Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Pardinus: A temporal relational model finder

Nuno Macedo · **Julien Brunel** · **David Chemouil** · **Alcino Cunha**

Received: date / Accepted: date

Abstract This article presents Pardinus, an extension of the popular Kodkod [53] relational model finder with linear temporal logic (including past operators), to simplify the analysis of dynamic systems. Pardinus includes a SAT-based bounded-model checking engine and an SMV-based complete model checking engine, both allowing iteration through the different instances (or counter-examples) of a specification. It also supports a decomposed parallel analysis strategy that improves the efficiency of both analysis engines on commodity multi-core machines.

Keywords Model Checking · Model Finding · Relational Logic · Temporal Logic

1 Introduction

High-level model finders and constraint solvers are becoming increasingly useful in software engineering. The ability to specify properties of a system in some expressive logic and then automatically find solutions (models) that satisfy such properties is useful in many applications, ranging from early system design validation to test-case generation. Kodkod [53] is an example of such model finders, supporting a range of features that make it quite popular:

Work financed by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation (COMPETE2020) and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT) within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007.

Nuno Macedo
INESC TEC and Faculdade de Engenharia da Universidade do Porto
E-mail: nfmmacedo@di.uminho.pt

Julien Brunel
ONERA DTIS and Université fédérale de Toulouse
E-mail: julien.brunel@onera.fr

David Chemouil
ONERA DTIS and Université fédérale de Toulouse
E-mail: david.chemouil@onera.fr

Alcino Cunha
INESC TEC and Universidade do Minho
E-mail: alcino@di.uminho.pt

- Problems are described using the single unifying concept of *relation* (of arbitrary arity), considerably simplifying the syntax and semantics of the language.
- Constraints are expressed in *relational logic*, first-order logic enriched with relational algebra and closure operators, enabling a terse, but still readable, style of specification.
- It allows the user to *iterate* over alternative solutions of the problem, also implementing a symmetry breaking mechanism (to avoid the generation of equivalent solutions) which makes it useful for scenario exploration.
- *Partial instances* can be provided *a priori*, in the form of lower- and upper-bounds for relations, enabling its application to configuration-solving tasks, where the goal is to find a full instantiation of a partial description of a system.

The best-known application of Kodkod is in the analysis of Alloy 4 and 5 specifications. Alloy [25] is a language that shares some of Kodkod's features – the *everything is a relation* motto and the usage of relational logic – but that also supports higher-level constructs to further simplify the description of a system, namely a type system with inheritance.

Despite its usefulness and popularity, Kodkod can only be directly applied to analyse structural designs. Analysis of behavioural designs is possible, but cumbersome and error-prone. The state and traces of the system must be explicitly modelled and temporal properties and (bounded) model checking must be specified directly using transitive closure over the traces. This approach is often viable for checking simple safety properties, but properly checking liveness properties is tricky and mostly avoided. Moreover, given the bounded nature of the analysis, complete model checking could only be directly supported by setting a bound that covers all reachable states, is infeasible for most examples.

This article presents the Pardinus model finder, an extension of Kodkod that addresses this limitation. It allows the declaration of mutable relations and the usage of *temporal relational logic* in the specification of properties, an extension of relational logic with linear temporal logic with past operators (PLTL). Pardinus problems can currently be analysed by two model finding backends that implement satisfiability checking for temporal relational logic: the first translates Pardinus problems back to plain Kodkod problems, by resolving the temporal domain and implementing a procedure that essentially amounts to bounded model checking with SAT [3]; the second resolves the first-order domain, and reduces Pardinus satisfiability checking to PLTL model checking over a universal model of a system (one that allows all possible behaviours) [45], using the concrete SMV syntax. Like Kodkod, Pardinus allows iteration over different satisfiable solutions, enabling the user to quickly explore behavioural scenarios, either different valid execution instances of a system or different counter-examples to a broken expected property. To speedup analysis, Pardinus also implements a decomposed solving procedure that splits a problem into two parts, one containing only immutable relations that is used to enumerate the configurations of the system, which are then incorporated in the remainder mutable part and analysed in parallel. To further speedup the analysis, users can provide more precise partial instances about mutable relations using *symbolic bounds*, which generalize the bounds supported by Kodkod with expressions referring to the value of immutable relations, thus incorporating in a partial instance information about the system configuration.

The main application of Pardinus is in the analysis of Alloy 6[1] specifications. This new version of Alloy adds support for mutable relations and temporal relational logic, an extension previously known as Electrum [30]. The architecture of Alloy 6 and Pardinus is depicted in Fig. 1, with the scope of this article is captured by thick lines and arrows. Pardinus is also used as a backend in Forge [50], a system to prototype formal methods tools.

[1] https://github.com/AlloyTools/org.alloytools.alloy/releases/tag/v6.0.0

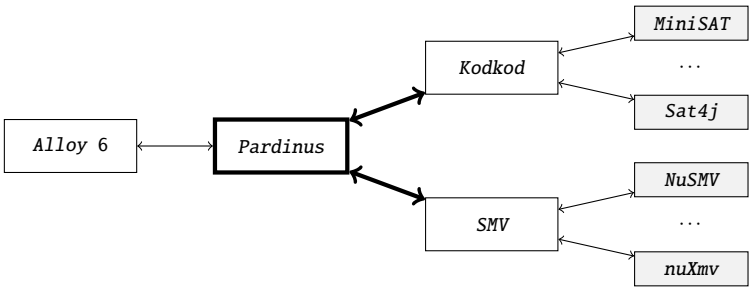


Fig. 1: Alloy 6 and Pardinus architecture

This article has four main contributions, when compared to previous publications presenting Pardinus and Electrum:

- A unified and complete presentation of both analysis backends (bounded and unbounded model checking). The paper that introduced Electrum [30] briefly mentions how specifications can be model checked, but at the time Pardinus did not exist and the two backends were not unified (bounded model checking was done via a translation to plain Alloy, and the SMV model checking backend was not as optimised as the current version). In a more recent tool paper about the current version of the Electrum Analyzer [5], Pardinus is already mentioned as the underlying model finder but not described.
- A novel path iteration mechanism, that returns only non-isomorphic solutions, and that is efficiently implemented using incremental SAT solving. Trace iteration was approached in [6], but only for single state updates and without an efficient implementation.
- A decomposed analysis technique that relies on symbolic bounds and parallel execution to speed up verification. The support for symbolic bounds and the decomposed analysis technique was introduced in [33], but only for plain Kodkod problems.
- An extended evaluation, with several new examples and case-studies, providing more confidence about the effectiveness of the proposed techniques.

The article is structured as follows. Section 2 presents Pardinus problems for temporal relational model finding. The analysis of such problems is discussed in Section 3, including the presentation of the two backends. Section 4 describes how instance (or counter-example) iteration is supported. The parallel decomposed analysis strategy and symbolic bounds are presented in Section 5. Section 6 presents and discusses the results of evaluating Pardinus over several examples and case-studies. Related work is discussed in Section 7. Section 8 closes the paper, presenting some conclusions and ideas for future work.

2 Pardinus problems

A Kodkod model finding *problem* consists of a set of relation declarations plus a single relational logic formula defined over those (free) relations, whose satisfiability is to be checked. To make the problem decidable every free relation must be given an upper-bound – the set of tuples that may be present in the relation in valid bindings. Tuples are sequences of *atoms* (uninterpreted identifiers) drawn from a finite universe, that must also be declared upfront. When declaring a relation it is also possible to specify a lower-bound – the set of tuples that must be present in the relation – useful to capture *a priori* partial knowledge about the solution. Pardinus problems extend Kodkod ones as follows:

after ϕ	ϕ is true in the next instant
always ϕ	ϕ will always be true
eventually ϕ	ϕ will eventually be true
ϕ **until** ψ	ψ will eventually be true and ψ will always be true until then
ϕ **releases** ψ	ψ will be true at least until ϕ is true
before ϕ	there was a previous instant and ϕ was true in it
historically ϕ	ϕ was always true
once ϕ	ϕ was once true
ϕ **since** ψ	ψ was once true and ψ was always true afterwards
ϕ **triggered** ψ	ψ was always true at least since ϕ was true

Fig. 2: Pardinus temporal operators

```
{I0,I1,I2,I3,P0,P1,P2,P3}

    Id      :1 {(I0),(I1),(I2),(I3)} {(I0),(I1),(I2),(I3)}
    next    :2 {(I0,I1),(I1,I2),(I2,I3)} {(I0,I1),(I1,I2),(I2,I3)}
    Process :1 {} {(P0),(P1),(P2),(P3)}
    id      :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),...,
                   (P3,I0),(P3,I1),(P3,I2),(P3,I3)}
    succ    :2 {} {(P0,P0),(P0,P1),(P0,P2),(P0,P3),...,
                   (P3,P0),(P3,P1),(P3,P2),(P3,P3)}
var outbox  :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),...,
                   (P3,I0),(P3,I1),(P3,I2),(P3,I3)}
var Elected :1 {} {(P0),(P1),(P2),(P3)}

id in Process → Id and
all p : Process | one p.id and
all i : Id | lone id.i and
succ in Process → Process and
all p : Process | one p.succ and
all p : Process | Process in p.^succ and

outbox = id and
always some p : Process, i : (succ.p).outbox | outbox' =
                    outbox − succ.p → i + p → (i − ^next.(p.id)) and

always Elected = {p : Process | once (p.id in p.outbox and
                                      before not (p.id in p.outbox))}
```

Fig. 3: A leader election protocol in Pardinus

- Mutable relations, whose value changes over time, can be declared with keyword **var**.
- Formulas can use (past and future) linear temporal operators to express behavioural constraints. The informal meaning of these operators is presented in Fig. 2.
- A relational expression can be primed to denote its value in the succeeding time instant.

The value of the immutable relations, that remains constant in a trace after being fixed at start, constitutes a so-called *configuration* of the system. The syntax[2] and semantics of Pardinus problems[3] will be illustrated through a simple example shown in Fig. 3, a specification of a leader election protocol. This protocol, first proposed by Chang and Roberts [10], assumes a ring network of processes (or nodes) with unique comparable identifiers.

[2] In practice, Kodkod and Pardinus are Java libraries and problems are defined programatically.

[3] Kodkod (and Pardinus) also have some limited support for integers which we omit in this presentation.

Specifying configurations (ll. 1–9, 14–19) The immutable portion of the problem is essentially pure Kodkod and specifies networks following the ring topology, amounting to the configuration of the protocol. To bound the problem, only rings with up to four nodes will be considered in the example. Thus, the mandatory universe declaration (l. 1) introduces four atoms to denote the processes (`P0` to `P3`) and four atoms for the identifiers (`I0` to `I3`). Next, a set of free relations can be declared that are the target of the model finding process. For each relation, besides its name, one must declare its arity (the length of the tuples it can contain), and its lower- and upper-bounds as tuple sets of the same arity. This problem declares two immutable sets (sets are simply normal unary relations) – `Id` (l. 3) and `Process` (l. 5) to denote the set of identifiers and processes, respectively, that will effectively exist in each solution – and three immutable binary relations – `next` to capture the total order between identifiers (l. 4), `id` to associate processes with their identifiers (l. 6), and `succ` to represent the desired topology, associating each process with its successor in the ring (l. 8).

By setting the lower-bound equal to the upper-bound, relations `Id` and `next` are declared as constants of the problem, with `next` fixing a particular total order between the four possible identifiers. This is an extreme usage of lower-bounds: in many situations the lower-bound is a smaller subset of the upper-bound, forcing only some of the possible tuples to be present in the valid bindings. In the remaining three immutable declarations, the lower-bound falls on the other extreme, being empty to allow any possible valuation within the respective upper-bound. `Process` is restricted to be any subset of the four possible process atoms (recall we intend to specify all rings with up to four processes), `id` to contain pairs where the first component is a process and the second is an identifier, and `succ` to only contain pairs of processes. The upper-bounds usually encode (loose) typing restrictions, but are not sufficiently expressive to restrict valid valuations. For instance, the upper-bound of `id` alone does not ensure that its tuples only relate processes that are effectively assigned to `Process`, which needs to be enforced in the problem's constraint. However, it still considerably speeds up the analysis by restricting upfront possible valuations.

Then, constraints of the problem are specified with a temporal relational logic formula, whose free variables are the relations previously declared. Relational logic combines the standard first-order logic primitives (Boolean connectives and first-order quantifiers) with relational algebra operators that are used to combine relational expressions. Except for the closure operators, relational operators do not increase the expressiveness of first-order logic, but simplify the writing of some constraints. Besides the standard binary set operators – union (`+`), intersection (`&`), difference (`-`) – relational expressions can also be combined with the Cartesian product (`->`), that concatenates all pairs of tuples from two expressions, the transpose (`~`), that reverses all tuples in a binary expression, and the quintessential composition or *dot join* (`.`), that joins all pairs of tuples from two expressions whenever they have the same last and first atom (dropping this matching atom). Given a binary relational expression, it is also possible to compute its transitive closure (`^`) or its reflexive-transitive closure (`*`). Three relational constants are predefined, namely **`univ`**, to denote the set of all atoms in the universe, **`none`** to denote the empty set, and **`iden`**, the identity binary relation over **`univ`**. It is also possible to define relations by comprehension with the usual syntax.

Atomic formulas are either cardinality checks with the multiplicity operators **`some`** (the relational expression has at least one tuple), **`lone`** (at most one tuple), **`one`** (exactly one tuple), or **`no`** (the relational expression is empty), or testing if a relational expression is a subset of another (**`in`**), with equality being just an alias to checking inclusion in both directions. For readers not accustomed with this Alloy style of relational logic, it is perhaps surprising that membership check is not part of the atomic formulas. This is due to the fact that scalars (for example, variables introduced by quantifiers) are actually (singleton) sets, which means that

membership is subsumed by inclusion. This also allows scalars to be used in relational expressions, with composition subsuming relation application. For example expression `p.id` (l. 15) determines the identifiers of process `p`. Complex formulas are then created through common Boolean constants (**`true`** and **`false`**) and connectives (**`not`**, **`and`**, **`or`** and **`implies`**), and first-order universal (**`all`**) and existential (**`some`**) quantifications (occasionally we abuse the notation and write multi-variable quantifications).

The specification of the ring topology consists of a conjunction of six sub-formulas over the immutable relations. The first forces `id` to be a subset of the (Cartesian) product between `Process` and `Id` (l. 14). As already noted, the upper-bound of `id` alone does not ensure this, since the actual binding for `Process` is unknown when defining the problem (the same does not apply to `next` that is already exactly bound). The second constraint forces every process to have exactly one identifier, or, in other terms, forces `id` to be a function from processes to identifiers (l. 15). The next constraint ensures that `id` is also injective, by stating that every identifier must be related by `id` to at most one process (l. 16). The next two constraints (ll. 17–18) force `succ` to be a function relating each process to exactly one other process. To ensure that `succ` forms a ring, the last constraint (l. 19) relies on the transitive closure to state that the set of all processes must be reachable through `succ` from every process.

Specifying behaviour (ll. 10–12, 21–26) The remaining of the problem specifies the evolution of the protocol over a given configuration. Note that Pardinus problems do not explicitly specify a state machine. Instead, behaviour is enforced through arbitrary temporal constraints that restrict which traces are acceptable in the system being modelled.

The protocol is uniform (every process performs the same operations) and works correctly if no failures occur (eventually one and at most one leader is elected). The protocol starts with each process ready to send its own identifier to its successor in the ring. When a process receives an identifier, it compares it with its own. If it is higher it propagates; otherwise it discards it. A process that receives back its own identifier is the elected leader. To model this behaviour, a mutable `outbox` binary relation is declared (l. 10) to associate each process with the identifiers it should propagate along the ring. As in [25], where this protocol is used to illustrate the Alloy 5 language following an explicit state idiom, we abstract away the inbox of each process and will merge the event of sending an identifier with that of the respective successor processing the identifier. A mutable `Elected` set is also declared (l. 12) to contain the processes that are elected leaders (hopefully, at most one).

With mutable relations, the constraints of a problem can rely on temporal operators. Relational expressions can be "primed" to retrieve their value in the succeeding state, and formulas are composed using the past and future temporal operators described in Fig. 2.

The dynamics of the protocol is specified with two constraints. The one in l. 21 specifies the initial value of the `outbox` relation (formulas without temporal operators must hold in the first state), stating it should be the same as relation `id`, *i.e.*, each process should start by sending its own identifier to the successor. The formula in ll. 22–23 specifies valid transitions, stating that at each time instant some process `p` should pick and process one of the identifiers in the outbox of its predecessor `succ.p`. The selected identifier `i` is removed from the outbox of the predecessor of `p` (by subtracting the tuple `succ.p`→`i` from `outbox`) and added to the outbox of `p` (by adding the tuple `p`→`i`), but only if it is greater than or equal to its identifier. The expression `^next.(p.id)` denotes the set of all identifiers that are smaller than the identifier of `p`, and by subtracting it from `i` (in the expression that computes the tuple to be added) the desired behaviour is ensured. Notice that, given the way the update on `outbox` is specified (with a constraint that defines the full global value of `outbox` in the succeeding instant), exactly one process will propagate exactly one identifier at each instant.

$$
\begin{array}{ll}
\pi, i \models \mathbf{true} & \\
\pi, i \models \Gamma \mathbf{\ in\ } \Delta & \equiv [\![\Gamma]\!]^i_\pi \subseteq [\![\Delta]\!]^i_\pi \\
\pi, i \models \mathbf{some\ } \Gamma & \equiv \left|[\![\Gamma]\!]^i_\pi\right| \geq 1 \\
\pi, i \models \mathbf{lone\ } \Gamma & \equiv \left|[\![\Gamma]\!]^i_\pi\right| \leq 1 \\
\pi, i \models \mathbf{not\ } \phi & \equiv \pi, i \not\models \phi \\
\pi, i \models \phi \mathbf{\ and\ } \psi & \equiv \pi, i \models \phi \wedge \pi, i \models \psi \\
\pi, i \models \mathbf{all\ } x : \Gamma \mid \phi & \equiv \forall t \in [\![\Gamma]\!]^i_\pi \cdot \pi \oplus \mathbb{N}_0 \mapsto x \mapsto \{t\}, i \models \phi \\
\pi, i \models \mathbf{after\ } \phi & \equiv \pi, i+1 \models \phi \\
\pi, i \models \phi \mathbf{\ until\ } \psi & \equiv \exists i \leq k \cdot \pi, k \models \psi \wedge \forall i \leq j < k \cdot \pi, j \models \phi \\
\pi, i \models \mathbf{before\ } \phi & \equiv 0 < i \wedge \pi, i-1 \models \phi \\
\pi, i \models \phi \mathbf{\ since\ } \psi & \equiv \exists 0 \leq k \leq i \cdot \pi, k \models \psi \wedge \forall k < j \leq i \cdot \pi, j \models \phi
\end{array}
$$

Fig. 4: Semantics of Pardinus formulas

$$
\begin{array}{ll}
[\![r]\!]^i_\pi & = \pi_i(r) \\
[\![x]\!]^i_\pi & = \pi_i(x) \\
[\![\mathbf{univ}]\!]^i_\pi & = \{(a) \mid a \in \mathscr{A}\} \\
[\![\mathbf{none}]\!]^i_\pi & = \{\} \\
[\![\mathbf{iden}]\!]^i_\pi & = \{(a,a) \mid a \in \mathscr{A}\} \\
[\![\sim\Gamma]\!]^i_\pi & = \{(b,a) \mid (a,b) \in [\![\Gamma]\!]^i_\pi\} \\
[\![\hat{}\Gamma]\!]^i_\pi & = \{(a,b) \mid \exists c_1, \ldots, c_n \cdot (a,c_1), (c_1,c_2), \ldots, (c_n,b) \in [\![\Gamma]\!]^i_\pi\} \\
[\![\Gamma + \Delta]\!]^i_\pi & = [\![\Gamma]\!]^i_\pi \cup [\![\Delta]\!]^i_\pi \\
[\![\Gamma \mathbin{\&} \Delta]\!]^i_\pi & = [\![\Gamma]\!]^i_\pi \cap [\![\Delta]\!]^i_\pi \\
[\![\Gamma - \Delta]\!]^i_\pi & = [\![\Gamma]\!]^i_\pi \setminus [\![\Delta]\!]^i_\pi \\
[\![\Gamma \to \Delta]\!]^i_\pi & = \{(a_1, \ldots, a_n, b_1, \ldots, b_m) \mid (a_1, \ldots, a_n) \in [\![\Gamma]\!]^i_\pi \wedge (b_1, \ldots, b_m) \in [\![\Delta]\!]^i_\pi\} \\
[\![\Gamma \mathbin{.} \Delta]\!]^i_\pi & = \{(a_1, \ldots, a_{n-1}, b_2, \ldots, b_m) \mid (a_1, \ldots, a_n) \in [\![\Gamma]\!]^i_\pi \wedge (b_1, \ldots, b_m) \in [\![\Delta]\!]^i_\pi \\
 & \qquad \wedge a_n = b_1\} \\
[\![\{x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi\}]\!]^i_\pi & = \{(a_1, \ldots, a_n) \mid (a_1) \in [\![\Gamma_1]\!]^i_\pi \wedge \ldots \wedge (a_n) \in [\![\Gamma_n]\!]^i_\pi \wedge \\
 & \qquad \pi \oplus \mathbb{N}_0 \mapsto x_1 \mapsto \{(a_1)\} \oplus \ldots \oplus \mathbb{N}_0 \mapsto x_n \mapsto \{(a_n)\}, i \models \phi\} \\
[\![\Gamma\text{'}]\!]^i_\pi & = [\![\Gamma]\!]^{i+1}_\pi
\end{array}
$$

Fig. 5: Semantics of Pardinus relational expressions ($\mathscr{A}$ is the declared universe)

The final constraint (ll. 25–26) defines the set of elected processes by comprehension at each instant, using a combination of future and past linear time operators: a process is considered elected if at some point in the past its identifier reappeared in its outbox.

