes and combine this in parallel with the approximative process representing all processes in the shade.
2. On this abstraction the formula is checked. If the check returns true or false, we are done. The result also holds for the non-abstracted, original system.
3. If the check returns "unknown", the abstraction needs to be refined. We either add a new predicate, or take one process out of the shade into the spotlight. Then we proceed with step 2.

Note that $P_\perp$ only modifies predicates about *global* variables. Thus, any set of shade processes which share the same operations on global variables V_g will give us the same approximative process $P_\perp^{V_g}$. For our semaphore example and the predicate $(y = 1)$ an arbitrary number of processes in the shade would be automatically abstracted into

$$P_\perp^{\{y\}} :: \left[\texttt{loop forever do} \quad (y=1) := \begin{cases} \mathit{false} & \texttt{if } (y=1) = \mathit{false} \\ \mathit{unknown} & \texttt{else} \end{cases} \right]$$

approximating the possible operations `request` and `release` on the global semaphore variable y.

For employing 3Spot in our verification procedure for parametric systems, we do not only have to instantiate process variables in the formula, but we also have to fix the number n of processes in $||_{i=1}^{n} P_i$. This number is set to $d+1$, i.e. one more than the process diameter of the formula, in the case of formula F_1 thus to 3. Now we check the following property with 3Spot

$$P_1 \,||\, P_2 \,||\, P_3 \models AG\neg(1@2 \wedge 2@2) \; ?$$

Since the property is only referring to processes 1 and 2, the abstraction starts with 1 and 2 in the spotlight. It turns out that throughout the whole model checking procedure process 3 is kept in the shade and the verification succeeds with a definite answer (yes) without ever considering process 3. More specifically, 3Spot shows the following to be true:

$$P_1 \mid\mid P_2 \mid\mid P_\perp^{\{y\}} \models AG\neg(1@2 \wedge 2@2)$$

Here, $P_\perp^{\{y\}}$ is the abstraction of process 3. As explained above this is also the abstraction of *any* number of parallel processes performing request and release operations on the global variable y. The correctness result for spotlight abstraction now lets us transfer this result to the original parallel system.

Theorem 2. *Let* $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ *be a spotlight abstraction of a symmetric system for which checking property* $\psi(1,\ldots,d)$ *yields true. Then for any* $N > d$ *we get*

$$||_{i=1}^{N}\ P_i \models \psi(1,\ldots,d)\ .$$

Proof: In [17] we have shown that if $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ is a spotlight abstraction of a parallel system with a *fixed* number of processes and $\psi(1,\ldots,d)$ yields true for $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ then we can transfer this result to the unabstracted system. Here, we consider *symmetric* systems of an *arbitrary* size. Due to symmetry, the approximative process $P_\perp^{V_g}$ and therefore the entire spotlight abstraction is the same for *any* number $N > d$ of processes. □

In the next step, we need to transfer the result for particular process identities to arbitrary process variables. Theorem 1 exactly allows us to do this: verification results for $(1,\ldots,d)$ also hold for all permutations of $(1,\ldots,d)$. Since all possible instantiations of process variables i_1 to i_d can be obtained this way (note that we required all variable values to be pairwise different), the verification result also holds in general.

Corollary 1. *Let* $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ *be a spotlight abstraction of a symmetric system for which checking property* $\psi(1,,\ldots,d)$ *yields true. Then the following holds:*

$$\forall\, N > d : ||_{i=1}^{N}\ P_i \models \forall\, i_1,\ldots,i_d, (i_l \neq i_j)_{1\leq l,j\leq d, l\neq j} : \psi(i_1,\ldots,i_d)$$

An analogue result is achieved for negative outcomes of the model checking runs: outcome "false" can also be transferred to the full system. In summary, we have the following verification steps: For a symmetric parameterised system $||_{i=1}^{n}\ P_i$ and a formula F execute the following steps for checking $\forall\, N : ||_{i=1}^{N}\ P_i \models F$:

1. Determine the process diameter d of F.
2. Instantiate F with process identities 1 to d for i_1 to i_d: $\psi(1,\ldots,d)$.
3. Check $P_1 \mid\mid \ldots \mid\mid P_d \mid\mid P_{d+1} \models \psi(1,\ldots,d)$.
4. If the result is "yes"/"no" and P_{d+1} is *not* in the spotlight in the final abstraction, return *true*/*false*. Else return *unknown*.

