A Method for Testing Partial-Order Reduction Theories in Alloy

Mara Miulescu[0009−0004−0637−1563] and Thomas Neele[0000−0001−6117−9129]

Eindhoven University of Technology, Eindhoven, the Netherlands
{m.c.miulescu, t.s.neele}@tue.nl

Abstract. Partial-order reduction (POR) is a technique for tackling the state-space explosion problem in model checking. The search space can be significantly reduced by disabling certain events at the state level. Recent research has brought to light issues in the theory behind several POR approaches. In this work, we demonstrate a method for rigorously testing the soundness of POR theories using Alloy, a SAT-based formal specification tool. We apply this method to several POR theories and formalisms. To explain the modelling challenges that we faced, we show two Alloy models in detail. In our experiments, Alloy manages to produce counterexamples to all theorems that we know to be flawed. This indicates that our ideas are viable for preventing comparable issues in the future development of POR methods.

Keywords: Partial-order reduction · Parity games · Alloy

1 Introduction

In model checking, *partial-order reduction* (POR) theory studies methods for tackling the state-space explosion problem that occurs in the context of verifying (interleaving) concurrent systems. The common intuition behind most POR approaches is that many interleavings can be considered equivalent, and it suffices to explore only one interleaving per equivalence class. This can greatly reduce the amount of effort required for verification. Though a mature field, with notable techniques introduced by Valmari [32], Peled [27] and Godefroid [13], POR theory remains difficult to understand because of its complexity. A proper understanding of any formal theory is essential for judging its correctness.

Recently, a number of correctness issues [22,24,30] (five in total) were uncovered in various POR theories. It thus turns out that the complexity of POR theory indeed impacts it correctness, and traditional pen and paper methods are not sufficient for reliably developing such theory. Therefore, we devise a methodology for modelling and rigorously testing POR theories using an *automated theorem prover* (ATP). Inspired by [30], we use the Alloy analyser [16]. In our setting, a *POR theory* comprises (i) a formalism (some form of transition system), (ii) a set of conditions under which reduction may be applied, and (iii) a theorem that states what properties are preserved in the reduced system

(usually a behavioural equivalence such as *stutter-trace equivalence*). Accordingly, the generic implementation comprises a model of transition systems and, to enable modelling the reduction conditions and behavioural equivalence, an approach for handling quantifications over paths. Such higher-order quantifications are not directly supported by Alloy. We validate our idea by applying it to the aforementioned flawed theories and show that Alloy can reproduce all the issues automatically. Our contributions are [1]:

- A generic Alloy model of transition systems and related concepts. We show how to model paths so that it is possible to write quantifiers over them. This is essential to modelling most POR theories.
- We show in detail how this can be applied to the *stubborn set* methods for *labelled-state transition systems* [33,34] (repaired in [22]) and *parity games* [23] (repaired in [24]). These ideas carry over straightforwardly to other flawed theories we have modelled (but do not discuss in detail here): *ample sets* for labelled Kripke structures [1] (found by [21]) and stubborn sets for reachability games [4] (found by [25], repaired in [5]).
- We run a SAT solver on each of the models and show that counterexamples can be found fully automatically for all of the known flawed theories. Moreover, testing the models of repaired theories reveals no new counterexamples up to the bounds that we set. The models are available online[2].

Considering the variety of theories we implemented as well as the results we obtained from our experiments, we believe our approach is a viable tool for testing the soundness of POR theories.

Related work Correctness issues have been surfacing in the field of formal theories for a long time. An early account of errors in published set theory results dates back to 1979 by De Millo *et al.* [10]. They argue that a proof is only a step in the direction of confidence, stating: "the point is not that mathematicians make mistakes; that goes without saying. The point is that mathematicians' errors are corrected, not by formal symbolic logic, but by other mathematicians".

In the field of POR, a multitude of issues have been found. First, the combination of *on-the-fly* partial-order reduction and nested-depth first search [27,28] proved to be incompatible by default and needed amending [14]. A formalisation in Isabelle/HOL (an interactive theorem prover) of the same on-the-fly approach by Brunner and Lammich [7] revealed that one supporting lemma is not correct, although they were not able to assess correctness of the main theorem. Finally, Siegel [30] was able to produce a counterexample to the theorem using Alloy. Siegel's formalisation is limited to ample sets for Büchi automata. To capture more powerful theories (in particular those involving stubborn sets or games), our Alloy models add support for non-deterministic transition systems and quantification over paths and strategies.

[1] This paper is based on the MSc thesis of the first author [20].
[2] See `https://doi.org/10.5281/zenodo.19134774` for the reproduction artifact.

Interestingly, at an earlier point Chou and Peled [8] formalised the correctness proofs of the *offline* algorithm of [27] in HOL, but did not find any issues. Later, other issues were found [24,22] in methods for labelled transition systems, Kripke structures, parity games, reachability games.

In other areas within formal methods, issues have been found in various places, including in a derivation system for consistent consequence for Boolean equation systems [9], in an antichain algorithm for failures refinement [18], in an algorithm for computing simulation preorder [12] and in a unifying framework for timed automata [17].

Overview First, in Section 2 we introduce the background of our work, including relevant Alloy formalisations. We expand on this in Sections 3 and 4, discussing our Alloy formalisation of POR. Finally, we present our experiments in Section 5 and conclude in Section 6.

2 Preliminaries

This section introduces the background concepts used in the remainder of the paper, in particular the Alloy language and labelled-state transition systems.

Alloy Alloy is an open source language and an automated theorem prover [16]. Given a specification and a property to be verified within a given set of bounds, it creates a *Boolean satisfiability* (SAT) [3] problem. Our approach boils down to transforming the question "is this theory sound?" into a SAT problem: "is there an instance that shows the theory is unsound?". We refer to such an instance as a *counterexample*.

