Model checking of hyperproperties for high-level relational models

NUNO MACEDO*, Universidade do Minho & INESC TEC, Portugal
HUGO PACHECO*, Universidade do Minho & INESC TEC, Portugal

Many properties related to security or concurrency must be encoded as so-called hyperproperties, temporal properties that allow reasoning about multiple traces of a system. However, despite recent advances on model checking hyperproperties, there is still a lack of higher-level specification languages that can effectively support software engineering practitioners in verifying properties of this class at early stages of system design.

Alloy is a lightweight formal method with a high-level specification language that is supported by automated analysis procedures, making it particularly well-suited for the verification of design models at early development stages. It does not natively support, however, the verification of hyperproperties.

This work proposes HyperPardinus, a new model finding procedure that extends Pardinus – the temporal logic backend of the Alloy language – to automatically verify hyperproperties over relational models by relying on existing low-level model checkers for hyperproperties. It then conservatively extends Alloy to support the specification and automatic verification of hyperproperties over design models, as well as the visualization of (counter-)examples at a higher-level of abstraction. Evaluation shows that our approach enables modeling and finding (counter-)examples for complex hyperproperties with alternating quantifiers, making it feasible to address relevant scenarios from the state of the art.

CCS Concepts: • **Software and its engineering** → **Model checking**; **Specification languages**; • **Security and privacy** → *Logic and verification.*

Additional Key Words and Phrases: model checking, hyperproperties, formal software design, security

1 Introduction

Many relevant system properties related to information flow, concurrency or robustness cannot be expressed by looking at individual execution traces of a system, and require reasoning about the relationship between multiple executions. This so-called class of *hyperproperties* [8] generalizes traditional trace properties to sets of traces and has seen increased applications across different research areas.

There are many widely used tools for checking specific hyperproperties for particular domains [27], but work on checking properties specified in general hyperlogics is more recent. For that purpose, classical temporal logics have been extended with operators that allow universal and existential quantification over traces, such as the extension of *Linear-time Temporal Logic* (LTL) known as HyperLTL. A recent overview on the landscape of hyperlogics, their expressive power and algorithms is given in [15]. The decidability of model checking for HyperLTL has led to the development of several model checking tools for hyperlogics [4, 5, 17, 21]. Other recent research focused on the deductive verification of imperative programs with respect to general hyperproperties, often requiring user guidance [1, 10, 11, 23].

Research on model checking HyperLTL properties – the focus of this work – remains vibrant with algorithmic advances and novel model checking procedures, but has unfortunately not yet reached the point where general high-level modeling tools can support the verification of properties written in hyperlogics. All existing model checkers for hyperlogics [4, 17, 21] work with compact, machine-readable formats common in the model checking community for describing automata, circuits or state machines, such as fragments of the SMV language [33]. They support HyperLTL

*Both authors contributed equally to this research.

Authors' Contact Information: Nuno Macedo, nmacedo@di.uminho.pt, Universidade do Minho & INESC TEC, Braga, Portugal; Hugo Pacheco, hpacheco@di.uminho.pt, Universidade do Minho & INESC TEC, Braga, Portugal.

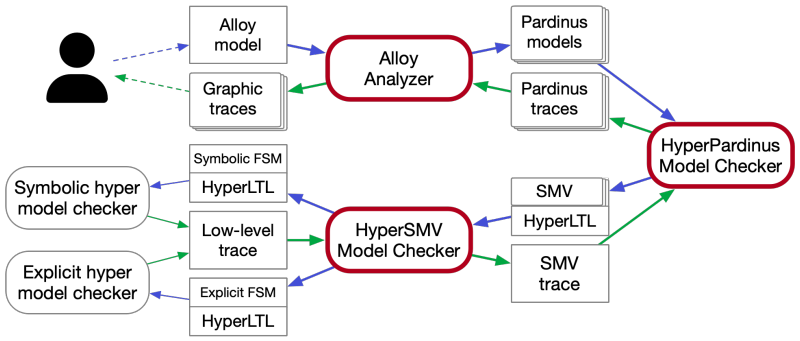

Fig. 1. Overview of the proposed toolchain. Rectangles are interchangeable formats, rounded rectangles software components, red identifies contributions.

formulas written over such representation, which requires users to reason at a very low level of abstraction. Moreover, automatic verification of hyperproperties involving alternations is particularly challenging [4, 22], and many approaches, such as MCHyper [17] or the approach proposed by Lamport and Schneider [29] based on TLA$^+$ [28], require user input to generate witnesses for existential quantifications. As a consequence, there is a considerable gap between the conceptual model of the system a user expects to verify, and the artefacts that have to be developed for these tools, hindering their adoption. In order to reach the wider software engineering community, there is a need for modeling and specification languages that act at a higher-level of abstraction, as well as verification tools that are able to handle such representations, either through novel procedures or translations into lower-level ones.

Alloy [25] is a high-level formal specification language that has become popular due to its simplicity and flexibility, whose Analyzer quickly provides intuitive feedback. As of version 6 [32][1], Alloy supports models with mutable elements, whose values can change over time, and the specification and verification of LTL properties. The Alloy Analyzer provides a GUI through which users write specifications, run analysis commands, and interpret feedback graphically, but actual analysis is decoupled from the frontend and performed by a backend *relational model finder*, Pardinus [31]. Such model finders provide a higher-level interface for the analysis of formal specifications, themselves typically relying on lower-level solvers. In the case of Pardinus, it supports the same relational temporal logic as Alloy 6, stripped away of certain syntactic constructs, and provides two alternative analysis procedures: one reducing to SAT solving through the relational model finder Kodkod [36], and another reducing to SMV and relying on off-the-shelf symbolic model checkers for LTL. Unfortunately, the whole Alloy toolchain focuses on the analysis of LTL properties for individual traces, rendering the analysis of arbitrary hyperproperties impossible.

In this paper, we extend Pardinus and the surrounding ecosystem to allow for the automatic verification of hyperproperties and the inspection of (counter-)examples for models written in a high-level specification language[2], whose overview is shown in Fig. 1. Specifically, our main contributions are:

- HyperPardinus, a high-level model finding tool for *hyper relational temporal logic*, a hyper generalization of the underlying logic of Alloy 6 supporting a superset of HyperLTL properties with flexible occurrences of trace quantifiers. As far as we know, this is the first model checking tool for a first-order hyperlogic.

[1] github.com/AlloyTools/org.alloytools.alloy/releases/tag/v6.0.0
[2] The tools and benchmarks are publicly available at https://github.com/haslab/HyperPardinus-benchmarks.

- HyperSMV, a SMV model checking tool that integrates with both explicit-state and symbolic state-of-the-art model checkers for HyperLTL, significantly improving the expressiveness of supported SMV models and the scalability of the verification process.
- A minimal extension to Alloy 6 and its Analyzer, to support analysis with HyperPardinus and visualization of multiple trace (counter-)examples, allowing the verification of hyperproperties for high-level models.
- An evaluation of our toolchain by modeling and verifying case studies that showcase hyperproperties from diverse domains, including low-level models from the literature and high-level models specifically crafted to evaluate its ability to handle more complex models.

The rest of the paper is organized as follows. Section 2 motivates the approach with an example. Section 3 discusses related work on hyperproperties. Section 4 formalizes the notion of hyper model finding problem, and Section 5 presents the HyperPardinus model finder for such problems. Section 6 presents the lower-level model checker HyperSMV. Section 7 presents the extended Alloy Analyzer that supports analysis through HyperPardinus. Section 8 presents the evaluation of HyperPardinus and HyperSMV against the state of the art. Section 9 concludes the paper and points direction for future work.

2 Motivating example

Conference management systems (CMSs) such as EasyChair or HotCRP are expected to enforce certain security policies, and there has been quite some work on using hyperproperties to model and verify their security. Famously, CoCon [35] has found bugs in production systems using a theorem prover to check confidentiality properties, expressed in a variant of *nondeducibility*. This has made CMSs a target of work on hyperproperties, such as in the verification of CMS workflows written in a multi-agent framework [16, 34], where special emphasis is placed on workflows with loops and different restrictions on agent behavior.

Following this inspiration, we start by defining a high-level Alloy model of a CMS that will illustrate our approach. Later, we will use this same example to highlight the limitations of the state of the art. This abstract version considers assignment of articles to reviewers, and we wish to analyze whether the process of submitting reviews and reaching decisions on submissions ensures fine-grained confidentiality policies.

2.1 Modeling a CMS

The `CMS` model presented in Fig. 2 is a possible specification of a CMS in our extended Alloy, whose syntax is mostly conformant with regular Alloy. In an Alloy model, structure is introduced by the declaration of *signatures* (keyword **`sig`**)—defining the entities of the model—and *fields* declared within signatures—representing the relationships between entities. In ll. 1–3, signatures are declared for the main entities of our model: reviewers, articles and decisions. During solving, signatures are assigned abstract, uninterpreted, *atoms* according to a scope defined by the user. For this model, the user will have to set a scope for reviewers and articles, since decisions are declared as an exact enumeration signature. Additionally, we introduce a distinguished reviewer, `Agent` (a sub-signature with multiplicity **`one`**), to define security properties in his perspective. Signature `CMS` then encapsulates the content of a CMS: it `assigns` a set of articles to reviewers, registers `reviews` which are reviewer-proposed decisions for articles, and notifies final `decisions` made by the editor on articles. In Alloy, fields encode relations with arbitrary arity and can have multiplicity constraints imposed. Declaring `assigns` (l. 6) within `CMS` with type `Reviewer` **`some`** → **`some`** `Article` results in a ternary relation between `CMS`s, `Reviewers` and `Articles`, with the additional constraints that there are no articles without reviewers nor reviewers without articles assigned (literally, for all

```
1  sig Reviewer, Article {}
2  one sig Agent in Reviewer {}      // a designated reviewer whose view is used in the NI/GNI properties
3  enum Decision { Reject, Major, Accept }
4
5  trace sig CMS { // a CMS includes static assignment of reviewers to papers and dynamic paper reviews
6    assigns : Reviewer some → some Article,
7    var reviews : Article → Reviewer → lone Decision,
8    var decisions : Article → lone Decision }
9
10 pred cms[s:CMS] {                                    // determine valid traces of a CMS
11   some Article - Agent.(s.assigns)                   // there are articles not assigned to agent
12   no s.reviews and no s.decisions                    // initial state
13   always some r:Reviewer, a:Article, d:Decision |    // possible events at each state
14     review[s,r,a,d] or decide[s,a,d] or stutter[s]
15   eventually s.decisions.Decision = Article }        // force eventual decisions
16
17 pred review[s:CMS, r:Reviewer, a:Article, d:Decision] {  // reviewer submits a review
18   a in r.(s.assigns)                                 // guard: a is assigned to r
19   no r.(a.(s.reviews))                               // guard: r has not submitted a review for a
20   s.reviews' = s.reviews + a→r→d                     // effect: add the review for this article
21   s.decisions' = s.decisions }                       // frame condition: decisions unchanged
22
23 pred decide[s:CMS, a:Article, d:Decision] {         // the editor makes a decision
24   a not in s.decisions.Decision                      // guard: editor has not decided on a
25   a.(s.reviews).Decision = s.assigns.a               // guard: all a reviews submitted
26   d in criteria[s,a]                                 // guard: d is according to specified criteria
27   s.decisions' = s.decisions + a→d                   // effect: add the decision for a
28   s.reviews' = s.reviews }                           // frame condition: reviews unchanged
29
30 pred stutter[s:CMS] {                               // the system stutters (does not change the state)
31   s.reviews' = s.reviews and s.decisions' = s.decisions }
32
33 fun low[s1, s2:CMS] : set Article {                 // articles are low security if assigned to agent
34   Agent.(s1.assigns + s2.assigns) }
35
36 fun high[s1, s2:CMS] : set Article {                // articles are high security if not assigned to agent
37   Article - low[s1,s2] }
38
39 pred same_assigns[s1, s2:CMS, arts:set Article] { // whether arts have the same assignments in two CMS
40   s1.assigns :> arts = s2.assigns :> arts }
```

Fig. 2. The CMS system in extended Alloy

`Reviewers` there are **`some`** `Articles` in `assigns`, and vice-versa); in contrast, `reviews` (l. 7) is a quaternary relation between `CMS`s, `Articles`, `Reviewers` and `Decisions`, where a reviewer submits at most one decision per article (**`lone`**). Our traces start at a point where articles have already been assigned to reviewers, so `assigns` is declared as a static relation (the default); in contrast, `reviews` and `decisions` will change as the systems progresses, and thus are declared as mutable (**`var`**), one of the temporal features introduced in Alloy 6. This signature is explicitely marked as a **`trace`**, making it prone to be quantified in a hyperproperty, as we shall see in the next section.

Once the structure of the model is defined, the next step is to specify valid `CMS` traces. In our model this is encoded by the auxiliary **`pred`**icate `cms` (ll. 10–15). Alloy is based on relational logic, and includes set operators such as union (`+`) and difference (`-`), and relational join (`.`) that allows composition of relations with arbitrary arity. For example, expression `Agent.assigns` is a unary relation (i.e., a set) containing all articles related with `Agent` through `assigns`, while `Article - Agent.assigns` represents all articles not assigned to `Agent`. Atomic formulas are created through inclusion tests (**`in`**) or cardinality tests, which are then combined through Boolean

```
1  assert NI {
2    all s1,s2:CMS | cms[s1] and cms[s2] and aligned[s1,s2,low[s1,s2]] implies
3      (same_assigns[s1,s2,low[s1,s2]] and always same_reviews[s1,s2,low[s1,s2]]) implies
4        always same_decisions[s1,s2,low[s1,s2]]
5    }
6
7  assert GNI {
8    all s1,s2:CMS | cms[s1] and cms[s2] and aligned[s1,s2,low[s1,s2]] implies some s3:CMS {
9      cms[s3]
10     same_assigns[s1,s3,low[s1,s3]] and always same_reviews[s1,s3,low[s1,s3]]
11     always same_decisions[s1,s3,low[s1,s3]]
12     same_assigns[s2,s3,high[s2,s3]] and always same_reviews[s2,s3,high[s2,s3]]
13   } }
```

Fig. 3. Hyperproperties in extended Alloy over the CMS system

operators, quantifications, and temporal operators. The `cms` constraint in l. 11 thus states that there must exist **`some`** articles that are not assigned to `Agent`, as the security properties will become vacuous when the `Agent`'s view of the system is empty. The one in l. 12 restricts the initial state to be empty: properties over mutable elements outside a temporal operator apply to the first state. The one in ll. 13–14 states how the system is allowed to evolve: at any state (**`always`**), either a review is submitted, the editor makes a decision, or the system stutters (does not change its state). These events are specified as auxiliary predicates that specify how succeeding states can be related, through formulas that compare the current state with the next using primed expressions (`'`). Finally, we force a decision to be **`eventually`** reached on every article (l. 15).

At this point, this could be a regular Alloy model, and we could execute commands to animate and validate this model, such as the one below that asks for a valid `CMS` with up to 3 elements in each signature. In Alloy all execution traces are infinite, represented as a finite prefix of the trace with a looping segment, and by default are obtained through bounded model checking (BMC).

```
run example { some s:CMS | cms[s] } for 3 but 1 CMS
```

To postulate security properties over our model, we also need to specify which information is considered confidential. The `Agent` in our model will serve precisely this role. We assume that articles assigned to the `Agent` are low-security (i.e., public), and dually that all other articles are high-security (i.e., secret). This security policy is encoded in **`fun`**ctions `low` and `high`, which declaratively select the joint public/secret view of the `Agent` in two `CMS`s. The policy extends to associated article information (assignments, reviews and decisions), as encoded in the remaining **`pred`**icates. For example, `same_assigns` (ll. 39–40) defines when two `CMS`s have the same assignments for a set of articles (operator `:>` restricts the range of a relation to a certain set).