Semantics A model of a Kodkod problem (or a solution) is a binding from the (free) relations to constants that respects the declared bounds and that satisfies the formula. As usual in model checking, a model of a Pardinus problem is an infinite path, *i.e.*, an infinite sequence of bindings from the declared relations to constants that always respects the declared bounds and satisfies the temporal formula, capturing an execution path of the system.

A path π is a mapping from naturals to bindings, themselves a mapping from relations to tuple sets. For a path π, π_i shall denote its i-th state. A path π is said to respect a set of relation declarations $\mathscr{D}$, denoted by $\pi \models \mathscr{D}$, if for every $r : a\ l\ u \in \mathscr{D}$, $\forall i \geq 0 \cdot l \subseteq \pi_i(r) \subseteq u$. If r is declared as immutable, its value must also not change, *i.e.*, $\forall i \geq 0 \cdot \pi_i(r) = \pi_0(r)$. The satisfaction of a formula ϕ at step i of a path π is denoted by $\pi, i \models \phi$ and defined in Fig. 4. The value of a relational expression Γ at time i of path π is denoted by $[\![\Gamma]\!]^i_\pi$ and defined

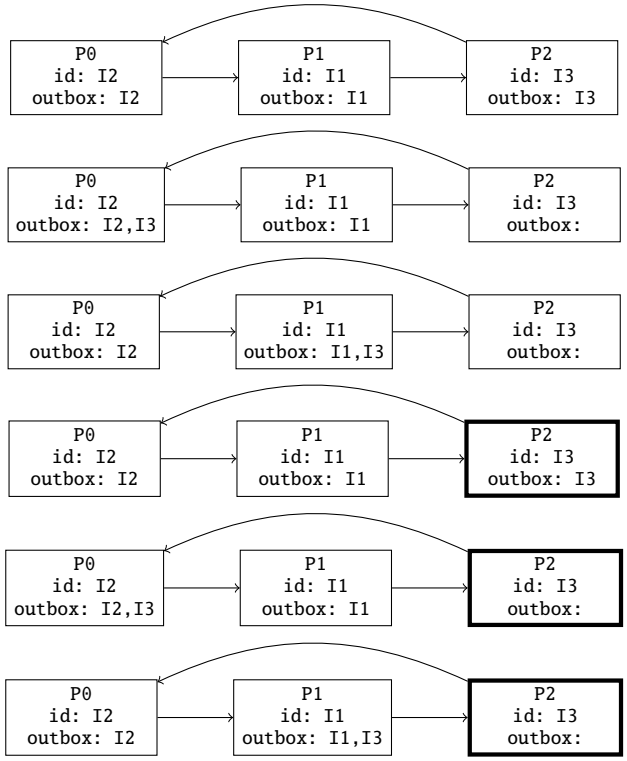


Fig. 6: A run of the leader election protocol

in Fig. 5 under a universe of atoms $\mathscr{A}$. To lighten the presentation, throughout the article we only detail the semantics of a kernel of operators, omitting the semantics of those easily defined out of others. This includes temporal operators, where, for instance, **`eventually`** ϕ can be defined as **`true until`** ϕ, or ϕ **`releases`** ψ as **`not`** `(`**`not`** ϕ **`until not`** ψ `)`. The standard semantics is followed for both first-order connectives and (past and future) linear temporal operators (the semantics for past is strong, meaning that **`before`** is always false at the initial state). For two mappings m_1 and m_2, $m_1 \oplus m_2$ denotes the overriding in m_1 of the elements mapped in m_2, while $x \mapsto y$ denotes the singleton mapping from x to y (we abuse the notation and, for a set A, write $A \mapsto y$ to denote $\bigcup_{x \in A} x \mapsto y$).

A Pardinus problem $\mathscr{A}\ \mathscr{D}\ \phi$ is satisfiable under a path π if the bounds are respected ($\pi \models \mathscr{D}$) and the formula is valid from the initial state ($\pi, 0 \models \phi$). If a problem is satisfiable, Pardinus returns a valid solution, using the procedures defined in the next section.

The example protocol specification is satisfiable and a possible valid solution returned by Pardinus is depicted in Fig. 6. To simplify the presentation, `id` and `outbox` are depicted as attributes of each process. More precisely, for each process `p` we depict in its box the value of `p.id` and `p.outbox`. Processes in the `Elected` set are signalled with a thick border. The figure presents only the first 6 steps of the solution, with the following (omitted) sequence being an infinite repetition of the last 3 steps. This is a minimal run of the protocol for 3 processes, where the highest identifier keeps being propagated around the ring.

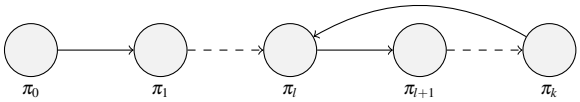


Fig. 7: Bounded witness of an infinite path

To check a particular temporal property, one should add its negation to the problem to try to find a counter-example. If none is found, the property is valid for the given bounds. The key safety property of this protocol is that at most one leader is elected. On first sight, one could specify this property as **`always lone`** `Elected`, stating that the set of elected processes never contains more than one process. However this is too weak, since it allows different processes to be considered elected at different points in time. A possible correct specification of this property is as follows, stating that if at any point in time a process `p` is considered elected, then afterwards the `Elected` set is restricted to contain at most `p`, *i.e.*,

```
always all p : Elected | always Elected in p .
```

Adding the negation of this formula to our problem reveals no counter-example.

To be useful, the protocol should also ensure that some leader is elected. This liveness property can be specified as

```
eventually some Elected .
```

Surprisingly, adding the negation of this formula to our problem reveals a counter-example for a configuration where the ring contains only one process: in this case, the process keeps sending its own identifier to itself, and thus is never added to `Elected` because the outbox always remains in a state that is indistinguishable from the initial one. This is not a problem with the protocol, but just a consequence of abstracting away the inbox in our specification and defining the `Elected` set over the outbox instead. Without adding the inbox, the alternative is to relax the definition of `Elected` to cover the special case of rings with just one process, for example adding the disjunct `p.succ = p` to the comprehension formula.

3 Temporal relational model finding

The goal of the model finding procedure of Pardinus is to find a valid execution path of a problem, according to the presented semantics. Given the bounded state space, the logic of Pardinus enjoys the Ultimately Periodic Model Property (UPMP) of Linear Temporal Logic (LTL) [18]: if a formula is satisfied by an infinite path, then it is also satisfied by a path that will eventually repeat itself. Such a path can thus be represented by a bounded witness with the shape depicted in Fig. 7, a finite sequence with $k+1$ states that loops back to position $l \leq k$, known as a (k,l)-*loop* (or more generally a k-loop if the looping state is not relevant). Thanks to the UPMP, Pardinus can limit the search to such paths without loss of generality. Now, like most model checkers for temporal logic, Pardinus can be run in *complete* mode – where it searches for any valid k-loop – or in *bounded* mode – where it searches for a valid k-loop where $k \leq n$ for a user-provided upper-bound n on the size of the witness.

The architecture of the complete model finding engine of Pardinus is depicted in Fig. 8, and essentially relies on a translation from a Pardinus problem to SMV, to be subsequently analysed by compatible model checkers such as NuSMV [9] or nuXmv [8]. This engine is abstracted by procedure `solveTRL` : *`Pardinus`* $\rightarrow$ *`path`*, that solves a temporal relational

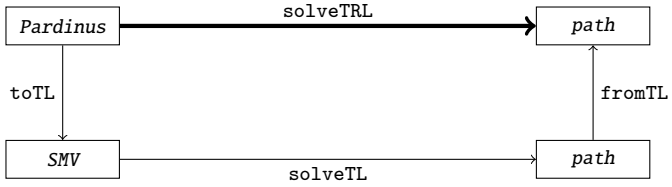


Fig. 8: Model finding with complete model checking

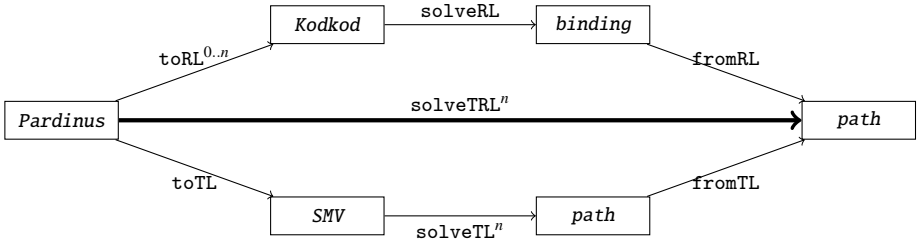


Fig. 9: Model finding with bounded model checking

logic Pardinus problem and obtains a valid path (or $\perp$ if the problem is unsatisfiable), and that is implemented using several auxilary procedures according to the following steps:

1. The problem is translated to SMV by eliminating the relational logic part and reducing it to a normal (propositional) temporal logic problem (procedure `toTL`), essentially by expanding relations to matrices of Boolean variables and the relational operators to operations on matrices. This process is explained in Section 3.2.
2. Any valid execution path found by SMV model checkers (here abstracted by procedure `solveTL`) must be converted back to a Pardinus path (procedure `fromTL`). The former is a sequence of Boolean bindings while the latter is a sequence of relational bindings. This translation is straightforward, reverting the process from the previous step and assembles the relations from the respective Boolean variables, so its description will be omitted.

The architecture of the bounded model finding engine of Pardinus, abstracted by procedure $\texttt{solveTRL}^n : \mathit{Pardinus} \rightarrow \mathit{path}$, parametrized by the upper-bound n on the size of the witness, is depicted in Fig. 9. SMV model checkers natively support bounded model checking (here abstracted by $\texttt{solveTL}^n$), so this procedure can actually be implemented using the translation described above for complete model finding (the sequence of steps in the lower path of Fig. 9). Nonetheless, an alternative mechanism is also provided (the steps in the upper path), that relies on a translation from Pardinus to standard Kodkod (to be subsequently translated to off-the-shelf SAT solvers), that works as follows:

1. Translate the problem to Kodkod by eliminating the temporal logic part and reducing it to a normal (static) relational logic problem (procedure $\texttt{toRL}^k$), essentially by introducing an explicit set of state identifiers and adding those as an extra dimension to mutable relations. This translation is parametrised by the exact size k of bounded witnesses (in number of steps until looping back), and is run iteratively from 0 up-to the maximum size n, stopping when a valid path is found. This process is detailed in Section 3.1.
2. Any valid binding found by the Kodkod model finder (here abstracted by procedure `solveRL`) must be converted back to a Pardinus path (procedure `fromRL`). Again, this

$\{(\mathtt{S}_i^j) \mid j \leftarrow 0..\lfloor\phi\rfloor, i \leftarrow 0..k\}$

`state`	$:1\ \{(\mathtt{S}_i^0) \mid i \leftarrow 0..k\} \cup \{(\mathtt{S}_k^j) \mid j \leftarrow 0..\lfloor\phi\rfloor\}\ \{(\mathtt{S}_i^j) \mid j \leftarrow 0..\lfloor\phi\rfloor, i \leftarrow 0..k\}$
`first`	$:1\ \{(\mathtt{S}_0^0)\}\ \{(\mathtt{S}_0^0)\}$
`last`	$:1\ \{(\mathtt{S}_k^{\lfloor\phi\rfloor})\}\ \{(\mathtt{S}_k^{\lfloor\phi\rfloor})\}$
`loop`	$:1\ \{\}\ \{(\mathtt{S}_i^{\lfloor\phi\rfloor}) \mid i \leftarrow 0..k\}$
`succ`	$:2\ \{(\mathtt{S}_i^0, \mathtt{S}_{i+1}^0) \mid i \leftarrow 0..k-1\}$ $\{(\mathtt{S}_i^j, \mathtt{S}_{i+1}^j) \mid j \leftarrow 0..\lfloor\phi\rfloor, i \leftarrow 0..k-1\} \cup \{(\mathtt{S}_k^j, \mathtt{S}_i^{j+1}) \mid j \leftarrow 0..\lfloor\phi\rfloor - 1, i \leftarrow 0..k\}$
`same`	$:2\ \{(\mathtt{S}_i^j, \mathtt{S}_i^0) \mid j \leftarrow 0..\lfloor\phi\rfloor, i \leftarrow 0..k\}\ \{(\mathtt{S}_i^j, \mathtt{S}_i^0) \mid j \leftarrow 0..\lfloor\phi\rfloor, i \leftarrow 0..k\}$

```
one loop and loop in state
succ in state → state
all s : state − last | one s.succ
no last.succ
state = first.*succ
all s : state | s.next.same = s.same.next
```

Fig. 10: New relations and constraints generated by procedure $\mathtt{toRL}^k$

straightforward translation just reverts the process from the previous step by projecting the static relations on the state identifiers, so its description will not be presented.

3.1 Temporal model finding with Kodkod

The translation from Pardinus to Kodkod explicitly models states and finite path prefixes, possibly with back loops, and allows the value of mutable relations to vary along the different states of a path by extending them with a state dimension. The explicitly modelling of state is similar to the idioms commonly used in Alloy for modelling behaviour before Alloy 6 was introduced, but these often do not consider the back loops (which are essential to enable the sound verification of liveness properties), In previous work we have presented this encoding in Alloy 4 for future-only relational temporal logic [16]. It is is based on the well-known procedure for bounded model checking with SAT [3], but at the higher-level of relational logic, taking advantage of transitive closure to encode the semantics of temporal logic operators. However, it cannot be reused directly in Pardinus because it is not sound when formulas contain past operators. As shown in [2], while the truth value of a future-only temporal formula in a state inside a loop is the same in all iterations of the loop, the same does not hold in presence of past operators. This is clear from the fact that, while the (infinite) future of a state inside a loop is always the same, the (finite) past of that same state differs in different iterations of the loop. Fortunately, a key result of [2] is that after a number of iterations equal to the maximum nesting depth of past operators in a formula its truth value can no longer be distinguished. Therefore, to perform sound bounded model checking it suffices to unroll the loop that many times. In the following presentation we assume that the past depth of a Pardinus formula ϕ is given by $\lfloor\phi\rfloor$.

Given a Pardinus problem with formula ϕ, procedure $\mathtt{toRL}^k$ starts by introducing new relations and constraints to explicitly specify bounded representations of infinite paths with exactly $k+1$ states and $\lfloor\phi\rfloor$ unrollings of the loop (from now on denoted simply as paths).

Recall that $\texttt{toRL}^k$ will be invoked iteratively by $\texttt{solveTRL}^n$ starting with $k=0$ up to $k=n$ until a valid solution is found. The new atoms, relations and constraints that are declared by $\texttt{toRL}^k$ are presented in Fig. 10, where some auxiliary notation is used to define bounds using set comprehension and union. First, a set `state` of explicit state identifiers is declared to denote the different instants in a possible execution path (including identifiers for the states in the unrollings). In particular, an atom $\texttt{S}_i^j$ identifies the state at position i of loop iteration j. Depending on the size of the looping segment, the number of identifiers can vary, hence the different lower- and upper-bounds: at minimum, solutions will have $k+1$ states for the initial path prefix plus the last state in each loop unrolling; and at most, when the path loops back to the first state, solutions will require $(k+1)\times(\lfloor\phi\rfloor+1)$ identifiers. Two singleton sets, `first` and `last`, are declared to denote the first and last state (before looping back) of the path, respectively. A singleton set `loop` will choose the state in the last unrolling to which the path will loop back to. An extra constraint ensures this set will contain exactly one state. The binary relation `succ` captures the total order between the state identifiers that will be part of the path. As the first $k+1$ states are always present, the lower-bound forces them to be ordered sequentially. The upper-bound ensures that the states in each of the unrollings, if present in a path, are also ordered sequentially. Also, the successor of the last state of each unrolling (or the initial prefix) can be any of the states in the next unrolling – the exact state will depend on the position of the back loop. Extra constraints ensure that `succ` is indeed a total order between `first` and `last`. Finally, `same` is a binary relation that captures the equivalence relation between state identifiers, in particular by mapping each state of the path to the equivalent state in the initial prefix. The final constraint takes advantage of this relation to ensure that each iteration of the loop is exactly the same (relation `next` is an auxiliary relation defined as `succ + last` $\rightarrow$ `loop`).