We now give a non-exhaustive overview of the semantics and syntax of the Alloy language, which is a first-order *relational* logic. The Alloy *universe* consists of *atoms*, and we determine the size of the universe by setting *bounds*. Alloy searches for an *instance* that meets the specification for the given bounds. An instance is an assignment of values to *expressions*. We use the notation $[\![\mathtt{e}]\!]$ to refer to the set of values of an expression `e`. Expressions may be of type Boolean (also called *formulas*), relational, or integer. Formulas can be combined with the standard operations $\neg, \vee, \wedge, \leftrightarrow, \rightarrow$ (`not`, `or`, `and`, `iff`, and `implies` in Alloy). Logical implication can be followed by the keyword `else` to capture the case where $[\![\mathtt{e}]\!]$ is `false`. Universal and existential quantifiers work as expected (respectively: `all x:T | e` and `some x:T | e`). Additionally we can express: there exists exactly one `x`, or there is no `x` (respective operators are `one` and `no`). We sometimes use the keyword `disj` inside quantifications. For instance, we write `all disj x, y: X` to quantify over disjoint $[\![\mathtt{x}]\!]$ and $[\![\mathtt{y}]\!]$. Operations that we frequently use on relations are transpose, transitive closure, reflexive transitive closure, set union, set difference, set intersection, join, cartesian product (`~e`, `^e`, `*e`, `e+e`, `e-e`, `e&e`, `e.e`, `e->e` in Alloy for some expression `e`). We use `in` to check set membership (e.g., `x in T`). A model in Alloy consists of declarations of *signatures*, *relations*, *functions*, *predicates*, *facts*, *assertions*, and *commands*.

Signatures define the types in an Alloy formalism. Their declaration can be preceded by keywords, like `abstract` and `one`, which indicate how many atoms of this type will be in the universe (no atoms and exactly one, respectively). Signatures may be extended with the `extends` keyword; for instance, we can write `sig A {}` and `sig B extends A {}`. We declare *fields* inside of a signature to model relations over that signature. To indicate multiplicities for fields we use `lone`, `one`, and `set`. For example, `sig A { r: lone B }` introduces a relation $r \subseteq A \times B$ containing, for each atom $a \in A$ at most one pair (a, b) for $b \in B$. The default behavior of r (if we omit the multiplicity) is to relate each atom a to some atom b. Relations can also be defined outside of signatures, by using `let` followed by a set comprehension or by declaring a (parameterised) function. Predicates are functions that evaluate to `true` or `false`. We may also use `let` to declare local variables inside of functions and predicates. We use facts to enforce that certain predicates, or simply expressions, hold true in the model. Assertions are statements containing the predicates to be checked. The commands for testing in Alloy are `run` and `check`, where the latter is used to find counterexamples for assertions. Both commands require setting bounds (upper or exact) for each signature, which is fundamental to how the Alloy analyser works.

Labelled-State Transition Systems Transition systems are commonly chosen in literature to formally reason about the behaviour of concurrent systems. Recall that a POR theory consists of (i) a formalism (some form of transition system), (ii) a set of conditions under which reduction may be applied, and (iii) a theorem that states what properties are preserved in the reduced system. The POR theories we consider reason about state *and* transition labels, hence we use a formalism containing both as the basis for our methodology. All definitions introduced in this section live in a parameterised Alloy module. When modelling a POR theory, we always start by extending this module. We will often show snippets of Alloy code next to the corresponding definitions.

Definition 1 (Labelled-state transition system). *A labelled-state transition system (LSTS) is a tuple* $(S, \hat{s}, \mathcal{L}, \mathcal{A}, T, L)$*, where:*

- S *is a finite set of states,*
- $\hat{s} \in S$ *is the initial state,*
- $\mathcal{L}$ *is a set of state labels,*
- $\mathcal{A}$ *is a set of actions,*
- $T \subseteq S \times \mathcal{A} \times S$ *is the transition relation,*
- $L : S \to \mathcal{L}$ *is a state labelling.*

```
module lib/lsts [Label, A]

abstract sig AState {
  label: set Label
}

sig Transition {
  src: one AState,
  label: one A,
  dest: one AState
}

let T = {
  s: AState, a: A, s": AState |
  some t: Transition | t.src = s and t.label
  ↪  = a and t.dest = s"
}
```

We introduce a signature for transitions in order to treat these as individual atoms. This is necessary for our encoding of paths. Note that we overload the field `label`. For transitions, the label captures the action executed by that transition,

whereas for states this corresponds to a subset of $\mathcal{L}$. We use a set comprehension to build the transition relation `T` in Alloy. If $(s, a, s') \in T$, then we also write $s \xrightarrow{a} s'$. We overload this notation to capture the *successor* relation $\rightarrow = \{(s, s') \in S \times S \mid \exists\, a \in \mathcal{A}.\ s \xrightarrow{a} s'\}$. We assume in our models that an LSTS is rooted at $\hat{s}$, i.e., all states can be reached from the initial state. This is specified using the reflexive transitive closure $\rightarrow^*$. Whenever $s \xrightarrow{a} s'$, we say that a is *enabled* in state s and denote this $a \in enabled(s)$.

Notice that the module `lib/lsts[Label, A]` defines signatures for S, L, and T, whereas $\mathcal{L}$ and $\mathcal{A}$ are parameters. This choice makes the approach generic enough to accommodate different formalisms. We can view e.g. parity and reachability games as LSTSs that differ in how their states and transitions are labelled.

Higher-order relations By design, Alloy does not support universal quantification over higher-order relations. This poses a challenge when modelling POR, because higher-order evaluation is required to encode most POR theories. As we will see in Sections 3 and 4), we often need to impose conditions on all *paths*. In an LSTS, a path is a finite or infinite sequence of transitions $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$. Alloy provides a utility for modelling sequences of atoms as a relation. Writing `seq Transition` produces all relations over `seq/Int->Transition`, where the values of `seq/Int` range from 0 to a bound that we set. Nevertheless, sequences are second-order relations, meaning that Alloy does not support quantifications such as `all path: seq Transition`. We tried to resolve this by using Alloy* [19], which is an extension of Alloy with support for higher-order logic. However, its performance proved insufficient to obtain meaningful results in our experiments. For these reasons, we introduce a signature that enables us to quantify over paths, i.e., `all path: Path`. This signature and the relevant predicates are included in the `lib/lsts` module.

`Path` has three fields to exactly identify where it starts, where it ends, and the sequence of transitions that it represents. Empty paths play a role in our encoding, therefore we define a relation to easily find those paths with no transitions (`P_e`). A *valid* path follows the transition relation, and its start and end must coincide with the start and end of its associated transition sequence. We capture this in the predicates `valid_paths` and `valid_trseq`, where `first`, `last`, and `inds` are defined in the `seq` module.

```
sig Path {
  start: one AState,
  end: one AState,
  tr: seq Transition
}
let P_e = { p: Path | no p.tr }

pred valid_paths {
  all p: P_e | p.start = p.end
  all p: Path - P_e {
    p.start = p.tr.first.src
    p.end = p.tr.last.dest
  }
  all p: Path | valid_trseq[p.tr]
}
pred valid_trseq[tr: seq Transition] {
  all i: tr.inds | let t1 = tr[i], t2 =
  ↪  tr[add[i,1]] |
  (some t1 and some t2) => t1.dest = t2.src
}
```

A subtlety of this approach is that just defining a path signature does not mean that *all* paths of the system will actually be explored in the analysis. Instead, we need to use *generator axioms* to populate the set of paths (we refer to [15,

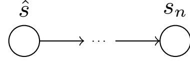

Fig. 1: A deadlock.