The obtained result is by symmetry then a valid result for the parameterised system. In this manner we can also disprove the liveness property F_2 by having only two processes in the spotlight.

3Spot might take the process P_{d+1} into the spotlight. In this case a definite outcome cannot be transferred to the full system because then there is no approximative process representing an arbitrary number of additional process instances. Hence, it is *unknown* whether ψ holds for *all* instantiations of the parameterised system. So this approach gives us a fully automatic, sound but not complete procedure for reasoning about parameterised, symmetric systems.

5 Generalisation

Our approach can be generalised to *classwise symmetric systems*. Such systems are not fully symmetric, but they consist of classes of symmetric processes. An example for a system consisting of two classes is the following solution to the readers/writers problem:

$$\texttt{global } y : \texttt{ sem where } y = n_{Rd}$$

$$\Big\|_{i \in PID^{Rd}} Reader_i :: \left[\begin{array}{l}\texttt{loop forever do}\\ \left[\begin{array}{ll}0: & \texttt{Non-Critical}\\ 1: & \texttt{request } (y,1);\\ 2: & \texttt{Critical-Write}\\ 3: & \texttt{release } (y,1);\end{array}\right]\end{array}\right]$$

$$\Big\|$$

$$\Big\|_{j \in PID^{Wrt}} Writer_j :: \left[\begin{array}{l}\texttt{loop forever do}\\ \left[\begin{array}{ll}0: & \texttt{Non-Critical}\\ 1: & \texttt{request } (y,n_{Rd});\\ 2: & \texttt{Critical-Write}\\ 3: & \texttt{release } (y,n_{Rd});\end{array}\right]\end{array}\right]$$

Here, we have two classes of processes, readers and writers. Furthermore, n_{Rd} is a parameter associated with the number of reader processes. The semantics of the semaphore statements `request` (y, c) and `release` (y, c) are given by $\langle$`await` $y \geq c;\ y := y - c\rangle$ and $\langle y := y + c\rangle$, respectively. Thus, the generalised semaphore y ensures that multiple (up to n_{Rd}) readers can enter the critical section at the same time, whereas if one writer is modifying data, no other process has access to the critical section. All readers and all writers execute the same program code. More generally, a classwise symmetric system consisting of k classes is defined as $P = \|_{m=1}^{k}\ P^m$ where $P^m = \|_{i \in PID^m}\ P_i$ is a parallel composition of fully symmetric processes of a class m with process identifiers from a set PID^m. All sets PID^m, where $m \in [1..k]$, are pairwise disjoint. Thus,

every process in a classwise symmetric system has a unique id. All classes share a set of global variables V_g whereas each class has a distinct set of local variables V_l^m with $\forall\, m_1, m_2 \in [1..k] : V_l^{m_1}, V_l^{m_2}$ pairwise disjoint. Furthermore, each V_l^m contains a dedicated program counter pc^m. The overall set of variables of P is $V = V_g \cup \bigcup_{m=1}^{k}(V_l^m \times PID^m)$.

Given this setting, we can define the semantics in a way similar to fully symmetric systems. States of a classwise symmetric system are defined as mappings $s : V \to D$. The local view of a process P_i in a state s is again denoted by $s[i]$, referring to its unique identifier i. Transitions in such a system caused by the execution of a statement in some process P_i are described by a predicate R_i on the primed and unprimed local views of P_i. Due to the symmetry of processes in a class m, predicates R_i are the same for all $i \in PID^m$.

Classwise symmetric systems can be represented as Kripke structures as well. As atomic propositions we use the same as before: $(v = d)$, $(i@l)$ and $(v@i = d)$. For a system $P = ||_{m=1}^{k}\ P^m$ with $P^m = ||_{i \in PID^m}\ P_i$ the corresponding Kripke Structure $K = (S, s_0, R, L)$ is defined as follows:

- States: $S := V \to D$,
- Initial state: $s_0 := s \in S$ with $s \restriction V_g \models \varphi_{ginit} \wedge \forall\, m \in [1..k], \forall\, i \in PID^m : s[i] \models \varphi_{linit^m}$,
- Transition function:

$$\begin{aligned} R(s, s') := \exists\, m \in [1..k], \exists\, i \in PID^m : R_i(s[i], s'[i]) \\ \wedge\ \forall\, i_2 \in PID^m, i_2 \neq i, \forall\, v \in V_l^m : s[i_2](v) = s'[i_2](v) \\ \wedge\ \forall\, m_2 \neq m, \forall\, j \in PID^{m_2}, \forall\, v \in V_l^{m_2} : s[j](v) = s'[j](v), \end{aligned}$$

