Verifying temporal relational models with Pardinus*

Nuno Macedo[1,3], Julien Brunel[4,5], David Chemouil[4,5], and Alcino Cunha[1,2]

[1] INESC TEC, Porto, Portugal
[2] University of Minho, Braga, Portugal
[3] Faculty of Engineering of the University of Porto, Porto, Portugal
[4] ONERA DTIS, Toulouse, France
[5] Université fédérale de Toulouse, Toulouse, France

Abstract. This short paper summarizes an article published in the *Journal of Automated Reasoning* [7]. It presents Pardinus, an extension of the popular Kodkod [12] 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 counterexamples) 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 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 [12] is an example of such model finders, supporting a range of features that make it quite popular:

- Problems are described using the single 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.

* 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 by the French Research Agency project FORMEDICIS ANR-16-CE25-0007 and by the research project CONCORDE of the Defense Innovation Agency (AID) of the French Ministry of Defense (2019650090004707501).

- 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*, as 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.

Kodkod is implemented as a Java API and is designed to be a plugin that can easily be incorporated as a backend of another tool. Its best-known application is the analysis of Alloy 5 specifications. Alloy [6] 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, which is infeasible for most examples.

This paper presents the Pardinus model finder, an extension of Kodkod that addresses this limitation. It allows the declaration of mutable relations and the specification of properties in *temporal relational logic*, 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 [1]; 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) [10], using the concrete SMV syntax [4].

The main application of Pardinus is in the analysis of Alloy 6[6] specifications. This new version of Alloy adds support for mutable relations and temporal relational logic, an extension previously known as Electrum [8,2]. The architecture of Alloy 6 and Pardinus is depicted in Fig. 1, with the scope of this paper captured by thick lines and arrows. Pardinus is also used as a backend in Forge [11], a system to prototype formal methods tools.

This article summarises [7], which 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 [8] briefly mentions how specifications can be model checked, but at the time Pardinus

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

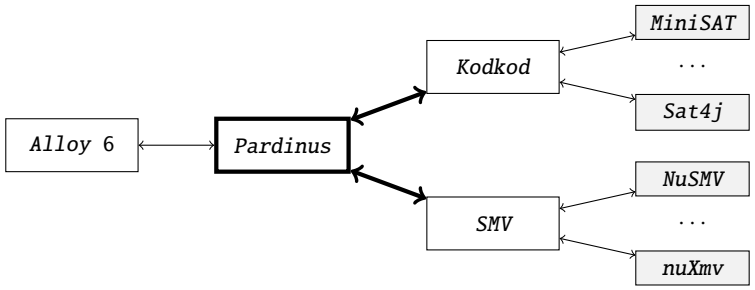

Fig. 1: Alloy 6 and Pardinus architecture

did not exist and the two backends were not unified. In a more recent tool paper about the current version of the Electrum Analyzer [2], 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 [3], 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. This technique was first introduced in [9], 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.

In the rest of this paper, we present how a Pardinus problem is described, taking a protocol as an illustrating example. We then briefly mention the different analyses which are performed by Pardinus. Details can be found in [7].

2 A Pardinus Problem

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 the tuples that may be present in the relation. Tuples are sequences of *atoms* (uninterpreted identifiers) drawn from a finite universe, that must also be declared upfront. A relation can also have a lower-bound, which is useful to capture *a priori* partial knowledge about the solution. Pardinus problems extend Kodkod ones as follows:

– 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.
– A relational expression in a formula can be primed to denote its value in the succeeding time instant.

```
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),..,(I2,I3)} {(I0,I1),..,(I2,I3)}
5      Process :1 {} {(P0),(P1),(P2),(P3)}
6      id      :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),..,
7                    (P3,I0),(P3,I1),(P3,I2),(P3,I3)}
8      succ    :2 {} {(P0,P0),(P0,P1),(P0,P2),(P0,P3),..,
9                    (P3,P0),(P3,P1),(P3,P2),(P3,P3)}
10 var outbox  :2 {} {(P0,I0),(P0,I1),(P0,I2),(P0,I3),..,
11                   (P3,I0),(P3,I1),(P3,I2),(P3,I3)}
12 var Elected :1 {} {(P0),(P1),(P2),(P3)}
13
14 id in Process → Id and
15 all p : Process | one p.id and
16 all i : Id | lone id.i and
17 succ in Process → Process and
18 all p : Process | one p.succ and
19 all p : Process | Process in p.^succ and
20
21 outbox = id and
22 always some p : Process, i : (succ.p).outbox |
23   outbox' = outbox − succ.p → i + p → (i − ^next.(p.id)) and
24
25 always Elected = {p : Process |
26     once (p.id in p.outbox and before not (p.id in p.outbox))}
```

Fig. 2: A leader election protocol in Pardinus

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. As an illustration, we consider the specification of a leader election protocol shown in Fig. 2. This protocol, first proposed by Chang and Roberts [5], assumes a ring network of processes (or nodes) with unique comparable identifiers.