The declaration of each mutable relation **`var`** r`:`$a\ l\ u$ in the problem is then changed by $\texttt{toRL}^k$: an extra `state` dimension is added to the correspondent (immutable) Kodkod relation, representing the value of the relation in each state of a path. For efficiency reasons, only the values for the states of the initial prefix are recorded: for the states in the succeeding unrollings the value of the relation can be accessed via the `same` relation, that returns the equivalent state in the initial prefix. Thus, a mutable declaration **`var`** r`:`$a\ l\ u$ is expanded as

$$r\texttt{:}(a+1)\ (l\times\{(\texttt{S}_i^0)\mid i\leftarrow 0..k\})\ (u\times\{(\texttt{S}_i^0)\mid i\leftarrow 0..k\})\ ,$$

where $\times$ is auxiliary notation denoting the Cartesian product of tuple sets.

Finally, the temporal operators in the problem formula ϕ are replaced by their trace semantics, defined over relations `next` for future operators and `succ` for past operators, since after unrolling, past operators should not consider the loop back. Like the semantics in Fig. 4, this *temporal expansion* of ϕ, denoted $\langle\phi\rangle_s$, is parametrized by the state s in the path where the truth value of the formula should be checked, and its definition is presented in Fig. 11. In this definition we assume that s_0 and s_1 are always fresh identifiers, and `upto[`s`,`s_0`]` and `downto[`s`,`s_0`]` are auxiliary definitions that compute the set of states between s and s_0 (not included) when advancing forwards via `next` and backwards via the converse of `succ`, respectively. Notice also that the **`before`** formulas must test whether the previous state exists. The temporal expansion of relational expressions is presented in Fig. 12, where the most interesting case corresponds to the expansion of mutable relation identifiers: as explained above, the value of a mutable relation r at state s is determined by projecting it over state s`.same`, the state in the initial prefix equivalent to s. Although the translation of the (omitted) derived temporal operators could be defined by translation to the kernel operators, in practice specialized direct translations are implemented to promote efficiency. Given these procedures, the problem formula ϕ is expanded to $\langle\phi\rangle_{\texttt{first}}$.

$$
\begin{array}{ll}
\langle \mathbf{true} \rangle_s & = \mathbf{true} \\
\langle \Gamma \ \mathbf{in} \ \Delta \rangle_s & = \langle \Gamma \rangle_s \ \mathbf{in} \ \langle \Delta \rangle_s \\
\langle \mathbf{some} \ \Gamma \rangle_s & = \mathbf{some} \ \langle \Gamma \rangle_s \\
\langle \mathbf{lone} \ \Gamma \rangle_s & = \mathbf{lone} \ \langle \Gamma \rangle_s \\
\langle \mathbf{not} \ \phi \rangle_s & = \mathbf{not} \ \langle \phi \rangle_s \\
\langle \phi \ \mathbf{and} \ \psi \rangle_s & = \langle \phi \rangle_s \ \mathbf{and} \ \langle \psi \rangle_s \\
\langle \mathbf{all} \ x : \Gamma \mid \phi \rangle_s & = \mathbf{all} \ x : \langle \Gamma \rangle_s \mid \langle \phi \rangle_s \\
\langle \mathbf{after} \ \phi \rangle_s & = \langle \phi \rangle_{s.\mathtt{next}} \\
\langle \phi \ \mathbf{until} \ \psi \rangle_s & = \mathbf{some} \ s_0 : s.\mathtt{*next} \mid \langle \psi \rangle_{s_0} \ \mathbf{and} \ \mathbf{all} \ s_1 : \mathtt{upto}[s, s_0] \mid \langle \phi \rangle_{s_1} \\
\langle \mathbf{before} \ \phi \rangle_s & = \mathbf{some} \ s_0 : \mathtt{succ}.s \mid \langle \phi \rangle_{s_0} \\
\langle \phi \ \mathbf{since} \ \psi \rangle_s & = \mathbf{some} \ s_0 : \mathtt{*succ}.s \mid \langle \psi \rangle_{s_0} \ \mathbf{and} \ \mathbf{all} \ s_1 : \mathtt{downto}[s, s_0] \mid \langle \phi \rangle_{s_1}
\end{array}
$$

Fig. 11: Temporal expansion of formulas (part of `toRL`)

$$
\begin{array}{ll}
\langle r \rangle_s & = \begin{cases} r.(s.\mathtt{same}) & \text{if } r \text{ mutable} \\ r & \text{otherwise} \end{cases} \\
\langle x \rangle_s & = x \\
\langle \mathbf{univ} \rangle_s & = \mathbf{univ} \\
\langle \mathbf{none} \rangle_s & = \mathbf{none} \\
\langle \mathbf{iden} \rangle_s & = \mathbf{iden} \\
\langle \sim \Gamma \rangle_s & = \sim \langle \Gamma \rangle_s \\
\langle \hat{} \Gamma \rangle_s & = \hat{} \langle \Gamma \rangle_s \\
\langle \Gamma + \Delta \rangle_s & = \langle \Gamma \rangle_s + \langle \Delta \rangle_s \\
\langle \Gamma \ \& \ \Delta \rangle_s & = \langle \Gamma \rangle_s \ \& \ \langle \Delta \rangle_s \\
\langle \Gamma - \Delta \rangle_s & = \langle \Gamma \rangle_s - \langle \Delta \rangle_s \\
\langle \Gamma \to \Delta \rangle_s & = \langle \Gamma \rangle_s \to \langle \Delta \rangle_s \\
\langle \Gamma \ . \ \Delta \rangle_s & = \langle \Gamma \rangle_s \ . \ \langle \Delta \rangle_s \\
\langle \{x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi\} \rangle_s & = \{x_1 : \langle \Gamma_1 \rangle_s, \ldots, x_n : \langle \Gamma_n \rangle_s \mid \langle \phi \rangle_s\} \\
\langle \Gamma' \rangle_s & = \langle \Gamma \rangle_{s.\mathtt{next}}
\end{array}
$$

Fig. 12: Temporal expansion of relational expressions (part of `toRL`)

3.2 Relational model finding with SMV

An SMV model (the format accepted by the NuSMV and nuXmv symbolic temporal model checkers) consists of the definition of a state space, a transition system (initial conditions and a transition relation) and a LTL formula (possibly including past operators) which is expected to be satisfied by all traces of the transition system. The translation from Pardinus to SMV basically consists in expanding the first-order and relational parts of formulas and expressions into disjunctions and conjunctions, resulting in a *propositional* LTL formula supported by SMV model checkers. In this scenario, the automaton considered by model checking would be the universal automaton, allowing all possible behaviours; in practice, the resulting formula is analysed to refine the state machine under analysis.

The state space is described by declarating Boolean variables corresponding to atomic propositions of the LTL formula produced by the translation. A Boolean variable r_t stands for the presence of tuple t in relation r. Such a variable is declared in the `VAR` (resp. `FROZENVAR`) section of the SMV format if the relation r is mutable (resp. immutable).

$$
\begin{array}{ll}
[\![r]\!]_\sigma & = \mathsf{up}(r) \\
[\![x]\!]_\sigma & = \sigma(x) \\
[\![\mathtt{univ}]\!]_\sigma & = \{(a) \mid a \in \mathscr{A}\} \\
[\![\mathtt{none}]\!]_\sigma & = \{\} \\
[\![\mathtt{iden}]\!]_\sigma & = \{(a,a) \mid a \in \mathscr{A}\} \\
[\![\sim\Gamma]\!]_\sigma & = \{(b,a) \mid (a,b) \in [\![\Gamma]\!]_\sigma\} \\
[\![\hat{}\Gamma]\!]_\sigma & = \{(a,b) \mid \exists c_1,\ldots,c_n \cdot (a,c_1),(c_1,c_2),\ldots,(c_n,b) \in [\![\Gamma]\!]_\sigma\} \\
[\![\Gamma + \Delta]\!]_\sigma & = [\![\Gamma]\!]_\sigma \cup [\![\Delta]\!]_\sigma \\
[\![\Gamma \,\&\, \Delta]\!]_\sigma & = [\![\Gamma]\!]_\sigma \cap [\![\Delta]\!]_\sigma \\
[\![\Gamma - \Delta]\!]_\sigma & = [\![\Gamma]\!]_\sigma \setminus [\![\Delta]\!]_\sigma \\
[\![\Gamma \to \Delta]\!]_\sigma & = \{(a_1,\ldots,a_n,b_1,\ldots,b_m) \mid (a_1,\ldots,a_n) \in [\![\Gamma]\!]_\sigma \wedge (b_1,\ldots,b_m) \in [\![\Delta]\!]_\sigma\} \\
[\![\Gamma \,.\, \Delta]\!]_\sigma & = \{(a_1,\ldots,a_{n-1},b_2,\ldots,b_m) \mid (a_1,\ldots,a_n) \in [\![\Gamma]\!]_\sigma \wedge (b_1,\ldots,b_m) \in [\![\Delta]\!]_\sigma \wedge a_n = b_1\} \\
[\![\Gamma']\!]_\sigma & = [\![\Gamma]\!]_\sigma \\
[\![\{x_1 : \Gamma_1,\ldots,x_n : \Gamma_n \mid \phi\}]\!]_\sigma & = [\![\Gamma_1 \to \ldots \to \Gamma_n]\!]_\sigma
\end{array}
$$

Fig. 13: Computing the constant upper-bound of bounding expressions

The `LTLSPEC` section of the SMV format contains the formula that is expected to hold. The negation of the propositional LTL expansion of a formula ϕ of a Pardinus problem containing the system definition and the property to check, denoted by $[\phi]$ and presented shortly, could be directly assigned to the `LTLSPEC` section. However, according to our experience, NuSMV and nuXmv perform better if eligible sub-formulas are allocated to specific SMV sections that restrict the transition system, instead of checking the whole LTL formula over the transition system that allows any sequence of states. Formula ϕ often has the shape of a top-level conjunction $\phi_1 \wedge \ldots \wedge \phi_n$ and so does its expansion $[\phi]$. According to the syntactic shape of each sub-formula $[\phi_i]$, the appropriate SMV section is used, namely:

- A conjunct $[\phi_i]$ not including temporal operator is allocated to the `INIT` section, which characterizes initial states.
- A conjunct $[\phi_i]$ that only consists of present- and next-time sub-formulas is allocated to the `TRANS` section, which characterizes the transition relation.[4]
- For a conjunct $[\phi_i]$ of the form $\mathsf{G}\,\psi_i$, where ψ_i does not include temporal operators, ψ_i is allocated to the `INVARIANT` section for formulas assumed to hold in any state.
- The negation of the conjunction of the sub-formulas $[\phi_i]$ that do not fit in any other section is allocated to the `LTLSPEC` section, the arbitrary temporal formula characterizing properties expected to hold in executions satisfying the constraints in the sections above.

The expansion of a constraint ϕ requires a substitution σ of variables bound in a given context (*e.g.*, under quantifiers) to atoms. The update of a substitution, introducing a mapping from x to t, is written $\sigma[x \mapsto t]$. The translation of a constraint ϕ also requires the computation of the lower- and upper-bounds of relational expressions, denoted by $[\![\cdot]\!]_\sigma$ and $[\![\cdot]\!]_\sigma$, respectively. The latter is defined in Fig. 13. The upper-bound of an expression is computed using the upper-bounds of its sub-expressions (except for the difference operator, where the lower-bound of the right sub-expression is used, since a larger right sub-expression entails a smaller difference expression). When a relation r is referenced, its upper-bound is the one declared in the problem (here $\mathsf{low}(r)$ and $\mathsf{up}(r)$ select declaration $r{:}a\ l\ u$ from the problem definition and return l or u, respectively). The (omitted) definition

[4] The `ASSIGN` section is not used as its restricted syntax makes it a complex target for generation from Pardinus.

$$
\begin{array}{ll}
[r]_\sigma(t) & = \begin{cases} \top & \text{if } t \in \llbracket r \rrbracket_\sigma \\ r_t & \text{if } t \in \lceil r \rceil_\sigma \setminus \llbracket r \rrbracket_\sigma \\ \bot & \text{otherwise} \end{cases} \\
[x]_\sigma(t) & = \begin{cases} \top & \text{if } t = \sigma(x) \\ \bot & \text{otherwise} \end{cases} \\
[\mathbf{univ}]_\sigma(t) & = \top \\
[\mathbf{none}]_\sigma(t) & = \bot \\
[\mathbf{iden}]_\sigma((a,b)) & = \begin{cases} \top & \text{if } a = b \\ \bot & \text{otherwise} \end{cases} \\
[\sim\Gamma]_\sigma((a,b)) & = [\Gamma]_\sigma((b,a)) \\
[\hat{}\Gamma]_\sigma(t) & = [\Gamma + \ldots + \underbrace{\Gamma . \ \ldots \ . \Gamma}_{\mathsf{maxLen}(\lceil\Gamma\rceil_\sigma)}]_\sigma(t) \\
[\Gamma + \Delta]_\sigma(t) & = [\Gamma]_\sigma(t) \vee [\Delta]_\sigma(t) \\
[\Gamma \,\&\, \Delta]_\sigma(t) & = [\Gamma]_\sigma(t) \wedge [\Delta]_\sigma(t) \\
[\Gamma - \Delta]_\sigma(t) & = [\Gamma]_\sigma(t) \wedge \neg[\Delta]_\sigma(t) \\
[\Gamma \to \Delta]_\sigma((a_1,\ldots,a_n)) & = [\Gamma]_\sigma((a_1,\ldots,a_{|\Gamma|})) \wedge [\Delta]_\sigma((a_{|\Gamma|+1},\ldots,a_n)) \\
[\Gamma \,.\, \Delta]_\sigma(t) & = \bigvee_{\substack{(a_1,\ldots,a_n)\in\lceil\Gamma\rceil_\sigma \\ (b_1,\ldots,b_m)\in\lceil\Delta\rceil_\sigma \\ a_n=b_1 \\ t=(a_1,\ldots,a_{n-1},b_2,\ldots,b_m)}} ([\Gamma]_\sigma((a_1,\ldots,a_n)) \wedge [\Delta]_\sigma((b_1,\ldots,b_m))) \\
[\{x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi\}]_\sigma((a_1,\ldots,a_n)) & = [\Gamma_1]_\sigma((a_1)) \wedge \ldots \wedge [\Gamma_n]_\sigma((a_n)) \wedge [\phi]_{\sigma[x_1\mapsto(a_1)]\ldots[x_n\mapsto(a_n)]} \\
[\Gamma']_\sigma(t) & = \mathsf{X}[\Gamma]_\sigma(t)
\end{array}
$$

Fig. 14: Characteristic-function expansion of relational expressions (part of `toTL`)

of function $\llbracket\cdot\rrbracket_\sigma$ is dual to $\lceil\cdot\rceil_\sigma$, except for comprehension: since ϕ is an arbitrary formula, the lower-bound of a comprehension expression is conservatively defined as the empty set, while the upper-bound can be computed from the upper-bound of the quantified variables.

Each relational expression Γ is translated into its *characteristic function* $[\Gamma]_\sigma(\cdot)$, which determines whether a tuple belongs to an expression Γ under a context σ, and is presented in Fig. 14[5]. One should not confuse $\lceil\Gamma\rceil_\sigma$ and $[\Gamma]_\sigma(\cdot)$: the former represents the upper-bound for Γ calculated at compile time, *independent from time*, while the latter is a predicate asserting that its argument belongs to (the denotation of) Γ in a particular state. The translation of a relation identifier produces an atomic formula expressing whether the given tuple is in the given relation (if the tuple is not in the upper-bound of the relation, the translation directly returns $\bot$). The other rules are mostly straightforward. For instance, a tuple t belongs to the Cartesian product of Γ and Δ if its left-hand-side sub-tuple (of arity $|\Gamma|$) belongs to Γ while its right-hand-side sub-tuple (of arity $|\Delta|$ and starting at index $|\Gamma|+1$) belongs to Δ. For the join of expressions, a tuple t is in $\Gamma.\Delta$ if there are tuples in Γ and in Δ (of adequate types) that coincide on the appropriate columns and such that t is their concatenation (without the said columns). Since relations are finite, the transitive closure of a relational expression is necessarily finite as well. Hence, it can be replaced by the iterated union of powers of the said expression (where the join operation is the product), up to a certain exponent. This exponent is equal to the maximum number of non-empty joins that can be done between pairs in the upper-bound of the expression. That is, there is a sequence of pairs $t_1, t_2, \ldots, t_{\mathsf{maxLen}}$

[5] Remark that we use mathematical symbols rather than concrete SMV syntax for better readability. However, future (X, G, F, U and R) and past (Y, H, O, S and T) temporal operators follow the standard textual notation also used in SMV.

$$
\begin{array}{ll}
[\mathbf{true}]_\sigma & = \top \\
[\Gamma\ \mathbf{in}\ \Delta]_\sigma & = \bigwedge_{t \in \lceil\lceil\Gamma\rceil\rceil_\sigma} ([\Gamma]_\sigma(t) \Rightarrow [\Delta]_\sigma(t)) \\
[\mathbf{some}\ \Gamma]_\sigma & = \mathtt{count}\{[\Gamma]_\sigma(t) | t \in \lceil\lceil\Gamma\rceil\rceil_\sigma\} \geq 1 \\
 & \\
[\mathbf{lone}\ \Gamma]_\sigma & = \mathtt{count}\{[\Gamma]_\sigma(t) | t \in \lceil\lceil\Gamma\rceil\rceil_\sigma\} \leq 1 \\
[\mathbf{not}\ \phi]_\sigma & = \neg[\phi]_\sigma \\
[\phi\ \mathbf{and}\ \psi]_\sigma & = [\phi]_\sigma \wedge [\psi]_\sigma \\
[\mathbf{all}\ x : \Gamma \mid \phi]_\sigma & = \bigwedge_{t \in \lceil\lceil\Gamma\rceil\rceil_\sigma} \left([\Gamma]_\sigma(t) \Rightarrow [\phi]_{\sigma[x \mapsto t]}\right) \\
[\mathbf{after}\ \phi]_\sigma & = \mathsf{X}[\phi]_\sigma \\
[\phi\ \mathbf{until}\ \psi]_\sigma & = [\phi]_\sigma\, \mathsf{U}\, [\psi]_\sigma \\
[\mathbf{before}\ \phi]_\sigma & = \mathsf{Y}[\phi]_\sigma \\
[\phi\ \mathbf{since}\ \psi]_\sigma & = [\phi]_\sigma\, \mathsf{S}\, [\psi]_\sigma
\end{array}
$$

Fig. 15: Relational expansion of formulas (part of `toTL`)

such that $t_1 \mathbin{.} t_2 \mathbin{.} \ldots \mathbin{.} t_{\mathsf{maxLen}}$ is not empty and there is no such sequence of greater length. We do not detail the exact computation here, abstracted by maxLen, but it essentially depends on atoms that are both in the domain and the codomain of the expression, as these are the only atoms that may appear both as the source and target of edges in the path and thus contribute to lengthen it. In the worst case, it is the size of the universe.

The translation of a constraint ϕ for a context σ is denoted by $[\phi]_\sigma$ and presented in Fig. 15. The initial $[\phi]$ denotes the expansion of ϕ with an empty substitution (if ϕ has no free variables). Γ **in** Δ is translated to a conjunction (of implications) over the set $\lceil\lceil\Gamma\rceil\rceil_\sigma$ of possible values for elements of expression Γ. Each implication then says that each such element that is in Γ is also in Δ. First-order quantifiers are translated in a similar way, with the domain of quantification for variable x translated into a membership precondition and the ϕ sub-formula evaluated in a substitution where x is mapped to t. Propositional and temporal connectives (including past ones) are translated in a straightforward way, as propositional LTL with past is natively implemented in SMV. The translation for multiplicity formulas makes use of the operator `count`, which is not an LTL operator strictly speaking but is an SMV operator. It designates the number of true formulas within a set.

