Synchronisation in Language-level Symmetry Reduction for Probabilistic Model Checking

Ivaylo Valkov$^{1[0000-0003-1116-875X]}$,
Alastair F. Donaldson$^{2[0000-0002-7448-7961]}$, and Alice Miller$^{1[0000-0002-0941-1717]}$

^{1} University of Glasgow, UK
`2064491v@student.gla.ac.uk`, `alice.miller@glasgow.ac.uk`
^{2} Imperial College London, UK
`alastair.donaldson@imperial.ac.uk`

Abstract. The *generic representatives* (or *counter abstraction*) approach has been shown to be an effective symmetry reduction method for model checking. This method was extended to a probabilistic setting via a specialised language, *Symmetric Probabilistic Specification Language* (SPSL) and an associated tool, GRIP, for use with the PRISM model checker. However, SPSL does not support synchronisation-based communication, making this method inapplicable to systems that require synchronisation. We show how synchronisation can be added to SPSL, and develop new counter abstraction translation rules for synchronous statements. We extend GRIP accordingly and demonstrate the feasibility and effectiveness of the new abstraction rules via a range of examples. This extends the applicability of the generic representatives technique to the wide class of probabilistic systems that rely on synchronisation. Experimental results show that our approach works well for systems that are composed of a large number of simple symmetric modules that feature a small amount of synchronisation-based communication.

Keywords: Probabilistic model checking · Symmetry reduction · Generic representatives · Counter abstraction · Synchronisation · PRISM.

1 Introduction

Model checking [8, 26, 9] is an automatic technique for verifying hardware and software systems by checking temporal logic properties against a finite state model of a system. Explicit state model checkers [21] such as SPIN [20] store each state individually, whereas symbolic model checkers such NuSMV [7] use a symbolic representation of states, typically through the use of Binary Decision Diagrams (BDDs) [3, 4]. Probabilistic model checkers, such as PRISM [24] and Storm [19] incorporate probabilities and quantitative aspects into the (symbolic) verification process by using Multi-terminal Binary Decision Diagrams (MTBDDs) [10] to store vectors of probabilities.

Model checking suffers from the so-called *state-space explosion problem*—the number of states increases exponentially with the number of components in a system. The symbolic approach goes some way to address this issue [5]. However, in many cases replicated components in the system under analysis can lead to large portions of the state-space that are symmetric, and a symbolic representation does nothing to avoid redundancy in analysis arising from this symmetry.

Symmetry reduction [11, 16, 25, 28] is a technique that was originally introduced for explicit state model checking to combat state-space explosion arising from this kind of replication of components. Symmetries of the system are used to partition the state-space into equivalence classes. The model checker then only needs to explore one representative state from each equivalence class. The construction of the equivalence classes is done by identifying a suitable relation known as the *orbit relation*.

Working with the orbit relation can be challenging—especially in the case of symbolic model checking where it has been shown that a BDD encoding of the orbit relation has size exponential in the number of replicated components [11]. An alternative approach using *generic representatives* [16, 17], also known as *counter abstraction*, allows symmetry reduction to be applied without the construction of the orbit relation. A system specification is translated into a reduced form known as *generic form*. In the generic form, the full set of individual local variables is replaced by a much smaller set of variables called *counters*, which record the number of components in each local state. Both the reduced specification and resulting model can be significantly smaller than the original.

Symmetry reduction tools have been developed for a variety of model checkers [18, 2, 14]. In the probabilistic context there have been two notable tools: PRISM-symm [23] and GRIP (Generic Representatives In PRISM) [12, 15]. PRISM-symm uses an efficient algorithm for the construction of quotient models from an original, non-reduced model. Property checking on the reduced model can then be performed more efficiently in comparison to property checking on the non-reduced model. However, this approach depends on it being feasible to construct the non-reduced model in the first place. In contrast, the GRIP tool is based on an extension of the generic representatives approach to the probabilistic setting. While PRISM-symm can work well for model specifications that consist of a small number of complex modules, GRIP excels in the context of a large number of relatively simple modules.

The extension of generic representatives to a probabilistic setting on which GRIP is based leverages a specialised language, *Symmetric Probabilistic Specification Language* (SPSL) [13]. This allows for the specification of a probabilistic system comprising multiple communicating modules in such a way that the applicability of the generic representatives technique is guaranteed. An algorithm for direct translation of SPSL specifications of symmetric multi-module probabilistic systems into generic form is also provided [13].

A major limitation of SPSL and the associated GRIP tool is that they only support symmetric systems in which modules communicate with one another through the use of shared variables. An alternative, widely-used communication mechanism involves *synchronisation*, where when multiple modules are ready to take a particular named action they all execute a statement related to this action simultaneously, in a synchronous fashion. This limitation means that there is a large class of systems to which SPSL and GRIP cannot be applied. This problem is more than just a tooling limitation: as we explain in this paper, the problem of how to encode inter-module synchronisation using generic representatives is difficult and—to our knowledge—has not been studied before.

In this paper we show how SPSL and the associated translation algorithm [13] can be extended to include synchronisation. To allow experimenting with the feasibility of the translation in practice, we have extended the GRIP tool to use our new translation method, and present experimental results comparing our updated version of GRIP with PRISM-symm (which already supports synchronisation) on three case studies that rely on inter-module synchronisation.

The experimental results show that our approach enables the symmetry reduction of specifications dependant on synchronisation but comes with an additional overhead for each synchronisation instance. We conclude that the technique works well for systems composed of a large number of simple symmetric modules that feature a small amount of synchronisation-based communication.

2 Background

2.1 Symmetry reduction via generic representatives

The generic representatives approach [16, 17], also known as *counter abstraction*, allows symmetry reduction to be applied without the construction of an orbit relation. Specifically, this process involves replacing a specification in which multiple symmetric modules[3] are each individually represented by a single *generic module*. The variables of the generic module represent the number of symmetric modules at a given local state.