2.2 Specifying & verifying hyperproperties

The model from Fig. 2 is written in the temporal dialect of Alloy. Still, it is impossible for Alloy to test hyperproperties relating multiple traces, as it can only test whether a temporal property holds in each individual trace. To allow higher-order quantifications, we introduced a new annotation **`trace`** for signatures, highlighted in red: quantifications over such signatures are lifted into quantifications over traces. `CMS` is marked as one such signature in Fig. 2 (l. 5).

Consider, for instance, a classical security hyperproperty, known as *noninterference* (NI), postulated as a relation on traces: any two traces that have the same low inputs must also have the same low outputs. It declaratively encodes the fact that the system appears deterministic to a low user (our `Agent`), in the sense that low outputs can only depend on low inputs. This is encoded for

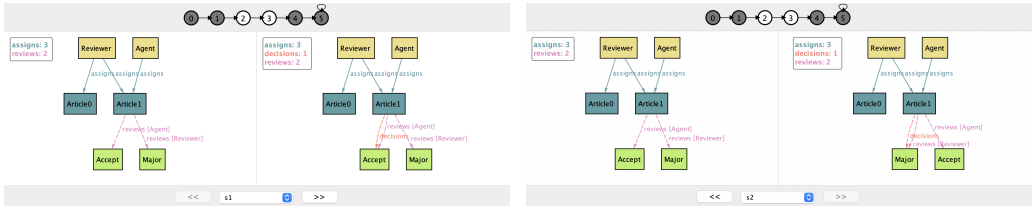

Fig. 4. Possible counter-example for the NI property of CMS

CMS as shown in ll. 1–4 of Fig. 3. We consider article assignments and reviews to be inputs of the system, and the editor's decision to be the outputs[3].

Hyperproperties that use temporal operators to compare events occurring in multiple traces require such traces to be aligned at least in the occurrence of low events; if the system is allowed to stutter, this still allows an arbitrary number of high events to occur between low events (see [29] for a deeper discussion). Thus, NI literally asserts that for any two CMS traces that behave according to cms and are aligned on low events, if the low inputs are the same at every point of the traces, so must be the low outputs.

To check the NI assertion, we can write a command with the desirable scopes, such as the following for up to 3 reviewers and articles.

```
check NI for 3 Reviewer, 3 Article
```

Whether this property holds for CMS depends on the behavior of the editor when making a decision. To explore the consequences of alternative designs, namely which information the editor has available to make a decision, in CMS (l. 26) this behavior was left as an auxiliary **fun**ction `criteria[s:CMS, a:Article]` that determines the possible decision an editor may take for an article. For the NI property to hold, an idealized editor would have to make the same decision with the same information, hence looking deterministic to the Agent, who knows all reviews and decisions of articles assigned to him. Let us model an optimistic editor that always chooses the best decision from those submitted by reviewers to the given article a, which could be expressed as follows.

```
fun criteria[s:CMS, a:Article] : set Decision {
  max[Reviewer.(a.(s.reviews))] }
```

Here, expression `a.(s.reviews)` retrieves decisions submitted by reviewers for article a in CMS s. `Reviewer.(a.(s.reviews))` then retrieves all submitted decisions regardless of the reviewer; elements of an enumeration signature are ordered in the order they are declared, so we can then select the maximum decision.

Our tool will report the **check** above to be valid, within the specified scope. However, we stated that any non-determinism on the low output would invalidate NI. To observe this, we could allow an editor to arbitrarily select a decision from those submitted, changing criteria to the following

```
fun criteria[s:CMS, a:Article] : set Decision {
  Reviewer.(a.(s.reviews)) }
```

Our extension of Alloy will now quickly report NI to be invalid and provide a counter-example, which for NI is composed by traces s1 and s2. Figure 4 shows how such a counter-example can be inspected in the visualizer of the Alloy Analyzer. Observing this counter-example, the Agent

[3]Note that this abstract system naturally supports reactive behavior, e.g., a reviewer can submit his review after viewing his assignments.

knows the two decisions for `Article1` (his own `Accept` and `Reviewer`'s `Major`); nonetheless, the fact that the editor selected differently among the two in the two traces `s1` and `s2` is sufficient to invalidate `NI`.

A more complex confidentiality hyperproperty is *generalized noninterference* (GNI), which relaxes NI to allow non-deterministic behavior, by postulating that any observable low behavior must be compatible with any sequence of high inputs. It can be formalized by stating that, for any two traces of the system, there must exist a third trace that combines the low information from the first and the high information from the second. This could be encoded for `CMS` as shown in ll. 6–10 of Fig. 3. Our extension of Alloy will report `GNI` as valid for both `criteria` functions defined above.

Intuitively, this shows that non-determinism is the sole explanation for differences in the output. To break `GNI`, the editor would have to consult secret information – for instance, to enforce a quota of accepted articles – to make the decision on an article. To observe this, we could let the editor select the decision not only from article `a`, but from all articles in the system. If `criteria` is changed to the following, our tool will then report `GNI` to be invalid.

```
fun criteria[s:CMS, a:Article] : set Decision {
  Reviewer.(Article.(s.reviews)) }
```

As this example illustrates, our lightweight extension to the Alloy language enables the specification of properties in a first-order hyperlogic over high-level relational models. This significantly increases expressiveness compared to the languages supported by current state-of-the-art tools, as shall become clear in Section 8.2. At the same time, the added expressiveness can impose considerable analysis costs. The remainder of the paper presents a pipeline for analyzing these abstract representations using state-of-the-art lower-level solvers.

3 Related work

3.1 Model checking

HyperLTL [7], as the most studied logic for expressing hyperproperties, has seen the development of various general verification methods in recent years (see [15] for an extensive review). This vibrant landscape includes various recent model checking tools, which are of particular interest to this paper.

The seminal MCHyper [17] supports the verification of alternation-free HyperLTL via model composition, but requires explicit witnesses to resolve existential quantifications. Strategy-based verification, where players of a game represent distinct traces of a joint property, can be used to support quantifier alternations [2]. HyperQB [21] supports BMC for HyperLTL by reducing to QBF (*Quantified Boolean Formula*) solving, but is incomplete as it is limited to finite, non-looping, prefixes of infinite traces. The same authors [22] propose a method with loops for one quantifier alternation, which is not integrated into HyperQB. Models are provided in SMV but counter-examples provided at the QBF level. AutoHyper [4, 5] is an explicit-state automata-based model checker which supports complete verification of *Hyper Quantified Propositional Temporal Logic* (HyperQPTL), in which trace and propositional quantifiers can be freely interleaved. Models are provided in a non-declarative subset of SMV or a low-level representation of an explicit-state system; counter-examples are generated at the level of models.

In terms of expressiveness, these tools [4, 17, 21] support HyperLTL or HyperQPTL restricted to prenex form (trace quantifiers appear in outermost positions of the formula). Our approach flexibly blends trace quantifications with *First-Order Logic* (FOL), and is a natural hyper generalization of *First-Order LTL* (FOLTL).

The visualization of hyperproperty counter-examples is a challenge on itself. HYPERVIS [19] is one of the first to address this, providing explanations for hyperproperty violations as minimal

causative sub-formulas [9]. Nonetheless, note that all of the mentioned model checkers work with low-level models, while ours support specification at a much higher level of abstraction. This becomes even more evident when counter-examples of tools like HyperQB are the raw transcript of the underlying QBF solver. In contrast, HyperPardinus is fully integrated into the Analyzer, which provides high-level counter-examples via its graphical user interface.

3.2 Deductive verification

Beyond model checking, significant effort has been dedicated to the deductive verification of imperative programs with respect to hyperproperties. These approaches generally navigate a trade-off between the expressiveness of the property language and the degree of automation.

On the highly expressive end, Hypra [11] introduces *Hyper Hoare Logic*, a logic capable of verifying general hyperproperties. However, this expressiveness comes at the cost of automation, often requiring manual proof guidance or detailed invariants from the user. To improve automation, ForEx [1] targets a specific subset of properties known as *hyperliveness*, by restricting the specification language to a less liberal ∀-∃ *Hoare Logic*.

Another class of approaches reframes the problem of hyperlogic verification to game-based interpretations. To capture asynchronous behaviors, HyPA [3] enables the verification of *Observation-based HyperLTL* (OHyperLTL). This approach aligns traces based on observable events rather than strict ordering, allowing more natural verification of properties where traces proceed at different speeds. HyHorn [23] focuses on a subset of OHyperLTL with one quantifier alternation over LTL safety properties. By reducing verification to the satisfiability of *Constrained Horn Clauses* (CHCs), and using off-the-shelf CHC solvers, it has shown to outperform HyPA in many benchmarks. In contrast, Hy∃n∀ [10] considers properties in a subset of OHyperLTL with arbitrary quantification over LTL safety properties, and employs symbolic execution to automate the search for violations (or "hyperbugs") in asynchronous programs.

These tools, much like traditional deductive verification, focus primarily in reasoning at the level of the operational semantics of imperative programs, often requiring user-provided invariants. We take a fundamentally different approach, and focus in the specification and automated verification of hyperproperties for high-level, declarative models expressed directly in first-order logic.

4 Hyper relational model finding

The analysis procedures of the Alloy Analyzer are powered by a relational temporal model finder, Pardinus [31] (itself an extension of the static model finder Kodkod [36]). This section formalizes hyper relational model finding problems as an extension of the non-hyper counterpart, allowing for the animation and verification of hyperproperties over relational models.

A (non-hyper) model finding *problem* $P = \langle \mathcal{A}, \mathcal{D}, \phi \rangle$ is composed of a universe $\mathcal{A}$ of uninterpreted atoms; the declaration $\mathcal{D}$ of a set of free relations $\mathcal{R}$ with upper- and lower-bounds restricting their possible values; and a relational temporal logic constraint ϕ over $\mathcal{R}$ variables that is expected to hold for the generated instances[4]. A *trace* $I : \mathbb{N} \to \mathcal{R} \mapsto \mathbb{P}(\mathcal{A}^*)$ is an infinite sequence of states assigning to each relation a tuple set. A trace I is an instance of a problem P, $I \models P$, iff the relation bounds and the constraint hold, whose semantics are detailed next. A hyper model finding problem builds on this formalism for non-hyper problems. It considers a set of non-hyper problems P_i over which traces will be quantified, and an additional problem H with a hyper relational temporal logic constraint Φ whose trace quantifications range over P_i problems; thus, Φ may refer to both the relations declared in H and those from models P_i.

[4]Model finders search for instances of a problem, so checking the validity of a property ψ amounts to creating a problem that searches for (counter-)examples that break it. This is done by negating ψ in the problem's constraint ϕ.

Definition 4.1 (Hyper model finding problem). For a domain of discourse $\mathcal{A}$, a *hyper model finding problem* is composed of

- a set of problems $P_i = \langle \mathcal{A}, \mathcal{D}_i, \phi_i \rangle$, where $\mathcal{D}_i$ declares a set of relations $\mathcal{R}_i$ bounded by tuples from $\mathcal{A}$ and ϕ_i is a (non-hyper) relational temporal logic constraint over $\mathcal{R}_i$;
- a problem $H = \langle \mathcal{A}, \mathcal{D}, \Phi \rangle$, where $\mathcal{D}$ declares a set of relations $\mathcal{R}$ bounded by tuples from $\mathcal{A}$ and Φ is a hyper relational temporal logic constraint over $\mathcal{R} \cup \bigcup \mathcal{R}_i \cup \bigcup \{P_i\}$.

The verification of the NI hyperproperty for the CMS model from Section 2, for instance, would result in two non-hyper models, each a copy of `CMS` trace signature and the restriction of their behaviour by predicate `cms`, and a hyper model finding problem with the hyperproperty defined in assertion `NI`. The translation of the higher-level Alloy model into this is addressed in Section 7.

4.1 Relation bounds

Syntax. The declaration of each free relation $r \in \mathcal{R}$ bounds the possible values that r may take, namely through a lower-bound denoting tuples that *must* belong to r, and an upper-bound, denoting tuples that *may* belong to r. In Pardinus, bounds are either tuple sets or an expression referring to other relations [31]. We have extended the upper-bounds to support *multiplicity constraints* over arbitrary n-ary relations. These are available at the Alloy level (as can be seen in Fig. 2, ll. 6–8), but are expanded before reaching Pardinus; by supporting them natively, we are able to generate finer state machines for the backends[5].

Definition 4.2 (Relation bounds). The declaration of an n-ary relation $r \in \mathcal{R}$ has the shape $\square\ r :_n L, U$, where $\square$ is either `static` or `mutable`, denoting whether the value of r can change over time, and L and U are the lower- and upper-bounds of r, defined as:

$$
\begin{array}{lcl}
L & ::= & \mathbb{P}(\mathcal{A}^n) \mid \Gamma \\
U & ::= & \mathbb{P}(\mathcal{A}^n) \mid \mathit{conjunction} \\
\mathit{conjunction} & ::= & \mathit{conjunction} \wedge \mathit{arrow} \mid \mathit{arrow} \\
\mathit{arrow} & ::= & \mathit{arrow}\ \mathit{mult}\ \rightarrow\ \mathit{mult}\ \mathit{arrow} \mid \mathit{mult}\ \Gamma \\
\mathit{mult} & ::= & \textbf{set} \mid \textbf{lone} \mid \textbf{some} \mid \textbf{one} \mid \textbf{no}
\end{array}
$$

where Γ is any relational expression over variables with constant bounds.

Roughly, multiplicity bounds specify a relation between atoms of the domain and the range of a relation. For instance, a multiplicity constraint `A lone → one B` defines an injective function from `A` to `B`: it must relate every `A` atom with exactly one `B` atom, and every `B` atom with at most one `A` atom. For a relation of arity > 2, this process is applied inductively, and the conjunction of multiple multiplicity constraints is supported. The syntax and semantics of relational expressions Γ are presented in the next section. For instance, the `CMS` model from Section 2 could be represented by the following declarations, assuming a scope of exactly 2 reviewers and 2 articles that would result in the universe of atoms $\mathcal{A} = \{\mathsf{A}_0, \mathsf{A}_1, \mathsf{R}_0, \mathsf{R}_1, \mathsf{D}_0, \mathsf{D}_1, \mathsf{D}_0\}$.

`static Article`	$:_1$	$\{(\mathsf{A}_0), (\mathsf{A}_1)\}$	$\{(\mathsf{A}_0), (\mathsf{A}_1)\}$
`static Reviewer`	$:_1$	$\{(\mathsf{R}_0), (\mathsf{R}_1)\}$	$\{(\mathsf{R}_0), (\mathsf{R}_1)\}$
`static Agent`	$:_1$	$\{(\mathsf{R}_0)\}$	$\{(\mathsf{R}_0)\}$
`static Decision`	$:_1$	$\{(\mathsf{D}_0), (\mathsf{D}_1), (\mathsf{D}_2)\}$	$\{(\mathsf{D}_0), (\mathsf{D}_1), (\mathsf{D}_2)\}$
`static assigns`	$:_2$	$\{\}$	`Reviewer some → some Article`
`mutable reviews`	$:_3$	$\{\}$	`Article → Reviewer → lone Decision`
`mutable decisions`	$:_2$	$\{\}$	`Article → lone Decision`

[5] Pardinus already had preliminary, undocumented, support for this, but we have extended it to richer multiplicities of arbitrary arity.