The translation described above would produce uselessly-large formulas. Therefore, the translation exploits static knowledge to reduce the size of generated formulas. In addition to basic compile-time short-circuiting simplifications, lower-bounds of relational expressions are leveraged to produce smaller formulas. For instance, instead of bluntly expanding a universal quantification into a conjunction over the upper-bound of the quantification range, we distinguish between the lower-bound – standing for tuples that *must* be in (the denotation of) the range – and the tuples that belong to the upper-bound but not to the lower one – the tuples that *may* be in the range:

$$
[\mathbf{all}\ x : \Gamma \mid \phi]_\sigma = \bigwedge_{t \in \lfloor\lfloor\Gamma\rfloor\rfloor_\sigma} \left([\Gamma]_\sigma(t) \Rightarrow [\phi]_{\sigma[x \mapsto t]}\right) \wedge \bigwedge_{t \in \lceil\lceil\Gamma\rceil\rceil_\sigma \setminus \lfloor\lfloor\Gamma\rfloor\rfloor_\sigma} \left([\Gamma]_\sigma(t) \Rightarrow [\phi]_{\sigma[x \mapsto t]}\right) \quad ,
$$

which simplifies to:

$$
[\mathbf{all}\ x : \Gamma \mid \phi]_\sigma = \bigwedge_{t \in \lfloor\lfloor\Gamma\rfloor\rfloor_\sigma} [\phi]_{\sigma[x \mapsto t]} \wedge \bigwedge_{t \in \lceil\lceil\Gamma\rceil\rceil_\sigma \setminus \lfloor\lfloor\Gamma\rfloor\rfloor_\sigma} \left([\Gamma]_\sigma(t) \Rightarrow [\phi]_{\sigma[x \mapsto t]}\right) \quad .
$$

Another improvement relates to the transitive closure operator. When computing the transitive closure of a relational expression Γ, we do not need to compute any power of Γ until the length of the longest possible path (maxLen) is reached. By applying the so-called *iterative square* procedure, we can reach a path of length 2^n with n iterations of a simple operation, which consists in performing one union and one join from the expression obtained in the previous iteration. This procedure allows us to have a significant gain in terms of the size of the generated LTL formula (in particular in the number of joins), and is defined as

$$[\hat{}\Gamma]_\sigma(t) = [\widehat{\Gamma}_{\mathsf{maxLen}(\lceil\Gamma\rceil_\sigma)}]_\sigma(t) \quad \text{where} \begin{cases} \widehat{\Gamma}_0 = \mathbf{none} \ \to \ \mathbf{none} \\ \widehat{\Gamma}_1 = \Gamma \\ \widehat{\Gamma}_n = \widehat{\Gamma}_{\lceil n/2\rceil} + \widehat{\Gamma}_{\lceil n/2\rceil} \cdot \widehat{\Gamma}_{\lceil n/2\rceil} \end{cases} .$$

As explained at the beginning of Section 3, the translation described above is the same for complete and bounded model checking as NuSMV and nuXmv both feature algorithms dedicated to the two approaches. As of this writing, for bounded model checking, both tools rely on the same incremental algorithm (`check_ltlspec_bmc_inc`). For complete model checking, NuSMV relies on a reduction from LTL model checking to CTL model checking (`check_ltlspec`) [9] and nuXmv on a *k*-liveness algorithm (`check_ltlspec_ic3`) [4].

4 Scenario exploration

As mentioned in Section 1, a key feature of Kodkod is solution iteration: after obtaining a valid binding, the solver can be instructed to find for a different one, and the process be repeated until all possible solutions are exhausted. By incorporating a symmetry breaking mechanism, Kodkod tries to rule out all bindings that are equivalent to a previously generated one, modulo a permutation of atoms, resulting in solutions that are in general truly different. This is extremely useful, for example, to allow a user to explore alternative scenarios of a design (or different counter-examples) or as a backend to generate test cases from specifications. This section explains how a similar feature was implemented in Pardinus.

4.1 Kodkod solution iteration

Pardinus' solution iteration procedure builds on that of Kodkod, which is efficiently implemented at the SAT solving level. This section briefly introduces this Kodkod procedure before presenting its extension to the Pardinus context in the succeeding sections.

Previously the solving procedure of Kodkod was abstracted as a function `solveRL` : *Kodkod* $\to$ *binding*. When iterating, the Kodkod problem is actually also updated in the process so that in the next invocation a different binding is returned. In practice a problem is updated in-place, but to ease the presentation we abstract this procedure as a pure function

$$\texttt{solveRL} : \mathit{Kodkod} \to \mathit{Kodkod} \times \mathit{binding} \ ,$$

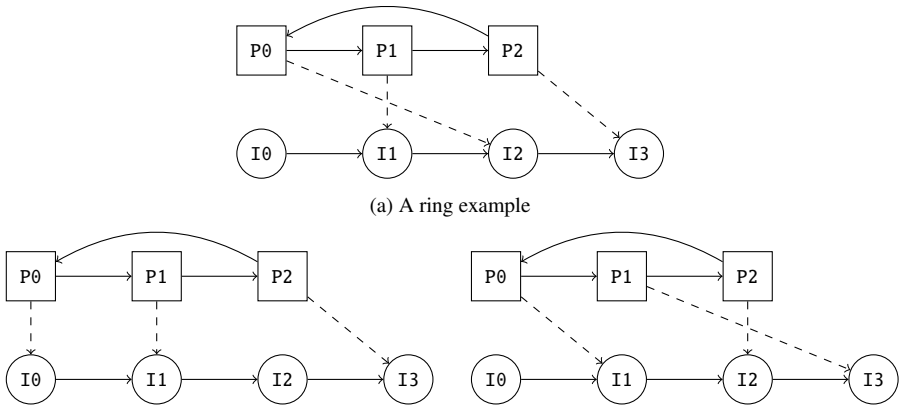
that, besides a binding solution to the current problem, also returns a novel problem where that binding is excluded from the search space. This can be achieved by mapping a binding back to a formula that exactly characterizes it (*i.e.*, that holds only for that binding) and force its negation in the constraint ϕ of the problem.

```
{I0,I1,I2,I3,P0,P1,P2,P3}

Id      :1 {(I0),(I1),(I2),(I3)} {(I0),(I1),(I2),(I3)}
next    :2 {(I0,I1),(I1,I2),(I2,I3)} {(I0,I1),(I1,I2),(I2,I3)}
Process :1 {} {(P0),(P1),(P2),(P3)}
id      :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),...,
               (P3,I0),(P3,I1),(P3,I2),(P3,I3)}
succ    :2 {} {(P0,P0),(P0,P1),(P0,P2),(P0,P3),...,
               (P3,P0),(P3,P1),(P3,P2),(P3,P3)}

id in Process → Id and
all p : Process | one p.id and
all i : Id | lone id.i and
succ in Process → Process and
all p : Process | one p.succ and
all p : Process | Process in p.^succ and
```

Fig. 16: Configuration problem of the leader election protocol



(a) A ring example

(b) A ring non-symmetric to the one in Fig. 17a

(c) A ring symmetric to the one in Fig. 17a

Fig. 17: Alternative ring examples for immutable portion of the problem in Fig. 3

Besides that concrete instance, other instances considered isomorphic (equivalent) should also be removed from the search space. Formally, since atoms are uninterpreted, a permutation $P : \mathscr{A} \to \mathscr{A}$ is a *symmetry* of a Kodkod problem $\mathscr{A}\ \mathscr{D}\ \phi$ if for all bindings s, $s \models \mathscr{D}$ and $s \models \phi$ iff $P(s) \models \mathscr{D}$ and $P(s) \models \phi$ under $\mathscr{A}$, where $P(s)$ denotes the application of P to every atom in every tuple of every relation bound in s. Two instances s and s' of a problem are considered isomorphic if there exists a symmetry P such that $s' = P(s)$. Recall the model finding problem defined in Fig. 3. If we restrict it to the (immutable) configuration specification we obtain a pure Kodkod problem, as depicted in Fig. 16. A possible binding returned by `solveRL` for that problem is depicted as a graph in Fig. 17a. Atoms in `Process` appear in boxes, atoms in `Id` appear in circles, and the binary relations `succ`, `next`, and `id` are shown as edges, with the latter being dashed. One symmetry of this problem is the permutation that preserves identifier atoms and permutes process atoms as $\texttt{P2} \mapsto \texttt{P1}, \texttt{P1} \mapsto \texttt{P0}, \texttt{P0} \mapsto \texttt{P2}$. This renders the solution depicted in Fig. 17c isomorphic to the one in Fig. 17a.

To understand how a formula that excludes the set of isomorphic instances can be defined, consider the instance in Fig. 17a as an example. It could be mapped to a formula defining what are the values of all the free relations in the problem, such as

```
some p0,p1,p2,p3,i0,i1,i2,i3 : univ |
    univ    = p0 + p1 + p2 + p3 + i0 + i1 + i2 + i3 and
    Id      = i0 + i1 + i2 + i3 and
    next    = i0 → i1 + i1 → i2 + i2 → i3 and
    Process = p0 + p1 + p2 and
    id      = p0 → i2 + p1 → i1 + p2 → i3 and
    succ    = p0 → p1 + p1 → p2 + p2 → p3                    .
```

By defining the value of **univ**, this formula ensures that the existentially quantified variables denote all the 8 different atoms of the problem's universe. Since quantified variables may be assigned any atom combination for which the formula holds, it also holds for all isomorphic ones, like the one in Fig. 17c.

We will assume the existence of a function $\langle\!\langle \cdot \rangle\!\rangle : \mathit{binding} \to \mathit{form}$ that, given a solution, returns a relational logic formula that restricts the values of relations to those of the solution. Free variables are created for each atom occurring in the binding, resulting in formulas such as the one inside the quantifier in the example just presented. Another parametrized function $\langle\!\langle \cdot \rangle\!\rangle_{\mathscr{A}} : \mathit{binding} \to \mathit{form}$ then existentially quantifies those variables over **univ**, resulting in a formula such as the one above, that exactly describes all instances isomorphic to the solution. For an input problem $\mathscr{A}\ \mathscr{D}\ \phi$ and a generated binding s, the new problem calculated by `solveRL` can be now described as $\mathscr{A}\ \mathscr{D}\ (\phi$ **and not** $\langle\!\langle s \rangle\!\rangle_{\mathscr{A}})$.

The model finding procedure of Kodkod reduces the problem to a propositional logic satisfiability problem, enabling the use of any off-the-shelf SAT solver [53]. Each relation r of arity a is first encoded as a matrix with a dimensions, with each entry denoting a possible tuple t in the relation: the entry for t is set to *false* if t is not allowed by the upper-bound, to *true* if t is required by the lower-bound, or otherwise contains a Boolean variable r_t whose value is to be determined by the SAT solver. Then the relational algebra operators are computed as matrix operations (for example, using matrix multiplication for composition), until atomic formulas are reached and expanded to Boolean formulas (for example, inclusion tests are expanded to a conjunction of point-wise implications between the two computed matrices). Finally, the Boolean connectives are applied, universal quantifiers are unrolled to conjunctions, and existential quantifiers are skolemized when possible or unrolled to disjunctions. The whole process is highly optimised by resorting to special data-structures to represent the (usually quite sparse) matrices and quantifier-free Boolean formulas.

For efficiency reasons, iteration in Kodkod is also implemented directly at the propositional logic level, taking advantage of incremental SAT solving. This means that, in fact, `solveRL` is not stateless but keeps a SAT solver alive between executions (which we abstract in our representation of *Kodkod* problems). To obtain the next solution Kodkod just adds a new clause to the SAT problem asking for a different valuation of at least one of the Boolean variables. For example, the clause added to obtain the next solution after the one of Fig. 17a, where `id` was assigned `{(P0,I2),(P1,I1),(P2,I3)}`, would include conjunct

$$\begin{array}{c}\mathtt{id}_{(\mathtt{P0,I0})} \vee \mathtt{id}_{(\mathtt{P0,I1})} \vee \neg\mathtt{id}_{(\mathtt{P0,I2})} \vee \mathtt{id}_{(\mathtt{P0,I3})} \vee \mathtt{id}_{(\mathtt{P1,I0})} \vee \neg\mathtt{id}_{(\mathtt{P1,I1})} \vee \mathtt{id}_{(\mathtt{P1,I2})} \vee \mathtt{id}_{(\mathtt{P1,I3})} \\ \vee \\ \mathtt{id}_{(\mathtt{P2,I0})} \vee \mathtt{id}_{(\mathtt{P2,I1})} \vee \mathtt{id}_{(\mathtt{P2,I2})} \vee \neg\mathtt{id}_{(\mathtt{P2,I3})} \vee \mathtt{id}_{(\mathtt{P3,I0})} \vee \mathtt{id}_{(\mathtt{P3,I1})} \vee \mathtt{id}_{(\mathtt{P3,I2})} \vee \mathtt{id}_{(\mathtt{P3,I3})}\ .\end{array}$$

This procedure removes from the search space the instance from Fig. 17a but not other isomorphic solutions. To that purpose, Kodkod introduces in the SAT problem a symmetry breaking clause [53], which has also proved to improve the efficiency of the analysis by reducing the search space. A symmetry detector (which here will be abstracted as a procedure

SD) tries to finds all permutations that are symmetries of a problem by processing only the bounds of declarations $\mathscr{D}$. For efficiency reasons, this detector only attempts to find a polynomially computable subset of all possible symmetries, but for most problems it detects all of them. For each symmetry detected by SD, a procedure SP creates a lexicographic leader symmetry breaking predicate [15] that given an order on Boolean variables, guarantees that only the lexicographically smallest solution is allowed. For the relational context this can be simplified by comparing the variables representing the presence of tuples containing the permuted atoms [49]. The order on relations determines which relations preserve their values when symmetry is broken, and in Kodkod lower arity relations are given priority.

For instance, consider a problem with a pair of relations `a :1 {} {(A0),(A1)}` and `r :2 {} {(A0,A0),(A0,A1),(A1,A0),(A1,A1)}`. `A0` and `A1` would be identified as symmetric by SD and a permutation $P = \texttt{A0} \mapsto \texttt{A1}$ returned. Let a Boolean variable r_t determine the presence of tuple t in relation r. Then SP generates the symmetry breaking predicate

$$[\texttt{a}_{\texttt{(A0)}}, \texttt{r}_{\texttt{(A0,A0)}}, \texttt{r}_{\texttt{(A0,A1)}}] \leq [\texttt{a}_{\texttt{(A1)}}, \texttt{r}_{\texttt{(A1,A1)}}, \texttt{r}_{\texttt{(A1,A0)}}] \ ,$$

meaning that, for instance, `a = {(A1)}` and `r = {(A1,A1)}` could be returned, but not `a = {(A0)}` and `r = {(A0,A0)}` which is isomorphic. When iterating, bindings `a = {(A1)}` and `r = {(A0,A0)}` and `a = {(A0)}` and `r = {(A1,A1)}` are possible new solutions but are isomorphic with each other. Due to the order on relations in the symmetry breaking predicate, the former would be selected, preserving the value of `a`. This predicate is easily implemented at the propositional level as

$$\begin{aligned} \texttt{a}_{\texttt{(A0)}} &\rightarrow \texttt{a}_{\texttt{(A1)}} && \wedge \\ (\texttt{a}_{\texttt{(A0)}} = \texttt{a}_{\texttt{(A0)}}) &\rightarrow (\texttt{r}_{\texttt{(A0,A0)}} \rightarrow \texttt{r}_{\texttt{(A1,A1)}}) && \wedge \\ (\texttt{a}_{\texttt{(A0)}} = \texttt{a}_{\texttt{(A0)}} \wedge \texttt{r}_{\texttt{(A0,A0)}} = \texttt{r}_{\texttt{(A1,A1)}}) &\rightarrow (\texttt{r}_{\texttt{(A0,A1)}} \rightarrow \texttt{r}_{\texttt{(A1,A0)}}) && \ . \end{aligned}$$

4.2 Pardinus path iteration

Having a similar approach to iteration in Pardinus (*i.e.*, return any different path) would often fail to incorporate the users expectations when exploring alternative paths. In our experience, scenario exploration is often performed in distinct stages. For instance, the user may first explore different configurations, each framing the context over which the path can evolve, and then explore alternative paths for a selected configuration, trying to find an interesting evolution scenario. In this section (and the next), we show how to specify (and implement) in Pardinus two concrete iteration operations that support different kinds of scenario exploration. Other operations could be implemented following a similar approach[6].

Unlike Kodkod, the constraints introduced in each iteration step will not always be cumulative: for instance, when iterating over alternative paths a fixed configuration is enforced, which must be reset when an alternative configuration is requested. Thus, we extend Pardinus problems as $\mathscr{A}\ \mathscr{D}\ \phi \mid \psi$, where a distinguished formula ψ represents a temporary constraint that characterises the current configuration. As in Kodkod, although in practice a Pardinus problem is updated in-place, we abstract this process by pure functions with type $\mathit{Pardinus} \rightarrow \mathit{Pardinus} \times \mathit{path}$ that, besides the solution path, also return an updated

[6] For instance, Pardinus also implements an operation to change a segment of a path, which is used by the Alloy 6 Analyzer to change the initial state and for forking paths.

problem. Abusing notation, the specification of the iteration operations will rely on a function $\langle\!\langle\cdot\rangle\!\rangle : \mathit{path} \to \mathit{form}$ (defined shortly) that maps a given path solution π to a temporal formula that restricts the values of the declared relations, and a parametrized function $\langle\!\langle\cdot\rangle\!\rangle_{\mathscr{A}} : \mathit{path} \to \mathit{form}$ that then quantifies over **univ** and returns a formula that characterizes all solutions isomorphic to the given path. The notion of isomorphic path is also defined by extending the notion of isomorphic binding. Formally, a permutation P on atoms is said to be a symmetry of a Pardinus problem $\mathscr{A}\ \mathscr{D}\ \phi$ if for all paths π, $\pi \models \mathscr{D}$ and $\pi \models \phi$ iff $P(\pi) \models \mathscr{D}$ and $P(\pi) \models \phi$, where $P(\pi)$ applies the permutation to all bindings π_i for all $i \geq 0$. Two paths π and π' are isomorphic if there exists a symmetry P such that $\pi' = P(\pi)$.