We illustrate the generic representatives approach using an example. Consider a mutual exclusion algorithm for six identical modules, each with three local states: neutral (N), trying (T) and critical (C). The global states (N, N, N, T, T, C), (N, T, T, N, N, C) and (C, N, N, T, N, T) are symmetrically equivalent and have generic representative $(3N, 2T, 1C)$. A generic representative indicates how many modules are in each local state, without referring to individual modules. In our example, the generic representative $(3N, 2T, 1C)$ merely records that there are three modules in the N state, two modules in the T state and one module in the C state, without keeping track of which particular module is in each state.

[3] Throughout the paper we use the term “module” to mean what is often called a “process” in the model checking literature. This is because the implementation of our ideas is in the context of the PRISM model checker, which uses the term “module” for this concept.

The idea of the generic representatives approach is to rewrite a specification initially expressed as multiple individual symmetric modules into one that is based on counter variables. The resulting specification represents the original symmetric modules via a *single* module that uses the counter variables to keep track of the number of original symmetric modules that are in each state. This has the effect of exploiting symmetry at the source code level. As a result, there is no need to do symmetry reduction when actually model checking, and so a symbolic approach can be directly applied. This avoids the need to construct a BDD for the orbit relation (which, as discussed above, is prohibitively expensive). It also avoids the need to build an unreduced model and then apply symmetry reduction to it (which is the approach taken by PRISM-symm), allowing the verification of systems for which constructing an unreduced model in the first place is intractable.

2.2 GRIP: generic representatives in PRISM

The generic representatives approach has been extended to probabilistic model checking [13, 15]. A language, *Symmetric Probabilistic Specification Language* (SPSL), was introduced which allowed for the specification of probabilistic systems in such a way that the applicability of the technique is guaranteed. SPSL specifications can be directly translated into generic form using a defined algorithm.

This translation has been implemented for the probabilistic model checker PRISM and its modelling language [24] via the as the GRIP (Generic Representatives in PRISM) tool. GRIP supports specifications that are defined in *Symmetric PRISM* (henceforth SP). This is a subset of the PRISM modelling language that is analogous to SPSL. Specifications are reduced using a translation corresponding to the SPSL translation rules. Fig. 1 shows the structure of the workflow of GRIP (before we applied our modifications). When the source code for GRIP is compiled, an abstract syntax tree representation of the model is created. This is then translated to the reduced specification.

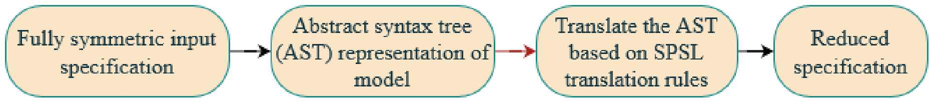
Fully symmetric input specification
Abstract syntax tree (AST) representation of model
Translate the AST based on SPSL translation rules
Reduced specification

Fig. 1: GRIP workflow.

Symmetric PRISM (SP) Input specifications for GRIP must be written in SP for the translation process to be applicable. An example input specification for GRIP can be seen in Listing 1.1. Note that, as is required by PRISM, the first line denotes the model type, which in this case is `mdp`—i.e. a Markov Decision Process (MDP). Other model types include `dtmc` for a Discrete Time Markov

Chain (DTMC) and `ctmc` for a Continuous Time Markov Chain (CTMC). All model types are described in full in [24].

Consider the following simple model. Two devices call heads or tails for the flip of a coin. They make their decisions at random and with equal probability, at which point they terminate (we do not model any consequence of the coin toss). All devices start at a state 0 (*initial state*), and can move to state 1 (*chosen to call heads*) or state 2 (*chosen to call tails*). Once the devices have both reached state 1 or state 2 they can move to state 3 (*end state*). An SP specification for this system is shown in Listing 1.1. Note that si=j is a guard that will be true if and only if device i is currently in state j. The model is defined by specifying one concrete module, `device1`, and then an additional module, `device2`, by *renaming* `device1`.

```
dtmc

module device1
  s1 : [0..3] init 0;

  [] s1=0 -> 0.5 : (s1'=1) + 0.5 : (s1'=2);
  [] s1=1 -> (s1'=3);
  [] s1=2 -> (s1'=3);
  [] s1=3 -> (s1'=3);

endmodule
module device2 = device1[s1=s2,s2=s1] endmodule
```

Listing 1.1: Example: coin toss model.

The core syntax of SPSL is shown in Table 1. The table is based on the original syntax present in [13] with updates (highlighted in red) made to support the translation of synchronised statements. The updates allow synchronisation labels to be optionally attached to statements. Statements with the same label are executed simultaneously, i.e. all symmetric modules must be in a state that satisfies the guard of at least one such statement, and the updates of those statements get executed simultaneously. The basic structure of an SPSL specification otherwise remains unchanged. A detailed explanation of SP and example specifications can be found in [27].

Translation process We describe how a model specified in SP is translated into generic form. First we consider how the local variables for each individual module are replaced by counter variables. We then consider the translation of transitions between states of the original specification to transitions between states defined as values of the counter variables.

An abstract syntax tree is constructed from the original specification based on the SP grammar. As shown in Fig. 2, a walk is then performed on this tree and each element is translated into the reduced specification. The model type (i.e. *DTMC*, *CTMC*, *MDP*, etc.) of the two specifications is the same. Similarly, the global variables and the non-symmetric module declarations are directly copied from the original model.