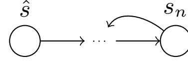

Fig. 2: A lasso.

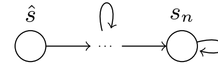

Fig. 3: An ignored path.

Section 5.3] for more details about the need for such axioms in Alloy). The generator axiom `all_paths_exist` builds paths inductively.

For each state $s \in S$, there must exist one empty path containing s. For every path p and every transition t, if appending t to p yields a *valid* path, that path must also exist.

```
pred all_paths_exist {
  all s: AState | one p: Path | p.start = s
  ↪  and p.end = s and no p.tr

  all p: Path, t: Transition |
    valid_path[p,t] => some q: Path | q.tr =
    ↪  p.tr.add[t] }
```

Above, `valid_path[p,t]` first checks whether the sequence represented by `t` appended to `p.tr` still follows the transition relation. Additionally, we cannot generate all paths because we consider LSTSs with infinite behaviour, i.e., with loops. Because we are in a bounded setting, we unfold each loop only once. To keep the bound on paths low and the formalisation manageable in complexity, we choose to only generate *deadlocking* paths (Fig. 1) and *lassos* (Fig. 2). We do this below by adding an extra condition to the predicate `valid_path`. The function `stateset` returns, in no particular order, the states occurring on a path. The last line of `valid_path` checks that `tr" = p.tr.add[t]` has the right shape.

Either `tr"` is a deadlock: its last state has no enabled actions, and furthermore there are no state repetitions. Otherwise, it is a lasso: the last state of the path also occurs earlier in the path and no other states are repeated.

```
fun stateset[p: Path] : set AState {
  p.start + p.tr.dest.elems }
pred valid_path [p: Path, t: Transition] {
  ...
  add[#(tr".inds),1] = #(stateset[p] +
  ↪  t.dest) or (add[#(p.tr.inds),1] =
  ↪  #stateset[p] and t.dest in
  ↪  stateset[p]) }
```

We claim that this restricted setting is sufficient for modelling POR conditions. In particular, our generator does not populate `Path` with paths shaped like the one in Fig. 3. However, it does lead to an under-approximation of path-based behavioural equivalences. We will discuss the consequence of this in Section 5.

Reduced Labelled-State Transition Systems Above, we provided a generic formalism for encoding POR theories in Alloy. We continue with another concept common to all theories: the *reduced* formalism. At a most basic level, the approach of POR, when performing state-space exploration in some state s, is to pick one or more outgoing transitions of s (based on the action labels) and ignore the others. If parts of the state space now become unreachable, we have effectively reduced the number of states to explore. This is captured in the *reduction function* and the *reduced LSTS*.

Definition 2. *Given an LSTS* $TS = (S, \hat{s}, \mathcal{L}, \mathcal{A}, T, L)$ *and a* reduction function $r : S \to 2^{\mathcal{A}}$*, the* reduced LSTS *is defined as* $TS_r = (S_r, \hat{s}, \mathcal{L}, \mathcal{A}, T_r, L_r)$*, where* L_r *is the restriction of* L *on* S_r*, and* S_r *and* T_r *are defined as the smallest sets satisfying* $S_r = \{\hat{s}\} \cup \{s' \in S \mid \exists s \in S_r, a \in r(s).(s, a, s') \in T_r\}$ *and* $T_r = \{(s, a, s') \in S_r \times \mathcal{A} \times S_r \mid a \in r(s)\}$*.*

To encode TS_r in Alloy, we extend TS by adding a field `r` to the signature `AState`. This field captures, for each state, the actions which will remain enabled in that state in TS_r. We extend our notation of transitions to reduced LSTSs such that $s \xrightarrow{a}_r s'$ iff $(s, a, s') \in T_r$ (`succ_r` models $\to_r$).

```
module lib/lsts [Label, A]

abstract sig AState {
  label: set Label,
  r: set A
}
...

let succ_r = {
  s, s": AState | some a: s.r | s->a->s" in
↪  T }
```

Notice how this approach "embeds" the reduced LSTS in the full LSTS, meaning that S_r, T_r, and L_r can simply be computed based on r.

3 Modelling POR for Labelled-State Transition Systems

Formalising POR in Alloy involves creating a model that (i) defines signatures for $\mathcal{L}$ and $\mathcal{A}$, (ii) imports the module `lib/lsts [Label, A]`, possibly extending the base signatures with fields or new signatures for the chosen formalism, (iii) defines a predicate for each reduction condition, and (iv) encodes the correctness theorem. Because steps (iii) and (iv) tend to vary with each theory, they cannot be generalised as easily as the LSTS. For this reason, this section demonstrates our method by encoding a specific POR variant: stubborn sets for LSTS.

We write `open lib/lsts [AP, Action]` to import the base module and specify that states are labelled with atomic propositions, and transitions, with actions. To help encode the equivalence relation between TS and TS_r later, we assume that states have exactly one label. The signature `Init` corresponds to $\hat{s}$.

```
module stubborn_lsts

open lib/lsts [AP, Action] as lsts

sig AP {}

sig Action {}

sig State extends AState {}{
  one label
}

one sig Init extends State {}
```

The generic nature of a reduction function allows also nonsensical reductions such as $r(s) = \emptyset$ for all s. Intuitively, the goal of POR is that from every class of equivalent interleavings, at least one is preserved. To achieve this, we need to impose some conditions on r: here, we present the *stubborn set* method. This method was originally presented in [33,34] and repaired in [22] by strengthening a condition called **D1**. We first define *key actions* and *(in)visible* actions.

An action a is *key* for $r(s)$ in s iff a is enabled in s' on all paths $s \xrightarrow{a_1 \dots a_n} s'$ where $a_1 \notin r(s), \dots, a_n \notin r(s)$. In this definition $n = 0$ is allowed, thus a is also required to be enabled in s.

```
fun enabled[s: AState]: set A { s.T.AState }

pred key_action[a: Action, s: State] {
  let reach = { t,t": State | some b: Action
  ↪   - s.r | t->b->t" in T } |
    all s": s.*reach |
      a in enabled[s"]
}
```

In Fig. 4, a is a key action in $\hat{s}$ because it is enabled in $\hat{s}$ and in all states that can be reached from $\hat{s}$ by performing actions that are not in $r(\hat{s})$.