Semantics. For a trace I, $I \models \mathcal{D}$ holds iff for every $r \in \mathcal{R}$ and $i \in \mathbb{N}$, $L(r) \subseteq I(i)(r) \subseteq U(r)$ (that is, the bounds are valid in each state of the trace). If a bound is a tuple set $\mathbb{P}(\mathcal{A}^n)$, then $\subseteq$ is just subset relation. If a bound is an expression Γ over relations with constant bounds, we interpret the bound as a *set of valuations* that satisfy the imposed multiplicity constraints; and a relation valuation is valid if it belongs to that set. These sets of valuations are formalized as follows.

Given a universe of atoms $\mathcal{A}$, a valuation for an n-ary relation r is a tuple set in $\mathbb{P}(\mathcal{A}^n)$. A multiplicity bound for such an n-ary has n unary relational expressions Γ_i over relations with constant bounds at the leaves. Thus, it is possible to statically evaluate the value of all *domains* Γ_i into unary tuple sets, which we denote by $[\![\Gamma_i]\!] \in \mathbb{P}(\mathcal{A}^1)$ (following the semantics for relational expressions from the next section); the domain of r is the Cartesian product of all its unary domains. The set of all possible valuations of r is the powerset $\mathcal{V}(r) \triangleq \mathbb{P}([\![\Gamma_1]\!] \times \ldots \times [\![\Gamma_n]\!])$. We call a *valuation set* of r any $V \subseteq \mathcal{V}(r)$. We recover the domain of a valuation set by taking the union of all the tuples in its valuations as $\bigcup V$.

To each multiplicity annotation $m \in \texttt{mult}$ we associate a predicate μ_m to constrain the cardinality of a valuation v:

$$\begin{aligned} \mu_{\textbf{set}}(v) &\triangleq \top & \mu_{\textbf{some}}(v) &\triangleq |v| \geq 1 \\ \mu_{\textbf{one}}(v) &\triangleq |v| = 1 & \mu_{\textbf{no}}(v) &\triangleq |v| = 0 \\ \mu_{\textbf{lone}}(v) &\triangleq |v| \leq 1 \end{aligned}$$

The *multiplicity restriction* of a valuation set V by m keeps only the valuations satisfying μ_m:

$$V \upharpoonright m \triangleq \{\, v \in V \mid \mu_m(v) \,\}$$

Whenever μ_m is satisfiable on $\mathcal{V}(r)$ for a non-empty domain (i.e., for every multiplicity except **no** on a non-empty $[\![\Gamma_1]\!] \times \ldots \times [\![\Gamma_n]\!]$), then $\bigcup(\mathcal{V}(r) \upharpoonright m) = [\![\Gamma_1]\!] \times \ldots \times [\![\Gamma_n]\!]$.

Given valuation sets $V_1 \subseteq \mathcal{V}(r_1)$ and $V_2 \subseteq \mathcal{V}(r_2)$ with domains $[\![\Gamma_1]\!] \times \ldots \times [\![\Gamma_k]\!]$ and $[\![\Gamma_{k+1}]\!] \times \ldots \times [\![\Gamma_n]\!]$, respectively, we define their *matched Cartesian product* $V_1 \otimes V_2 \subseteq \mathcal{V}(r_1 \times r_2)$ as the set of valuations where every tuple from V_1 has a match in V_2, and vice-versa.

$$V_1 \otimes V_2 \triangleq \left\{ v \subseteq \bigcup V_1 \times \bigcup V_2 \;\middle|\; \begin{array}{l} \forall\, (a_1, \ldots, a_k) \in \bigcup V_1 \mid \{(b_{k+1}, \ldots, b_n) \mid (a_1, \ldots, b_n) \in v\} \in V_2 \\ \forall\, (b_{k+1}, \ldots, b_m) \in \bigcup V_2 \mid \{(a_1, \ldots, a_k) \mid (a_1, \ldots, b_n) \in v\} \in V_1 \end{array} \right\}$$

Finally, the semantic function $[\![\cdot]\!]$ maps each multiplicity expression to a valuation set. For an arrow a with underlying relation of arity n, $[\![a]\!] \subseteq \mathcal{V}(\mathcal{U}^n)$.

$$\begin{aligned} [\![m\, \Gamma]\!] &\triangleq \mathbb{P}(\Gamma) \upharpoonright m \\ [\![a_1\ m_1 \rightarrow m_2\ a_2]\!] &\triangleq [\![a_1]\!] \upharpoonright m_1 \otimes [\![a_2]\!] \upharpoonright m_2 \end{aligned}$$

A conjunction whose top-level constructor is an arrow a inherits the arrow's semantics; an explicit intersection is interpreted as the set intersection of the two valuation sets:

$$[\![c \wedge a]\!] \triangleq [\![c]\!] \cap [\![a]\!],$$

As an example, let us calculate the set of possible valuations for relation binary relation `assigns` of `CMS`, whose upper-bound is `Reviewer` **`some`** $\rightarrow$ **`some`** `Article`, meaning it has two unary domains, `Reviewer` and `Article`.

$$
\begin{aligned}
&[\![\texttt{Reviewer } \textbf{some} \rightarrow \textbf{some } \texttt{Article}]\!] \\
=&[\![\texttt{Reviewer}]\!] \upharpoonright \textbf{some} \otimes [\![\texttt{Article}]\!] \upharpoonright \textbf{some} \\
=&[\![\texttt{Reviewer}]\!] \upharpoonright \textbf{some} \otimes \{\{\}, \{(A_0)\}, \{(A_1)\}, \{(A_0), (A_1)\}\} \upharpoonright \textbf{some} \\
=&[\![\texttt{Reviewer}]\!] \upharpoonright \textbf{some} \otimes \{\{(A_0)\}, \{(A_1)\}, \{(A_0), (A_1)\}\} \\
=&\{\{\}, \{(R_0)\}, \{(R_1)\}, \{(R_0), (R_1)\}\} \upharpoonright \textbf{some} \otimes \{\{(A_0)\}, \{(A_1)\}, \{(A_0), (A_1)\}\} \\
=&\{\{(R_0)\}, \{(R_1)\}, \{(R_0), (R_1)\}\} \otimes \{\{(A_0)\}, \{(A_1)\}, \{(A_0), (A_1)\}\} \\
=&\{\{(R_0, A_0), (R_1, A_1)\}, \{(R_0, A_1), (R_1, A_0)\}, \\
&\{(R_0, A_0), (R_0, A_1), (R_1, A_0)\}, \{(R_0, A_0), (R_0, A_1), (R_1, A_1)\}, \{(R_0, A_0), (R_1, A_0), (R_1, A_1)\}, \{(R_0, A_1), (R_1, A_0), (R_1, A_1)\}, \\
&\{(R_0, A_0), (R_0, A_1), (R_1, A_0), (R_1, A_1)\}\}
\end{aligned}
$$

4.2 Hyper relational temporal logic constraints

Syntax. Hyper model finding supports constraints specified in a extension of HyperLTL with first-order and relational operators, as well as more flexible trace quantification placement.

Definition 4.3 (Hyper relational temporal logic syntax). A hyper relational temporal logic formula Φ is defined as follows, where $x \in \mathcal{R} \cup \bigcup \mathcal{R}_i \cup bv(\Phi) \cup \bigcup bv(\phi_i)$ are free or bound relational variables, and $\pi \in \bigcup \{P_i\}$ are trace variables.

$$
\begin{array}{lcl}
\Phi & ::= & \textbf{not}\ \Phi \mid \Phi\ \textbf{and}\ \Phi \mid \textbf{all}\ \pi : P \mid \Phi \mid \phi \\
\phi & ::= & \textbf{true} \mid \Gamma\ \textbf{in}\ \Gamma \mid \textbf{some}\ \Gamma \mid \textbf{lone}\ \Gamma \mid \textbf{not}\ \phi \mid \phi\ \textbf{and}\ \phi \mid \textbf{all}\ x : \Gamma \mid \phi \\
 & \mid & \textbf{after}\ \phi \mid \phi\ \textbf{until}\ \phi \mid \textbf{before}\ \phi \mid \phi\ \textbf{since}\ \phi \\
\Gamma & ::= & x \mid \Gamma[\pi] \mid \textbf{none} \mid \textbf{iden} \mid {\sim}\Gamma \mid {\hat{}}\Gamma \mid \Gamma{+}\ \Gamma \mid \Gamma\ \&\ \Gamma \mid \Gamma\ {-}\ \Gamma \mid \Gamma \rightarrow \Gamma \mid \Gamma\ .\ \Gamma \mid \{x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi\} \mid \Gamma'
\end{array}
$$

The fragment not highlighted in red corresponds to the non-hyper relational temporal logic fragment and allows the definition of properties in relational temporal logic, an extension of first-order temporal logic with relational and transitive closure operators. Here, for the sake of brevity, we focus on a kernel of the language into which all other operators can be translated. For expressions, this includes the converse (~), union (+), intersection (&), difference (-), product (→), join (.) and transitive closure (^) relational operators, as well as the temporal prime operator (') and relations by comprehension. For formulas, this includes atomic inclusion tests (**in**) and cardinality tests (**some** and **lone**), and combinations with negation (**not**), conjunction (**and**) and (first-order) universal quantifications (**all**), as well as future (**until** and **after**) and past (**since** and **before**) temporal operators. In relational model finding everything is seen as a relation, including quantified variables. This allows the same operators to be used for relations and variables, since variables are simply singleton tuple sets (a set with exactly one tuple of arity 1). This is the reason why there is no set membership, but only the inclusion operator **in**. The hyper fragment introduces trace quantifications and a trace selector to determine under which trace variable an expression is evaluated. Notice that trace quantifications can be combined by Boolean connectives, but not nested inside temporal operators nor first-order quantifications.

Semantics. The semantics of formulas and expressions are shown in Fig. 5, with entries highlighted in red denoting the hyper fragment. They are evaluated under a set of traces, given by a context σ for each trace variable, the current trace variable π being evaluated, and an index i of the current state being evaluated. The validity of a formula follows standard semantics for relational logic and LTL with future and past operators. First-order quantification produces static variables, and for a tuple t, we write $\pi \mapsto \mathbb{N} \mapsto x \mapsto \{t\}$ to assign t to x at every state of trace variable π. The value of a relational expression Γ – a tuple set – is denoted by $[\![\Gamma]\!]_\sigma^{\pi,i}$ (the universe $\mathcal{A}$ is given implicitly). The hyper fragment introduces a new trace into σ for each quantification over a model P, while

$$
\begin{array}{lcl}
\sigma,\pi,i \models \textbf{not}\ \Phi & \equiv & \sigma,\pi,i \not\models \Phi \\
\sigma,\pi,i \models \Phi_1\ \textbf{and}\ \Phi_2 & \equiv & \sigma,\pi,i \models \Phi_1 \wedge \sigma,\pi,i \models \Phi_2 \\
\sigma,\pi,i \models \textbf{all}\ \pi_1 : P \mid \Phi & \equiv & \forall I \cdot I \models P \Rightarrow \sigma \oplus \pi_1 \mapsto I, \pi, i \models \Phi
\end{array}
$$

$$
\begin{array}{lcl}
\sigma,\pi,i \models \Gamma_1\ \textbf{in}\ \Gamma_2 & \equiv & [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \subseteq [\![\Gamma_2]\!]^{\pi,i}_{\sigma} \\
\sigma,\pi,i \models \textbf{some}\ \Gamma & \equiv & \left|[\![\Gamma]\!]^{\pi,i}_{\sigma}\right| \geq 1 \\
\sigma,\pi,i \models \textbf{lone}\ \Gamma & \equiv & \left|[\![\Gamma]\!]^{\pi,i}_{\sigma}\right| \leq 1 \\
\sigma,\pi,i \models \textbf{not}\ \phi & \equiv & \sigma,\pi,i \not\models \phi \\
\sigma,\pi,i \models \phi_1\ \textbf{and}\ \phi_2 & \equiv & \sigma,\pi,i \models \phi_1 \wedge \sigma,\pi,i \models \phi_2 \\
\sigma,\pi,i \models \textbf{all}\ x : \Gamma \mid \phi & \equiv & \forall t \in [\![\Gamma]\!]^{\pi,i}_{\sigma} \sigma \oplus \pi \mapsto \mathbb{N} \mapsto x \mapsto \{t\}, \pi, i \models \phi \\
\sigma,\pi,i \models \textbf{after}\ \phi & \equiv & \sigma,\pi,i+1 \models \phi \\
\sigma,\pi,i \models \phi_1\ \textbf{until}\ \phi_2 & \equiv & \exists i \leq l \cdot \sigma,\pi,l \models \phi_2 \wedge \forall i \leq j < l \cdot \sigma,\pi,j \models \phi_1 \\
\sigma,\pi,i \models \phi_1\ \textbf{since}\ \phi_2 & \equiv & \exists 0 \leq l \leq i \cdot \sigma,\pi,l \models \phi_2 \wedge \forall l < j \leq i \cdot \sigma,\pi,j \models \phi_1
\end{array}
$$

$$
\begin{array}{lcl}
[\![x]\!]^{\pi,i}_{\sigma} & = & \sigma(\pi)(i)(x) \\
[\![\Gamma[\pi_1]]\!]^{\pi,i}_{\sigma} & = & [\![\Gamma]\!]^{\pi_1,i}_{\sigma} \\
[\![\textbf{none}]\!]^{\pi,i}_{\sigma} & = & \{\} \\
[\![\textbf{iden}]\!]^{\pi,i}_{\sigma} & = & \{(a,a) \mid a \in \mathcal{A}\} \\
[\![\sim\Gamma]\!]^{\pi,i}_{\sigma} & = & \{(b,a) \mid (a,b) \in [\![\Gamma]\!]^{\pi,i}_{\sigma}\} \\
[\![\hat{}\Gamma]\!]^{\pi,i}_{\sigma} & = & \{(a,b) \mid \exists c_1,\ldots,c_n \cdot (a,c_1),(c_1,c_2),\ldots,(c_n,b) \in [\![\Gamma]\!]^{\pi,i}_{\sigma}\} \\
[\![\Gamma_1 + \Gamma_2]\!]^{\pi,i}_{\sigma} & = & [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \cup [\![\Gamma_2]\!]^{\pi,i}_{\sigma} \\
[\![\Gamma_1 \,\&\, \Gamma_2]\!]^{\pi,i}_{\sigma} & = & [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \cap [\![\Gamma_2]\!]^{\pi,i}_{\sigma} \\
[\![\Gamma_1 - \Gamma_2]\!]^{\pi,i}_{\sigma} & = & [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \setminus [\![\Gamma_2]\!]^{\pi,i}_{\sigma} \\
[\![\Gamma_1 \rightarrow \Gamma_2]\!]^{\pi,i}_{\sigma} & = & \{(a_1,\ldots,a_n,b_1,\ldots,b_m) \mid (a_1,\ldots,a_n) \in [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \wedge (b_1,\ldots,b_m) \in [\![\Gamma_2]\!]^{\pi,i}_{\sigma}\} \\
[\![\Gamma_1 \,.\, \Gamma_2]\!]^{\pi,i}_{\sigma} & = & \{(a_1,\ldots,a_{n-1},b_2,\ldots,b_m) \mid \exists c \cdot (a_1,\ldots,c) \in [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \wedge (c,\ldots,b_m) \in [\![\Gamma_2]\!]^{\pi,i}_{\sigma}\} \\
[\![\{x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi\}]\!]^{\pi,i}_{\sigma} & = & \\
\multicolumn{3}{l}{\quad \{(a_1,\ldots,a_n) \mid (a_1) \in [\![\Gamma_1]\!]^{\pi,i}_{\sigma} \wedge \ldots \wedge (a_n) \in [\![\Gamma_n]\!]^{\pi,i}_{\sigma} \wedge \sigma \oplus \pi \mapsto \mathbb{N} \mapsto (x_1 \mapsto \{(a_1)\}, \ldots, x_n \mapsto \{(a_n)\}), \pi, i \models \phi\}} \\
[\![\Gamma']\!]^{\pi,i}_{\sigma} & = & [\![\Gamma]\!]^{\pi,i+1}_{\sigma}
\end{array}
$$

Fig. 5. Formula and expression semantics, a, b, c are atoms from an implicitly defined $\mathcal{A}$.

context selection changes the trace variable being evaluated. A trace instance of a hyper problem H is a valuation I for the variables declared in H such that the bounds and constraint hold.