Translating the symmetric modules into a single generic module is more complicated. The local variables are substituted by counter variables, one for each

$$
\begin{array}{rcl}
\mathsf{specification} & ::= & \mathsf{global\text{-}variables}^{?}\ \mathsf{module}^{+} \\
\mathsf{global\text{-}variables} & ::= & \texttt{globals}\ \{\ \mathsf{var\text{-}decl}^{+}\ \} \\
\mathsf{module} & ::= & \texttt{module}\ M[\mathsf{number}]\{\ \mathsf{var\text{-}decl}^{*}\ \mathsf{statement}(M)^{+}\ \} \\
\mathsf{var\text{-}decl} & ::= & \mathsf{name} : \mathsf{type}\ \texttt{init}\ \mathsf{constant} \\
\mathsf{type} & ::= & [\mathsf{number..number}]\ \mid\ \texttt{bool} \\
\mathsf{constant} & ::= & \texttt{true}\ \mid\ \texttt{false}\ \mid\ \mathsf{number} \\
\mathsf{statement}(M) & ::= & []\ \mathsf{expr}(M_i) \rightarrow \mathsf{stoch\text{-}update}(M) \\
 & \mid & \textcolor{red}{[\mathsf{name}]\ \mathsf{expr}(M_i) \rightarrow \mathsf{update}(M)} \\
\mathsf{stoch\text{-}update}(M) & ::= & \mathsf{expr}(M_i)\text{:}\mathsf{update}(M) + \ldots + \mathsf{expr}(M_i)\text{:}\mathsf{update}(M) \\
\mathsf{update}(M) & ::= & \texttt{skip}\ \mid\ (\mathsf{name} := \mathsf{expr}(M_i)) \parallel \ldots \parallel (\mathsf{name} := \mathsf{expr}(M_i)) \\
\mathsf{symm\text{-}expr} & ::= & \mathsf{constant}\ \mid\ \mathsf{global\text{-}name} \\
 & \mid & \bigcirc_{1\le j\le \#N} \mathsf{loc\text{-}expr}(N)_j\ \text{(for some module type } N\text{)} \\
 & \mid & \mathsf{symm\text{-}expr} \bowtie \mathsf{symm\text{-}expr} \mid \neg\mathsf{symm\text{-}expr}\ \mid\ (\mathsf{symm\text{-}expr}) \\
\mathsf{loc\text{-}expr}(M) & ::= & \mathsf{constant}\ \mid\ \mathsf{local\text{-}name} \mid \mathsf{loc\text{-}expr}(M) \bowtie \mathsf{loc\text{-}expr}(M) \\
 & \mid & \neg\mathsf{loc\text{-}expr}(M) \mid (\mathsf{loc\text{-}expr}(M)) \\
\mathsf{expr}(M_i) & ::= & \mathsf{loc\text{-}expr}(M)_i\ \mid\ \mathsf{symm\text{-}expr} \mid \bigcirc_{1\le j\ne i\le \#M} \mathsf{loc\text{-}expr}(M)_j \\
 & \mid & \mathsf{expr}(M_i) \bowtie \mathsf{expr}(M_i) \mid \neg\mathsf{expr}(M_i)\ \mid\ (\mathsf{expr}(M_i))
\end{array}
$$

Table 1: Syntax of Symmetric Probabilistic Specification Language (SPSL). PCTL-specific syntax is omitted. Updates shown in red.

state a symmetric module can be in. Each counter variable keeps track of how many symmetric modules are in the state associated with it. Each transition statement of the original symmetric module is translated into one or more reduced statements. These update the counter variables according to the original statement. Listing 1.2 shows the output produced by GRIP based on the model specification from Listing 1.1.

```
probabilistic

module generic_process
  no_0 : [0..2] init 2;    // No modules in state (0)
  no_1 : [0..2] init 0;    // No modules in state (1)
  no_2 : [0..2] init 0;    // No modules in state (2)
  no_3 : [0..2] init 0;    // No modules in state (3)

  [] (no_0>0) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
               + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
  [] (no_0>1) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
               + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
  [] (no_1>0) -> (no_1'=no_1-1)&(no_3'=min(no_3+1,2));
  [] (no_1>1) -> (no_1'=no_1-1)&(no_3'=min(no_3+1,2));
  [] (no_2>0) -> (no_2'=no_2-1)&(no_3'=min(no_3+1,2));
  [] (no_2>1) -> (no_2'=no_2-1)&(no_3'=min(no_3+1,2));
  [] (no_3>0) -> true;
  [] (no_3>1) -> true;
endmodule
```

Listing 1.2: Example output specification for a model of a coin tossing scenario. Output is generated by GRIP from the input specification shown in Listing 1.1.

Lines 4 to 7 declare the counter variables replacing the local variables of the eight symmetric modules. Lines 9 to 18 are the translated transition statements. Note that each transition's guard checks the number of symmetric modules in a state associated with a particular counter variable. The update denotes the transfer of one module from one state to another by incrementing/decrementing the associated counter variables.

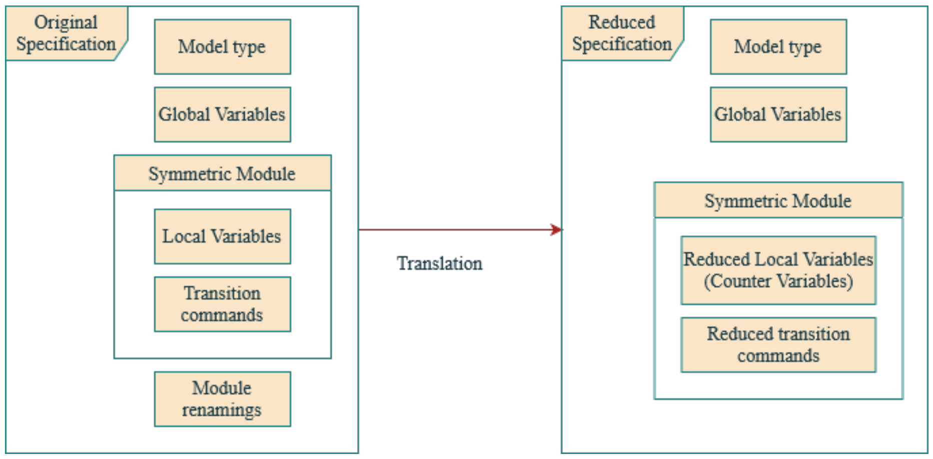

Fig. 2: Visualisation of the translation steps of the SPSL symmetry reduction algorithm. Entities with the same name in the two specifications are direct copies. We include a family of symmetric modules but no asymmetric modules (for simplicity).