An action a is part of a set $\mathcal{I}$ of invisible actions if, but not necessarily only if, $s \xrightarrow{a} s'$ implies that $L(s) = L(s')$ for all $s, s' \in S$. When $a \notin \mathcal{I}$, we say that a is visible.

```
let Viz = {
  a: Action | some t: Transition |
    t.label = a and not t.src.label =
    ↪  t.dest.label
}

let Inv = Action - Viz
```

In Fig. 4, action a is allowed to be in $\mathcal{I}$ because it always occurs between states having the same label. However, b is visible because it "changes" the labels, for instance $L(\hat{s}) \neq L(s_2)$.

The reduction r is called a *stubborn set* iff r satisfies the conditions **D1**, **D2w**, **V**, **I**, **L** in every state s. The particular variant that we consider here is called a weak stubborn set in [34]. We now introduce each condition and its corresponding predicate in Alloy. **D1** ensures that the actions that are selected for the stubborn set do not disable other paths. For instance, $r(\hat{s})$ does not disable b in Fig. 4 as witnessed by $\hat{s} \xrightarrow{ab} s_2$. Its Alloy predicate first finds the non-empty paths p of shape $s \xrightarrow{a_1 \dots a_n} s_n \xrightarrow{a} s'_n$. Then, it requires that every p has a *commuting* path: the actions occur in the same order as in p (except for a). Below, recall that `P_e` is the set of empty paths. We can quantify over sequences of transitions because we use an existential quantifier.

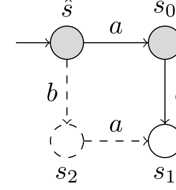

Fig. 4: An LSTS where $r(\hat{s}) = \{a\}$ and $r(s) = \mathcal{A}$ for all other states. Dashed states and transitions are not in T_r and S_r, thus they are eliminated by the reduction.

D1 *For all $s_1, \dots, s_n, s'_n$ and all $a \in r(s)$ and $a_1 \dots, a_n \notin r(s)$, if $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n \xrightarrow{a} s'_n$, then there are $s', s'_1, \dots, s'_{n-1}$ such that $s \xrightarrow{a} s' \xrightarrow{a_1} s'_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s'_n$.*

```
pred D1 { all s: State, a: s.r |
  let P = { p: start.s - P_e | (no
  ↪   p.tr.label.elems & s.r) and a in
  ↪   enabled[p.end] } |
  all p: P |
  some t": seq Transition | (
  valid_trseq[t"]
  and t".first.label = a
  and t".first.src = s
  and t".last.dest = a.(p.end.T)
  and t".rest.label = p.tr.label
}
```

The repaired condition, which we refer to as **D1'**, has an additional requirement on those $a \in r(s)$ that are invisible to prevent the so-called *inconsistent labelling problem* [22].

D1' *... Furthermore, if a is invisible, then $s_i \xrightarrow{a} s'_i$ for every $1 \leq i < n$.*

```
pred D1" {
  ... and (a in Inv => all i: p.tr.inds |
  ↪  p.tr[i].dest->a->t"[add[i,1]].dest in
  ↪  T)
}
```

Condition **D2w** ensures that the reduction does not introduce new deadlocks. By selecting key actions, we guarantee that something will happen in the future.

D2w *If $enabled(s) \neq \emptyset$, then $r(s)$ contains a key action in s.*

```
pred D2w { all s: State | some enabled[s]
↪  implies some a: s.r | key_action[a,s] }
```

Condition **V** ensures that the commuting paths in **D1** have the same sequence of visible actions.

V *If $r(s)$ contains an enabled visible action, then it contains all visible actions.*

```
pred V { all s: State | some enabled[s] &
↪  s.r & Viz implies Viz in s.r }
```

Condition **I** guarantees that invisible actions will eventually be performed.

I *If an invisible action is enabled in s, then $r(s)$ contains an invisible key action.*

```
pred I { all s: State |
  let key = { a: s.r | key_action[a,s] } |
  some enabled[s] & Inv implies some Inv &
  ↪  key }
```

Condition **L** prevents the *ignoring problem* [11,32], which happens when a visible action is infinitely often not chosen for the stubborn set. Here `P_r = {p: Path | all t: p.tr.elems | t.label in t.src.r}` selects paths that exist in the reduced LSTS.

L *For every visible action a, every cycle in the reduced LSTS contains a state s' such that $a \in r(s')$.*

```
pred L { let cycles = P_r & lassos |
  all a: Viz, p: cycles |
  some s: stateset[p] | a in s.r }
```

The goal of the reduction is to preserve *stutter equivalent* paths. The *trace* of a path $p = \hat{s} \to s_0 \to \dots$ is the sequence of state labels observed on p, $L(\hat{s})L(s_0)\dots$. Two paths are stutter equivalent iff they are either both finite (deadlock) or both infinite (lasso), and their traces are equal modulo *stuttering*, i.e., we drop consecutive label repetitions. We sketch the approach of the predicate `stutter_eq`. Checking two deadlocking paths is straightforward: we eliminate repetitions of labels and compare the resulting stutter-free traces. For two lassos this is more involved: it requires distinguishing the initial part of the lasso and the repetitive part before normalising them. Normalisation helps to identify

lassos that enter the loop at a different place (e.g., $a(ba)^\omega$ becomes $(ab)^\omega$) or that show a different number of unrollings (e.g., $ab(ab)^\omega$ becomes $(ab)^\omega$).

A path is *initial* when it starts in the initial state. A path is *complete* if it is deadlocking or a lasso (set `P_c = {p: Path | no p.end.enabled or is_lasso[p]}`, where predicate `is_lasso[p]` models the definition from Section 2). Complete reduced paths contain only stubborn actions labels (`P_c_r = {p: Path | (no p.end.enabled & p.end.r or is_lasso[p]) and all t: p.tr.elems | t.label in t.src.r}`). We relate an LSTS to its reduced LSTS by checking that every path in the full system has an equivalent in the reduced system. This correctness condition is captured in the following flawed theorem.

Theorem 1. *[33, Theorem 2] For every labelled-state transition system TS, given a stubborn set r (under conditions* **D1**–**L***), it holds that for every complete initial path p in TS, there is a complete initial path q in TS_r such that p and q are stutter equivalent.*

```
pred correctness {
  all p: start.Init & P_c | some q: start.Init & P_c_r |
  ↪  stutter_eq[p,q] }
```

4 Modelling POR for Parity Games

We continue by extending the model from the previous section to the POR method of stubborn sets for *parity games* [23,24]. Parity games are a versatile framework for encoding various decision problems, including deciding whether an LSTS satisfies a given temporal property (e.g. an LTL or μ-calculus formula). A parity game extends an LSTS: states are now owned by a *player* ("even" $\Diamond$ or "odd" $\Box$) and furthermore labelled with a *priority*, a natural number.