Definition 4.4. A trace $I : \mathbb{N} \rightarrow \mathcal{R} \mapsto \mathbb{P}(\mathcal{A}^*)$ is an *instance* of an hyper model finding problem over problems P_i and $H = \langle \mathcal{A}, \mathcal{D}, \Phi \rangle$, denoted $I \models H$, iff $I \models \mathcal{D}$ and $\pi_0 \mapsto I, \pi_0, 0 \models \Phi$, for some distinguished trace variable π_0 not occurring in Φ. A problem H is satisfiable, $\models H$, if it has any instance I, and unsatisfiable, $\not\models H$, otherwise.

Notice that if Φ has no hyper constructs, the semantics degenerates into that of (non-hyper) relational temporal logic problems evaluated under a single trace variable. So, for a non-hyper problem P, $I \models P$, iff $I \models \mathcal{D}$ and $\pi_0 \mapsto I, \pi_0, 0 \models \phi$.

Finite prefix semantics. In model checking instance traces are always infinite, but some BMC techniques draw conclusions from finite prefixes of infinite traces. This requires the semantics to be adapted [6], depending on assumptions made about the unknown continuation of the prefix. HyperQB [21] implements BMC for HyperLTL, and our approach considers the same variants of the semantics. For instance, for a prefix where p holds at every state, a *pessimistic* semantics will assume that p may not hold in the future and consider `always p` trivially false; in contrast, an *optimistic* semantics will assume p will continue to hold and consider `always p` trivially true. Since certain identities (such as `always p` ≡ `not eventually not p`) no longer hold, an additional temporal operator, **`releases`**, is added to the kernel language. In those cases, results are still reported but they are inconclusive. The pessimistic semantics for a prefix of size k is as follows:

$$\begin{array}{lcl}
\sigma, \pi, i \models_k \textbf{after}\ \phi & \equiv & i < k \wedge \sigma, \pi, i+1 \models_k \phi \\
\sigma, \pi, i \models_k \phi_1\ \textbf{until}\ \phi_2 & \equiv & \exists i \leq l \leq k \cdot \sigma, \pi, l \models_k \phi_2 \wedge \forall i \leq j < l \cdot \sigma, \pi, j \models_k \phi_1 \\
\sigma, \pi, i \models_k \phi_1\ \textbf{releases}\ \phi_2 & \equiv & \exists i \leq l \leq k \cdot \sigma, \pi, l \models_k \phi_2 \wedge \forall i \leq j \leq l \cdot \sigma, \pi, j \models_k \phi_1
\end{array}$$

The optimistic semantics is defined in a dual way. In classical BMC, if the state machine representing the system is total (i.e., there are no deadlock states) then the outcome is conclusive when a finite trace is found under the pessimistic semantics. However, as we have seen, relational model finding problems do not have an explicit state machine. Instead, behavior is enforced declaratively in constraint ϕ, making it hard to test for totality.[6] Under the optimistic semantics the outcome is conclusive if no trace is found. HyperQB actually introduces two other variants of bounded semantics for models without loops other than self-loops in *halting* states. In these variants, which we also support, a sound conclusion can also be made for traces that reach a halting state.

5 Model finding with HyperPardinus

HyperPardinus is our implementation of a hyper model finder, and is the first step of our analysis procedure. For a hyper model finding problem[7], it generates a set of SMV models and a HyperLTL formula, lower level formats already consumable by existing HyperLTL model checkers and our own HyperSMV presented in the next section. By relying on standard formats, our technique is expected to benefit from further advances in model checking techniques.

5.1 Hyper model finding algorithm

Existing model checkers for HyperLTL follow the traditional approach of having a distinct notation for the state machine over which traces are quantified, and the hyperproperty to be checked. Most advanced ones (e.g., AutoHyper and HyperQB) support the definition of the state machine in (a fragment of) the SMV language. SMV models can be defined in two styles: one more imperative based on direct **`ASSIGN`** clauses, and another declarative based on **`INIT`**, **`INVAR`** and **`TRANS`** restrictions. The property to be verified is specified in an **`LTLSPEC`** clause.

Figure 6 presents a possible (manual) encoding of a CMS similar to the one presented in Alloy in Section 2. Since SMV has no first-order features, the scope of the model is fixed; here we consider 2 reviewers and 2 articles. SMV supports atomic types, and words/arrays of fixed length. Here, we define `assigns` as a 2×2 array of Booleans to represent papers assigned to reviewers; it is defined as a `FROZENVAR`, meaning its value will not change in the trace. Both `review` and `decision` are defined as arrays with 4 possible values (0 meaning unassigned); they are defined as variables (**`VAR`**) whose value is assigned by the transitions of the system. `a`, `r` and `d` are input variables (**`IVAR`**), whose value is not controlled by the state machine but rather assigned arbitrarily; they are used to represent inputs that guide the selection of events. To enforce the behavior of the system, we employ the imperative style using **`ASSIGN`** clauses. These determine the next value of each variable independently, making it cumbersome to model events with multiple effects. Here, a `review` may change value if it is still unassigned; a `decision` may change value if all assigned reviewers have submitted decisions and the decision is within those reviews (the second criteria defined in Section 2).

Pardinus already provides a translation of (non-hyper) model finding problems into declarative SMV by expanding the relational fragment of the language into (propositional) LTL [31]. Although Pardinus models do not support an explicit definition of the state machine, this translation identifies

[6] Our tool will actually be able to perform this test, but requires infinite trace analysis, which defeats the purpose of BMC; thus, it is disabled by default.

[7] As a backend solver, model finders typically do not provide a concrete syntax for the definition of problems, which are defined programmatically through an API.

```
FROZENVAR
  assigns : array 0..1 of array 0..1 of boolean;
VAR
  review   : array 0..1 of array 0..1 of 0..3;
  decision : array 0..1 of 0..3;
IVAR
    a : 0..2;
    r : 0..2;
    d : 1..3;
ASSIGN
    init(review[0][0]) := 0;
    next(review[0][0]) := case
                            r = 0 & a = 0 & assigns[0][0] & review[0][0] = 0: d;
                            TRUE: review[0][0];
                          esac;
    init(review[0][1]) := 0;
    next(review[0][1]) := case
                            r = 1 & a = 0 & assigns[0][1] & review[0][1] = 0: d;
                            TRUE: review[0][1];
                          esac;
    …
    init(decision[0]) := 0;
    next(decision[0]) := case
                           r = 2 & a = 0 & decision[0] = 0 &
                           (assigns[0][0] -> review[0][0] != 0) & (assigns[0][1] -> review[0][1] != 0) &
                           (review[0][0] = d | review[0][1] = d | review[1][0] = d | review[1][1] = d): d;
                           TRUE: decision[0];
                         esac;
    …
```

Fig. 6. Possible encoding of a CMS model in SMV for 2 reviewers and 2 articles

conjuncts with no temporal operators (referring to the initial state), and those only under **always** operator (denoting invariants and transitions if non-nested prime operators occur), into a declarative description of the state machine; the remaining conjuncts go to the **LTLSPEC** clause.

HyperPardinus builds on this procedure, as shown in Algorithm 1 which translates a hyper model finding problem into a set of (SMV) state machines and a HyperLTL formula. We abstract the translation of relational temporal logic to LTL translation by a procedure RL2LTL that is identical to the one provided by Pardinus except that it propagates the trace selectors from relational expressions down to propositional variables. We also assume that a procedure annotates a non-hyper problem P with a trace variable π, denoted by $P[\pi]$; this connects a problem P with a particular hyper quantification $\pi : P$ in H.

A mismatch between existing HyperLTL model checkers and hyper model finding problems is that the former only support hyperproperties in prenex normal form. Quantifiers can be trivially moved to the outermost position if their domain is guaranteed to be non-empty. The algorithm starts by converting constraint Φ from H into prenex form, and separating it between the quantifiers $\mathbf{Q}$ and the non-hyper fragment ψ; this non-hyper fragment of the outermost model H is translated into a state machine M and an LTL constraint ψ^0. Then, for each hyper quantification $\pi : P$, it annotates P with the correct trace variable and converts it into a lower-level problem consisting of a state machine M and a formula an LTL ϕ^0. To guarantee that the prenex transformation was sound, the LTL problem is checked for non-emptiness, which can be performed by a standard SMV model checker. All ϕ^0 formulas that could not be encoded in the state machine are added as premises of formula ψ^0. The final formula applies the hyper quantifiers $\mathbf{Q}$ to ψ^0, resulting in a HyperLTL constraint Φ^0.

Input: Hyper model finding problem with problems P_i and $H = \langle \mathcal{A}, \mathcal{D}, \Phi \rangle$.
Output: State machine definitions $\mathscr{M}$, HyperLTL formula Φ^0, or $\bot$

$Q, \psi \leftarrow$ `prenex`(Φ); `// convert to prenex and extract quantifiers`
$M, \psi^0 \leftarrow$ `RL2LTL`$(\langle \mathcal{A}, \mathcal{D}, \psi \rangle)$; `// translate non-hyper fragment into LTL`
$\mathscr{M} \leftarrow [M]$;
foreach $Q\,\pi : P \in$ `reverse`(Q) **do**
  $M, \phi^0 \leftarrow$ `RL2LTL`$(P[\pi])$; `// translate problem into LTL`
  if `empty`(M, ϕ^0) **then return** $\bot$; `// invalid state machine`
  if $Q = \forall$ **then**
    $\psi^0 \leftarrow \phi^0$ **`implies`** (ψ^0);
  else
    $\psi^0 \leftarrow \phi^0$ **`and`** (ψ^0);
  $\mathscr{M} \leftarrow [M] + \mathscr{M}$;
$\Phi^0 \leftarrow Q \mid \psi^0$; `// prefix hyper quantifiers and problem constraints`
return $\mathscr{M}, \Phi^0$

Algorithm 1: Overview of the HyperPardinus-to-SMV procedure

Theorem 5.1. *Let* `LTL` *and* `HyperLTL` *be model checking procedure for LTL and HyperLTL, respectively. Assume that, for any* $M, \phi^0 =$ `RL2LTL`(P), $\models P$ *iff* `LTL`$(M, \neg\phi^0)$ *finds counter-examples. Then, if* $\mathscr{M}, \Phi^0$ *is obtained from problems* P_i *and* H *through Algorithm 1,* $\models H$ *iff* `HyperLTL`$(\mathscr{M}, \neg\Phi^0)$ *finds counter-examples.*

Proof (draft). We assume that the `RL2LTL` translation implements the relational temporal logic semantics from Fig. 5; that is, P is satisfiable iff `LTL` finds a counter-example for $\neg\phi^0$ over a state machine M (model checkers, unlike model finders, attempt to falsify properties). The conversion into prenex normal form is sound if the quantification ranges are non-empty; for trace quantifiers, this amounts to $\models P$, which due to the previous assumption can be performed through an emptiness check with an LTL model checker. After conversion to prenex normal form, the non-hyper subformula of Φ is translated with `RL2LTL`, preserving the semantics of the relational temporal logic fragment. Hyper model finding problems range over problems P, while `HyperLTL` ranges over state machines M; a problem P may not fully be encoded as a state machine M, requiring the extra ϕ^0 constraint produced by `RL2LTL`; these are imposed as premises of the property passed to `HyperLTL` in order to only consider valid traces of P problems (as conjunction for existential quantifications, and implication for universal ones). □

5.2 Optimizations

In the rest of this section, we describe further optimizations related to this translation.

Composing quantifications. Sequential quantifications of the same type (universal or existential) can be merged into a single quantification over the product of the individual models. This composition results in more complex models but fewer trace quantifications, which usually results in more efficient analysis. The product of two model finding problems $P_1 \times P_2$ is obtained by simply uniting the universes and declarations, and conjoining the constraints, apart from renamings to avoid clashes (in particular, in the self composition $P \times P$). Outermost existential quantifications can also be merged into the hyper problem H. So, for instance, a hyper problem with models P_1, P_2 and H, a constraint Φ of the shape **`some`** `x, y` $: P_1 \mid$ **`all`** `w, z` $: P_2 \mid \phi$, becomes $H \times P_1 \times P_1$ with formula **`all`** `wz` $: P_2 \times P_2 \mid \phi$. As a side-effect, the generated trace instances will now also instantiate these outermost existential quantifications (such as `s1` and `s2` in the motivating example).

Multiplicity constraints. The default RL2LTL translation from Pardinus produces Boolean-encoded SMV models in which the presence of each tuple of an n-ary relation R is recorded by an independent Boolean variable. This encoding represents the full space $\mathcal{V}(R) = \mathbb{P}(R)$ of valuations, and thus corresponds to an unconstrained declaration $[\![\textbf{set}\ R]\!]$. Our extension exploits the multiplicity constraints attached to field declarations to encode instead the smaller space $[\![R]\!]$, retaining only the valuations that respect the declared multiplicities. We realise this by replacing the per-tuple Booleans by SMV integer variables ranging over $[\![R]\!]$.

For instance, a binary relation $R : A \rightarrow \textbf{one}\ B$ – a total function from A to B – has exactly $|[\![R]\!]| = |B|^{|A|}$ valuations, and can be encoded as $|A|$ integer variables of size $|B|$ (one per atom of A). This will be crucial for explicit backends such as AutoHyper, whose scalability is dominated by the explored state space, and will also yield a more compact representation for symbolic backends such as HyperQB: even after the customary translation to a Boolean encoding ("bit blasting"), each integer of size $|B|$ takes only $\lceil \log_2 |B| \rceil$ Booleans, so the $|A|$ integers together require $|A| \cdot \lceil \log_2 |B| \rceil$ Booleans, against the $|A| \cdot |B|$ Booleans generated by default by RL2LTL.

In general, $|[\![R]\!]| \leq |\mathcal{V}(R)| = 2^{|R|}$, whose worst case for an n-ary R over domains of size $|D_1|, \ldots, |D_n|$ is $2^{|D_1| \cdots |D_n|}$. Assuming $|D_i| = k$, this is 2^{k^n} – doubly exponential in the arity n. Materialising $[\![R]\!]$ explicitly is therefore only feasible for relations with tight multiplicity bounds. Our implementation accordingly manipulates relations symbolically, and applies reduction rules for the common multiplicity patterns; explicit enumeration is reserved as a last resort for more intricate constraints, typically over small domains.

Symmetry breaking. An important feature of Pardinus, inherited from Kodkod, is the detection of symmetries to avoid the generation of isomorphic solutions. The introduction of symmetry breaking predicates considerably reduces the search space, and has shown to largely improve the performance of solvers [36]. We also explore this feature for hyper problems, but it can only be applied to the outermost model, since breaking symmetries in the inner models would be dependent on the valuations of the outermost relations. However, by applying this detection after merging quantifications we can actually apply symmetries over multiple traces. For instance, in the CMS example, articles are considered symmetric in the outermost quantification `s1`, so having the `Agent` assigned to `Article0` or `Article1` is indistinguishable. In `s2`, articles would be considered symmetric if they are not distinguishable in `s1`, but `s2` has no knowledge of `s1`. Composing `s1` and `s2` together enables symmetry breaking in the latter. Remark that state-of-the-art model checkers for HyperLTL also support merging quantifications but at much lower level; applying symmetry breaking over merged quantifications is one of the domain-specific advantages of starting the analysis from a higher-level model.