For MDPs there is a one-to-one relationship between statements and translated statements. In the case of DTMCs however (as for this example) each statement is translated into a number of reduced statements (one for each symmetric module present in the original specification). This is necessary to correctly model the fact that a counter variable with a higher value is more likely to change in the next transition.

3 Synchronisation and Generic Representatives

Our goal is to add synchronisation to both the grammar and translation rules of SPSL. To do this we first introduce some notation (Section 3.1). We then give some initial intuition towards the development of our new translation rules for SPSL via an example in SP (Section 3.2). The new translation rules for SPSL are defined in Section 3.3.

3.1 Notation

For ease of presentation We assume that our model is for a single family of symmetric modules $M = \{M_1, M_2, \ldots, M_m\}$ that have access to a set of shared global variables.

If the set of states of M is $S(M)$, the set of local states of each M_i is $S(M_i)$, and the set of realisations of the global variables is G, then $S(M) = (\prod_{1 \le i \le m} S(M_i)) \times G$. Every state is a tuple of the form

$$(s_1, s_2 \ldots s_m, g)$$

We may assume that, for each i, $S(M_i) = \{s_{i,1}, s_{i,2}, \ldots, s_{i,r}\}$, and for any i and j, the states $s_{i,k}$ and $s_{j,k}$ are identical after module renaming for any $1 \le k \le r$.

Counter variables `count`$\_M\_k$ $: k \in \{1, 2, \ldots, r\}$ record at any state the number of modules in a given local state. Specifically they record the number of modules, M_i whose local state is $s_{i,k}$. Counter function f maps each element of $S(M)$ to the appropriate r-tuple of counter variable values.

For each transition statement c in M_i, let $\mathsf{SAT}_{M_i}(c) = \{l \in S(M_i) : l \models c\}$, i.e. the subset of local states of M_i that satisfy its guard. Similarly, for an expression e appearing in a guard, we use $\mathsf{SAT}_{M_i}(e)$ to refer to the subset of local states that satisfy e.

3.2 Basic synchronisation example

Consider the basic coin toss example from Listing 1.1; however, this time with synchronisation labels present. The two devices again make their decisions at random and with equal probability, but now must simultaneously reveal their choices, at which point they terminate (we again do not model any consequence of the coin toss).

```
 1 dtmc
 2
 3 module device1
 4   s1 : [0..3] init 0;
 5
 6   []  s1=0 -> 0.5 : (s1'=1) + 0.5 : (s1'=2);
 7   [a] s1=1 -> (s1'=3);
 8   [a] s1=2 -> (s1'=3);
 9   []  s1=3 -> (s1'=3);
10
11 endmodule
12 module device2 = device1[s1=s2,s2=s1] endmodule
```

Listing 1.3: Example: coin toss with synchronisation

Without synchronisation labels, each device could progress to state 3 (c.f. lines 7 and 8 of Listing 1.3) as soon as it enters state 1 or 2. With the labels, they would need to wait until the other device were ready. Furthermore, synchronisation requires all updates to be executed simultaneously.

Lines 13 to 16 of Listing 1.2 have been created by GRIP for the reduced specification of the coin toss model without synchronisation. The updates are simple: in each of them the value of the `no_3` counter is increased by one, i.e. a module moves to state 3. In two of them that particular module has arrived at that state from state 1, in the other two it has arrived from state 2. The guard of line 13 is equivalent to “there are 1 or 2 modules in state 1” and of line 14 to “there are 2 modules in state 1”. Similarly the guards of lines 15 and 16 are equivalent to “there are 1 or 2 modules in state 2”, and “there are 2 modules in state 2” respectively. Out of these, lines 14 and 16 would be acceptable even with synchronisation. However, the guards of the other two would need to be tightened. Merely requiring that one module is in state 1 (or in state 2) is insufficient – in this case the guard should also state that the other module is in the corresponding (other) state. That is: when exactly one module is in state 1, the other module would need to be in state 2 and vice versa. Listing 1.4 shows what a reduced version of the specification could look like if the guards were strengthened to accommodate this observation.

```
1  probabilistic
2  global total : [ 0 .. 2 ] init 0 ;
3
4  module generic_process
5    no_0 : [0..2] init 2;     // No modules in state (0)
6    no_1 : [0..2] init 0;     // No modules in state (1)
7    no_2 : [0..2] init 0;     // No modules in state (2)
8    no_3 : [0..2] init 0;     // No modules in state (3)
9
10   []  (no_0>0) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
11                 + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
12   []  (no_0>1) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
13                 + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
14   [a] (no_1=0) & (no_2=2) -> (no_2'=0)&(no_3'=min(no_3+2,2));
15   [a] (no_1=1) & (no_2=1) -> (no_1'=0)&(no_2'=0)&(no_3'=min(no_3+2,2));
16   [a] (no_1=2) & (no_2=0) -> (no_1'=0)&(no_3'=min(no_3+2,2));
17   []  (no_3>0) -> true;
18   []  (no_3>1) -> true;
19 endmodule
```

Listing 1.4: Example: reduced coin toss specification with synchronisation

Note that in this example, the updates for all synchronised statements in the output specification are the same, suggesting that the three statements could be combined. However this is not true in general, as synchronised statements may have different updates. The reduced updates will therefore consist of a number of distinct assignments each requiring a distinct statement. For each synchronisation label, guard and update combination, we must consider the possible ways of allocating the symmetric modules between the states accepted by the guards. The number of reduced statements arising from a synchronised block of statements is exponential in both the number of symmetric modules and in the number of local states that satisfy the guards of the local statements. This is an important observation (hence we state it again below). It means that adding synchronisation to our translation algorithm comes at a significant cost - although its inclusion is necessary when synchronous protocols are to be modelled.