Definition 3 (Parity game). *A parity game is an LSTS $(S, \hat{s}, \mathbb{N} \times \{\Diamond, \Box\}, \mathcal{A}, T, L)$, where:*

- *$\mathbb{N}$ is the set of natural numbers,*
- *$\{\Diamond, \Box\}$ represents the two players.*

```
open lib/lsts[Int, Action]

sig Action {}
one sig Even {}
one sig Odd {}
sig State extends AState {
  player: one { Even + Odd }
}{
  label > 0 and label < 4
  one label
}
one sig Init extends State {}
```

Compared to the generally accepted definition of parity games, we added actions on the transitions, which is required for our POR theory. Note that the Alloy formalisation of L deviates from the definition because of a limitation of Alloy where a binary relation (in this case, the set $\mathbb{N} \times \{\Diamond, \Box\}$) may not be used as a module argument. While in our theory each state is labelled with a pair $(priority, player)$, in practice we use `s.label` to denote the priority, and `s.player` for the player owning s. We restrict the priority to values from $\{1, 2, 3\}$, as that suffices for the small examples we consider. Let $S_\Diamond$ denote the set of states owned by player $\Diamond$, i.e., $S_\Diamond = \{s \in S \mid \mathcal{P}(s) = \Diamond\}$. This set is $\{\hat{s}, s_1\}$ in Fig. 5.

During the game, players move a token from one state to the next: the owner of the state where the token currently resides may choose along which outgoing transition to move the token. This yields an infinite or finite (in case the token ends up in a state without successors) path that the token has traversed. An infinite path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots$ is won by player $\Diamond$ iff the least priority that occurs infinitely often along π is even. A finite path $s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ is won by player $\Diamond$ iff $\mathcal{P}(s_n) = \Box$. Otherwise, these paths are won by $\Box$. A *strategy* for a player p is a function $\sigma : S_p \to S$ such that, for all $s \in S_p$, $\sigma(s)$ is a successor of s (i.e. $s \to \sigma(s)$)[3]. A path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots$ is *consistent* with a strategy σ iff $\sigma(s_i) = s_{i+1}$ for all i such that $\sigma(s_i)$ is defined. Finally, a strategy σ is *winning* for player p in a state s iff all complete paths that start from s and are consistent with σ are won by player p. A state s is won by p iff there is a winning strategy for p in s. These concepts are modelled in Alloy as follows. In predicate `win_state` (and the analogous `r_win_state` for reduced games), we select the relevant paths by intersecting the set of paths starting in s (`start.s`) with the set of complete paths (`P_c`). Note that `win_path` and `win_state` concern player $\Diamond$.

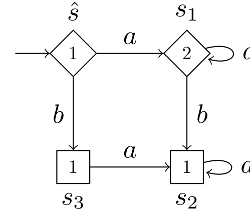

Fig. 5: A four-state parity game. Priorities are inscribed in the states.

```
pred win_path [pi: Path] {
  some pi
  is_lasso[pi] implies not
  ↪   odd_priority[min_priority[pi]] else
  ↪   pi.end.player = Odd }
```

```
pred win_state [s: State] {
  some st: Strategy {
    all p: start.s & P_c |
    consistent[p, st.move] implies
    ↪   win_path[p] } }
```

In Fig. 5, $\Diamond$ can win state s_1 by choosing to never move the token from s_1, and similarly it can win the initial state by always moving the token to s_1. However, $\Diamond$ has no winning strategy for the other states, because the token is trapped in an infinite repetition of an odd priority.

Similar to paths, strategies must be modelled as atoms. So, we introduce a strategy signature and an accompanying generator axiom. We do not need to model strategies for $\Box$, since any results for $\Diamond$ carry over by duality. Thus, strategies in Alloy look like `sig Strategy { move: S_e -> S }`, where `S_e` is $S_\Diamond$ and `move` models σ.

The generator `all_strategies_exist` below has two steps. First, we require the existence of some σ such that $\sigma(s) = s'$ for every transition $s \to s'$. For example, for the parity game in Fig. 5, this generates $\sigma_1(\hat{s}) = s_1$, $\sigma_1(s_1) = s_1$ and $\sigma_2(\hat{s}) = s_3$, $\sigma_2(s_1) = s_2$. This is insufficient: a strategy where $\hat{s}$ moves to s_1 and s_1 to s_2 is missing, among others. Thus, we also ensure the presence of all combinations of strategies generated in the first step by comparing them pairwise and calculating their differences.

[3] We use Zielonka's theorem about *positional determinacy* [35] (either $\Diamond$ or $\Box$ wins in any state s) to forego histories and only reason about *memoryless* strategies.

```
pred all_strategies_exist {
  all s: S_e, s": s.succ | some st: Strategy | s.(st.move) = s"
  all disj s1, s2: Strategy {
    let diff1 = s2.move - s1.move | all s: diff1.State | some s3: Strategy |
      s.(s3.move) = s.diff1 and all t: S_e-s | t.(s3.move) = t.(s2.move)
    let diff2 = s1.move - s2.move | all s: diff2.State | some s3: Strategy |
      s.(s3.move) = s.diff2 and all t: S_e-s | t.(s3.move) = t.(s2.move) } }
```

On top of the conditions **D1'**–**L** presented in Section 3, stubborn sets for parity games require adding condition **P** [24]. This is necessary for preserving the winning player by taking into account that control of the token may move between the players.

P *If there is an action* $a \in r(s)$ *and a state* t *such that* $s \xrightarrow{a} t$ *and* $\mathcal{P}(s) \neq \mathcal{P}(t)$*, then* $r(s) = \mathcal{A}$.

```
pred P { all s: State |
  (some a: s.r, t: State | s->a->t in T and
  ↪  not s.player = t.player)
  implies s.r = Action }
```

The goal of the reduction is to preserve the winning states of each player. We restate the flawed theorem and give the Alloy predicate that captures the preservation of winners. By duality and positional determinacy, the predicates `win_state` and `r_win_state` only need to check $\Diamond$'s strategies.