6 Model checking with HyperSMV

HyperSMV is the second step of our analysis pipeline (Fig. 7). It consumes the set of SMV models $\mathscr{M}$ and HyperLTL formula Φ^0 produced by HyperPardinus, and discharges the verification problem using one of two backends: an infinite explicit-state backend (Exp) and a finite symbolic backend (Sym).

Applying existing HyperLTL model checkers to our higher-level examples turned out to be far from straightforward. The SMV models and HyperLTL formulas that arise from high-level Alloy specifications are significantly larger and more complex than the low-level benchmarks those tools were designed for: they use the full declarative SMV language (including **`INIT`**, **`INVAR`** and **`TRANS`** clauses with integer-valued variables), whereas AutoHyper only accepts imperative **`ASSIGN`**-style models, and HyperQB struggled to cope with the size of the resulting QBF instances. Scaling these

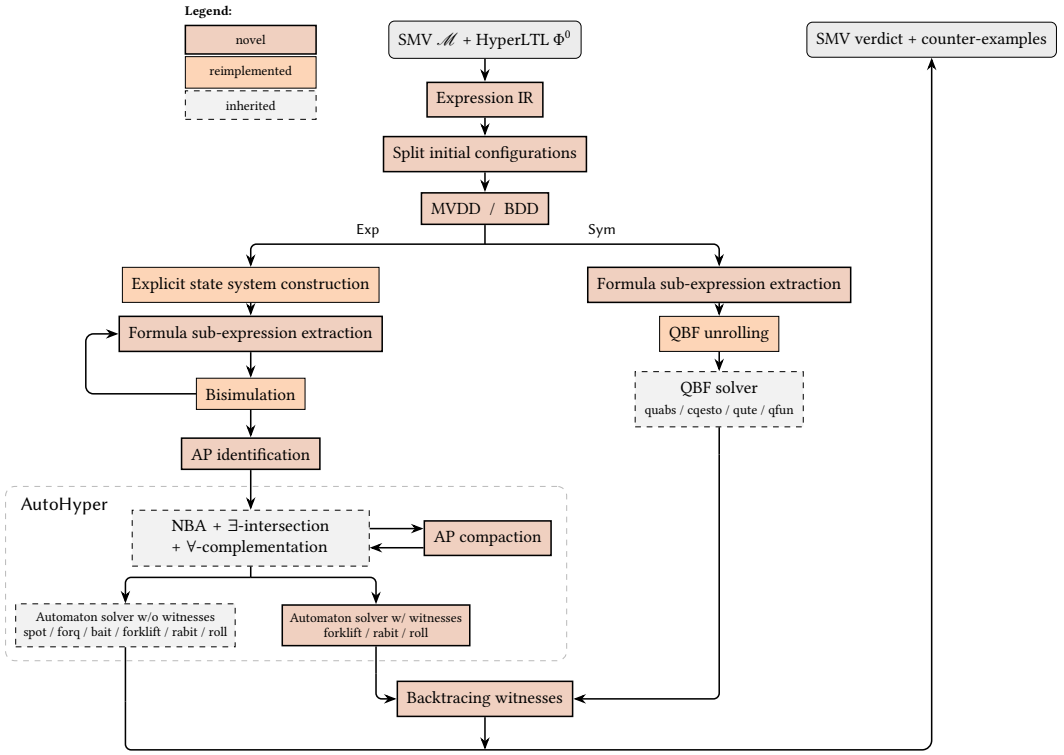

Fig. 7. Architecture of HyperSMV. Thick red borders mark novel contributions; orange solid borders mark reimplemented components with significant extensions; dashed grey borders mark inherited components. Inside AutoHyper, the witness track for counter-example generation during inclusion checking is novel.

tools to our setting required significant reengineering of their pipelines; HyperSMV is the result of that effort.

HyperSMV is designed to be a retrocompatible drop-in replacement: it accepts the same input formats (SMV + HyperLTL) and calls the same underlying solvers — AutoHyper directly for Exp, and off-the-shelf QBF solvers for Sym — while supporting a much more liberal class of SMV models and applying several optimizations along the way, as described below and summarized in Fig. 7.

Beyond the technical pipeline improvements, the two main usability contributions of HyperSMV are support for *general SMV models* (removing the restriction to imperative style) and *SMV-level witnesses* (the ability to translate solver counter-examples back to readable SMV traces), which was a major hindrance in the existing tools and is essential for the Alloy Analyzer visualization described in Section 7.

Representing expressions as decision diagrams. We first apply integer-level simplifications – such as equality tests and range checks – to all non-hyper, non-temporal sub-expressions of SMV models and HyperLTL formulas at an intermediate expression level, before any decision diagram is constructed. We then convert the simplified expressions into a shared internal representation based on *Multi-Valued Decision Diagrams* (MVDD), using hash-consing to promote memory sharing of sub-expressions, and implementing horizontal (conjunctive) and vertical (disjunctive) partitioning for scalable processing. As an optional preprocessing step, we can convert and specialize to *Binary Decision Diagrams* (BDDs) via nuXmv.

The SMV integer variables produced by the multiplicity-aware encoding of Section 5 are carried into the MVDD as multi-valued nodes; this keeps the diagram shallower and avoids inflating the node count without reducing the underlying state space. The explicit-state backend (Exp) manipulates this representation directly. The symbolic backend (Sym) bit-blasts multi-valued nodes into Boolean gates at encoding time.

6.1 Explicit-state analysis (Exp)

Regarding our explicit backend, we have encountered two major challenges in trying to connect HyperPardinus directly to AutoHyper. The first challenge is that AutoHyper only supports imperative SMV models, being incompatible with those generated by Pardinus[8]. The second challenge concerns the generation of counter-examples. Succinctly, AutoHyper converts the formula to a *Non-deterministic Büchi Automata* (NBA) and iteratively intersects it with the automaton of each $\exists$ quantification; $\forall$ quantifiers are handled using automata complementation. Since NBA complementation is a very expensive operation, the outermost $\forall$ quantification is checked via automaton inclusion. This optimization is however disabled when computing witnesses, and AutoHyper instead relies on the spot [13] automaton library to compute witnesses as any valid trace of the final NBA.

Constructing explicit-state systems. The first step in Exp is thus to convert each SMV model into an explicit-state system, which essentially corresponds to an NBA. The MVDD representation greatly eases the implementation of this pass. Conversion proceeds by enumerating all initial SMV states and recursively following all transitions. The above HyperPardinus optimizations ensure that the input SMV models are already as compact as possible.

Reducing explicit-state systems. The next pass of AutoHyper, replicated in Exp, is to compress models by computing bisimulation quotients. This consists in projecting away state variables not mentioned by the formula, jointly with transitions that only depend on projected-away variables. We refine quotients to observe the formula's non-temporal sub-expressions (namely APs, introduced below) rather than the raw variables, which is strictly coarser since these are functions of the variables.

Compressing formulas. Remember that the solving process of AutoHyper consists in converting the formula to an NBA, to then be intersected with the NBA of each quantification. To reduce the size of the formulas, arbitrary non-temporal sub-expressions can be treated as variables – termed *Atomic Propositions (APs)* – in the resulting NBA. Since the choice of APs can greatly affect the performance of the conversion and the subsequent solving process, AutoHyper allows users to directly annotate APs in hyper formulas. In Alloy models – and later at the level of abstraction of HyperPardinus – there are no such annotations, and HyperSMV thus infers APs automatically. This inference is performed on formula expressions, before any decision diagram is built. AP boundaries are placed syntactically following two rules: 1) a maximal non-temporal sub-expression mentioning a single trace becomes one AP; 2) families of multi-trace Boolean expressions corresponding to relational operations such as equality or comparisons (that the Alloy-to-SMV translation has previously expanded) are grouped into a single AP. We also ensure that semantically identical APs are shared.

Compressing APs. Compressing the formula is not sufficient, because AutoHyper introduces new APs as it solves: intersecting the NBA with the automaton of a quantification $\pi_2 : M$ eliminates that trace variable, and a multi-trace AP such as $x[\pi_1] = x[\pi_2]$ is then left behind as one AP $x[\pi_1] = v$

[8]It is possible to convert a declarative SMV model to an imperative one [26], but this is more complex than necessary for our purposes.

for *every* value v that x takes in M. The number of APs thus grows with the size of the models rather than with the formula, even though the whole family is determined by the single value that x takes in trace π_2: for an equality at most one of them can hold, and for a comparison such as $x[\pi_1] \leq x[\pi_2]$ the residual tests $x[\pi_1] \leq v$ are ordered by implication. In either case, most combinations of these APs are impossible. This increase in APs can make the subsequent complementations blow up, and we address it by tuning AutoHyper in two ways. First, we discard operations that no reachable value of the model can satisfy, and replace variables with a single reachable value by constants. Second, much like on the HyperSMV side, we recognize the families whose APs are determined by such a value – equalities and comparisons between variables or against constants, and their Boolean combinations – and replace each family by $\lceil \log_2 n \rceil$ APs, one per bit of an SMV integer variable of size n, as in the bit blasting of Section 5, rewriting the operation bitwise, so that the APs are fixed in advance. Unrecognized shapes fall back to the original APs. This rewriting is applied to the automaton rather than to the formula, so the LTL-to-NBA conversion still sees the original APs.

Calling AutoHyper and computing witnesses. We then call AutoHyper over the reduced systems and formula. When computing witnesses, AutoHyper originally relied solely on automata complementation, defaulting to it whenever witnesses are requested. This is because mapping witnesses over bisimulation-compressed models back to the original uncompressed ones is non-trivial. Since complementation quickly becomes a bottleneck for larger models and more complex formulas, we extended AutoHyper to produce SMV-level witnesses from the output of inclusion checkers RABIT, forklift and ROLL, which natively provide automaton-level witnesses.

Backtracing witnesses. Finally, we translate witness traces over reduced systems back to the original systems, and then to the SMV level. Since the reduced system is a projection of the original, this can be framed as a reachability problem over the original system that always has a solution. Still, bisimulation entails that witness traces may not have the same length, since one state in the reduced witness trace may correspond to multiple transitions in the original system, namely the source states in the same equivalence class (that have been compressed to the same reduced state). Witnesses for zero or more outermost traces – always with the same quantifier type – are handled similarly under composition.

6.2 Symbolic analysis (Sym)

The main challenge for Sym is that the HyperQB pipeline was not coping with larger models.

Constructing QBF problems. We reimplemented the HyperQB [21] algorithm to convert models and formulas to QBF, represented as a Boolean circuit with $\forall$ and $\exists$ quantifiers in prenex form. For a prefix length k, we define $k+1$ copies of the variables of each model and construct a circuit that unrolls the initial states, transitions and invariants. We then unroll the formula over states $0..k$, assuming an optimistic or pessimistic semantics at step $k+1$. For the halting semantics, we additionally test that the circuit corresponding to the transition from k to $k+1$ is equivalent to the identity relation.

Scaling to larger models. As for Exp, the simplicity of this pass is greatly empowered by our MVDD representation. In a similar fashion, HyperQB builds on the PyNuSMV library to construct a symbolic internal BDD representations of input SMV models, using it to first enumerate all initial states, and then find all reachable states. It then constructs circuits of the form $\bigwedge(vars(i) \rightarrow vars(i+1))$ for each transition from state i to its next state $i+1$. This enumeration is comparable to the process of constructing an explicit-state system performed by AutoHyper, and may also help explaining why both tools achieve comparable performance in our benchmarks later in Section 8. In contrast, we convert each MVDD/BDD to a circuit by applying a simple Shannon Expansion

that mirrors its structure, preserving sharing. Larger expressions in the models and formula are naturally handled via partitioning.

Calling QBF solver and backtracing witnesses. We then call an off-the-shelf (non-CNF) QBF solver, such as Quabs, cqesto, qute or qfun. The result of the solver will include a partial sequence of assignments, that we convert back to the original SMV model by mapping each QBF variable to a different variable of a different state in the witness trace. At the source SMV level, we present the validity of the formula, a witness SMV trace and whether the outcome is conclusive or not.

6.3 Optimizations

We now describe additional HyperSMV optimizations that proved to be critical for pratical efficiency.

Moving formula sub-expressions. The translation performed by HyperPardinus moves into the SMV definition sub-expressions of the problems' constraints that represent initial conditions, invariants and transitions. The remaining constraints are moved into the HyperLTL property Φ^0 to be checked. To further simplify the formula, we extract additional sub-expressions that refer to only one trace and move them to the respective state machine. For Exp, this further reduces the size of the explicit-state system. We apply the same LTL-to-NBA conversion and intersection with the NBA of the model performed by AutoHyper for $\exists$ quantifications, and iteratively move formula components and apply bisimulation until no further reduction is possible. For Sym, the performance gain is not so direct as both the state machine and the formula are encoded as Boolean expressions in the resulting QBF problem. However, the same expression may have different semantics depending on whether it appears in the state machine or in the formula, due to the bounded semantics; pushing sub-expressions from the formula into the state machine can therefore improve the conclusiveness of the analysis. For example, a formula **some** `a:A` | **always** $\phi[a]$ **and eventually** $\psi[a]$ will become false under the pessimistic semantics, but if ϕ is moved to the state machine of `A` then **some** $a{:}\{A_\phi\}$ | **eventually** $\psi[a]$ may find a witness `a` that satisfies ψ in a finite prefix. This optimization is sound only if the totality of the state machine is preserved — i.e., that every prefix has at least one possible continuation; this may be violated, for instance, when ϕ – that will act as an invariant – rules out all outgoing transitions from some state. In Section 8 we discuss how this is handled in our selected benchmarks.

Splitting initial configurations. For larger SMV models, a significant bottleneck lies in the processing and state-space exploration of the full system; therefore, HyperSMV adopts a partitioning strategy [18, 30] by decomposing the initial state predicate into disjoint subsets. This effectively splits the input model into smaller submodels with restricted initial configurations that can be solved independently. We implement several splitting strategies to control granularity, such as explicitly computing all initial states, partitioning based only on variables referenced in the **INIT** predicate, or splitting only "frozen" variables that remain constant throughout a trace. While our current implementation checks these submodels sequentially, the approach is sound because the total set of traces is the union of the submodels' traces. Consequently, to refute a $\forall$ property or witness an $\exists$ property, it is sufficient to find a single submodel that respectively invalidates or satisfies the formula, allowing the Exp backend to prune the search space and scale better for complex models.

7 The extended Alloy Analyzer

As seen in Section 2, Alloy is based on the same relational linear temporal logic of Pardinus, but introduces additional syntactic features to ease the design of software systems. For a thorough presentation of Alloy see [24, 32].

When an Alloy command is executed, the model is converted into a single Pardinus problem $P - \mathcal{A}$ is derived from the command's scope, $\mathcal{D}$ from the signature hierarchy, and ϕ from conjoining the command's formula with other model restrictions. To generate a hyper model finding problem instead, we must identify portions of the Alloy model that represent P_i models for trace quantifications. The only extension implemented in the Alloy language is a keyword **`trace`** to identify signatures that represent a P_i model; any quantification over such signatures results in a trace quantification $\pi_i : P_i$ in the hyperproperty from H. Occurrences of such variables will result in context selections, and thus π_i can only be used when composed with fields of P_i. To obtain the constraint ϕ of each P_i problem – which defines its behavior declaratively – we convert the original formula into negation normal form and extract top-level conjuncts where π_i is the only trace variable mentioned. This extraction is performed syntactically over the whole formula without requiring any user annotation; moving conjuncts into individual models can only further constrain them, so no solutions are lost. It may even move into the models parts of the formula to be checked in the command, resulting in simpler hyperproperties.[9] Additionally, we also search the conjuncts for any multiplicity constraints over the declared relations, which are then moved from constraint ϕ to the bound declarations $\mathcal{D}$.