Observation Although the method we present allows the generic representatives approach to symmetry reduction to be applied in the presence of synchronised statements, its scalability is limited. This is because our method results in an exponential blow-up in the size of the text of the reduced specification: the blow-up is exponential in both the number of symmetric modules, and the number of statements that use a particular synchronisation label. Although the state space associated with the reduced specification will be smaller than the original state space (thanks to symmetry reduction), the overhead associated with processing the larger specification text in order to build this state space may outweigh the benefits brought by symmetry reduction.

3.3 Translating synchronisation

We now develop new SPSL translation rules for synchronised statements to be added to the original rules introduced in [13]. For notation see Section 3.1. As the translation process is more complex for DTMCs, we assume that our model type is either an MDP or a CTMC for ease of presentation. However, our implementation in the GRIP tool does support DTMCs.

Counter abstraction replaces each family M of m symmetric modules by a single generic module with a set of r counter variables, each of which ranges from 0 to m. All counter variables are initialised to 0, except for that corresponding to the initial state, which is initialised to m.

Without loss of generality we can assume that a synchronised statement has the form:

$$[label] \quad \mathsf{local\text{-}expr}(M) \wedge \mathsf{symm\text{-}expr}(M) \rightarrow \mathsf{stoch\text{-}update}(M) \tag{1}$$

where $\mathsf{local\text{-}expr}(M)$ is an expression over local variables only and $\mathsf{symm\text{-}expr}(M)$ may also include some global variables. Either can be set to *true* if no such expression is present. Expression $\mathsf{symm\text{-}expr}(M)$ must be fully symmetric for translation to be applicable.

Consider a statement in the original specification with $\mathsf{local\text{-}expr}(M) = e$, and $\mathsf{symm\text{-}expr}(M) = s$. Without synchronisation, GRIP splits the translation process into cases, one per $l \in \mathsf{SAT}_M(e)$ (see Fig. 3). For each case, a separate reduced generic statement is generated by the following process. Expression e is replaced with a condition $\texttt{count\_}M\texttt{\_}f_M(l) > 0$. (Although we are assuming that our model is an MDP, it is worth noting that, in the case of DTMCs, each statement gets translated into m statements which are identical except that the ith statement has guard $\texttt{count\_}M\texttt{\_}f_M(l) > i$, for $0 \leq i \leq m-1$.) This condition asserts that some member of M has a local state required to satisfy the local part of the guard of this statement. Both s and the stochastic update $\mathsf{stoch\text{-}update}(M)$ are translated in the context of the state l, so their reduced counterparts do not make use of any variables local to M. Updates affecting variables local to M are replaced with updates to two counter variables: if l is the local state considered in the current case, and l' is the local state reached by performing all local variable updates on l, then the resulting reduced update must decrement $\texttt{count\_}M\texttt{\_}f_M(l)$ and increment $\texttt{count\_}M\texttt{\_}f_M(l')$ (representing modules leaving l and arriving at l' respectively).

We approach synchronised statements using a similar methodology (see Fig. 4). First, we assume that all updates of synchronised statements will

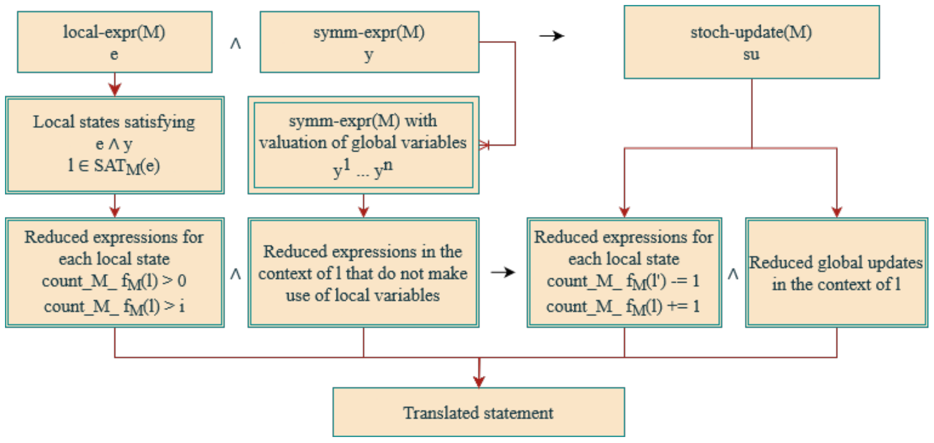

Fig. 3: Translation of a single non-synchronised statement. The top row represents the components a statement in the original specification, while the bottom row represents the corresponding translated components that form a reduced statement.

only change the local state of a module. Note that PRISM does not allow synchronised statements to perform updates to global variables, so this is a reasonable assumption. For brevity, we only consider synchronised statements with a single update (i.e. of the form $p_1 : u_1$ where $p_1 = 1$) rather than a stochastic choice of updates. A discussion of how multiple updates could be included is given in [27].

To translate statements with synchronisation we must consider all statements with the same label at the same time, rather than translating them individually. Again we assume that M consists of a single family of symmetric modules M_i, $1 \leq i \leq m$. For any statement in M_1 (say), the same statement (under module renaming) is present in all modules. For this reason we base our reasoning on statements in M_1.