Theorem 2. *[23, Theorem 1] For every parity game PG, given a stubborn set r (under conditions* **D1'-L***), it holds that for every state s in the reduced game* PG_r*, the winner of s in* PG_r *is equal to the winner of s in PG.*

```
pred correctness {all s: State | win_state[s] iff r_win_state[s]}
```

5 Testing POR in Alloy

This section covers our approach to testing POR theories in Alloy. We discuss our experimental set-up, considerations for defining the size of the experiments, and the results. We modelled all the above theory, as well as the theories of [1,4] and their repaired version where available [5]. These additional models do not contain new constructs compared to what was presented above. We focused on two types of tests, (1) reproducing counterexamples to flawed theories from the literature [21,25], and (2) checking for the absence of counterexamples for the corrected theories. We also replicated an Alloy model of another POR technique [27,28] with corrections [30]. We used Alloy 6.2.0 with Kissat 4.0.3 [2] as an external SAT solver. All tests were run five times on an Intel Xeon Gold 6136 with a timeout of 48 hours.

We list some factors for setting experiment bounds. Kodkod [31] is the engine responsible for the translation between the Alloy language and CNF. It requires that n^k is smaller than the maximum integer value in Java, where n is the size of the Alloy universe and k is the arity of a relation in our model. Our formalisations model relations of arity 5: conditions **D1** and **D1'** for LSTS and parity games, and **W** for reachability games. Thus the size of the universe can be at most 73. Furthermore, the `seq` bound has to coincide with the S bound. Recall our

Table 1: Testing set-up, size of the generated CNF, and average running time

Theory	Bound							Symm. breaking				CNF		SAT	Time
	seq	S	$\mathcal{A}$	$\mathcal{L}$	T	Path	Strat	$\mathcal{L}$	T	Path	Strat	#var	#clause		
LSTS[33]	5	5	2	2	9	24	-	-			-	671 500	2 186 341	✓	53m
								-	✓	✓	-	604 430	2 028 322	-	t-o
PG[23]	5	5	4	3	9	26	6	-				1 111 592	3 351 940	✓	43m
								-	✓	✓	✓	1 008 957	3 094 087	-	t-o
RG[4]	4	4	2	3	4	10	1	✓				76 603	181 450	✓	3s
								✓	✓	✓	✓	62 258	152 413	✓	6s
LKS[1]	4	4	2	1	3	8	-	✓			-	18 550	43 815	✓	0.1s
								✓	✓	✓	-	15 718	38 420	✓	0.1s
LSTS✓[22]	5	5	2	2	9	24	-	-			-	722 133	2 282 564	✓	11h48m
								-	✓	✓	-	655 063	2 124 545	-	t-o
PG✓[24]	4	4	3	3	6	17	4	-				256 757	692 198	✗	1h54m
								-	✓	✓	✓	227 567	624 244	-	t-o
	5	5	4	3	9	26	6	-				1 111 902	3 352 376	-	t-o
								-	✓	✓	✓	1 009 267	3 094 523	-	t-o
RG✓[5]	4	4	2	3	4	10	1	✓				76 465	181 216	✗	6s
								✓	✓	✓	✓	62 140	152 199	✗	2m
PA✓[30]	4	4	4	2	5	1	-	✓			-	18 761	36 364	✗	0.1s
								✓	✓		-	17 740	34 634	✗	0.1s

assumption that each state is reachable from $\hat{s}$. Then the longest possible path is a lasso of length $|S|$, which is encoded as an Alloy sequence of length $|S|$. The bound of T should also accommodate this. The *Path* scope has a lower bound of $|S| + |T|$ because we generate at least the empty path for each state and a path containing each transition. Due to the generator axiom, the upper bound must be high enough to allow all possible paths. In our experiments we often overestimated this bound, to avoid the model being trivially UNSAT. For games, $\mathcal{L}$ has bound 3 since we always draw the label from a set of three values.

To reduce the runtime needed for larger instances, we prune the search space using *symmetry breaking*. By default, the atoms belonging to signatures are unordered, meaning that any satisfying assignment is part of an equivalence class of assignments obtained by permuting the atoms. Alloy always tries to eliminate symmetries internally, but this built-in mechanism might not perform optimally unless we explicitly order our signatures. To impose an order, we import the built-in `ordering` module (by writing `open util/ordering[sig] as ord_sig` in the same file that defines that signature). This module creates a lexicographic order [29] on the signature and it enforces a predicate that holds true only for the lexically smallest element. However, it also forces the signatures to be exactly bounded by the number we set, instead of upper-bounded. In practice, `ordering` creates relations `First` and `Next` for the considered signature, which helps expose the symmetries to the solver. We always apply the ordering to the

sets S and $\mathcal{A}$ and (where applicable) to $\mathcal{L}$. In a separate run, we also apply it to T (specifically to `sig Transition`, not `fun T`), *Path* and *Strategy*. For some theories, a lexicographic ordering cannot be imposed on $\mathcal{L}$ due to a name clash: the `ordering` functions `next` and `last` clash with our encoding of parity games, where Alloy's integer module used for $\mathcal{L}$ also defines a function `next`. It also clashes with our LSTS model, because we use the `seq` function `last`. Therefore, we do not import `ordering` on $\mathcal{L}$ for those models.

Table 1 summarises our results. We refer here to each theory by abbreviating the name of its formalism: labelled-state transition system (LSTS), parity game (PG), reachability game (RG), labelled Kripke structure (LKS), product automaton (PA). The first four rows in the "Theory" column concern tests of type (1) and the last four, tests of type (2). In the column "Bound" we list the most important signature bounds, and we indicate with a dash when the Alloy encoding does not use that signature. Each row is then divided in two, where the top corresponds to a run with only some symmetry breaking, and the bottom, to a run with symmetry breaking on all signatures. The next column shows whether a lexicographic order was applied for that particular test. Here we again use dashes when the signatures are not used in the encoding. Since we always apply it to S and $\mathcal{A}$, we omit this in the table. The column "CNF" shows the size of each generated CNF formula in terms of variables and clauses. The "SAT" column shows whether the formula is satisfiable or unsatisfiable. For type (1) tests we wish for SAT, which means that a counterexample exists. Conversely, for type (2) tests we wish for the opposite. Finally, the last column contains the running time averaged over the five runs (a timeout is denoted "t-o").

Alloy found every counterexample from the literature for the flawed theories, or at least one that comes close to it. We first discuss the result for stubborn sets for labelled-state transition systems in detail, and then briefly address the others. Fig. 6 shows the counterexample that we set out to reproduce, and Fig. 7, the Alloy counterexample for the following specification.

```
pred test {
  D1 and D2w and I and V and L
  some Init.r
  some enabled[Init] - Init.r
  some enabled[Init] & Init.r }
```

```
check lsts {
  test => correctness
} for 5 seq, 5 State, 2 Action, 2 AP, 9
↪  Transition, 24 Path
```

To avoid generating instances where the conditions are trivially satisfied, in the last three lines of predicate `test` we require that there is some reduction in the initial state. In the `check` command, `correctness` corresponds to Theorem 1.