- Labelling function:

$$L(s,p) := \begin{cases} s(v) = d & \text{for } p \mathrel{\widehat{=}} (v = d) \\ s(pc^m, i) = l & \text{for } p \mathrel{\widehat{=}} (i@l), i \in PID^m \\ s(v, i) = d & \text{for } p \mathrel{\widehat{=}} (v@i = d), i \in PID^m, v \in V^m \end{cases}$$

For classwise symmetric systems $P = ||_{m=1}^{k}\ P^m$ we like to show properties that refer to distinct classes but arbitrary processes in each class. More precisely, a property formula takes the following form:

$$F = \forall\, i_1^1, \ldots, i_{d_1}^1, \ldots, \forall\, i_1^k, \ldots, i_{d_k}^k : \psi(i_1^1, \ldots, i_{d_1}^1, \ldots, i_1^k, \ldots, i_{d_k}^k)$$

where $\psi \in$ CTL and i_1^m to $i_{d_m}^m$ with $m \in [1..k]$ are pairwise different variables for process identifiers in PID^m. Thus, for every class m referred in F there is a distinct process diameter d_m. For our readers/writers example we have two classes of symmetric processes: *readers* with the corresponding set of process identifiers PID^{Rd} and *writers* with PID^{Wrt}. A safety property about mutual exclusion (no reading and writing at the same time, and no writing concurrently) can be formalised as: $F_3 : \forall\, i \in PID^{Rd}, \forall\, j_1, j_2 \in PID^{Wrt} : AG\neg(i@2 \wedge j_2@2) \wedge AG\neg(j_1@2 \wedge j_2@2)$ where the process diameter is 1 for readers and 2 for writers.

Since we like to show such properties for an arbitrary number of processes from each class running in parallel, we want exploit the symmetry again. As the considered systems are not fully but classwise symmetric, we need a new notion of process permutations.

Definition 6. *A* class-sensitive process permutation *is a bijective function* $\pi : \bigcup_{m=1}^{k} PID^m \to \bigcup_{m=1}^{k} PID^m$ *with* $i \in PID^m \Leftrightarrow \pi(i) \in PID^m$ *for all* $m \in [1..k]$ *and* $i \in PID^m$.

Class-sensitive permutations preserve the class affiliation of process identifiers. Hence, on classwise symmetric systems all class-sensitive process permutations are symmetries. Alike the fully symmetric case, we get the following result:

Theorem 3. *Let* $||_{m=1}^{k} P^m$ *be a classwise symmetric system,* $K = (S, s_0, R, L)$ *the corresponding Kripke structure over AP and* π *a class-sensitive process permutation which is a symmetry for* K*. Moreover, let* $s \in S$ *and* ψ *be a CTL formula referring to concrete process identifiers* $p_j^m \in PID^m$, $j \in [1..d_m]$, $m \in [1..k]$*. Then*

$$\begin{aligned} & K, s \models \psi(p_1^1, \ldots, p_{d_1}^1, \ldots, p_1^k, \ldots, p_{d_k}^k) \\ \Leftrightarrow\ & K, \pi(s) \models \psi(\pi(p_1^1), \ldots, \pi(p_{d_1}^1), \ldots, \pi(p_1^k), \ldots, \pi(p_{d_k}^k)) \ . \end{aligned}$$

Thus, applying class-sensitive permutations preserves the validity of CTL properties referring to *particular* processes of a classwise symmetric system. Via the spotlight technique we can extend this result to formulas referring to *arbitrary* processes. The process $P_{\perp}^{V_g}$ now approximates all operations on global variables occurring in *any* class of the considered system. Here, $PID_{d_m}^m$ denotes an arbitrary subset of PID^m with d_m elements.