All statements in M_1 with label α have the form $[\alpha] \quad e \wedge y \rightarrow p_1 : u_1$ (c.f. Eq. (1)). If there are z such statements, with local parts $e_1, e_2, \ldots, e_z$, then define

$$\mathsf{SAT}_{M_1}(\alpha) = \bigcup_{1 \leq j \leq z} \mathsf{SAT}_{M_1}(e_j)$$

Synchronisation over α can only take place at state $s = (s_1, s_2, \ldots, s_m, g)$ if, for each module, at least one statement with label α is enabled at s, i.e. if there exists an $s_{i,k} \in \mathsf{SAT}_{M_i}(\alpha)$ for $1 \leq i \leq m$. For each j and local state $l \in \mathsf{SAT}_{M_1}(e_j)$ we define w_j^l to be the number of modules in their corresponding local state before the synchronised transition is executed, where $0 \leq w_j^l \leq m$. Similarly for any state $l \in \mathsf{SAT}_{M_1}(\alpha)$ define $x^l = \sum_{j : l \in \mathsf{SAT}_{M_1}(e_j)} w_j^l$. Each translated statement then has the condition $\bigwedge_{l \in \mathsf{SAT}_{M_1}(\alpha)} \texttt{count\_}M\_f_M(l) = x^l$.

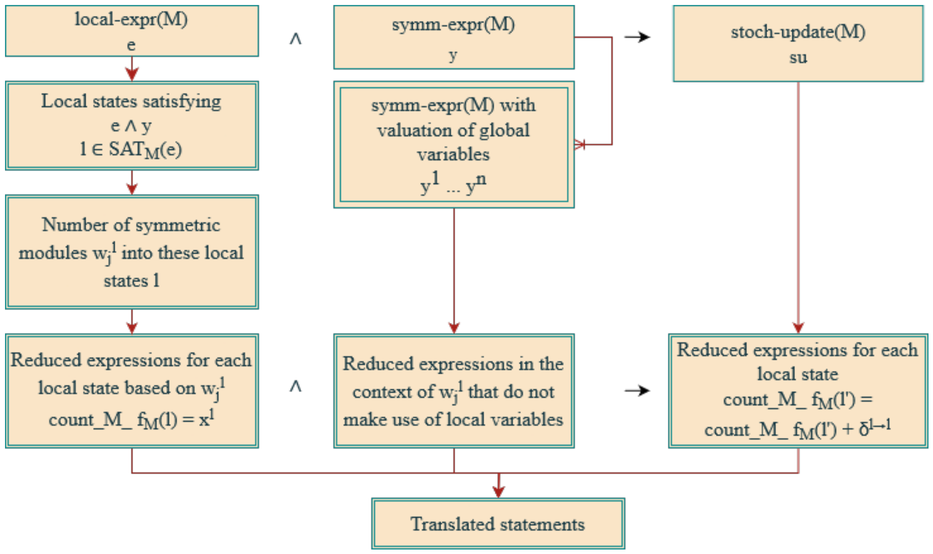
local-expr(M)
e
∧
symm-expr(M)
y
→
stoch-update(M)
su
Local states satisfying
e ∧ y
l ∈ SAT_M(e)
symm-expr(M) with valuation of global variables
y^1 ... y^n
Number of symmetric modules w_j^l into these local states l
Reduced expressions for each local state based on w_j^l
count_M_ f_M(l) = x^l
∧
Reduced expressions in the context of w_j^l that do not make use of local variables
→
Reduced expressions for each local state
count_M_ f_M(l') =
count_M_ f_M(l') + δ^{l→l'}
Translated statements

Fig. 4: Translation of a block of synchronised statements. Top and bottom rows represent components in a statement in a synchronised block in the original specification in the reduced model respectively.

The global part of the guards, symm-expr(M), is then translated by considering only those parts which belong to a statement that is being executed, and evaluated in the context of the corresponding local state l. For each local state l with $w_j^l > 0$, we translate the global guard corresponding to e_j. We then combine the translated global guards. The individual translation process here is the same as for the global part of a non-synchronised statement. The translation of updates follows a similar idea: for each $w_j^l > 0$, we find the states resulting from applying the update associated with the statement whose guard is e_j. For any such updated state l' we are interested in two quantities: the number of modules that were in state l' before the updates were applied, $w_j^{l'}$, and the number of modules in states which change to state l' after the updates are applied. The translated updates have the following form $\bigwedge_l \texttt{count\_}M\texttt{\_}f_M(l)\texttt{:=count\_}M\texttt{\_}f_M(l) + \delta^{l \to l'}$, where $\delta^{l \to l'}$ is the difference between the modules changing into state l' and the number of modules changing out of it.

4 Experimental Results

We have implemented our new translation techniques in an updated version of GRIP: GRIP 3.0. The input PRISM specification (in SP) may now contain any number of synchronised statements across multiple modules. There may be multiple blocks of synchronised statements (each with a different label).

We now investigate how well GRIP 3.0 performs compared to PRISM and PRISM-symm. Our first example (Rock-Paper-Scissors) is a purpose-built small example to show a complete SP specification with synchronisation. The other two examples (CSMA/CD, and a synchronous version of a randomised Byzantine agreement protocol) are based on examples from [23] and demonstrate where GRIP does badly compared to PRISM-symm, and where it does well. The experiments were performed on a 2.60 GHz PC with 16 GB RAM, running PRISM version 4.8 under Windows. The maximum memory of the CUDD library was set to 1 GB (PRISM default) and the Java maximum memory was set to 6 GB.

We note that all of the PRISM case studies [1] suitable for symmetry reduction using GRIP have already been considered in the previous version of the tool (for more information see [27]). Many of the other PRISM case studies are not suitable for the generic representatives approach as they possess *ring symmetry*, rather than the full symmetry we require [11].