The first operation, $\texttt{solveTRL}_\texttt{C}$, allows iteration at the configuration level, forcing changes in the immutable relations and allowing the remaining relations to adapt freely to the new configuration. The behaviour of this operation can be formalized as

$$\begin{aligned}\texttt{solveTRL}_\texttt{C}(\mathscr{A}\ \mathscr{D}\ \phi \mid \psi) = (\mathscr{A}\ \mathscr{D}\ \phi' \mid \psi', \pi)\\ \text{where } \pi = \texttt{solveTRL}(\mathscr{A}\ \mathscr{D}\ (\phi\ \textbf{and not}\ \psi))\\ \phi' = \phi\ \textbf{and not}\ \psi\ \textbf{and not}\ \langle\!\langle\pi\rangle\!\rangle_{\mathscr{A}}\\ \psi' = \langle\!\langle\pi|_S\rangle\!\rangle_{\mathscr{A}}\ ,\end{aligned}$$

where $\pi|_S$ retrieves the configuration from a path π – *i.e.*, a binding for immutable relations. The new solution π is calculated with the negation of the temporary restriction ψ, which encodes the previous configuration being explored, thus removing it from the state space. The permanent constraint is also updated with this information so that this configuration is not visited again. Lastly, the temporary constraint is reset to fix only the newly generated configuration, and the permanent constraint updated to exclude the new path. This operation is assumed to be the first in any iteration session, so that the temporary constraint is properly initialized with the configuration of the first found solution (initially the temporary restriction is $\bot$, which will enable any configuration to be generated).

The second operation, $\texttt{solveTRL}_\texttt{P}$, keeps the current configuration but forces a change in the path, and is specified by equation

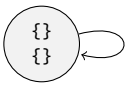
$$\begin{aligned}\texttt{solveTRL}_\texttt{P}(\mathscr{A}\ \mathscr{D}\ \phi \mid \psi) = (\mathscr{A}\ \mathscr{D}\ \phi' \mid \psi', \pi)\\ \text{where } \pi = \texttt{solveTRL}(\mathscr{A}\ \mathscr{D}\ (\phi\ \textbf{and}\ \psi))\\ \phi' = \phi\ \textbf{and not}\ \langle\!\langle\pi\rangle\!\rangle_{\mathscr{A}}\\ \psi' = \psi\ .\end{aligned}$$

Here, to calculate the new solution π the temporary constraint is considered positively so that the current configuration is kept. The permanent constraint – where all previously seen paths are already excluded – is updated with the negation of the new path, while the temporary constraint is preserved, fixing the same configuration.

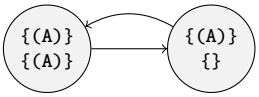
It remains to formalize function $\langle\!\langle\cdot\rangle\!\rangle$, which must take into consideration the notion of isomorphic instances defined above. Besides atom permutations as in Kodkod, here we must also forbid solutions that represent the same path, either by having a different number of unrollings of the looping segment, or that loop back to a different state, but that actually represent the same looping behaviour. For a (k,l)-loop path π, $\langle\!\langle\pi\rangle\!\rangle$ can be defined as

$$\begin{aligned}\langle\!\langle\pi\rangle\!\rangle = &\bigwedge_{0\leq i\leq k} \textbf{after}^i\ \langle\!\langle\pi_i\rangle\!\rangle\ \textbf{and}\\ &\textbf{after}^l\ \textbf{always}\ \bigwedge_{l\leq i\leq k} (\langle\!\langle\pi_i\rangle\!\rangle\ \textbf{implies}\ \textbf{after}^{k-l+1}\ \langle\!\langle\pi_i\rangle\!\rangle)\ ,\end{aligned}$$

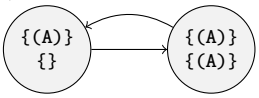
where $\textbf{after}^i$ denotes the nesting of i **after** operators. The first component of this formula fixes the prefix of the path, yielding a formula for each state $\pi_0, \ldots, \pi_k$ with the appropriate



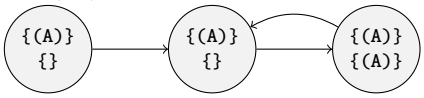
(a) A possible $(0,0)$-loop path returned by an initial `solveTRL`$_\mathtt{C}$



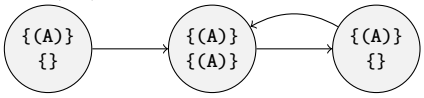
(b) A possible $(1,0)$-loop path returned by `solveTRL`$_\mathtt{C}$ on Fig. 18a



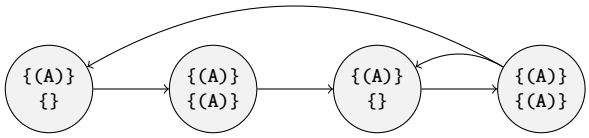
(c) A possible $(1,0)$-loop path returned by `solveTRL`$_\mathtt{P}$ on Fig. 18b



(d) A possible $(2,1)$-loop path returned by `solveTRL`$_\mathtt{P}$ on Fig. 18b



(e) A $(2,1)$-loop path symmetric to Fig. 18c, *not* returned by `solveTRL`$_\mathtt{P}$



(f) Two 3-loop paths symmetric to Fig. 18c, *not* returned by `solveTRL`$_\mathtt{P}$

Fig. 18: An iteration session example

temporal offset. The second component addresses the looping segment $\pi_l,\ldots,\pi_k$, forcing these states to repeat every $k-l-1$ steps. This encoding can be simplified for relations declared as immutable since their state does not change. In this case their restriction can be moved to outside the temporal quantifications. The full formula $\langle\!\langle \pi \rangle\!\rangle_{\mathscr{A}}$ is then computed by existentially quantifying over all atoms of $\mathscr{A}$ and fixing **`univ`**, as in the Kodkod case.

As an example, let us consider a very simple Pardinus problem with two relations `r :1 {} {(A)}` and **`var`** `s :1 {} {(A)}` and constraint **`always`** `s` **`in`** `r`, a solution for which is the $(0,0)$-loop depicted in Fig. 18a (relation `r` on the top and `s` on the bottom). Applying `solveTRL`$_\mathtt{C}$ and solving the resulting problem could result, for example, in the path from Fig. 18b, where the configuration was forced to change due to the negation of $\langle\!\langle \pi|_S \rangle\!\rangle_{\{\mathtt{A}\}}$, which, considering some simplifications, is equivalent to

```
all A : univ | univ != A or some r
```

a formula that forces the value of at least one of the immutable relations, in this case **`univ`** and `r`, to be different than in Fig. 18a. Since **`univ`** cannot change, `r` will be forced to change.

Searching for alternative paths on the new configuration using `solveTRL`$_\mathtt{P}$ would fix the value of the immutable relations and force a path change with the negatation of $\langle\!\langle \pi \rangle\!\rangle_{\{\mathtt{A}\}}$, a formula equivalent to

```
some A : univ | univ = A and r = A

all A : univ |
    univ != A or r != A or
    s != A or (after some s) or
    eventually ((s = A and after after s != A) or
                (no s and after after some s))
```

which forces either the valuations of s in the prefix or the looping behaviour to change from that of Fig. 18b, while preserving the value of the immutable relations. Solving the new extended problem could result in paths such as those in Figs. 18c and 18d. If we repeat the process for the path in Fig. 18c, the formula generated by $\langle\!\langle \pi \rangle\!\rangle_{\{\mathtt{A}\}}$ would hold not only for that $(1,0)$-loop, but also for any other (k,l)-loop encoding the same infinite path, such as, for example, the $(2,1)$-loop in Fig. 18e or the $(3,0)$- and $(3,2)$-loops in Fig. 18f. Thus, adding its negation to a problem guarantees that the next solution is not one of those. Note that when alternating between these operations solutions are no longer guaranteed to be returned with increasing prefix length, since calling $\mathtt{solveTRL_C}$ after calls to $\mathtt{solveTRL_P}$ may find a new configuration for which there are shorter paths. Moreover, if $\mathtt{solveTRL_C}$ is only called after exhausting all calls to $\mathtt{solveTRL_P}$, the process is complete, iterating over all valid solutions.

Some of these operations can be efficiently implemented by exploiting the incremental features of solvers, as will be presented in the next section. Iteration with non-incremental solvers (such as SMV model checkers) or ones whose semantics always requires backtracking (such as those from [6]) can always be implemented by restarting the solver, but this will lead to poor efficiency in iteration and are thus outside the scope of this paper.

4.3 Pardinus SAT iteration

Similarly to Kodkod, the iteration operations are implemented directly at the propositional logic level in Pardinus' SAT backend: a SAT solver is kept alive between enumeration steps so that incremental solving capabilities can be exploited. However, since the constraints introduced by iteration are not always cumulative we cannot rely uniquely on incremental SAT solving – in particular, since $\mathtt{solveTRL_C}$ discards temporary configuration constraints ψ, its application following a $\mathtt{solveTRL_P}$ step must restart the SAT solver. For $\mathtt{solveTRL_P}$ steps, and for consecutive $\mathtt{solveTRL_C}$ steps, solving is always performed incrementally.

The solving procedure is iterative on the prefix length i, and thus at each step we may need to exclude from the search space previously found (k,l)-loops with $k \leq i$ (notice that within an iteration session the prefix length is never reduced). The key challenge is thus, for any path π represented as a (k,l)-loop, to implement the translation of $\langle\!\langle \pi \rangle\!\rangle_{\mathscr{A}}$ formulas into propositional logic under arbitrary $i \geq k$ prefix lengths, particularly when addressing looping behaviour. A particular (k,l)-loop is fully described at the SAT level by the value of the Boolean variables that define the free relations (one set of variables for the immutable relations and $k+1$ sets for the mutable ones), plus Boolean variables defining the value of the loop relation (to simplify the presentation let us assume that no past operators occur in the problem, and thus loops need not be unrolled). For instance, for our trivial problem `r :1 {} {(A)}` and **var** `s :1 {} {(A)}` with $k = 2$, we create a single variable $\mathtt{r_{(A)}}$, denoting whether A belongs to the configuration, 3 variables $\mathtt{s_{(A),}}i$, denoting whether A belongs to s in each state, and 3 variables $\mathtt{loop}_i$ for the loop relation.

Iterating over a (k,l)-loop with $\mathtt{solveTRL_C}$ relies on **not** $\langle\!\langle \pi|_S \rangle\!\rangle_{\mathscr{A}}$ formulas to remove the valuation of immutable relations. For this, it suffices to require a different value for at

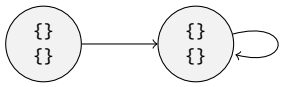


Fig. 19: A non-canonical $(1,0)$-loop path

least one of the variables corresponding to immutable relations. For instance, for the $(0,0)$-loop from Fig. 18a, **not** $\langle\!\langle \pi|_S \rangle\!\rangle_{\mathscr{A}}$ for any i prefix length would be encoded simply as formula $\mathtt{r}_{(\mathtt{A})}$, forcing the value of the configuration to change by adding `A` to relation `r`.

Iterating over a (k,l)-loop with $\mathtt{solveTRL_P}$ for a prefix length i requires fixing a configuration with $\langle\!\langle \pi|_S \rangle\!\rangle_{\mathscr{A}}$ and removing the previous infinite looping behaviour with **not** $\langle\!\langle \pi \rangle\!\rangle_{\mathscr{A}}$. The former is straightforward by fixing the Boolean variables that represent the immutable relations. For the latter we must either force a change of the value of prefix states or a change in the looping behaviour. To impose a change on the relations in the prefix, it suffices to negate the valuation of the Boolean variables corresponding to the declared relations for segment $\pi_0 \dots \pi_i$, possibly unrolling the looping states from the (k,l)-loop if $i > k$. To change the looping behaviour, all identical loops that might have arisen from unrolling the (k,l)-loop up to i must be removed from the search space. These are any states j after the non-looping prefix, *i.e.*, $j \geq l$, whose distance from the last state i is a multiple of the looping segment length $k-l+1$, *i.e.*, $(i-j+1) \mod (k-l+1) = 0$. Due to the path formalization at least one looping state must exist (Fig. 10), so it suffices to add a constraint stating that at least one of the Boolean variables representing the non-identical loops must be true.

Getting back to our example and the $(1,0)$-loop from Fig. 18c, whose looping segment has size 2, the following conjunct would be added under $i = 1$

$$\mathtt{r}_{(\mathtt{A})} \wedge (\mathtt{s}_{(\mathtt{A}),0} \vee \neg \mathtt{s}_{(\mathtt{A}),1} \vee \mathtt{loop}_1) \ ,$$

that is, *i*) fix the valuation of the configuration (`A` must belong to `r`) and either change *ii*) the value of the mutable relations (either add `A` to `s` in state 0 or remove it from state 1) or *iii*) the looping behaviour (state 0 is excluded as a valid loop). With $i = 2$ a state would be unrolled, but still only state 1 excluded as an acceptable looping state, since this is the single looping segment with size multiple of 2 that fits $i = 2$, resulting in formula

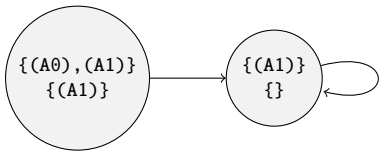
$$\mathtt{r}_{(\mathtt{A})} \wedge (\mathtt{s}_{(\mathtt{A}),0} \vee \neg \mathtt{s}_{(\mathtt{A}),1} \vee \mathtt{s}_{(\mathtt{A}),2} \vee \mathtt{loop}_0 \vee \mathtt{loop}_2) \ ,$$

excluding the solution from Fig. 18e, as expected. Under $i = 3$ there are now two looping states with size multiple of 2 that represent the same path, states 0 and 2, resulting in formula

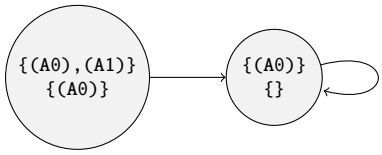
$$\mathtt{r}_{(\mathtt{A})} \wedge (\mathtt{s}_{(\mathtt{A}),0} \vee \neg \mathtt{s}_{(\mathtt{A}),1} \vee \mathtt{s}_{(\mathtt{A}),2} \vee \neg \mathtt{s}_{(\mathtt{A}),3} \vee \mathtt{loop}_1 \vee \mathtt{loop}_3) \ ,$$

which excludes both paths from Fig. 18f, as expected.

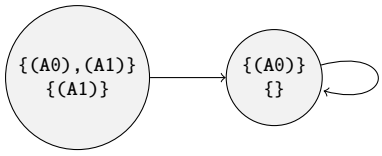
This process removes all representations of a path π resulting from the unrolling of the provided (k,l)-loop, but one may wonder whether there are alternative (k,l')-loops for π that may not be removed by this translation (note that to be efficient, this process is performed without inspecting the concrete states, considering only the k and l values). For instance, consider the path depicted in Fig. 19. Changing the looping state to 0 would actually represent the same path, and our SAT encoding would not exclude it from the search space. However, our solving procedure naturally guarantees that such non-canonical paths – in the sense that changing the looping state l results in the same path – are never returned in the first place. Due to the iterative nature on k of the solving process, we are guaranteed to previously have seen the (k,l)-loop representation of a path π with a minimal k – for Fig. 19,



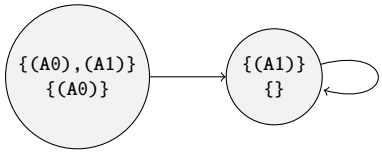
(a) A 2-loop path



(b) A solution symmetric to Fig. 20a, excluded



(c) A solution non-symmetric to Fig. 20a, returned



(d) A solution symmetric to Fig. 20c, excluded

Fig. 20: Symmetric paths for **var** `a = {} {(A0),(A1)}` and **var** `b = {} {(A0),(A1)}`

it would have been the one in Fig. 18a – and it can be shown that in that case no alternative looping states l would represent the same path. If a path with prefix length k can be described by two looping segments of different sizes l_1 and l_2, then it is guaranteed that that same path had been found before for a smaller k with a looping segment of size $\gcd(l_1, l_2)$, a result that easily follows from Bézout's lemma. In fact, while iterating with $i = k$ (*i.e.*, before exhausting the solution space at prefix length i and having to increase it) there is no need to check additional unrollings and we can simply negate the $\texttt{loop}_l$ Boolean variable.

Lastly, the symmetry breaking mechanism of Kodkod must also be adapted to the temporal context of Pardinus. Let us consider another concrete example, a problem with two mutable relations declared, **var** `a :1 {} {(A0),(A1)}` and **var** `b :1 {} {(A0),(A1)}` for which `SD` will return $P = \texttt{A0} \mapsto \texttt{A1}$. Let a Boolean variable $r_{t,s}$ determine the presence of tuple t in relation r at state s. Symmetry breaking in traces should act state-wise, meaning that an atom is symmetric to another if it has been so in all preceding states, and stops being so once a relation in a state breaks that symmetry. This could be achieved by adapting `SP` to generate a symmetry breaking propositional formula such as the following for the example

$$[\texttt{a}_{\texttt{(A0)},0}, \texttt{b}_{\texttt{(A0)},0}, \texttt{a}_{\texttt{(A0)},1}, \texttt{b}_{\texttt{(A0)},1}] \leq [\texttt{a}_{\texttt{(A1)},0}, \texttt{b}_{\texttt{(A1)},0}, \texttt{a}_{\texttt{(A1)},1}, \texttt{b}_{\texttt{(A1)},1}] \ .$$

If there were immutable relations in the problem, they would be given the higher priority when ordering the relations, since immutable relations establish the configuration on which behaviours will act. A possible solution is depicted in Fig. 20a, with the value of a in the top

and b in the bottom. The solution depicted in Fig. 20b is isomorphic to this one and would be excluded from the search space. Solutions from Fig. 20c and Fig. 20d are isomorphic to each other, but not isomorphic to Fig. 20a, so when iterating after this path only one of them will be returned. With the symmetry breaking predicate defined above, Fig. 20c is selected, prioritizing changes as late as possible in the trace, and thus not requiring the user to fully reinterpret the new trace starting from the initial state.

It should be noticed that such predicate does not need to take into consideration loops. For a prefix length k, after state k the path inevitably loops back into a previous state. Thus, if the symmetry was preserved in all states up to k, it will continue to be preserved once path loops back to a previous state without any additional constraint.

5 Parallel decomposed analysis

Configurations, determined by the immutable relations, are initially arbitrary, but remain constant as the system evolves. This enables a decomposed analysis of Pardinus problems that first solves for configurations and afterwards, for each configuration, solves for possible behaviours. Depending on the analysis and number of configurations, this decomposition can yield substantial performance benefits. Such decomposed analysis is also amenable for parallelisation using commodity hardware, since different configurations can be solved independently in different cores. Moreover, since commonly the values of mutable relations depend on those of immutable ones, if these dependencies were explicit, the configurations could be used as partial instances for the succeeding stage, further speeding up analysis. For that purpose, Pardinus allows users to declare *symbolic bounds* for mutable relations, so that dependencies on the immutable relations can be made explicit. This strategy was initially proposed by us [33] for Kodkod; here we extend it to the temporal context, providing an automatic decomposition criterion on the immutable/mutable relations of the problem. This criterion has two main advantages: it enables the deployment of the best-suited solvers for each stage – Kodkod for the configuration and alternative temporal backends for the remaining – and is compatible with the iteration operations proposed in the previous section.

Symbolic bounds are essentially non-temporal relational expressions where only immutable relations can appear as free variables. Thus, arbitrary expressions without primed expressions and relations by comprehension can be used in symbolic bounds[7]. Symbolic bounds could be introduced in our running example from Fig. 3 to impose the type of mutable relations `outbox` and `Elected`, by replacing their declaration with

```
var outbox  :2 {} {Process → Id}
var Elected :1 {} {Process}  .
```

Likewise constant bounds, symbolic bounds allow users to provide additional partial knowledge about a problem in order to further improve the performance of the solving procedures.