The LSTS in Fig. 7 captures both the full and the reduced system: the dashed states and transitions are exactly those which are unreachable in the reduced LSTS. The gray-coloured states $\hat{s}, s_0, s_3$ are labelled with $\{l\}$ and the others unlabelled, and we have two actions $\mathcal{A} = \{a, b\}$, with $\mathcal{I} = \{a\}$. The stubborn set is $r(\hat{s}) = \{a\}$ and $r(s) = \mathcal{A}$ for all other $s \in S$, i.e., we only reduce in the initial state. We now explain why this is a valid counterexample by arguing that r satisfies the reduction conditions, while the correctness theorem is violated. Condition **D1** holds because $r(\hat{s})$ does not disable b: the path $\hat{s} \xrightarrow{b} s_2 \xrightarrow{a} s_1$

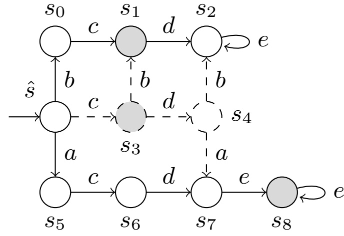

Fig. 6: An LSTS from the literature that violates [33, Theorem 2].

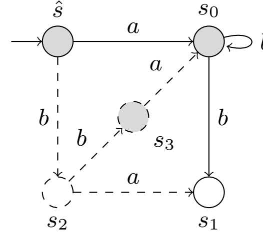

Fig. 7: An Alloy LSTS that violates [33, Theorem 2].

commutes with $\hat{s} \xrightarrow{a} s_0 \xrightarrow{b} s_1$, and similarly $\hat{s} \xrightarrow{b} s_2 \xrightarrow{b} s_3 \xrightarrow{a} s_0$ commutes with $\hat{s} \xrightarrow{a} s_0 \xrightarrow{b} s_0 \xrightarrow{b} s_0$. Action a is key in $\hat{s}$, thus **D2w** holds. **V** is satisfied because the visible action b occurs in the same order on the commuting paths considered in **D1**. **I** holds as $a \in r(\hat{s})$ is an invisible key action in $\hat{s}$. **L** is satisfied because the only cycle in the reduced LSTS is in state s_0, where no reduction is applied. Thus, the stubborn set is sound but the trace $\{l\}\emptyset\{l\}\{l\}\emptyset$ from the full LSTS (to find this trace in Alloy, we open the instance in the evaluator and write `correctness_p.tr._trace`) has no stutter-free equivalent in the reduced LSTS. This new, Alloy-generated LSTS is exactly half the size of the original counterexample. Although the manually-constructed counterexample is larger, it may offer more insight because it was built "backwards" from the proof of the unsound theorem. Moreover, it is easier to understand because it is deterministic. On the other hand, Alloy found a counterexample in under an hour without access to the proof, while the flaw lay undiscovered for close to three decades.

For parity games we found an equivalent, slightly simpler counterexample to [23, Theorem 1] than the literature counterexample [24, Fig. 2(d)]. For reachability games we managed to generate the exact counterexample from [22, Fig. 12(b)] which violates [4, Theorem 3.6]. Lastly, for labelled Kripke structures we found a similar counterexample to the one in the literature [22, Fig. 12(a)] where [1, Theorem 6.5] is violated.

In the second type of experiment we check that the CNF generated from the corrected theory is unsatisfiable. For example, for LSTS this means using the same predicate `test`, except that on the first line we replace `D1` by `D1"`. We test each theory with the same bounds as for the previous experiments. For reachability games, we find no counterexamples and terminate in a matter of seconds. For labelled-state transition systems, Alloy found an *invalid* counterexample (Fig. 8). We call a counterexample invalid when we can manually verify that it does not actually violate the checked assertion. In this case, the stubborn set satisfies the conditions and the correctness theorem actually holds. This is a consequence of the way we modelled paths: the path $\hat{s} \xrightarrow{a} s_0 \xrightarrow{b} s_2 \xrightarrow{b} s_0 \xrightarrow{a} s_1$ is shaped like the path in Fig. 3 due to the repetition of s_0. Thus it is not contained in the signature `Path`. Thus it is missed in the computation of stutter equivalence, and, according to our model, the reduction yields a system that is

not stutter equivalent to the full LSTS. To resolve this, an alternative approach would be to take each path in the full LSTS and reorder its actions to compute a matching path in the reduced LSTS. After all, partial-order reduction exploits exactly those interleavings where the actions can be reordered. We leave this for future work.

For parity games, the bounds of our first experiment result in timeouts. Therefore, we also checked the corrections up to a lower set of bounds derived from [24, Fig. 2(c)]. Lastly, we modelled ample sets applied to product automata [27,28]. We adapted Siegel's existing Alloy model [30] to our framework, including the proposed corrections, and checked correctness up to the bound of the original counterexample. Compared to the other theories, which required higher-order constructions for paths and/or strategies, ample sets for product automata can be formalised without generating paths, making the model simpler and faster.

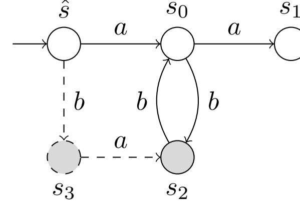

Fig. 8: An invalid counterexample to the repaired theory of stubborn sets for LSTS [22].

It is clear from Table 1 that the experiments timed out or took longer to terminate when we enforced orderings on T, *Path*, and *Strategy*. We have also experimentally found that enabling this one-by-one still impacts the runtime negatively. Given that the CNF always becomes strictly smaller with this optimisation, it is counter-intuitive for the performance to be worse. A possible explanation could be that, in our models, the individual atoms of those signatures are not used meaningfully; we are only interested in their fields. We therefore speculate that applying an ordering predicate in this case is not beneficial in any way.

6 Conclusion

We presented a method for reducing errors in formal theories, specifically partial-order reduction. We can conclude from our results that modelling theories in Alloy is a step in that direction. Our experience is that our approach is significantly less time-intensive than using proof assistants, and that it also provides useful support for troubleshooting and making repairs.

In further research, we intend to improve our formalisation of paths and stutter equivalence as discussed above. The LSTS module could be extended with generalisations of commonly-used definitions, like key actions, (in)visible actions, and strategies. Furthermore, we would like to investigate how Alloy records [6] could improve the efficiency of our implementation of generated signatures. It would also be interesting to see whether applying optimisations at the SAT-solving stage could help in obtaining correctness results for higher bounds, in particular for parity games. A different direction would be to use this methodology to model complex algorithms, such as bisimilarity reduction [26].