Rock-Paper-Scissors We first consider a model of a Rock-Paper-Scissors game. Modules represent participants, who choose between three options: rock, paper and scissors. When all choices have been made, they are evaluated. If all choices are different or all are the same, the game continues for another round. Otherwise an outcome is announced, based on the choices made. Synchronised statements are required to ensure that all choices are made before the result is evaluated. The PRISM model is shown in Listing 1.5.

```
dtmc
global r : [0..1] init 0;
global p : [0..1] init 0;
global s : [0..1] init 0;

module player1
// choice: 0-undecided, 1-rock, 2-paper, 3-scissors
ch1 : [0..3];
// local phase
ph1 : [1..2];
// winner: 1-rock, 2-paper, 3-scissors
res1 : [0..3];
// make choice
[]((ch1=0)&(ph1=1)&(res1=0)) -> 1/3: (ch1'=1) & (r'=1)
                              + 1/3: (ch1'=2) & (p'=1)
                              + 1/3: (ch1'=3) & (s'=1);
// determine outcome
[decided] ((ph1=1) & (res1=0)) -> (ph1'=2) ;
[]((ph1=2)&(res1=0))&((r=1)&(p=0)&(s=1))->(ch1'=0)&(res1'=1);
[]((ph1=2)&(res1=0))&((r=1)&(p=1)&(s=0))->(ch1'=0)&(res1'=2);
[]((ph1=2)&(res1=0))&((r=0)&(p=1)&(s=1))->(ch1'=0)&(res1'=3);
[]((ph1=2)&(res1=0))&((r=0)&(p=0)&(s=0))->(ch1'=0);
[]((ph1=2)&(res1=0))&((r=1)&(p=0)&(s=0))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
[]((ph1=2)&(res1=0))&((r=0)&(p=1)&(s=0))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
[]((ph1=2)&(res1=0))&((r=0)&(p=0)&(s=1))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
[]((ph1=2)&(res1=0))&((r=1)&(p=1)&(s=1))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
// reset for next round if needed
[reset] ((ch1=0)&(ph1=2)&(res1=0)) -> (ph1'=1) ;
endmodule
module player2=player1[ch1=ch2,ch2=ch1,ph1=ph2,ph2=ph1,res1=res2,res2=res1]
    endmodule
```

Listing 1.5: Rock-Paper-Scissors model. Multiple module renamings are not shown.

Table 2 shows the model sizes and execution times for the Rock-Paper-Scissors model described above for m participants, for $2 \leq m \leq 10$, using PRISM, PRISM-symm and GRIP respectively. The property verified is: "what is the probability that the winning outcome is *rock?*'.

Compared to PRISM, the GRIP specification is more complex in all cases but the resulting model has far fewer states (when $m > 2$). Consequently the build times increase for reduced models and the times taken for model checking decrease. On this example GRIP is significantly out-performed by PRISM-symm. We suspect that this is because, given the size of the example, the synchronisation dominates. (Recall from the observation in Section 3.2 that synchronisation results in an exponential increase in translated statements). We expect our approach to be most beneficial for models that involve a majority of non-synchronised statements. However, we do achieve a significant improvement in comparison to standalone PRISM. The example also serves to demonstrate the correctness of GRIP's new support for synchronisation.

RPS	Model size (MTBDD)			Model build time (sec.)			Model check time (sec.)		
	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP
m		-symm			-symm			-symm	
2	453	280	605	0.03	0.066	0.01	0.01	0.03	0.03
3	1774	749	1029	0.04	0.109	0.15	0.01	0.05	0.03
4	4311	1513	2156	0.05	0.112	0.35	0.02	0.04	0.04
5	8021	2394	2880	0.07	0.185	0.88	0.05	0.04	0.05
6	12902	3335	3672	0.08	0.207	2.98	0.15	0.06	0.13
7	18951	4360	4593	0.09	0.339	6.66	0.59	0.07	0.13
8	26153	5442	7240	0.15	0.428	20.73	5.16	0.08	0.08
9	34526	6593	8611	0.16	0.545	30.68	148.21	0.09	0.33
10	44067	7816	10198	0.18	0.712	54.09	261.71	0.12	0.55

Table 2: Model size and build times for the Rock-Paper-Scissors model for m participants, obtained by PRISM, PRISM-symm and GRIP 3.0.

Carrier Sense, Multiple Access with Collision Detection Protocol (CS-MA/CD) PRISM-symm has been applied to a variety of case studies [23]. However, until now, it has not been possible to compare the performance of PRISM-symm to GRIP for one of those case studies due to the lack of support for synchronisation in GRIP. That example is the IEEE 802.3-2002 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) communication protocol (*csma*) [22]. We now investigate applying GRIP 3.0 for that example.

The PRISM specification for *csma* is of the type that GRIP is not well suited for. GRIP excels at a larger number of symmetric copies of a simpler module, while PRISM-symm is best for a smaller number of more complex modules [13]. Hence we would not expect GRIP to compare favourably to PRISM-symm in this case.

Examining the *csma* specification closely, we note that the symmetric module has $|S(M)| = 118$ local states, an order of magnitude larger than the typical GRIP case studies [13]. Additionally, most of its synchronised statements have loose guards; for example, a single synchronised statement can have its guard satisfied by modules in 60 out of the 118 possible states. As the number of reduced statements increases with the number of local states satisfying the guards of synchronised statements (again, see the observation in Section 3.2), the number of reduced statements is very large.

Our attempt to apply GRIP 3.0 to this example resulted in approximately eight million reduced statements being generated for a single synchronisation label of the *csma* specification. A complete model would take over an hour to build and is unlikely to offer any improvement over PRISM. We conclude that while GRIP 3.0 could now be applied to the *csma* case study, it would be ill-advised.

Randomised Byzantine Agreement protocol Our final example is an adaptation of a Byzantine agreement protocol [6]. The original protocol was an example for which GRIP performed favourably compared to PRISM-symm [13]. We have added a common synchronisation label to three of the statements that have updates to local states only and compare performance again. Results for a range of numbers of participants m are shown in Table 3. Despite the additional synchronisation, GRIP still performs well for this example compared with PRISM-symm.