Figure 21 presents an overview of the decomposed analysis procedure for complete model finding, denoted by $\texttt{solveTRL}_\texttt{D}$ (the bounded version $\texttt{solveTRL}^n_\texttt{D}$ is obtained by just replacing the calls to solveTRL for $\texttt{solveTRL}^n$). It relies on 3 new procedures: `config`, that extracts the configuration specification of a Pardinus problem; $\oplus$, that *integrates* a binding in a Pardinus problem, restricting the original bounds; and `resolve`, that *resolves* the symbolic bounds with constant tuple sets. Algorithm 1 presents an abstract view of this process.

[7] Although omitted for simplicity, bounding expressions may also refer to concrete atoms as regular constant bounds. These are ignored during symmetry breaking.

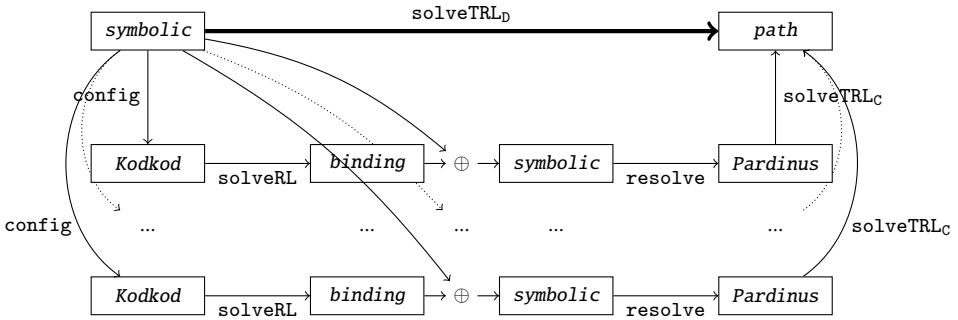


Fig. 21: Complete model finding with the decomposed strategy

Input: A Pardinus problem $\mathscr{A}\ \mathscr{D}\ \phi$.
Output: A new solution or $\bot$
$\mathscr{A}\ \mathscr{D}'\ \rho \leftarrow \texttt{config}(\mathscr{A}\ \mathscr{D}\ \phi)$;
$\pi \leftarrow \bot$;
repeat
  $(\mathscr{A}\ \mathscr{D}'\ \rho, s) \leftarrow \texttt{solveRL}(\mathscr{A}\ \mathscr{D}'\ \rho)$;
  if $s \neq \bot$ **then**
    $\mathscr{A}\ \mathscr{D}\ \phi \leftarrow \mathscr{A}\ \mathscr{D}\ (\phi \oplus s)$;
    $\pi \leftarrow \texttt{solveTRL}(\texttt{resolve}(\mathscr{A}\ \mathscr{D}\ \phi))$;
until $\pi \neq \bot \vee s = \bot$;
return π;

Algorithm 1: Abstract $\texttt{solveTRL}_\texttt{D}$ sequential procedure

The first step is to extract the configuration problem of a Pardinus problem with symbolic bounds using `config`, which will include the declarations of immutable relations and the top level conjuncts of the formula that do not refer mutable relations (nor contain temporal operators). This results in a normal Kodkod problem that can be solved by `solveRL`. Solving the configuration problem will yield a binding s that assigns a concrete value to all immutable relations. Procedure $\oplus$ will then integrate this binding into the original problem, that is, replace the lower- and upper-bounds of each immutable relation r by $s(r)$ (and also remove the top-level conjuncts that were extracted by `config`). The result of this step is a new Pardinus problem with symbolic bounds that specifies the behaviour of a system for one concrete configuration. In the last step, the symbolic bounds of the integrated problem are resolved, so that it can be analysed with any of the Pardinus engines (complete or bounded) described in Section 3. To resolve symbolic bounds one essentially replaces every declaration in the problem by one where the symbolic lower- and upper-bounds are replaced by their constant lower- and upper-bounds. These can be computed using the $\llbracket\cdot\rrbracket_\sigma$ and $\lceil\cdot\rceil_\sigma$ operations (presented in Section 3.2) with an empty context σ, that is

$$\begin{aligned}\texttt{resolve}(r\texttt{:}a\ c\ c) &= r\texttt{:}a\ c\ c\\ \texttt{resolve}(\textbf{var}\ r\texttt{:}a\ \Gamma\ \Delta) &= \textbf{var}\ r\texttt{:}a\ \llbracket\Gamma\rrbracket_\emptyset\ \lceil\Delta\rceil_\emptyset\ .\end{aligned}$$

We assume that this procedure is only called once the static portion has been solved, and immutable relations already have exact bounds assigned. Notice that this process is efficient as it simply evaluates symbolic expressions that ultimately depend on constant expressions. This typically results on tighter bounds for the mutable relations. Note that not all configurations may be extended into a full path solution, so this process may explore several con-

```
1  {I0,I1,I2,I3,P0,P1,P2,P3}
2
3      Id       :1 {(I0),(I1),(I2),(I3)} {(I0),(I1),(I2),(I3)}
4      next     :2 {(I0,I1),(I1,I2),(I2,I3)} {(I0,I1),(I1,I2),(I2,I3)}
5      Process  :1 {(P0),(P1),(P2)} {(P0),(P1),(P2)}
6      id       :2 {(P0,I2),(P1,I1),(P2,I3)} {(P0,I2),(P1,I1),(P2,I3)}
7      succ     :2 {(P0,P1),(P1,P2),(P2,P0)} {(P0,P1),(P1,P2),(P2,P0)}
8  var outbox   :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),...,
9                      (P2,I0),(P2,I1),(P2,I2),(P2,I3)}
10 var Elected  :1 {} {(P0),(P1),(P2)}
11
12 outbox = id and
13 always some p : Process, i : (succ.p).outbox | outbox' =
14                       outbox - succ.p → i + p → (i - ^next.(p.id)) and
15
16 always Elected = {p : Process | once (p.id in p.outbox and
17                                         before not (p.id in p.outbox))}
```

Fig. 22: Result of integrating and resolving the configuration of Fig. 17a

figurations before returning a solution (or exhausting the configurations). A consequence of this process is that solutions are no longer guaranteed to have minimal prefix length, since each independently analysed configuration may have minimal paths of different length.

The configuration problem extracted from our running example is similar to the immutable portion already presented in Fig. 16. Let us assume that the configuration of Fig. 17a is returned when solving this problem. Then result of integrating it in the Pardinus problem with symbolic bounds and resolving is shown in Fig. 22. Notice that relations `outbox` and `Elected` now have smaller upper-bounds, resulting from the resolution of its symbolic bounds for the concrete configuration.

Although the presented algorithm is sequential, in practice these integrated problems are analysed in parallel, according to a (configurable) limit imposed on the number of analysis threads running concurrently, since the number of configurations can be quite high. Once one of these threads returns a satisfying path, it is pushed into a blocking queue that the user can inspect. Other integrated problems keep being analysed and launched in the background until the blocking queue fills up, providing a buffer of full solutions. However, this procedure can perform poorly if there is an overwhelming number of configurations that needs to be explored before finding one that has a solution (or the problem is unsatisfiable, a result that requires the analysis of all configurations). In such situations, the analysis of the original Pardinus problem, where all configurations are *amalgamated* and explored in a single solver run, can be more efficient. To address this issue, Pardinus supports a *hybrid* analysis strategy, where the parallel decomposed analysis is run concurrently with a thread solving the full amalgamated problem. This strategy resembles portfolio parallel SAT solving [23], where identical solvers with different parameters competitively solve the same problem. Note that it is not the case that the hybrid approach will, in the worst case, run at least as fast as the sequential (isolated) analysis of the original, since it is expected to have slightly deteriorated performance due to cache interference.

As in the amalgamated procedures, to support iteration an updated problem is also returned by $\texttt{solveTRL}_\texttt{D}$. In the decomposed context this will include both the state of the configuration problem – encoding which configurations have been explored – and the current integrated problem prior to bound resolution – encoding the current configuration and

whose behaviours have been explored. The iteration operations previously presented are still available under the decomposed strategy and exploit this problem division. $\texttt{solveTRP}_\texttt{C}$ discards the current integrated problem and searches for a new configuration that can be extended into a path solution; $\texttt{solveTRL}_\texttt{P}$ is directly applied to the current integrated problem following the procedure presented in Section 4.3. As expected, in practice these procedures act at the propositional logic level and take advantage of incremental SAT solving. However, unlike the incremental implementations proposed in Section 4.3, $\texttt{solveTRP}_\texttt{C}$ can actually be supported by SMV backends: the iteration of configurations is performed by Kodkod, which are then integrated into the full problem and solved with plain `solveTRL` procedures.

The set of solutions returned by the decomposed strategy must be identical to the one returned by amalgamated solving, thus symmetries must be calculated and applied accordingly. However, there are some issues that must be addressed in this scenario. First, since configurations are solved before behaviours, the order of relations considered in the symmetry breaking predicate must respect this; luckily, this is consistent with the order imposed in Section 4.3 for amalgamated analysis. Second, the declarations of the mutable relations may break symmetries which may affect the configuration problem; thus, procedure `SD` of the configuration problem must consider all relation declarations rather than just the immutable ones. Lastly, the integrated problems must consider the symmetries broken by the concrete configuration under analysis; since configurations are integrated by fixing the lower- and upper-bounds of the immutable relations, these will be naturally detected by `SD`.

6 Evaluation

To be useful, the proposed solving and iteration procedures must scale with the size of the problem and the maximum prefix length. This section discusses their performance for a set of example problems, aiming to answer the following research questions.

RQ1 How does bounded temporal model finding with past scale?
RQ2 How does complete temporal model finding with past scale?
RQ3 What are the gains of the decomposed strategies?
RQ4 How do the bounded iteration operations scale?

We considered 6 Pardinus problems with varying size m for this analysis, with satisfiable and unsatisfiable variants:

`Leader` the ring leader election protocol, being used as a running example; m denotes the number of identifiers and nodes; there are four variants, one satisfiable searching for a scenario (`Scenario`), one unsatisfiable checking the safety property (`Safety`), one satisfiable checking the liveness property in the bugged version (`LivenessBug`), and one unsatisfiable checking the liveness property after fixing the model (`LivenessFix`);

`Hotel` the hotel room locking system used in [25] and packaged with the Alloy Analyzer; in a problem with size m there are at most m guests and rooms and exactly $m+1$ keys available; the two variants check a safety property, one satisfiable that finds a counter-example (`Intervenes`) and another unsatisfiable where the property holds (`NoIntervenes`);

`Mutex` specifies Dijkstra's mutex ordering criterion, also packaged with the Alloy Analyzer; the model size m denotes the maximum number of processes and mutexes available; one version is satisfiable and searches for a scenario (`Show`), and the other checks a safety property that holds for the protocol (`Deadlocks`).

n	maximum trace length
m	model size
$\#T$	total number of configurations
$\#S$	total number of configurations expandable into a valid path
$\#C$	number of iterated configurations, up to 100
$\#P$	number of iterated paths for the 1st configuration, up to 100

AG	time for amalgamated analysis with Glucose
AX	time for amalgamated analysis with bounded nuXmv
AC	time for amalgamated analysis with complete nuXmv
PG	time for parallel analysis with Glucose
PX	time for parallel analysis with bounded nuXmv
PC	time for parallel analysis with complete nuXmv
HG	time for hybrid analysis with Glucose
HX	time for hybrid analysis with bounded nuXmv
HC	time for hybrid analysis with complete nuXmv
$C\#S$	time to iterate all configurations
$C100$	time to iterate the first 100 configurations
$P100$	time to iterate the first 100 paths of the 1st configuration

All times in milliseconds. An italic hybrid result means the integrated problems finished first.

Fig. 23: Key for the benchmark results

`Paxos` an abstract version of the Paxos consensus algorithms; in a model with size m there are at most m acceptors and $m/2$ quorums and ballots; there is a single unsatisfiable safety property that checks whether there is always a consensus (`Consensus`);

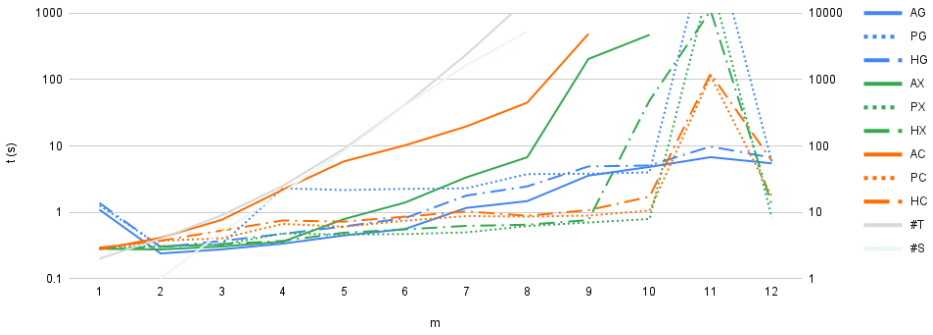
`Echo` a version of the Echo algorithm to form a spanning trees in a network; model size m determines the maximum number of nodes in the graph; one version checks a safety property (`SpanTree`) and another a liveness property (`Finish`), both unsatisfiable;

`SelfStable` Dijkstra's self-stabilizing algorithm; model size m determines the maximum number of nodes in the network; two versions check a safety property, one satisfiable which exhibits a bug (`BugStable`) and another unsatisfiable where the bug is fixed (`StaysStable`), and an unsatisfiable version that checks liveness (`WillStable`).

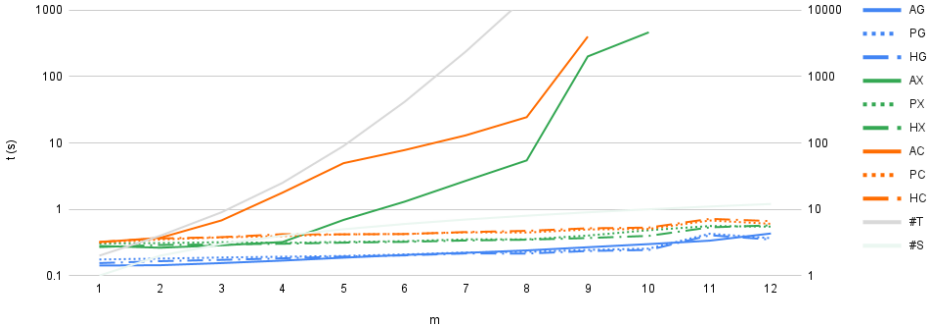
To evaluate solving and SAT-based iteration, commands were executed with different solvers, solving strategies, model sizes and maximum prefix lengths (in hybrid mode, iteration mode depends on whether the amalgamated or the integrated problems finishes first). For $\texttt{solveTRP}_\texttt{C}$ we iterated over all configurations (resulting in as many executions, since the first configuration is returned by the solving procedure, but an additional operation must be performed to identify that there are no more configurations available). For $\texttt{solveTRP}_\texttt{P}$ we executed up to 100 operations to the first solution returned. The key for the results is shown in Fig. 23. All tests were run in a 2.3 GHz Intel Core i5 with 16 GB RAM using Pardinus 1.2, and MiniSAT 2.2, Glucose 2.1, NuSMV 2.6 and nuXmv 2.0 in the backend. The timeout was set to 1000s[8]. Table 1 presents an excerpt of the results for solving, detailed in Figs. 24 and 25 for the `Leader` problem. Table 2 depicts an excerpt of the results for iteration over the satisfiable problems, detailed in Figs. 26 and 27 for the `Intervenes` problem., which has a richer variety of counter-examples. The number of configurations and the number of such configurations that can be extended into a valid path solution (0 if unsatisfiable) are shown to provide an idea of the complexity of each problem.

RQ1 Our experiments showed that Glucose consistently outperforms MiniSAT in non-trivial problems (*i.e.*, that take more than 1s to solve), so the presented results focus on the former. For example, for $n = 12$, in unsatisfiable `LivenessFix` at $m = 3$ both Glucose and MiniSAT take less than 2s, but by $m = 6$ MiniSAT already takes 200s while Glucose only 101s. In the bounded context, NuSMV and nuXmv have similar performance (as expected, since they implement the same algorithm). For instance, for `LivenessFix` at $n = 12$

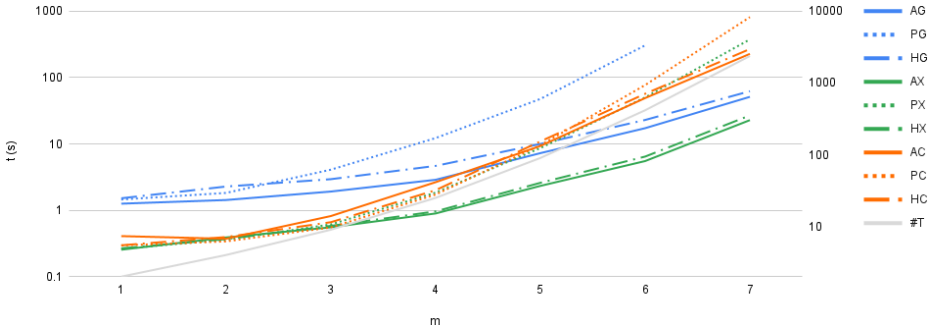
[8] The complete results are available online at `https://bit.ly/2YF6hWL`, and scripts to reproduce the results available at the Pardinus repository.