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, with `next` fixing a particular total order between the four possible identifiers. Then `Process` is restricted to be any subset of the four possible process atoms (recall that 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. Due to space constraints, we do not detail the logic here as it is essentially that of Alloy 6. The specification of the ring topology consists of a conjunction of six sub-formulas (ll. 14-19) over the immutable relations.

Specifying behaviour (ll. 10–12, 21–26) The remaining of the problem specifies the evolution of the protocol. 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 [6], 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 of LTL.

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 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.

Analyzing the problem If a problem is satisfiable, as in this example, Pardinus returns a solution. Additionally, in order to check a particular temporal property, one should add its negation to the problem to try to find a solution, also called counterexample in this case. If none is found, the property is valid for the specified bounds.

The key safety property of this protocol is that at most one leader is elected, which can be specified as **`always lone`** `Elected`, or as the stronger formula **`always all`** `p : Elected |` **`always`** `Elected` **`in`** `p`, which forbids different processes to be considered elected at different points in time. To be useful, the protocol should also ensure that some leader is elected. This liveness property can be specified as **`eventually some`** `Elected`.

3 Iteration on Solutions

As mentioned earlier, once a solution (or a counterexample) is computed by Pardinus, a mechanism allows for iteration over the set of solutions (or counterexamples). The Kodkod approach (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. Thus, Pardinus implements different navigation operations that focus on modifying different aspects of the path. To be efficient, these operations are directly implemented at the solver level, and also incorporate a symmetry breaking mechanism.

As an illustration, let us consider the leader election protocol illustrated in Section 2. Suppose that in the first solution that is computed by Pardinus, the set `Process` consists of the single process `P0`, the relation `succ` is a self-loop, *i.e.*, `succ = {(P0, P0)` and `P0` repeatedly sends its own identifier to itself. The user may ask Pardinus for another solution, having a different configuration. This returns a solution with a different number of processes. Notice that a solution with a single process different from `P0` would also correspond to a different configuration but is considered as symmetrical to the first solution, and is thus pruned out by Pardinus. Suppose that Pardinus provides a new solution in which `Process = {P0, P1, P2}` and `succ = {(P0,P1),(P1,P2),(P2,P0)}`. Suppose that in this solution, there are seven different states before a process is elected whereas the user wants to exhibit the shortest possible scenario to elect a process

(*i.e.*, with four different states in this case). The user may now ask for another solution with the same configuration. Pardinus then computes a solution where `Process` and `succ` are the same, but where the behavior, *i.e.* the sequence of operations executed by the processes, differs. As soon as the returned solution does not correspond to the scenario that the user has in mind, a new solution having the same configuration can be requested. If such a solution exists, it will necessarily be returned by Pardinus.

4 Parallel Decomposition

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. Evaluation shows that in certain contexts, 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.

5 Evaluation

We evaluated the scalability of Pardinus for the complete and bounded backends, the parallel decomposed strategy, and the iteration operations, with multiple variants of 6 different Pardinus problems.

Both the SAT and SMV bounded backends scaled to considerable model sizes and maximum trace lengths. The SAT backend seems to scale better with increasing model size, particularly for satisfiable problems. The SMV procedures do not seem to have considerable gains for satisfiable problems. As expected, the complete SMV backend performed worse, but closes on the performance of the bounded backends as the considered maximum trace length increases. This supports the application of complete analysis when enough confidence is obtained from the bounded backends. The parallel strategy shows considerable gains for satisfiable problems, particularly for the bounded and complete SMV backends. The gains for unsatisfiable ones are not as consistent, but the SMV backends seem to benefit more from it. A hybrid approach tames the negative outliers while preserving the gains otherwise.

Both iteration operations have shown to be feasible for interactive sessions with both strategies, although configuration iteration seems to be affected by the number of valid configurations. Configuration iteration performed better in the non-parallel approach, while path iteration fared better with the parallel one.

6 Conclusion

The full article [7] expands in detail on the topics mentioned before and also adds a substantial evaluation section, answering several research questions and demonstrating the relevance of the techniques implemented in Pardinus.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
2. 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)
3. 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)
4. 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`
5. 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)
6. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, 2nd edn. (2016)
7. Macedo, N., Brunel, J., Chemouil, D., Cunha, A.: Pardinus: A temporal relational model finder. Journal of Automated Reasoning **66**, 861–904 (2022)
8. 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)
9. Macedo, N., Cunha, A., Pessoa, E.: Exploiting partial knowledge for efficient model analysis. In: ATVA. LNCS, vol. 10482, pp. 344–362. Springer (2017)
10. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. STTT **12**(2), 123–137 (2010)
11. 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
12. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS. LNCS, vol. 4424, pp. 632–647. Springer (2007)