Byz	Model size (MTBDD)			Model build time (sec.)			Model check time (sec.)		
m	PRISM	PRISM -symm	GRIP	PRISM	PRISM -symm	GRIP	PRISM	PRISM -symm	GRIP
6	130,145	54,512	21,018	1.36	0.207	1.29	2.07	0.30	0.26
8	592,630	214,293	54,887	6.20	0.428	1.92	>10m	1.39	0.66
12	OOM	OOM	218,153	OOM	OOM	2.98	OOM	OOM	4.653
16	OOM	OOM	343,941	OOM	OOM	6.06	OOM	OOM	13.84

Table 3: Model size and build times for the Byzantine model obtained by PRISM, PRISM-symm and GRIP 3.0. OOM signifies models which resulted in an Out-of-Memory error.

5 Conclusion

We have defined new translation rules for synchronised statements in SPSL, and shown that they are sound. We have discussed the limitations of the method and shown that, in worst case, the counter-abstraction approach does not achieve a reduction in the size of the state space. We have updated GRIP (to version 3.0) to include support for specifications including synchronised statements. We have shown that our approach is feasible, and works well in some cases. Our approach is most beneficial for models that involve a majority of non-synchronised statements and a few synchronised ones.

We plan to add further features to GRIP. While the tool currently does not support translation of steady-state properties, we have conducted an initial investigation using manual translation of this type of property (not included in this paper). Our investigation has shown us that adding this feature in the future would be feasible. Similarly, we aim to introduce support for specification and analysis of properties based on costs and rewards. This would involve translating both the reward structures present in a model and the properties themselves. Specifically, (1) the translated reward structures should not reference any individual module, and (2) GRIP and SPSL would need to be extended to support the translation of reward-based properties.

Acknowledgments. Ivaylo Valkov was supported by the EPSRC Doctoral Training Partnership award EP/N007565/1 and by a grant from the UKRI Strategic Priorities Fund to the UKRI Research Node on Trustworthy Autonomous Systems Governance and Regulation [EP/V026607/1, 2020-2024]. Alastair Donaldson was supported by the EPSRC IRIS project (grant EP/R006865/1).

References

1. Prism - case studies. https://www.prismmodelchecker.org/casestudies/index.php, accessed: 2024-03-15
2. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric spin. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1885, pp. 1–19. Springer (2000). https://doi.org/10.1007/10722468_1
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
4. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992). https://doi.org/10.1145/136035.136043
5. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10ˆ20 states and beyond. Inf. Comput. **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
6. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In: Neiger, G. (ed.) Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA. pp. 123–132. ACM (2000). https://doi.org/10.1145/343477.343531
7. Cimatti, A., M. Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: a new symbolic model checker. STTT **2**, 410–425 (03 2000). https://doi.org/10.1007/s100090050046

8. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (apr 1986). https://doi.org/10.1145/5397.5399
9. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking, second edition. Cyber Physical Systems Series, MIT Press (2018), https://books.google.co.uk/books?id=qJl8DwAAQBAJ
10. Clarke, E.M., McMillan, K., Zhaor, X., Fujita, M., Yang, J.: Spectral transforms for large boolean functions with applications to technology mapping. In: Proceedings of the 30th ACM/IEEE Design Automation Conference. pp. 54–60. IEEE Computer Society Press (1993)
11. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design **9**, 77–104 (1996), https://api.semanticscholar.org/CorpusID:14472493
12. Donaldson, A.F., Miller, A.: Symmetry reduction for probabilistic model checking using generic representatives. In: Graf, S., Zhang, W. (eds.) Automated Technology for Verification and Analysis, 4th International Symposium ATVA 2006. Lecture Notes in Computer Science, vol. 4218, pp. 9–23. Springer (2006)
13. Donaldson, A.F., Miller, A., Parker, D.: Language-level symmetry reduction for probabilistic model checking. In: QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems. pp. 289 – 298. IEEE Computer Society (2009). https://doi.org/10.1109/QEST.2009.21
14. Donaldson, A.F., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: FM 2006: Formal Methods, 14th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 531–556. Springer (2006)
15. Donaldson, A.F., Miller, A., Parker, D.: GRIP: Generic representatives in PRISM. In: Proceedings of the 4th International Conference on Quantitative Evaluation of Systems (QEST'07). IEEE Computer Society. pp. 115–116 (2007)
16. Emerson, E.A., Trefler, R.J.: From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In: In Conference on Correct Hardware Design and Verification Methods (CHARME '99). pp. 142–156. Springer (1999)
17. Emerson, E.A., Wahl, T.: On combining symmetry reduction and symbolic representation for efficient model checking. In: Conference on Correct Hardware Design and Verification Methods (CHARME 2003). pp. 216–230. Springer (2003)
18. Hendriks, M., Behrmann, G., Larsen, K.L., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to Uppaal. In: First International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2003). Lecture Notes in Computer Science, vol. 2791, pp. 46–59. Springer (2003)
19. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker Storm. CoRR **abs/2002.07080** (2020), https://arxiv.org/abs/2002.07080
20. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 1st edn. (2011)
21. Holzmann, G.J.: Explicit-state model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 153–171. Springer International Publishing, Cham (2018)
22. IEEE Computer Society: IEEE standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. IEEE Std 802.3-2002 (Revision of IEEE Std 802.3, 2000 edn) pp. 1–1550 (2002). https://doi.org/10.1109/IEEESTD.2002.93570

23. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R. (eds.) Proc. 18th International Conference on Computer Aided Verification (CAV'06). LNCS, vol. 4114, pp. 234–248. Springer (2006)
24. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV 2011. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)
25. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in temporal logic model checking. ACM Computing Surveys **38**(3) (9 2006). https://doi.org/10.1145/1132960.1132962, http://eprints.gla.ac.uk/3197/
26. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming. pp. 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg (1982)
27. Valkov, I.: Formal Analysis of Communication Protocols for Wireless Sensor Systems. Phd thesis, University of Glasgow, Glasgow, UK (June 2024), to appear
28. Wahl, T., Donaldson, A.F.: Replication and abstraction: Symmetry in automated formal verification. Symmetry **2**(2), 799–847 (2010). https://doi.org/10.3390/SYM2020799