The Alloy Analyzer was extended to support this translation, launch HyperPardinus (choosing the Sym or Exp is done through the GUI as in regular Alloy), and present trace instances back to the user for satisfiable commands, as exemplified in Fig. 4. The counter-example shows the declarations of the outermost model H. When multiple outermost existential quantifiers are present, their domains are combined into a single composite model, so a single instance is displayed that ranges over all of them simultaneously (see Section 6).

8 Evaluation

This section intends to evaluate the feasibility of our approach by answering the following research questions:

RQ1 Can relevant hyperproperties be specified over rich models in our extension to Alloy?
RQ2 How do high-level Alloy models improve usability compared to low-level SMV ones?
RQ3 Are analyses and optimizations in the pipeline effective for high-level models?
RQ4 How does the performance of HyperSMV compare with state-of-the-art model checkers?

To answer these, we consider 2 sets of benchmarks, namely: 1) Alloy models with hyperproperties written by the authors (High) and 2) SMV models from a benchmark suite that is shared by HyperQB [20] and AutoHyper [4] (Low), as well as their automatic translation into Alloy[10]. For **RQ1**, High explores the flexibility of writing high-level models natively, and Low the support for examples from the state of the art. For **RQ2**, we analyze the readability of specifications and the interpretability of feedback, benchmarking the Alloy experience against standard SMV model checkers for demonstrative benchmarks. For **RQ3**, we run HyperPardinus for i) High models, ii) the Low Alloy models translated from SMV to analyze the overhead of the pipeline, and iii) High models with the HyperPardinus optimizations disabled both altogether and selectively to assess their impact. For **RQ4**, we compare HyperSMV against AutoHyper and HyperQB with i) the

[9] Since moving constraints into a P_i model is equivalent to restricting an initially unconstrained state machine, doing so can affect conclusiveness in the bounded semantics. User predictability is therefore essential. Unlike HyperSMV, which simplifies formulas prior to extraction, this Alloy-level transformation is conservative. We preserve the structure of the user's formula and restrict extraction to top-level conjuncts where π_i is the sole trace variable, to ensure that only the portions of the formula the user most naturally associates with P_i's behavior are moved there.

[10] We developed a specialized HyperSMV mode that performs a direct, syntactic translation from SMV to Alloy. Although Alloy is designed for high-level abstraction, it can also express these lower-level patterns, even if the resulting models are not considered idiomatic.

original SMVs from Low, and ii) the SMVs generated by HyperPardinus for High. All executions had witness reporting enabled and were performed in a MacBook Pro 13-inch, M1, 2020 16GB with a timeout of 200s.

8.1 Benchmark models

High includes the first 4 groups of models and consists of idiomatic examples written natively in Alloy to showcase the full expressive power of the language.

Conference management system. This is our running example from Section 2, with the variants for `criteria` that render NI and GNI valid or invalid: NI and GNI hold for CMS_{max}, CMS_{ndet} breaks NI, and CMS_{any} breaks GNI as well. The scope determines the number of *A*rticles and *R*eviewers.

Robotic path synthesis. Path synthesis is a typical use case of hyperproperties. Here we consider different variants, such as finding a shortest path [21] ($\text{Robot}_{\text{short}}$), robust paths which solve multiple goals [21] ($\text{Robot}_{\text{robust}}$), robust paths against adversaries [22] ($\text{Robot}_{\text{adv}}$), and paths with minimal penalties ($\text{Robot}_{\text{score}}$). We consider also a simpler $\text{Robot}_{\text{two}}$ property that checks if at least two distinct paths exist.

Scope *P*osition determines the size of the board, and in $\text{Robot}_{\text{score}}$ the maximum penalty. Below is the property for $\text{Robot}_{\text{adv}}$, searching for a `robot` trace against all possible `adversaries` traces.

```
some r:Rob | rob[r] and all a:Rob | adv[a] implies
  always (not Robot.(r.robot) in Robot.(a.robot))
```

In $\text{Robot}_{\text{score}}$, we allow robots to step onto obstacles with a penalty. The hyperproperty searches for a path with a smaller penalty than any other path reaching a goal.

```
some r:Rob | rob[r] and all o:Rob | rob[o] implies
  always ((o.pos in o.goals and r.pos in r.goals) implies lte[r.penalty,o.penalty])
```

Mutation testing. Hyperproperties have been used in mutation-driven test case generation [14]. We consider whether a mutant of a non-deterministic system is *possibly* or *definitely* killed by a test case, and 3 different vending systems for which the properties hold or fail [14] (scope determines the maximum *Q*uantity of items). Below is the definition of definitely killable mutant with 3 trace quantifiers stating, for a sequence of inputs, the output of the test for the mutant must differ from all possible original system outputs.

```
some s1:System | system[m1] and all m,s2:System |
  (mutant[m] and system[s2] and always (same_in[s1,m] and same_in[s1,s2])) implies
    eventually not same_out[m,s2]
```

Linearizability. Verifying that a concurrent system adheres to a sequential specification is a hallmark challenge in formal methods; such properties are hard to express compositionally and more easily framed as hyperproperties. We consider the SNARK concurrent queue [12], a data structure that is linearizable but not sequentially consistent. While sequential consistency requires a sequential execution that preserves the program order of all traces, linearizability is more flexible, allowing concurrent operations to be reordered to match a valid sequential execution. For this benchmark, the analysis complexity is determined by the scope of available *N*odes, possible *V*alues and concurrent *P*rocesses.

Low benchmarks. This is a set of SMV models and HyperLTL properties from a benchmark suite jointly used by AutoHyper and HyperQB. These include the Bakery3, Bakery5, SNARK, NI, NRP, Mutant and Robot benchmarks, and were also directly translated to Alloy.

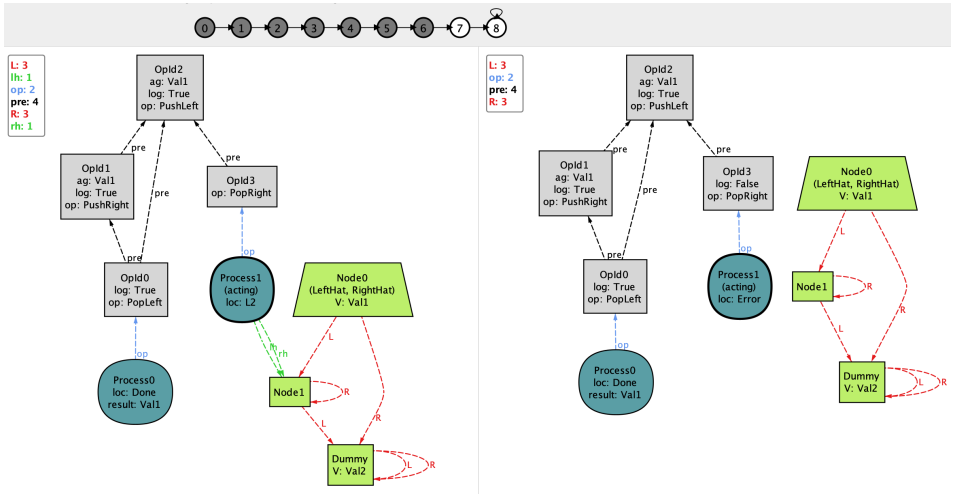

Fig. 8. Last transition of a known bug for SNARK [12], process 1 fails a pop even though the queue was never empty since it started, which is not linearizable; the precedences between the occurred operations are shown, required to test linearizability.

8.2 Qualitative evaluation

Regarding **RQ1**, our approach has shown to be suitable for modeling the selected systems at a high level of abstraction. This is patent, e.g., in the robot examples, where in Alloy the movement is defined declaratively for any current position and is easily readable, while the original SMVs encoded actions for each explicit position, and were thus inscrutable and required dedicated auxiliary scripts to be generated for various map sizes. Moreover, we cover varied classes of quantifier alternations. The different level of abstraction is also patent in the returned instances, which will become clear for the SNARK examples. While Alloy allows relational instances to be graphically visualized, SMV-level tools report traces with only Boolean or integer variables, and HyperQB provides just the output of the QBF solver.

Regarding **RQ2**, we have set to compare the experience of specifying models and interpreting the results using our extension of Alloy. For that purpose, we have chosen two demonstrative high-level examples that were inspired by existing low-level benchmarks.

Robotic path synthesis. In this family of problems, the models encode valid movements of a robot in a grid map with obstacles; the hyperproperty then defines different goals for the path that is to be synthesized. Here, we compare the encoding found in the existing, SMV-based benchmarks, with our based on Alloy.

Figure 9a shows an excerpt (focusing on horizontal movement) of the SMV encoding of a 10×10 map with 26 obstacles provided in the HyperQB benchmarks. SMV does not have first-order quantifications or auxiliary predicates, so any encoding of such a grid will require addressing each grid position explicitly. In the case of Fig. 9a, the authors defined an auxiliary constraint for each movement direction, determining when they might occur according to (implicit) grid constraints (here, `LEFT` and `RIGHT` are shown). Actual movement is then defined by assignments in an imperative flavour. Such a model is clearly unmanageable, since any changes to the obstacles requires changing all the auxiliary formulas, and changes to the grid size extending the assignment clauses. Evidence of this is that the authors actually relied on Python scripts to generate such SMV

files from a given dimension and list of obstacles. While other SMV encodings in a more declarative style could be explored (e.g., using **TRANS** constraints), they would still need to be defined for a concrete grid dimension.

Contrast this with the Alloy model shown in Fig. 9b (again, an excerpt focusing on horizontal movement). It imposes a total order on a signature position that is navigated through relations `next`/`prev`; the size of the grid is simply defined by the scope on this signature. Then, each movement action is described by a declarative predicate with two pre-conditions (the position to which the robot exists, and is not an obstacle), and an effect (the robot position is updated). The obstacles are defined in a separate predicate as a binary relation between positions (i.e., sets of coordinates); any update to the obstacles requires only a local change in this predicate.

Linearizability. Our largest Alloy case study involves modeling the SNARK concurrent deque [12]. This data structure implements a double-ended queue using two "hats" (pointers to the leftmost and rightmost elements) and supports concurrent push and pop operations across multiple processes. While intended to be linearizable, the original design contained a subtle bug that was later identified and corrected in [12]. The SNARK structure effectively illustrates the distinction between linearizability and sequential consistency. For example, if a process initiates a `pop` on an empty queue while another process concurrently completes a `push`, the `pop` might still return a failure. Such a trace is not sequentially consistent, as no sequential execution explains a failed `pop` on a non-empty queue; however, it is linearizable because concurrent operations can be reordered. While the specific bug found in [12] violates both properties, many valid linearizable executions do not satisfy sequential consistency.

Previous attempts to verify the SNARK structure [20] – as modelled in Low – relied on lower-level models and substituted the target linearizability property with the significantly stricter requirement of sequential consistency. We contend that this substitution is fundamentally problematic. First, it forces developers to verify a rigid proxy that misaligns with the actual design goals of concurrent data structures. Second, because many valid, linearizable traces do not satisfy sequential consistency, previous work had to impose ad-hoc restrictions on the search space to suppress these unwanted counter-examples. Such artificial narrowing of the state space is counter-productive to verification, as it requires the user to manually guide the tool away from legitimate concurrent behaviors.

To our knowledge, this is the first time the actual linearizability of the SNARK has been verified using a model checker for hyperproperties. To express the valid reorderings of concurrent operations, each trace maintains structures analogous to logical clocks, recording the history of initiated operations (`log`) and those that have completed when each operation initiates (`pre`). This allows us to express linearizability as a global hyperproperty as follows:

```
pred precedes[A:Con, B:Seq] {
    Process.(A.loc) in Done+Error implies (A.log = B.log and A.pre in B.pre) }

pred Linearizability {
  all A:Con | RunCon[A] implies
    some B:Seq | RunSeq[B] and sameOps[A,B] and always precedes[A,B] }
```

That is, for any concurrent execution, there exists some sequential execution that, at a stable state (no processes executing), recorded the same completed operations in an order that respects linearizability (order must be preserved between operations terminated before another operation initiates). Predicate `sameOps` (omitted) forces each operation atom to represent the same operation in both traces.

Within seconds, Alloy identifies the SNARK bug and provides a minimal, 9-state relational graphical instance (Fig. 8). In comparison, the output from state-of-the-art tools (for the SNARK

```
ASSIGN
  init(x_axis) := {0};
  next(x_axis) :=
    case
      (TOKEN=0) : x_axis;
      (x_axis=0 & RIGHT ) : {1};
      (x_axis=1 & LEFT & RIGHT) : {0,2};
      (x_axis=1 & LEFT & !RIGHT) : {0};
      (x_axis=1 & !LEFT & RIGHT) : {2};
      (x_axis=2 & LEFT & RIGHT) : {1,3};
      (x_axis=2 & LEFT & !RIGHT) : {1};
      (x_axis=2 & !LEFT & RIGHT) : {3};
      (x_axis=3 & LEFT & RIGHT) : {2,4};
      (x_axis=3 & LEFT & !RIGHT) : {2};
      (x_axis=3 & !LEFT & RIGHT) : {4};
      (x_axis=4 & LEFT & RIGHT) : {3,5};
      (x_axis=4 & LEFT & !RIGHT) : {3};
      (x_axis=4 & !LEFT & RIGHT) : {5};
      (x_axis=5 & LEFT & RIGHT) : {4,6};
      (x_axis=5 & LEFT & !RIGHT) : {4};
      (x_axis=5 & !LEFT & RIGHT) : {6};
      (x_axis=6 & LEFT & RIGHT) : {5,7};
      (x_axis=6 & LEFT & !RIGHT) : {5};
      (x_axis=6 & !LEFT & RIGHT) : {7};
      (x_axis=7 & LEFT & RIGHT) : {6,8};
      (x_axis=7 & LEFT & !RIGHT) : {6};
      (x_axis=7 & !LEFT & RIGHT) : {8};
      (x_axis=8 & LEFT & RIGHT) : {7,9};
      (x_axis=8 & LEFT & !RIGHT) : {7};
      (x_axis=8 & !LEFT & RIGHT) : {9};
      (x_axis=9 & LEFT & RIGHT) : {8,10};
      (x_axis=9 & LEFT & !RIGHT) : {8};
      (x_axis=9 & !LEFT & RIGHT) : {10};
      TRUE: x_axis;
    esac;
…
DEFINE
  GOAL := ((x_axis=7 & y_axis=5));
  LEFT := (!(x_axis=0)) & !(x_axis=7 & y_axis=6)
        & !(x_axis=8 & y_axis=6) & !(x_axis=5 & y_axis=5)
        & !(x_axis=4 & y_axis=4) & !(x_axis=6 & y_axis=4)
        & !(x_axis=2 & y_axis=8) & !(x_axis=7 & y_axis=4)
        & !(x_axis=5 & y_axis=9) & !(x_axis=3 & y_axis=6)
        & !(x_axis=5 & y_axis=6) & !(x_axis=2 & y_axis=5)
        & !(x_axis=7 & y_axis=3) & !(x_axis=3 & y_axis=9)
        & !(x_axis=9 & y_axis=5) & !(x_axis=4 & y_axis=5)
        & !(x_axis=4 & y_axis=1) & !(x_axis=6 & y_axis=0)
        & !(x_axis=9 & y_axis=6) & !(x_axis=2 & y_axis=6)
        & !(x_axis=4 & y_axis=6) & !(x_axis=4 & y_axis=8)
        & !(x_axis=2 & y_axis=9) & !(x_axis=3 & y_axis=8)
        & !(x_axis=8 & y_axis=4) & !(x_axis=2 & y_axis=3);
  RIGHT:= (!(x_axis=9)) & !(x_axis=5 & y_axis=4)
        & !(x_axis=7 & y_axis=6) & !(x_axis=7 & y_axis=5)
        & !(x_axis=4 & y_axis=0) & !(x_axis=4 & y_axis=4)
        & !(x_axis=6 & y_axis=6) & !(x_axis=6 & y_axis=4)
        & !(x_axis=2 & y_axis=8) & !(x_axis=2 & y_axis=4)
        & !(x_axis=3 & y_axis=6) & !(x_axis=1 & y_axis=6)
        & !(x_axis=8 & y_axis=1) & !(x_axis=2 & y_axis=1)
        & !(x_axis=5 & y_axis=6) & !(x_axis=2 & y_axis=5)
        & !(x_axis=0 & y_axis=3) & !(x_axis=5 & y_axis=3)
        & !(x_axis=3 & y_axis=9) & !(x_axis=1 & y_axis=8)
        & !(x_axis=0 & y_axis=6) & !(x_axis=0 & y_axis=9)
        & !(x_axis=2 & y_axis=6) & !(x_axis=1 & y_axis=9)
        & !(x_axis=0 & y_axis=8) & !(x_axis=3 & y_axis=5)
        & !(x_axis=0 & y_axis=5);
```