Theorem 4. *Let* $(||_{m=1}^{k}||_{i \in PID_{d_m}^m} P_i) \,||\, P_{\perp}^{V_g}$ *be a spotlight abstraction of a classwise symmetric system for which checking property* $\psi(p_1^1, ..., p_{d_1}^1, ..., p_1^k, ..., p_{d_k}^k)$ *yields true. Then the following holds:*

$$\begin{aligned} & \forall\, N_1 > d_1, ..., N_k > d_k : \\ & ||_{m=1}^{k}||_{i \in PID_{N_m}^m} P_i \models \forall\, i_1^1, ..., i_{d_1}^1, ..., \forall\, i_1^k, ..., i_{d_k}^k : \psi(i_1^1, ..., i_{d_1}^1, ..., i_1^k, ..., i_{d_k}^k) \end{aligned}$$

As before, negative outcomes can be transferred to the original system as well. This result lets us exploit once again symmetries to verify parameterised systems by using spotlight abstractions; now for a generalised notion of symmetric systems, called classwise symmetric systems. For our readers/writers example we can prove the safety property F_3 via 3Spot by taking just one reader process and two writer processes into the spotlight. The verification was performed in $0.83s$ (on a 2.40GHz Core 2 Duo Windows system with 3GB memory).

6 Conclusion

In this paper we have proposed a verification technique for parameterised systems. It provides for a sound proof technique for full CTL properties by a combination

of symmetry reduction and spotlight abstraction. In case that the spotlight abstraction draws too many processes into the spotlight, the validity of properties for the parameterised system is however left open. In the future we therefore intend to investigate whether this technique can give us more results when we use larger instantiations, i.e. larger than "1 plus process diameter". This might be essential for reasoning about several liveness properties: the formula $F_4 : \forall\, i : AG(i@1 \Rightarrow AF(i@2))$ refers to a single process only. Thus, the process diameter is one. However, for proving or disproving starvation freedom it is obviously necessary to have at least one "adversary" process in the spotlight and therefore an instantiation larger than $d+1$. Further investigation might allow us to find appropriate instantiation sizes for distinct types of temporal logic formulae. Moreover, our approach could be easily modified to an iterative but possibly infinitely running procedure: We gradually add processes $P_{d+2}, P_{d+3}, \ldots$ to the instantiation until we get a definite answer without having all process instances in the spotlight. This might not terminate in all cases (e.g. when the property cannot be proven on a finite-state abstraction). Anyway, the formula F_4 can be disproved for the mutual exclusion example in this way within two iterations and $0.25s$.

Related work. The generally undecidable problem of parameterised verification has received a lot of attention in research. Several approaches (e.g. [8,5,7,14]) try to bypass this problem by summarising the considered system by a finite instantiation, based on symmetry arguments. From Clarke et al. [5] we have taken the idea of exploiting symmetry under permutations. In our work permutations are furthermore applied to atomic propositions which is also done in [11]. The work most closely related to ours is that of Emerson and Kahlon [7] who reduce reasoning for parameterised systems of an arbitrary size to systems of a small cutoff size. Contrary to the process diameter in our technique the cutoff size does not depend on the property but on the description of the system. Moreover, their method is complete but restricted to processes of a certain structure and properties in a fragment of CTL$^*\backslash$X. Similar to [7], Namjoshi [14] proposes a cutoff-based verification technique for parameterised systems with less restrictions to the structure of the processes but limited to safety properties.

The principle of spotlight abstraction was first introduced by Wachter and Westphal [18] and later enriched with three-valued semantics in our previous work [17]. To the best of our knowledge, we are the first to combine symmetry reduction and spotlight abstraction in parameterised model checking. In former approaches symmetry reduction has been used in combination with partial order reduction [6] and heuristic search [12].

Acknowledgement. We thank Daniel Wonisch for help with extending 3Spot.

References

1. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Monotonic abstraction: on efficient verification of parameterized systems. Int. J. Found. Comput. Sci. 20(5), 779–801 (2009)

2. 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)
3. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
4. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: Safety and liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
5. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
6. Emerson, E., Jha, S., Peled, D.: Combining partial order and symmetry reductions. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 19–34. Springer, Heidelberg (1997)
7. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
8. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
9. Fitting, M.: Kleene's three valued logics and their children. Fundamenta Informaticae 20(1-3), 113–131 (1994)
10. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
11. Leuschel, M., Butler, M.J., Spermann, C., Turner, E.: Symmetry reduction for b by permutation flooding. In: B, pp. 79–93 (2007)
12. Lluch-Lafuente, A.: Symmetry reduction and heuristic search for error detection in model checking. In: 2nd Workshop on Model Checking and Artificial Intelligence (MoChArt), pp. 77–86 (2003)
13. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
14. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
15. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
16. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infinity)-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
17. Schrieb, J., Wehrheim, H., Wonisch, D.: Three-valued spotlight abstractions. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 106–122. Springer, Heidelberg (2009)
18. Wachter, B., Westphal, B.: The spotlight principle. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 182–198. Springer, Heidelberg (2007)