Acknowledgments. T. Neele is supported by NWO grant VI.Veni.232.224.

References

1. Beneš, N., et al.: Partial order reduction for state/event LTL with application to component-interaction automata. Sci. Comput. Program. **76**(10), 877–890 (2011). https://doi.org/10.1016/J.SCICO.2010.02.008
2. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling: Entering the SAT Competition 2020. In: SAT Competition 2020. pp. 50–53 (2020)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability - Second Edition. IOS Press (2021). https://doi.org/10.3233/FAIA336
4. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñiz, M., Srba, J.: Partial Order Reduction for Reachability Games. In: CONCUR. LIPIcs, vol. 140, pp. 23:1–23:15. Schloss Dagstuhl (2019). https://doi.org/10.4230/LIPICS.CONCUR.2019.23
5. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñiz, M., Srba, J.: Stubborn Set Reduction for Two-Player Reachability Games. Log. Methods Comput. Sci. **17**(1) (2021). https://doi.org/10.23638/LMCS-17(1:21)2021
6. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: Adding Records to Alloy. In: ABZ2023. LNCS, vol. 14010, pp. 212–219. Springer (2023). https://doi.org/10.1007/978-3-031-33163-3_16
7. Brunner, J., Lammich, P.: Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. J. Autom. Reason. **60**(1), 3–21 (2018). https://doi.org/10.1007/S10817-017-9418-4
8. Chou, C., Peled, D.A.: Formal Verification of a Partial-Order Reduction Technique for Model Checking. J. Autom. Reason. **23**(3-4), 265–298 (1999). https://doi.org/10.1023/A:1006225515062
9. van Delft, M., Geuvers, H., Willemse, T.A.C.: A Formalisation of Consistent Consequence for Boolean Equation Systems. In: ITP. LNCS, vol. 10499, pp. 462–478. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_29
10. DeMillo, R.A., Lipton, R.J., Perlis, A.J.: Social Processes and Proofs of Theorems and Programs. Commun. ACM **22**(5), 271–280 (1979). https://doi.org/10.1145/359104.359106
11. Evangelista, S., Pajault, C.: Solving the ignoring problem for partial order reduction. Int. J. Softw. Tools Technol. Transf. **12**(2), 155–170 (2010). https://doi.org/10.1007/S10009-010-0137-Y
12. van Glabbeek, R.J., Ploeger, B.: Correcting a Space-Efficient Simulation Algorithm. In: CAV. LNCS, vol. 5123, pp. 517–529. Springer (2008). https://doi.org/10.1007/978-3-540-70545-1_49
13. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, LNCS, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7
14. Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: The Spin Verification System. DIMACS Series, vol. 32, pp. 23–31. DIMACS/AMS (1996). https://doi.org/10.1090/DIMACS/032/03
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)
16. Jackson, D.: Alloy: a language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019). https://doi.org/10.1145/3338843
17. Keiren, J.J.A., Fontana, P., Cleaveland, R.: Corrections to "a menagerie of timed automata". ACM Comput. Surv. **50**(3), 42:1–42:8 (2017). https://doi.org/10.1145/3078809

18. Laveaux, M., Groote, J.F., Willemse, T.A.C.: Correct and Efficient Antichain Algorithms for Refinement Checking. Log. Methods Comput. Sci. **17**(1) (2021). https://doi.org/10.23638/LMCS-17(1:8)2021
19. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. Formal Methods Syst. Des. **55**(1), 1–32 (2019). https://doi.org/10.1007/S10703-016-0267-2
20. Miulescu, M.: Computer-Assisted Verification of Partial-Order Reduction Methods. Master's thesis, Eindhoven University of Technology (2025)
21. Neele, T., Valmari, A., Willemse, T.A.C.: The Inconsistent Labelling Problem of Stutter-Preserving Partial-Order Reduction. In: FoSSaCS. LNCS, vol. 12077, pp. 482–501. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_25
22. Neele, T., Valmari, A., Willemse, T.A.C.: A Detailed Account of The Inconsistent Labelling Problem of Stutter-Preserving Partial-Order Reduction. Log. Methods Comput. Sci. **17**(3) (2021). https://doi.org/10.46298/LMCS-17(3:8)2021
23. Neele, T., Willemse, T.A.C., Wesselink, W.: Partial-Order Reduction for Parity Games with an Application on Parameterised Boolean Equation Systems. In: TACAS (2). LNCS, vol. 12079, pp. 307–324. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_19
24. Neele, T., Willemse, T.A.C., Wesselink, W., Valmari, A.: Partial-order reduction for parity games and parameterised Boolean equation systems. Int. J. Softw. Tools Technol. Transf. **24**(5), 735–756 (2022). https://doi.org/10.1007/S10009-022-00672-0
25. Neele, T.S.: Reductions for parity games and model checking. Ph.D. thesis, Technische Universiteit Eindhoven (2020)
26. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987). https://doi.org/10.1137/0216062
27. Peled, D.A.: Combining Partial Order Reductions with On-the-fly Model-Checking. In: CAV. LNCS, vol. 818, pp. 377–390. Springer (1994). https://doi.org/10.1007/3-540-58179-0_69
28. Peled, D.A.: Combining Partial Order Reductions with On-the-Fly Model-Checking. Formal Methods Syst. Des. **8**(1), 39–64 (1996). https://doi.org/10.1007/BF00121262
29. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. Electron. Notes Discret. Math. **9**, 19–35 (2001). https://doi.org/10.1016/S1571-0653(04)00311-7
30. Siegel, S.F.: What's Wrong with On-the-Fly Partial Order Reduction. In: CAV (2). LNCS, vol. 11562, pp. 478–495. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_27
31. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: TACAS. LNCS, vol. 4424, pp. 632–647. Springer (2007). https://doi.org/10.1007/978-3-540-71209-1_49
32. Valmari, A.: Stubborn sets for reduced state space generation. In: Applications and Theory of Petri Nets. LNCS, vol. 483, pp. 491–515. Springer (1989). https://doi.org/10.1007/3-540-53863-1_36
33. Valmari, A.: A Stubborn Attack on State Explosion. Formal Methods Syst. Des. **1**(4), 297–322 (1992). https://doi.org/10.1007/BF00709154
34. Valmari, A., Hansen, H.: Stubborn Set Intuition Explained. Trans. Petri Nets Other Model. Concurr. **12**, 140–165 (2017). https://doi.org/10.1007/978-3-662-55862-1_7

35. Zielonka, W.: Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998). `https://doi.org/10.1016/S0304-3975(98)00009-7`