(a) SMV model

```
pred moveleft[b:Board] {
  some next.(b.robot)
  no next.(b.robot) & b.obstacles
  b.robot' = next.(b.robot) }

pred moveright[b:Board] {
  some prev.(b.robot)
  no prev.(b.robot) & b.obstacles
  b.robot' = prev.(b.robot) }
…
pred board[b:Board] {
  let p0=first, p1=p0.next, p2=p1.next, p3=p2.next,
      p4=p3.next, p5=p4.next, p6=p5.next, p7=p6.next,
      p8=p7.next, p9=p8.next {
    b.goals = p7→p5
    b.obstacles =
      p1→p3 + p1→p5 + p1→p6 + p1→p8 + p1→p9 +
      p2→p6 + p2→p8 + p2→p9 + p3→p1 + p3→p4 +
      p3→p5 + p3→p6 + p3→p8 + p4→p5 + p4→p6 +
      p4→p9 + p5→p0 + p5→p4 + p6→p3 + p6→p4 +
      p6→p6 + p7→p4 + p7→p6 + p8→p5 + p8→p6 +
      p9→p1
    b.robot = p0→p0 }

    always {
      moveleft[b] or moveright[b] or … or stutter[b]
    } }
```

(b) Alloy model

Fig. 9. Excerpt of the robot model, showing the same map and right/left actions

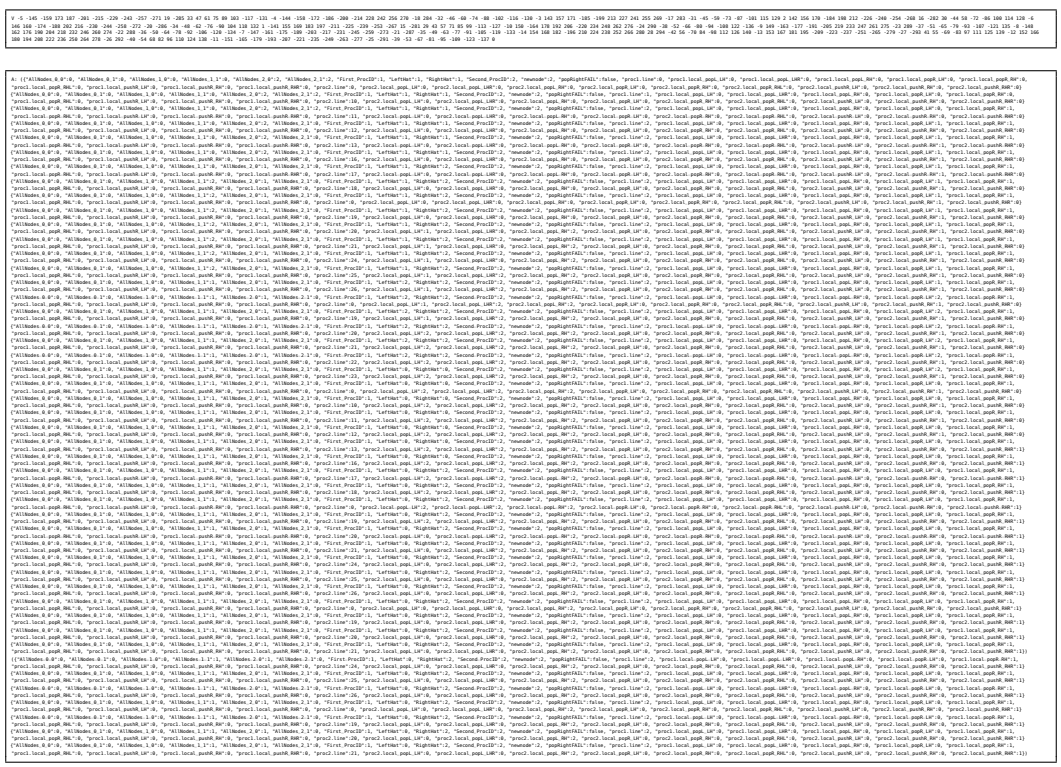

Fig. 10. Traces evidencing the SNARK bug for the $\text{SNARK}_{\text{lin}}$ benchmark provided by HyperQB (top) and by AutoHyper (bottom).

benchmark from Low) places a significant cognitive burden on the user, as shown in Fig. 10.[11] HyperQB, for instance, directly reports the raw output of the `QuAbS` solver. This feedback consists of quantifier-indexed Boolean variables where truth values are indicated by the sign of the index. Because the tool converts all model integers to Booleans and duplicates every variable for each step of the k-unrolling, the resulting QBF-level witness is essentially inscrutable without extensive manual reverse-mapping. Moving to AutoHyper, it reports a trace at the level of SMV variables but does not guarantee minimal instances. For the same SNARK bug, it produces a sprawling 47-state witness containing a 7-state loop. In contrast, the Alloy visualizer abstracts away the low-level solver state, mapping results back to the original model context in a human-readable format. By ensuring both minimality[12] and a high-level representation, our framework resolves the usability hindrances that previously made interpreting counter-examples in complex hyperproperties a challenging task.

[11] For clarity, the visualized instance contains a self-loop in the last state, which represents a valid trace under the model's stuttering semantics; the symbolic backend, however, produces only finite prefixes without loops.

[12] For the Sym backend, any discovered witness is guaranteed to be of size k, where k is the provided number of steps. Such a witness is inherently minimal, provided that the user follows an incremental search strategy by increasing k sequentially until a solution is found.

Model (scope)	Q*	Res	solver	k	Sem	HyperPardinus AutoHyper Size	t(s)	HyperPardinus HyperQB Size	t(s)	HyperPardinus- Exp Size	t(s)	HyperPardinus- Sym Size	t(s)	HyperPardinus Exp Size	t(s)	HyperPardinus Sym Size	t(s)
			(exp)		(sym)												
$\text{Mutant1}_{\text{pk}}(3_Q)$	∃∀	✔	forklift	6	pes	21×21	1.5	50KB	2.2	18×20	1.0	87KB	0.6	18×20	1.1	21KB	0.5
$\text{Mutant1}_{\text{dk}}(3_Q)$	∃∀∀	✔	forklift	6	pes	21×82	1.0	400KB	?	3×18×20	1.1	141KB	0.7	3×30	1.0	37KB	0.6
$\text{Mutant2}_{\text{pk}}(3_Q)$	∃∀	✔	forklift	6	pes	21×21	1.0	50KB	1.0	21×20	1.0	139KB	0.6	21×20	1.0	22KB	0.6
$\text{Mutant2}_{\text{dk}}(3_Q)$	∃∀∀	✗	forq	6	pes	21×105	1.0	550KB	?	3×21×20	0.9	142KB	?	3×45	0.9	40KB	?
$\text{Mutant3}_{\text{pk}}(3_Q)$	∃∀	✗	forq	6	pes	15×21	0.9	50KB	?	14×20	0.9	88KB	?	14×20	0.9	21KB	?
$\text{Mutant3}_{\text{dk}}(3_Q)$	∃∀∀	✗	forq	6	pes	21×45	1.0	200KB	?	3×14×20	1.0	142KB	?	3×21	0.9	36KB	?
$\text{Robot}_{\text{two}}(10_P)$	∃∃	✔	forklift	20	pes	310×310	128.9	5MB	3.0	310×310	45.1	7MB	8.3	310×310	18.3	4MB	4.8
$\text{Robot}_{\text{short}}(10_P)$	∃∀	✔	forklift	20	opt	71×71	2.2	750KB	?	71×71	42.3	7MB	?	71×71	1.7	4MB	?
$\text{Robot}_{\text{score}}(10_P{\times}6_Y)$	∃∀	✔	forklift	20	opt	570×570	4.9	10MB	?	✽	🕒	7MB	?	570×570	3.9	4MB	?
$\text{Robot}_{\text{robust}}(10_P)$	∃∀	✔	forklift	20	opt	310×293	🕒	5MB	🕒	310×293	25.0	6MB	?	310×293	2.4	4MB	?
$\text{Robot}_{\text{adv}}(10_P)$	∃∀	✔	forklift	20	opt	✽	🕒	7MB	?	100×36	🕒	10MB	?	100×36	3.4	6MB	?
$\text{CMSany}_{\text{NI}}(2_R{\times}2_A)$	∀∀	✗	forklift	4	opt	7202	6.8	✽	🕒	868×868	🕒	192KB	?	67	1.8	114KB	0.7
$\text{CMSany}_{\text{NI}}(3_R{\times}2_A)$	∀∀	✗	forklift	6	opt	✽	🕒	✽	🕒	23836×23836	🕒	322KB	?	✽	🕒	279KB	0.9
$\text{CMSany}_{\text{GNI}}(2_R{\times}2_A)$	∀∀∃	✗	forklift	4	opt	55037×868	🕒	✽	🕒	355×77×646	🕒	234KB	?	689×433	8.2	171KB	0.8
$\text{CMSany}_{\text{GNI}}(3_R{\times}2_A)$	∀∀∃	✗	forklift	5	opt	✽	🕒	✽	🕒	1417×1712×11842	🕒	392KB	?	✽	🕒	443KB	1.0
$\text{CMSndet}_{\text{NI}}(2_R{\times}2_A)$	∀∀	✗	forklift	6	opt	2450	1.6	✽	🕒	532×532	🕒	192KB	?	63	1.3	172KB	0.7
$\text{CMSndet}_{\text{NI}}(3_R{\times}2_A)$	∀∀	✗	forklift	6	opt	356968	131.0	✽	🕒	16036×16036	🕒	265KB	?	160	115.0	268KB	0.9
$\text{CMSndet}_{\text{GNI}}(2_R{\times}2_A)$	∀∀∃	✔	forq	4	opt	17957×532	🕒	✽	🕒	217×77×490	🕒	236KB	?	1631×490	6.0	161KB	?
$\text{CMSndet}_{\text{GNI}}(3_R{\times}2_A)$	∀∀∃	✔	forq	5	opt	✽	🕒	✽	🕒	16036×16036×16036	🕒	356KB	?	✽	🕒	446KB	?
$\text{CMSmax}_{\text{NI}}(2_R{\times}2_A)$	∀∀	✔	forq	6	opt	1568	1.7	✽	🕒	448×448	🕒	210KB	?	36	1.2	197KB	?
$\text{CMSmax}_{\text{NI}}(3_R{\times}2_A)$	∀∀	✔	forq	6	opt	127618	65.0	✽	🕒	10648×10648	🕒	308KB	?	80	61.7	282KB	?
$\text{CMSmax}_{\text{GNI}}(2_R{\times}2_A)$	∀∀∃	✔	forq	4	opt	10901×448	🕒	✽	🕒	224×77×406	🕒	265KB	?	1337×406	4.8	206KB	?
$\text{CMSmax}_{\text{GNI}}(3_R{\times}2_A)$	∀∀∃	✔	forq	5	opt	✽	🕒	✽	🕒	10648×10648×10648	🕒	403KB	?	✽	🕒	449KB	?
$\text{SNARK}_{\text{bug}}(3_N{\times}2_V{\times}2_P)$	∀∃	✗	forklift	8	opt	✽	🕒	✽	🕒	✽	🕒	5MB	🕒	✽	🕒	2MB	19.3
$\text{SNARK}_{\text{fix}}(3_N{\times}2_V{\times}2_P)$	∀∃	✔	forq	8	opt	✽	🕒	✽	🕒	✽	🕒	4MB	🕒	✽	🕒	2MB	🕒

Table 1. Execution results for High examples for the HyperPardinus pipeline with alternative SMV solvers.

Model	Q*	Res	solver	k	Sem	Original SMVs AutoHyper Size	t(s)	HyperQB Size	t(s)	Exp Size	t(s)	Sym Size	t(s)	Translated Alloy Models Exp Size	t(s)	Sym Size	t(s)
			(exp)		(sym)												
$\text{Bakery3}_{\text{S1}}$	∃∃	✗	forq	7	pes	167×167	0.6	700KB	?	112×112	0.4	200KB	?	112×112	1.2	300KB	?
$\text{Bakery3}_{\text{S2}}$	∀∃	✗	forklift	12	opt	167×167	0.9	1MB	1.1	112×112	1.6	350KB	0.2	112×112	1.6	450KB	0.8
$\text{Bakery3}_{\text{S3}}$	∃∀	✗	forq	20	opt	167×167	1.1	2MB	1.2	112×112	1.1	700KB	0.3	112×112	1.3	750KB	0.9
$\text{Bakery3}_{\text{sym1}}$	∀∃	✗	forklift	10	opt	167×167	0.7	1MB	1.0	112×112	0.7	550KB	0.2	112×112	1.3	350KB	0.8
$\text{Bakery3}_{\text{sym2}}$	∀∃	✗	forklift	10	opt	167×167	0.6	1MB	0.8	1×1	0.5	400KB	0.2	1×1	1.1	350KB	0.8
$\text{Bakery5}_{\text{sym1}}$	∀∃	✗	forklift	10	opt	996×996	0.9	11MB	5.3	478×478	1.0	950KB	0.5	478×478	1.8	600KB	1.1
$\text{Bakery5}_{\text{sym2}}$	∀∃	✗	forklift	10	opt	996×996	0.8	11MB	5.3	130×130	0.7	1MB	0.5	130×130	1.6	550KB	1.0
$\text{Mutant}_{\text{mut}}$	∃∀	✔	forklift	8	pes	32×32	0.8	200KB	0.6	21×21	0.6	50KB	0.1	21×21	1.1	100KB	0.7
$\text{NI}_{\text{correct}}$	∀∃	✔	forq	10	opt	64×64	0.8	350KB	?	64×64	0.5	300KB	?	64×64	1.2	300KB	?
$\text{NI}_{\text{incorrect}}$	∀∃	✗	forklift	57	pes	368×368	1.2	12MB	?	368×368	1.2	2MB	?	368×368	2.4	2MB	?
NRP_{fair}	∃∀	✔	forklift	15	opt	55×55	0.7	600KB	?	38×38	0.6	100KB	?	38×38	1.1	150KB	?
$\text{NRP}_{\text{unfair}}$	∃∀	✔	forklift	15	pes	54×54	4.0	600KB	0.8	36×36	0.7	100KB	0.1	36×36	1.2	150KB	0.7
$\text{SNARK}_{\text{lin}}$	∀∃	✗	forklift	26	opt	4914×548	3.6	132MB	?	71×137	1.2	5MB	?	71×137	5.1	7MB	?
Robot_{sp}	∃∀	✔	forklift	20	opt	146×146	0.9	2MB	?	124×124	1.1	650KB	?	124×124	2.3	800KB	?
Robot_{rb}	∃∀	✔	forklift	20	opt	266×266	1.7	3MB	?	258×258	1.3	950KB	?	258×258	2.8	2MB	?