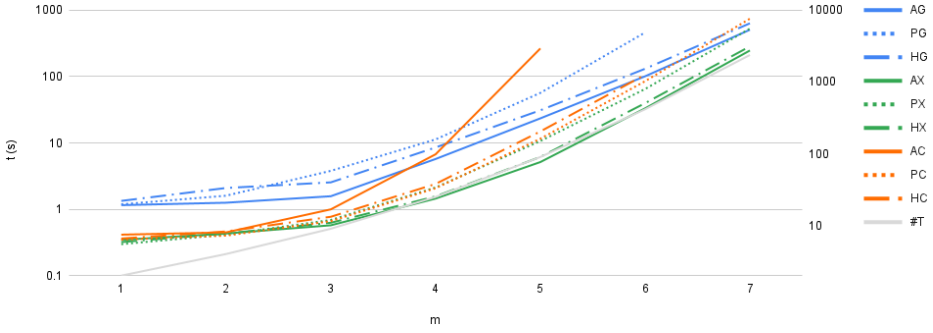
(a) `Scenario` problem for the `Leader` example



(b) `LivenessBug` problem for the `Leader` example



(c) `Safety` problem for the `Leader` example



(d) `LivenessFix` problem for the `Leader` example

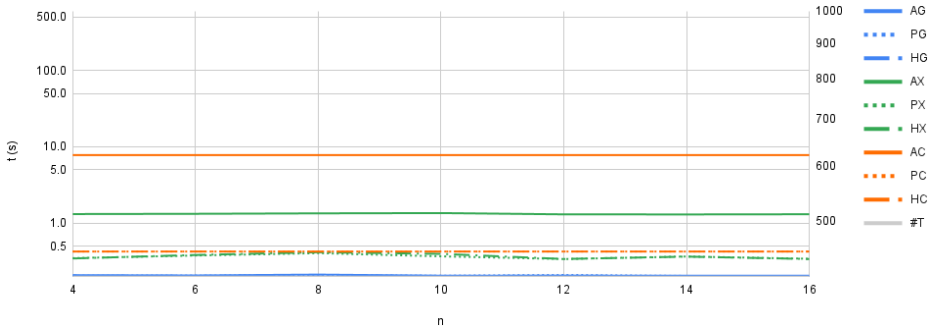
Fig. 24: Solving times for `Leader` variants, $n = 12$ and increasing model size m

problem	n	m	#T	#S	AG	AX	AC	PG	PX	PC	HG	HX	HC
Scenario	12	6	416	409	0.6	1.4	10.2	2.3	0.5	0.7	0.8	*0.6*	*0.9*
Scenario	12	7	2373	1645	1.2	3.4	19.6	2.3	0.5	0.9	1.8	*0.6*	*1.0*
Scenario	12	8	16073	5264	1.5	6.7	45.0	3.8	0.6	0.9	2.5	*0.6*	*0.9*
Safety	12	5	90	0	7.3	2.3	9.3	47.6	8.7	9.5	10.0	2.6	10.9
Safety	12	6	416	0	17.2	5.5	49.0	305.2	50.2	77.0	22.9	6.4	56.9
Safety	12	7	2373	0	51.2	22.8	227.1	> 999	375.5	814.0	62.2	26.8	268.8
LivenessBug	12	6	416	6	0.2	1.3	7.8	0.2	0.3	0.4	*0.2*	*0.3*	*0.4*
LivenessBug	12	7	2373	7	0.2	2.7	13.0	0.2	0.4	0.4	*0.2*	*0.3*	*0.5*
LivenessBug	12	8	16073	8	0.2	5.4	24.4	0.2	0.4	0.4	*0.2*	*0.3*	*0.5*
LivenessFix	12	4	25	0	5.7	1.4	6.7	11.3	2.1	2.1	8.5	1.6	*2.4*
LivenessFix	12	5	90	0	23.4	5.1	264.9	56.7	10.7	11.6	31.0	6.1	*14.9*
LivenessFix	12	6	416	0	100.9	33.2	> 999	461.0	64.8	85.2	131.1	39.7	*106.5*
Intervenes	10	4	255	78	0.8	2.1	45.9	0.3	0.4	1.8	*0.3*	*0.4*	*1.9*
Intervenes	10	5	1212	508	4.0	7.9	387.6	0.3	0.4	2.5	*0.3*	*0.5*	*2.7*
Intervenes	10	6	6132	3225	6.5	29.5	> 999	0.3	0.5	3.6	*0.3*	*0.6*	*3.7*
NoIntervenes	10	2	12	0	0.5	0.6	1.9	0.9	0.8	1.4	0.7	0.7	*1.7*
NoIntervenes	10	3	56	0	2.5	6.5	57.2	4.9	7.2	19.6	4.0	8.2	72.7
NoIntervenes	10	4	255	0	55.3	473.6	> 999	30.1	84.7	437.4	*38.9*	*131.2*	> 999
Show	15	9	9	8	1.0	2.0	13.2	0.5	0.4	2.4	*0.5*	*0.5*	*2.7*
Show	15	10	10	9	1.3	2.9	16.5	0.5	0.5	2.7	*0.5*	*0.5*	*3.0*
Show	15	11	11	10	1.8	4.1	27.8	0.6	0.5	3.5	*0.5*	*0.5*	*3.8*
Deadlocks	15	9	9	0	5.2	2.3	6.2	8.8	3.2	11.8	9.2	3.1	8.3
Deadlocks	15	10	10	0	6.6	3.2	7.2	11.9	5.0	15.7	10.9	4.3	9.6
Deadlocks	15	11	11	0	8.8	4.6	9.8	15.7	7.4	20.8	13.8	5.8	12.7
StaysStable	12	4	20	0	19.0	9.6	3.5	10.1	2.8	0.9	*10.5*	*2.9*	*1.1*
StaysStable	12	5	42	0	128.6	632.7	> 999	76.6	36.7	2.8	*88.4*	*42.7*	*4.0*
StaysStable	12	6	90	0	665.0	> 999	> 999	537.2	336.3	11.8	*676.2*	*435.4*	*21.6*
BugStable	12	5	42	3	4.8	3.1	> 999	13.0	4.3	474.4	7.5	4.4	*678.8*
BugStable	12	6	90	6	9.0	10.5	> 999	19.2	5.9	> 999	13.8	*9.6*	> 999
BugStable	12	7	240	6	27.2	48.5	> 999	68.6	22.8	> 999	40.5	*36.2*	> 999
WillStable	12	4	14	0	3.2	2.2	47.0	3.1	1.0	10.1	*3.7*	*1.2*	*10.4*
WillStable	12	5	28	0	10.7	12.1	> 999	9.4	3.2	> 999	*12.2*	*4.7*	> 999
WillStable	12	6	61	0	31.5	85.7	> 999	37.2	13.2	> 999	40.5	*18.4*	> 999
SpanTree	16	3	4	0	8.0	2.9	2.1	5.7	2.0	1.3	*6.4*	*2.3*	*1.4*
SpanTree	16	4	15	0	116.0	83.6	17.1	89.8	38.3	19.5	*114.8*	*49.5*	20.9
SpanTree	16	5	73	0	> 999	> 999	219.5	> 999	> 999	433.6	> 999	> 999	269.3
Finish	16	3	4	0	13.2	3.4	3.7	12.6	2.9	2.8	*13.9*	*3.2*	*3.3*
Finish	16	4	15	0	101.3	118.2	> 999	130.5	97.8	> 999	157.8	*115.4*	> 999
Finish	16	5	73	0	474.0	> 999	> 999	> 999	> 999	> 999	725.2	> 999	> 999
Consensus	12	3	21	0	2.2	1.0	3.2	5.9	2.0	3.2	3.0	1.4	1.4
Consensus	12	4	40	0	2.8	1.6	4.5	13.8	4.3	9.4	4.1	2.5	2.5
Consensus	12	5	294	0	33.1	36.0	493.4	587.5	202.0	> 999	44.4	51.7	51.7

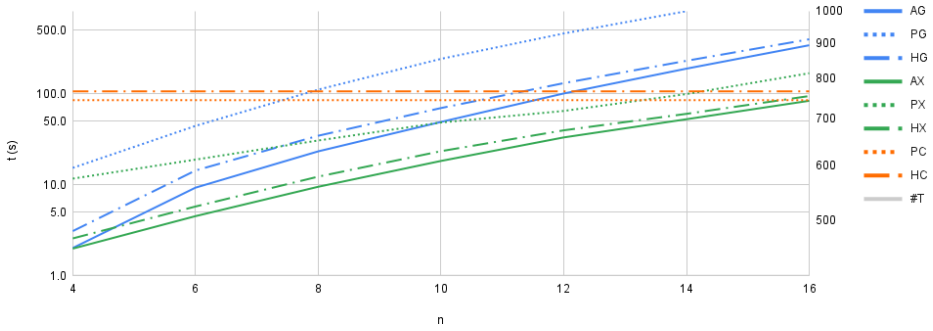
Table 1: Solving performance results for the selected Pardinus problems

and $m = 6$, NuSMV takes 35s and nuXmv 33s, while for Safety for the same parameters both around 6s. However, as we shall soon show, nuXmv largely outperforms NuSMV in complete mode, so the bounded SMV results presented also focus on the former.

It is clear that for satisfiable problems both bounded backends can scale to considerable model sizes, although in comparison the SMV backend seems to scale slighltly worse (*e.g.*, LivenessBug and Intervenes). Notice also that for satisfiable problems, once a solution is found increasing the prefix length does not affect performance, as expected from the iterative nature of the procedures (*e.g.*, LivenessBug). Considering unsatisfiable problems, scalability deteriorates with the model size, as expected. The SMV backend often outperforms the SAT one for smaller model sizes, but is eventually outperformed as m increases (*e.g.*, Safety), although for some examples the SAT backend outperforms the SMV one even for small m values (*e.g.*, NoIntervenes). No consistent pattern could be identified for increasing n values: *e.g.*, the SMV backend scales better in LivenessFix, Safety and Deadlocks, but considerably worse in NoIntervenes and StaysStable.



(a) `LivenessBug` problem for the `Leader` example



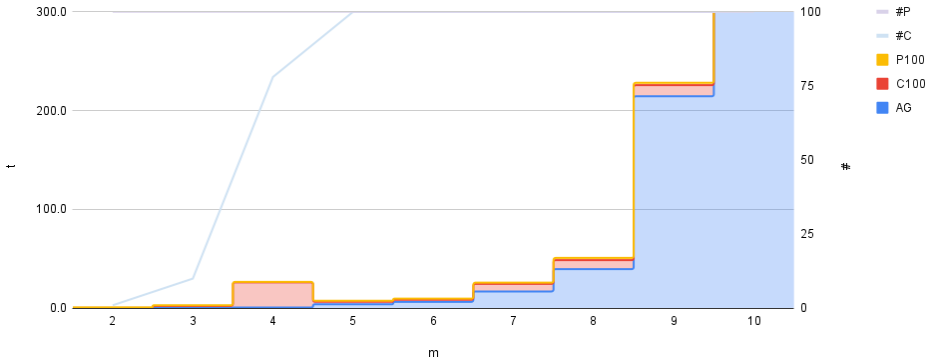
(b) `LivenessFix` problem for the `Leader` example

Fig. 25: Solving times for `Leader` variants, $m = 6$ and increasing maximum trace length n

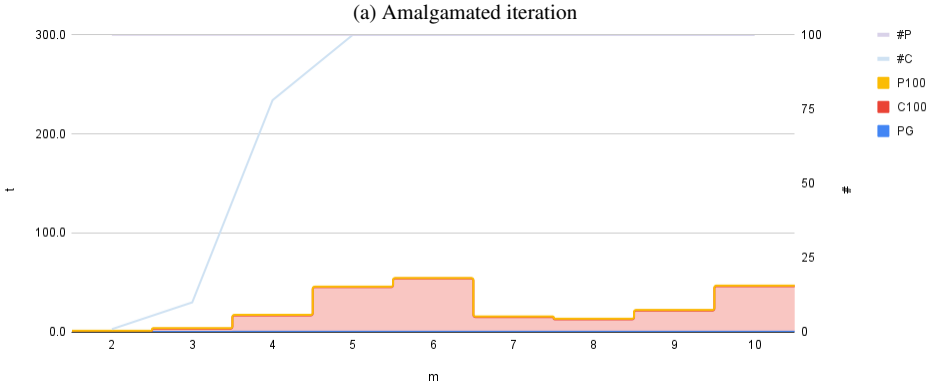
					amalgamated					parallel				
problem	n	m	$\#T$	$\#S$	AG	$C100$	$C\#S$	$\#P$	$P100$	PG	$C100$	$C\#S$	$\#P$	$P100$
`Scenario`	12	5	90	84	0.4	7.7	7.7	11	2.3	2.2	18.6	18.6	11	2.2
`Scenario`	12	6	416	409	0.6	9.0	34.9	11	3.0	2.3	31.8	236.5	11	2.1
`Scenario`	12	7	2373	1645	1.2	24.8	158.8	11	3.8	2.3	170.4	> 999	11	2.4
`LivenessBug`	12	5	90	5	0.2	21.2	21.2	1	2.4	0.2	53.7	53.7	1	2.2
`LivenessBug`	12	6	416	6	0.2	96.6	96.6	1	3.1	0.2	446.9	446.9	1	2.2
`LivenessBug`	12	7	2373	7	0.2	429.3	429.3	1	4.2	0.2	> 999	> 999	1	2.2
`Intervenes`	10	4	255	78	0.8	25.4	25.4	> 100	0.5	0.3	16.5	16.5	> 100	0.6
`Intervenes`	10	5	1212	508	4.0	2.5	786.0	> 100	1.0	0.3	45.0	120.5	> 100	0.5
`Intervenes`	10	6	6132	3225	6.5	2.2	> 999	> 100	1.1	0.3	53.7	> 999	> 100	0.5
`Show`	15	6	6	5	0.5	9.0	9.0	> 100	0.4	0.4	18.5	18.5	> 100	0.5
`Show`	15	7	7	6	0.7	109.6	109.6	> 100	0.4	0.4	185.3	185.3	> 100	0.5
`Show`	15	8	8	7	0.9	977.2	977.2	> 100	0.3	0.4	> 999	> 999	> 100	0.4
`BugStable`	12	5	42	3	4.8	9.4	9.4	> 100	0.7	13.0	3.4	3.4	> 100	0.5
`BugStable`	12	6	90	6	9.0	38.0	38.0	> 100	1.1	19.2	33.3	33.3	> 100	0.5
`BugStable`	12	7	240	6	27.2	183.1	183.1	> 100	1.5	68.6	178.7	178.7	> 100	0.2

Table 2: Iteration performance results for the selected Pardinus problems

RQ2 Our experiments showed nuXmv to almost always outperform NuSMV, so the presented data focuses on the former. For instance, for satisfiable `Scenario` and `LivenessBug` NuSMV in complete mode times out at $m = 4$, while nuXmv only at $m = 10$, and for unsatisfiable `Safety` and `LivenessFix` NuSMV times out at $m = 5$ and nuXmv at $m = 8$ and $m = 6$, respectively. Recall that maximum length n is irrelevant in the complete context.



(a) Amalgamated iteration



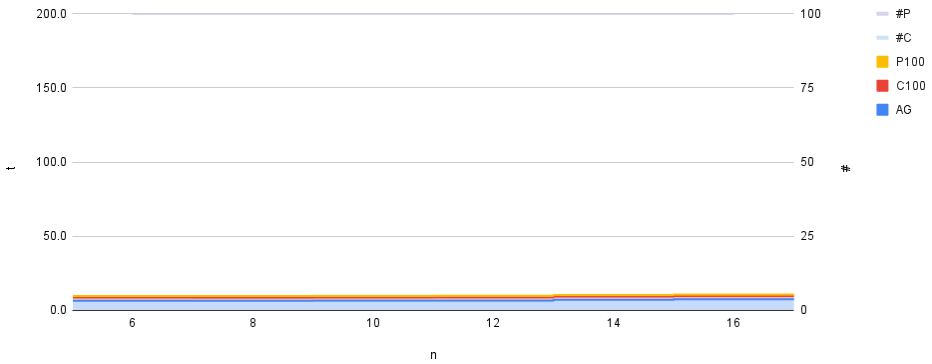
(b) Parallel iteration

Fig. 26: Iteration times for `Intervenes`, $n = 10$ and increasing model size m

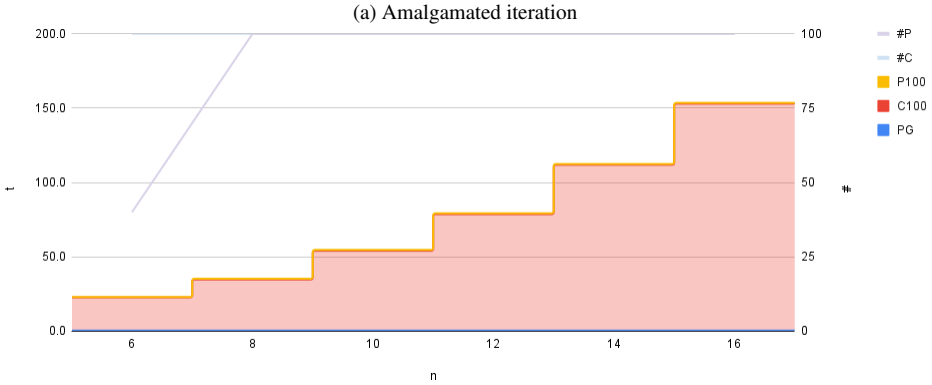
As expected, for small m values the scalability of the complete backend is worse than that of the bounded ones. Like the SMV backend in bounded mode, the fact that the problem is satisfiable does not seem to greatly improve performance. Recall however that the complete backend is most useful to verify desirable properties once the bounded procedures have produced enough confidence, so they are expected to be run less often (and not for knowingly satisfiable problems). Nonetheless, as the maximum prefix length n for the bounded backends increases, the performance of the complete backend closes on that of the bounded backends, occasionally actually outperforming them (*e.g.*, `SpanTree`). Thus, when searching for solutions with long prefixes, it may pay off to switch to a complete backend.

RQ3 For the decomposed strategies, Glucose was used for the static configuration problem. The problems were also solved without symbolic bounds to measure the impact of this feature (not shown in the table). Results show that the gains improve with model size, particularly for satisfiable problems. For instance, for satisfiable `Scenario` for $n = 12$ and $m = 12$ symbolic bounds have gains of 1.8x for the SAT and 2.2x for the bounded SMV backends, while for unsatisfiable `NoIntervenes` for $n = 10$ and $m = 4$ the gains are 1.1x for both.

The parallel strategy almost always improves performance considerably for non-trivial satisfiable problems, particularly for the SMV backend whose amalgamated performance scales worse for satisfiable problems (*e.g.*, `LivenessBug` and `Intervenes`). The same ap-



(a) Amalgamated iteration



(b) Parallel iteration

Fig. 27: Iteration times for `Intervenes`, $m = 6$ and increasing prefix length n

plies for the complete SMV backend (*e.g.*, `LivenessBug`). The parallel strategy performs slightly worse only for `Scenario` with the SAT backend. The results of the parallel strategy for unsatisfiable problems are not as consistent, with some problems having speedups (*e.g.* `NoIntervenes`) and others significant slowdowns (*e.g.* `LivenessFix`). Again, the SMV backends seem to benefit more from this strategy, almost always paying off with the complete backend. It should be noted that the performance of this strategy depends on the order in which the configurations are generated by the configuration problem and on how many of those can be extended into a path solution. For instance, for the `LivenessBug` problem, very few configurations exhibit the problem (those with a single node), but they seem to be generated within the first 4 configurations. In `Scenario` there are several configurations that can lead to an elected node, but as the size of the model increases, the $n = 12$ steps do not suffice to elect a leader in some configurations, which results in a less regular performance.

The hybrid approach tames the negative outliers while still preserving the gains otherwise. For satisfiable executions, all cases where the parallel execution outperformed the amalgamated execution continue to do so in hybrid mode, although with a slight deterioration of performance. The same applies to unsatisfiable problems, the exception being `NoIntervenes` for the complete backend. For cases where the amalgamated approach outperformed the parallel one, in hybrid mode the amalgamated process terminates before the smaller integrated problems, with some deterioration of performance. Nonetheless, in all

problems, in comparison to the amalgamated analysis the hybrid analysis never slowed down more than 0.5x and had speedups of up to 1317.6x.

RQ4 Both operations have shown to be feasible for interactive sessions with both strategies. Performance does not seem to be completely dependent on the model size m, as evident from the spikes in the graphs. For `Show`, $\texttt{solveTRP}_\texttt{C}$ has poorer scalability possibly due to the fact that it has few alternative configurations ($m-1$), since performance in `Ring` is also worse for `LivenessBug` than `Scenario`. $\texttt{solveTRP}_\texttt{P}$ seems to scale better in general for both strategies. Considering increasing maximum lengths, $\texttt{solveTRP}_\texttt{P}$ seems to be little affected, being in fact constant for `Hotel` and `Mutex`. In the case of $\texttt{solveTRP}_\texttt{C}$, this scalability seems to be affected by the number of configurations that can lead to complete path, showing the worse performance for `Scenario` and `Show`. Comparing the strategies, for $\texttt{solveTRP}_\texttt{C}$ the amalgamated mode consistently outperforms the parallel strategy, except for the occasional spike (*e.g.*, in `Scenario`). In contrast, $\texttt{solveTRP}_\texttt{P}$ in parallel mode almost always outperforms the amalgamated mode, which is expected since it is iterating over a less complex problem (an integrated problem whose configuration has already been fixed). The exception is `Show`, but running times for 100 paths are below 0.5s thus difference are not significant. Note, however, that the time taken to initially solve the problem before iteration should also be considered, and that the parallel approach for satisfiable problems often outperforms the amalgamated one. Note, however, that the performance of $\texttt{solveTRP}_\texttt{P}$ (and even the number of solutions available) is dependent on the first configuration returned, which may not be the same for amalgamated and parallel strategies, so no definitive conclusions can be drawn from this experiment. Calling $\texttt{solveTRP}_\texttt{C}$ operations after $\texttt{solveTRP}_\texttt{P}$ ones, as explained in Section 4.3, amounts essentially to solving a new problem as addressed in the previous RQs.

7 Related work

7.1 Model checking and model finding

Complete model checkers for temporal logics are currently robust and well-established tools, widely used to verify safety properties. They are often divided in two categories, those that explicitly represent the states of the system during analysis – such as SPIN [24] for Promela models, TLC [28] for TLA^+ models or ProB [29] for B models – and those that represent states symbolically as a formula, often relying on off-the-shelf solvers – such as NuSMV [9] or nuXmv [8] for SMV models. Bounded model checking has also been widely researched, and model checkers such as NuSMV, nuXmv or ProB also support bounded analyses. Unlike Pardinus, the traditional bounded semantics for temporal logic [3] is also defined for witnesses without the back loop, representing finite prefixes of (infinite) execution paths. The main advantage is the possibility of returning shorter counter-examples to safety properties (or instances of liveness properties). However, such semantics can be confusing. At least three different semantics – *weak*, *strong*, *neutral* – have been proposed [20]. In the (most commonly adopted) strong version, on finite prefixes $\textbf{always}\ \phi$ never holds and $\textbf{not}\ \textbf{after}\ \phi \not\equiv \textbf{after}\ \textbf{not}\ \phi$ because $\textbf{after}\ \phi$ is always false in the last known state. This would rarely be useful in Pardinus anyway: problems almost always include at least one **always** formula specifying the behaviour of the system, which rules out finite prefixes. Lastly, the performance advantage of having shorter counter-examples can still be achieved by allowing the traces to loop at any state through a stutter event.

Finite model finders for first-order logics have also been proposed, which, besides being used verification, are also commonly used for the generation of scenarios for validation. They are often categorized as either implementing specialized search procedures – such as SEM [56] or Mace4 [34] – or being based on the compilation into off-the-shelf solvers – such as Paradox [13] or Kodkod [53] (and Alloy, that uses Kodkod at its backend). However, support for model checking or model finding for first-order temporal logic specifications is scarce. Performing behavioural analysis in such model finders is possible but requires the explicit encoding of traces and temporal properties.

For Alloy specifically, [16] shows how to perform analysis of future-only LTL, and [55] how to perform complete analysis of CTL if the scope for states covers all reachable states. Besides being cumbersome and quickly becoming unmanageable, these approaches impair the application of dedicated analysis procedures, and considerable research has been dedicated to propose Alloy-like languages with native support for behavioural analysis [21, 11, 37]. These approaches compromise the flexibility of the language, introducing syntactic extensions that force users to adhere to specific idioms. DynAlloy [21] resorts to dynamic logic to specify behaviour, and liveness properties are not expressible. The technique by Near and Jackson [37] enhances Alloy with imperative constructs, analysable through a translation into regular Alloy. The one by Chang and Jackson [11] extends the relational logic of Alloy with CTL temporal logic and proposes a dedicated model checker, but system actions must be specified with a fixed imperative-flavoured idiom. Moreover, it disregards the rich structural properties introduced by the signature type system from regular Alloy. In DASH [48] the system is modelled as a hierarchical state machine over which CTL properties can be checked through a translation into Alloy. Electrum [30] was proposed by us to address these issues. It is a conservative extension that provides support for LTL constraints while preserving the flexibility of the language. It was the analysis of Electrum models that motivated the development of Pardinus, which is now used as the backend of its Analyzer.

In the context of model checking, as far as we are aware, TLC is the only one that natively supports a first-order temporal logic, the *Temporal Logic of Actions* (TLA) [27] at the core of TLA$^+$. TLA popularised the idea of modelling the system under analysis using a temporal logic formula with primed variables, instead of using a state-machine DSL as is usual in model checking. However, TLC only supports the verification of a subset of the temporal properties allowed in TLA that is less expressive than FOLTL. In particular, to enable the verification of refinement, specifications in TLA are restricted to be *stutter-invariant*: the temporal operator **`after`** is not supported and primes can only appear in so-called *action* predicates, applied directly to variables and inside a special invariant temporal operator that necessarily allows stuttering. Pardinus supports formulas that are not stutter-invariant and primes can be applied to any expression, giving more freedom to the user when specifying systems. Note also that, unlike in TLA, primes in Pardinus (and FOLTL in general) do not add expressive power, as they can always be encoded with a combination of quantification and **`after`** (although in a much more verbose manner). This is what enables the translation to SMV described in Section 3.2. SMV also supports a prime operator, but like in TLA it can only be applied to atomic variables and thus could not be used to directly encode Pardinus' prime. Still concerning TLC, it is worth noting that due to the explicit state nature of its model checking procedure, adding rich first-order structural properties to the system specification (e.g. constraining the initial state or the non-deterministic outcome of an event) may also considerably hinder its performance [31]. The B-method also supports first-order constraints but not temporal logic, so natively only invariants can be checked by ProB. It has however been extended to support the verification of LTL properties [39]

but such properties are not integrated in the B specification language and must be defined separately. Moreover, they cannot be used to restrict the model of the system under analysis.

7.2 Scenario exploration

Most model checkers provide an interactive simulation mode that allows users to explore the state space by choosing transitions. However, these modes ignore temporal properties that span the complete trace, and thus cannot be used to explore alternative witnesses or counter-examples for a particular property. Some techniques have been proposed for the iteration of traces satisfying a given temporal property, the simplest ones simply returning an alternative trace. For instance, SPIN can be instructed to ignore a certain number of violations when exploring the state space, Kromodimoeljo [26] proposes a technique to incrementally generate counter-examples, and the first version of the Electrum Analyzer allowed for the efficient exclusion of a trace through incremental SAT solving. However, the number of alternative solutions may be too large for this kind of "blind" iteration to be useful. Some approaches [19,7] for particular modelling languages infer from a counter-example a formula representing certain equivalence classes that represent similar solutions. The model checking process can then be restarted with the original problem conjoined with the negation of such formula, removing similar solutions from the search space. By introducing symmetry breaking predicates when launching the solving procedure, our approach guarantees that isomorphic traces are never returned. Nonetheless, since in our approach the model can be restricted by arbitrary temporal properties, iteration for more complex equivalence classes (such as all traces following the same control-flow path) could be easily implemented by directly restricting the problem's formula and restarting the process. More advanced approaches may rely on user input to guide the generation of new counter-examples. KEGVis [12] queries the model checker to provide an explanation to the counter-example in the shape of a CTL proof that the user can inspect and then ask for alternative proofs that result in different traces. Recently we have proposed an approach requiring less knowledge about proof systems, allowing the user to focus on a particular segment of the trace to change or ask for particular transitions to take place [6]. It has been implemented in the Electrum Analyzer over Pardinus by restarting the problem with an updated formula.

Iteration on static solutions has received considerable attention, including techniques aimed at the Alloy/Kodkod ecosystem. As already presented, Kodkod relies on incremental SAT solving and symmetry breaking predicates to remove isomorphic solutions from the search-space. Nonetheless, to tackle the possibly overwhelming number of solutions, more principled iterations have subsequently been proposed, with criteria such as minimality [38, 17,32,47] or coverage [40,46,52,41]. Others have focused on iterating over richer classes of equivalences, either through abstract views defined by users [51] or graphs derived from solutions [14]. These approaches are orthogonal to the problem addressed in Section 4, since path iteration operations can be combined with principled iteration over the individual states of the trace. Pardinus currently reuses Kodkod's mechanism, but many of these approaches act at the SAT-level and could be integrated into our incremental approach.

7.3 Incremental and decomposed analysis

Some work has also been developed to improve the performance of the Alloy/Kodkod toolkit through incremental analyses. Kato [54], a technique from which we drew inspiration, splits

Alloy models in two pairs of constraints, and solutions to the first are fed as partial information to the second. The best partition criteria is chosen by testing candidates at small scopes. The proposed solving process was purely sequential but already showed performance gains. Ganov et al. [22] proposed to apply this slicing recursively and performing parallel analysis, then relying on dedicated solvers for particular classes of constraints. Our technique also acts in parallel but additionally exploits partial instances to reduce the search space. The adopted decomposition criterion also allows the deployment the best-suited solvers for each stage. We also address iteration and symmetry breaking.

The partitioning and parallelisation of Alloy analysis procedures has also been proposed. In Ranger [44] each parallel problem solves the same constraints but within a restricted search space, defined by a range of solutions. Ranges are derived from the structure of the models, disregarding the constraints, resulting in unpredictable complexity. The number of partitions is equal to the number of available parallel processes, but since partitioning does not guarantee problems with similar complexity, some process may become idle while others are encumbered with more complex tasks. This issue is tamed by allowing the dynamic partition of problems. In our approach there are usually much more partitions, with reduced complexity, than available processes, so processes rarely become idle. In general, this renders their approach more suitable for unsatisfiable problems (where the complete search space must be searched) and ours for satisfiable problems. In [43] the same authors explore a technique, which they dub *tranScoping*, to infer partitions on the SAT propositional variables from high-level Alloy models with small scopes.

Approaches to incremental solving have also been developed in the context of Alloy model evolution. In Titanium [1] the solutions of a previous version of a model are used to tighten the bounds of the analysis of the revised version, while in Platinum [57] the results of slices of constraints from the analysis of previous versions is stored and reused when analysing a revised version. However, our proposed decomposed approach does not focus in the particular context of re-running analysis during model evolution.

Regarding providing Kodkod with additional partial knowledge, Montaghami and Rayside [36] proposed a technique to extract finer Kodkod partial solutions from high-level specifications, still relying on its constant tuple set bounds. They proposed an extension to Alloy for the specification of solutions, that can be mapped into Kodkod bounds. Our approach extends the expressiveness of partial solutions at the Kodkod/Pardinus level. Since Alloy natively supports binding expressions in the declaration of the relations, symbolic bounds are easily retrieved without any extension to the language.

8 Conclusion

The paper presented Pardinus, a temporal relational model finder backed by bounded and complete model checking engines, that can be used either by end users to validate designs or as a backend for techniques requiring automated instance generation. Pardinus extends the Kodkod relational model finder, still providing a simple language built around the notion of relation, supporting solution iteration operations with symmetry breaking, and enabling the provision of partial instances. Additionally, a decomposed solving strategy is also supported. Evaluation shows that the solving engines are scalable, particularly if the decomposed strategy is enabled, as well as the iteration operations.

Future work is planned to empirically evaluate the iteration operations, and possibly extend its catalogue. Pardinus is used in the backend of the Alloy 6 Analyzer, which provides

a higher-level specification language and a graphical visualizer for solutions that could support such evaluation with end users. We also to intend to keep exploring alternative backends solvers, such as the SMT support for model finding [42,35], to improve the performance of the analysis procedures and, possibly, support more expressive problems.

References

1. Bagheri, H., Malek, S.: Titanium: Efficient analysis of evolving Alloy specifications. In: SIGSOFT FSE, pp. 27–38. ACM (2016)
2. Benedetti, M., Cimatti, A.: Bounded model checking for past LTL. In: TACAS, *LNCS*, vol. 2619, pp. 18–33. Springer (2003)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS, *LNCS*, vol. 1579, pp. 193–207. Springer (1999)
4. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuXmv 2.0.0 User Manual. FBK (2019). `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`
5. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The Electrum Analyzer: Model checking relational first-order temporal specifications. In: ASE, pp. 884–887. ACM (2018)
6. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: Simulation under arbitrary temporal logic constraints. In: F-IDE@FM, *EPTCS*, vol. 310, pp. 63–69 (2019)
7. Castillos, K.C., Waeselynck, H., Wiels, V.: Show me new counterexamples: A path-based approach. In: ICST, pp. 1–10. IEEE (2015)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV, *LNCS*, vol. 8559, pp. 334–342. Springer (2014)
9. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.6 User Manual. FBK-IRST (2010). `http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`
10. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Communications of the ACM **22**(5), 281–283 (1979)
11. Chang, F.S., Jackson, D.: Symbolic model checking of declarative relational models. In: ICSE, pp. 312–320. ACM (2006)
12. Chechik, M., Gurfinkel, A.: A framework for counterexample generation and exploration. Int. J. Softw. Tools Technol. Transf. **9**(5-6), 429–445 (2007)
13. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: CADE-19 Workshop on Model Computation (2003)
14. Clarisó, R., Cabot, J.: Diverse scenario exploration in model finders using graph kernels and clustering. In: ABZ, *LNCS*, vol. 12071. Springer (2020)
15. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: KR, pp. 148–159. Morgan Kaufmann (1996)
16. Cunha, A.: Bounded model checking of temporal formulas with Alloy. In: ABZ, *LNCS*, vol. 8477, pp. 303–308. Springer (2014)
17. Cunha, A., Macedo, N., Guimarães, T.: Target oriented relational model finding. In: FASE, *LNCS*, vol. 8411, pp. 17–31. Springer (2014)
18. Demri, S., Goranko, V., Lange, M.: Temporal Logics in Computer Science: Finite-State Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2016). DOI 10.1017/CBO9781139236119
19. Dominguez, A.L.J., Day, N.A.: Generating multiple diverse counterexamples for an EFSM. Tech. Rep. CS-2013-06, University of Waterloo (2013)
20. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: CAV, *LNCS*, vol. 2725, pp. 27–39. Springer (2003)
21. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: Upgrading Alloy with actions. In: ICSE, pp. 442–451. ACM (2005)
22. Ganov, S.R., Khurshid, S., Perry, D.E.: Annotations for Alloy: Automated incremental analysis using domain specific solvers. In: ICFEM, *LNCS*, vol. 7635, pp. 414–429. Springer (2012)
23. Hölldobler, S., Manthey, N., Nguyen, V.H., Stecklina, J., Steinke, P.: A short overview on modern parallel SAT-solvers. In: ICACSIS, pp. 201–206. IEEE (2011)
24. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997)

25. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, 2nd edn. MIT Press (2016)
26. Kromodimoeljo, S.: Controlling the generation of multiple counterexamples in LTL model checking. Ph.D. thesis, The University of Queensland (2014)
27. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 872–923 (1994)
28. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
29. Leuschel, M., Butler, M.J.: ProB: A model checker for B. In: FME, *LNCS*, vol. 2805, pp. 855–874. Springer (2003)
30. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE, pp. 373–383. ACM (2016)
31. Macedo, N., Cunha, A.: Alloy meets TLA$^+$: An exploratory study. CoRR **abs/1603.03599** (2016)
32. Macedo, N., Cunha, A., Guimarães, T.: Exploring scenario exploration. In: FASE, *LNCS*, vol. 9033, pp. 301–315. Springer (2015)
33. Macedo, N., Cunha, A., Pessoa, E.: Exploiting partial knowledge for efficient model analysis. In: ATVA, *LNCS*, vol. 10482, pp. 344–362. Springer (2017)
34. McCune, W.: Prover9 and Mace4 (2005–2010). `http://www.cs.unm.edu/~mccune/prover9/`
35. Meng, B., Reynolds, A., Tinelli, C., Barrett, C.W.: Relational constraint solving in SMT. In: CADE, *LNCS*, vol. 10395, pp. 148–165. Springer (2017)
36. Montaghami, V., Rayside, D.: Extending Alloy with partial instances. In: ABZ, *LNCS*, vol. 7316, pp. 122–135. Springer (2012)
37. Near, J.P., Jackson, D.: An imperative extension to Alloy. In: ASM, *LNCS*, vol. 5977, pp. 118–131. Springer (2010)
38. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE, pp. 232–241. IEEE (2013)
39. Plagge, D., Leuschel, M.: Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. Int. J. Softw. Tools Technol. Transf. **12**(1), 9–21 (2010)
40. Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: SIGSOFT FSE, pp. 908–919. ACM (2016)
41. Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM, *LNCS*, vol. 10951, pp. 568–587. Springer (2018)
42. Reynolds, A., Tinelli, C., Goel, A., Krstic, S.: Finite model finding in SMT. In: CAV, *LNCS*, vol. 8044, pp. 640–655. Springer (2013)
43. Rosner, N., Pombo, C.G.L., Aguirre, N., Jaoua, A., Mili, A., Frias, M.F.: Parallel bounded verification of Alloy models by TranScoping. In: VSTTE, *LNCS*, vol. 8164, pp. 88–107. Springer (2013)
44. Rosner, N., Siddiqui, J.H., Aguirre, N., Khurshid, S., Frias, M.F.: Ranger: Parallel analysis of Alloy models by range partitioning. In: ASE, pp. 147–157. IEEE (2013)
45. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. STTT **12**(2), 123–137 (2010)
46. Saeki, T., Ishikawa, F., Honiden, S.: Automatic generation of potentially pathological instances for validating Alloy models. In: ICFEM, *LNCS*, vol. 10009, pp. 41–56 (2016)
47. Saghafi, S., Danas, R., Dougherty, D.J.: Exploring theories with a model-finding assistant. In: CADE, *LNCS*, vol. 9195, pp. 434–449. Springer (2015)
48. Serna, J., Day, N.A., Farheen, S.: DASH: A new language for declarative behavioural requirements with control state hierarchy. In: RE Workshops, pp. 64–68. IEEE Computer Society (2017)
49. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. Electron. Notes Discret. Math. **9**, 19–35 (2001)
50. Siegel, A., Santomauro, M., Dyer, T., Nelson, T., Krishnamurthi, S.: Prototyping formal methods tools: A protocol analysis case study. In: Protocols, Logic, and Strands: Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday, LNCS. Springer (2021). To appear
51. Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction: A modeling idiom to enhance a lightweight formal method. In: ICFEM, *LNCS*, vol. 11852, pp. 336–352. Springer (2019)
52. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: ICST, pp. 264–275. IEEE (2017)
53. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS, *LNCS*, vol. 4424, pp. 632–647. Springer (2007)
54. Uzuncaova, E., Khurshid, S.: Constraint prioritization for efficient analysis of declarative models. In: FM, *LNCS*, vol. 5014, pp. 310–325. Springer (2008)
55. Vakili, A., Day, N.A.: Temporal logic model checking in Alloy. In: ABZ, *LNCS*, vol. 7316, pp. 150–163. Springer (2012)
56. Zhang, J., Zhang, H.: SEM: A system for enumerating models. In: IJCAI, pp. 298–303. Morgan Kaufmann (1995)
57. Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: FASE, *LNCS*, vol. 12076, pp. 29–52. Springer (2020)