Table 2. Execution results for Low examples for alternative SMV solvers.

8.3 Quantitative evaluation

We now report quantitative results addressing **RQ3** and **RQ4**, comparing the performance of HyperPardinus and HyperSMV against state-of-the-art HyperLTL model checkers.

How to read the data. The results relevant for **RQ3** and **RQ4** are reported in Tables 1 and 2 for High and Low, respectively. An entry corresponds to an Alloy command for an example, and

Model	C-		M-		S-		F-		I-		B	
	Size	**t(s)**	**Size**	**t(s)**	**Size**	**t(s)**	**Size**	**t(s)**	**Size**	**t(s)**	**Size**	**t(s)**
Exp												
$\text{Mutant1}_{\text{pk}}(3_Q)$	18×20	1.2	18×20	1.0	18×20	1.2	21×21	1.6	–	–	18×20	1.0
$\text{Mutant1}_{\text{dk}}(3_Q)$	3×18×20	1.0	3×30	1.0	3×30	1.0	21×82	1.0	–	–	3×30	1.0
$\text{Mutant2}_{\text{pk}}(3_Q)$	21×20	1.0	21×20	1.0	21×20	1.0	21×21	1.1	–	–	21×20	1.0
$\text{Mutant2}_{\text{dk}}(3_Q)$	3×21×20	0.9	3×45	0.9	3×45	0.9	21×105	1.1	–	–	3×45	0.9
$\text{Mutant3}_{\text{pk}}(3_Q)$	14×20	0.9	14×20	0.9	14×20	0.9	15×21	1.0	–	–	14×20	0.9
$\text{Mutant3}_{\text{dk}}(3_Q)$	3×14×20	1.0	3×21	0.9	3×21	0.9	21×45	0.9	–	–	3×21	0.9
$\text{Robot}_{\text{two}}(10_P)$	–	–	310×310	45.0	310×310	18.8	310×310	134.2	–	–	310×310	18.4
$\text{Robot}_{\text{short}}(10_P)$	71×71	1.8	71×71	41.4	71×71	1.7	71×71	🕑	–	–	71×71	1.7
$\text{Robot}_{\text{score}}(10_P \times 6_Y)$	570×570	4.0	✹	🕑	570×570	4.0	570×570	🕑	–	–	570×570	3.8
$\text{Robot}_{\text{robust}}(10_P)$	310×293	2.4	310×293	24.8	310×293	2.5	310×293	🕑	–	–	310×293	2.4
$\text{Robot}_{\text{adv}}(10_P)$	100×36	3.4	✹	🕑	100×36	3.5	✹	🕑	–	–	100×36	3.3
$\text{CMSany}_{\text{NI}}(2_R \times 2_A)$	434×518	2.8	67	8.6	67	2.6	7202	2.7	–	–	67	1.7
$\text{CMSany}_{\text{NI}}(3_R \times 2_A)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	–	–	✹	🕑
$\text{CMSany}_{\text{GNI}}(2_R \times 2_A)$	434×77×646	🕑	✹	🕑	2485×646	66.6	110×868	🕑	3365×646	14.9	689×433	8.5
$\text{CMSany}_{\text{GNI}}(3_R \times 2_A)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑
$\text{CMSndet}_{\text{NI}}(2_R \times 2_A)$	266×298	2.0	63	5.0	63	1.9	2450	1.5	–	–	63	1.4
$\text{CMSndet}_{\text{NI}}(3_R \times 2_A)$	5993×3630	🕑	✹	🕑	✹	🕑	356968	134.7	–	–	160	113.9
$\text{CMSndet}_{\text{GNI}}(2_R \times 2_A)$	266×77×490	🕑	✹	🕑	1519×490	🕑	110×532	🕑	2141×490	6.8	1631×490	5.9
$\text{CMSndet}_{\text{GNI}}(3_R \times 2_A)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑
$\text{CMSmax}_{\text{NI}}(2_R \times 2_A)$	224×262	1.7	36	3.3	36	1.4	1568	1.3	–	–	36	1.1
$\text{CMSmax}_{\text{NI}}(3_R \times 2_A)$	3887×2994	🕑	✹	🕑	80	178.4	127618	62.6	–	–	80	63.1
$\text{CMSmax}_{\text{GNI}}(2_R \times 2_A)$	224×77×406	🕑	✹	🕑	1225×406	159.9	110×448	🕑	1847×406	14.9	1337×406	4.7
$\text{CMSmax}_{\text{GNI}}(3_R \times 2_A)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑	✹	🕑
$\text{SNARK}_{\text{bug}}(3_N \times 2_V \times 2_P)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	–	–	✹	🕑
$\text{SNARK}_{\text{fix}}(3_N \times 2_V \times 2_P)$	✹	🕑	✹	🕑	✹	🕑	✹	🕑	–	–	🕑	🕑
Sym												
$\text{Mutant1}_{\text{pk}}(3_Q)$	25KB	0.6	100KB	0.6	25KB	0.5	25KB	0.6	–	–	25KB	0.6
$\text{Mutant1}_{\text{dk}}(3_Q)$	50KB	0.6	200KB	0.8	50KB	0.6	50KB	0.6	–	–	50KB	0.6
$\text{Mutant2}_{\text{pk}}(3_Q)$	25KB	0.6	150KB	0.7	25KB	0.6	25KB	0.6	–	–	25KB	0.6
$\text{Mutant2}_{\text{dk}}(3_Q)$	50KB	?	200KB	?	50KB	?	50KB	?	–	–	50KB	?
$\text{Mutant3}_{\text{pk}}(3_Q)$	25KB	?	100KB	?	25KB	?	25KB	?	–	–	25KB	?
$\text{Mutant3}_{\text{dk}}(3_Q)$	50KB	?	200KB	?	50KB	?	50KB	?	–	–	50KB	?
$\text{Robot}_{\text{two}}(10_P)$	4MB	4.8	7MB	8.3	4MB	4.7	4MB	4.4	–	–	4MB	4.7
$\text{Robot}_{\text{short}}(10_P)$	4MB	?	7MB	?	4MB	?	4MB	?	–	–	4MB	?
$\text{Robot}_{\text{score}}(10_P \times 6_Y)$	4MB	?	7MB	?	4MB	?	4MB	?	–	–	4MB	?
$\text{Robot}_{\text{robust}}(10_P)$	4MB	?	6MB	?	4MB	?	4MB	?	–	–	4MB	?
$\text{Robot}_{\text{adv}}(10_P)$	7MB	?	10MB	?	7MB	?	7MB	?	–	–	7MB	?
$\text{CMSany}_{\text{NI}}(2_R \times 2_A)$	150KB	?	150KB	0.7	100KB	0.8	150KB	0.7	–	–	100KB	0.7
$\text{CMSany}_{\text{NI}}(3_R \times 2_A)$	350KB	?	250KB	0.9	250KB	0.9	300KB	0.9	–	–	300KB	0.8
$\text{CMSany}_{\text{GNI}}(2_R \times 2_A)$	150KB	?	300KB	0.8	250KB	0.8	150KB	0.8	–	–	150KB	0.8
$\text{CMSany}_{\text{GNI}}(3_R \times 2_A)$	350KB	?	400KB	0.9	450KB	1.0	450KB	1.1	–	–	450KB	1.1
$\text{CMSndet}_{\text{NI}}(2_R \times 2_A)$	200KB	?	200KB	0.6	200KB	0.7	150KB	0.7	–	–	150KB	0.6
$\text{CMSndet}_{\text{NI}}(3_R \times 2_A)$	350KB	?	250KB	0.8	250KB	0.8	300KB	0.8	–	–	250KB	0.8
$\text{CMSndet}_{\text{GNI}}(2_R \times 2_A)$	200KB	?	250KB	?	200KB	?	200KB	?	–	–	150KB	?
$\text{CMSndet}_{\text{GNI}}(3_R \times 2_A)$	550KB	?	450KB	?	400KB	?	400KB	?	–	–	450KB	?
$\text{CMSmax}_{\text{NI}}(2_R \times 2_A)$	150KB	?	200KB	?	150KB	?	150KB	?	–	–	200KB	?
$\text{CMSmax}_{\text{NI}}(3_R \times 2_A)$	350KB	?	250KB	?	250KB	?	250KB	?	–	–	300KB	?
$\text{CMSmax}_{\text{GNI}}(2_R \times 2_A)$	200KB	?	300KB	?	200KB	?	200KB	?	–	–	200KB	?
$\text{CMSmax}_{\text{GNI}}(3_R \times 2_A)$	450KB	?	350KB	?	400KB	?	400KB	?	–	–	450KB	?
$\text{SNARK}_{\text{bug}}(3_N \times 2_V \times 2_P)$	2MB	19.5	6MB	58.9	3MB	🕑	2MB	20.2	–	–	2MB	19.0
$\text{SNARK}_{\text{fix}}(3_N \times 2_V \times 2_P)$	3MB	🕑	6MB	🕑	2MB	🕑	3MB	🕑	–	–	3MB	🕑

Table 3. Ablation tests for High examples using the two HyperPardinus backends.

shows the quantifier alternation (**Q***, with bold quantifiers denoting when witnesses are generated), and the validity of the property (**Res**, ✔ if it is valid, ✗ otherwise). For explicit backends, we additionally show the used automaton **solver** (namely, forklift when witnesses are expected and

forq when not). For symbolic backends, the considered prefix size (**k**) and finite semantics (**Sem**, pes(simistic) or opt(imistic)) are shown. The execution results report the **Size** of the intermediate model representations (when model splitting is enabled, only the largest sub-model is being reported), in number of states in explicit backends (with HyperPardinus model composition, there may be less models than initially in $\mathbf{Q}^*$) and consumed memory in symbolic ones (✿ denotes inability to build these within the defined time period), and execution **t**ime (◔ denotes timeouts). Inconclusive results due to finite semantics are signaled with **?** (note that they may also be wrong, but that is meaningless in an inconclusive result). In both tables, the AutoHyper and HyperQB columns considers the off-the-shelf tools, with minor improvements. Table 2 shows the performance of HyperSMV (with Exp and Sym), AutoHyper and HyperQB when given original SMVs from the Low benchmark. Additionally, it shows the performance of the complete HyperPardinus pipeline (also with Exp and Sym) starting with the Alloy version of Low models. Table 1 considers only the full HyperPardinus pipeline. Here, each entry additionally shows the used scope (due to their origin, examples from Table 2 have a fixed universe). To assess the impact of the HyperPardinus optimizations, we also show the results with quantifier composition, symmetry breaking, and multiplicity constraints disabled (HyperPardinus-); and with the state-of-the-art model checkers for HyperLTL instead of HyperSMV. For a finer analysis of the optimizations employed along the pipeline, Table 3 compares the performance of High examples with only one optimization selectively disabled. For HyperPardinus, C- stands for quantifier composition, M- for multiplicity constraints, S- for symmetry breaking. For HyperSMV, F- stands for the new model constructions (formula sub-expressions, bisimulation quotients and AP compaction) and I- for splitting initial configurations; HyperPardinus is the baseline with all optimizations enabled, the same values presented in Table 1; – indicates that an optimization was not used for a benchmark.

HyperPardinus and High. Focusing on Table 1 and addressing **RQ3**, we can see that HyperPardinus was able to analyze models written at a much higher level of abstraction than those in Low, with Sym performing considerably better (though only considering finite prefixes and often inconclusive). For instance, Sym can verify all CMS entries below the timeout, while Exp reaches scope $3_R \times 2_A$ only for some NI properties. Regarding the impact of the HyperPardinus optimizations, Exp- is able to generate the state machines and terminate for most entries, SNARK being a notable exception. CMS is the demonstrative class for which the state machines can be generated, but their size becomes prohibitive; the worst case is $\text{CMSndet}_{NI}(3_R \times 2_A)$, with 16036×16036 versus 160 states. The gains from Sym- are less evident and harder to compare: the finite semantics is only conclusive when it finds a (counter-)example, and many temporal sub-expressions reduce to trivial statements when inconclusive, affecting the overall complexity; to avoid misleading comparisons, we therefore refrain from reporting those times; sizes are independent of the finite semantics. A notable exception is $\text{Robot}_{\text{two}}$, the only conclusive Robot entry, where disabling the optimizations still yields an answer but roughly doubles both time and size. The SNARK model exhibits too much non-determinism for any Exp backend to produce a state machine: the state space explodes because every distinct sequence of the 4 concurrent operations, combined with the full history required to model linearizability, yields a unique state. The $\text{SNARK}_{\text{lin}}$ model from Low is therefore far more constrained, disallowing operation reorderings (and thus requiring no history) and restricting the allowed sequences of operations. Nonetheless, for Sym, HyperPardinus finds the counter-example in $\text{SNARK}_{\text{bug}}$ for the much more complex High model in comparable time to the Low model, while HyperQB cannot even produce the QBF problem. More importantly, the High SNARK counter-example is conclusive while the Low one is not; we express linearizability as a safety property over explicit histories, so a violation is observed at a concrete state of the finite prefix and cannot be repaired by extending the trace. $\text{SNARK}_{\text{fix}}$ has no bugs, so it is intrinsically

inconclusive for Sym regardless of the chosen semantics; for opt it ultimately concludes in 5.5H – demonstrating that Sym is far more suitable for bug finding, as expected.

HyperSMV and High. Turning to **RQ4**, it is interesting to see that the HyperSMV optimizations are also essential to the pipeline: even with HyperPardinus optimizations enabled, there are many Robot and CMS examples where AutoHyper and HyperQB time out but the corresponding HyperSMV procedure returns an answer in a few seconds. It is worth noting that more declarative modeling seems to have a toll on the state-of-the-art tools. For instance, for $\text{Robot}_{\text{robust}}$ natively developed for Alloy both state-of-the-art tools timed out (unlike HyperSMV), but in the original low-level version Robot_{rb} (Table 2), they finished in few seconds. This is likely due to the level of non-determinism, as for instance robot movements in Robot_{rb} cannot happen freely in all directions, unlike in $\text{Robot}_{\text{robust}}$.

HyperSMV and Low. Focusing on Table 2 and addressing **RQ4**, HyperSMV consistently replicates the expected outcomes for Low, and has competitive performance with state-of-the-art tools. Focusing on Exp, the worst scenario was Bakery3_{S2}, becoming 2× slower. Exp is particularly successful in $\text{SNARK}_{\text{lin}}$, becoming 3× faster; the size of the state machine implies that optimizations were key. Sym was equivalent or better in all executions, with an evident improvement on the size of the models, spending 26× less memory in $\text{SNARK}_{\text{lin}}$. Nonetheless, the symbolic approaches (HyperQB and Sym) are naturally inconclusive for many of the examples, hindering the significance of the comparisons; this is the case for $\text{SNARK}_{\text{lin}}$, whose property is of the form $F\phi_0 \wedge G\phi_1$ and $F\phi_0$ becomes trivially true under the optimistic semantics.

HyperPardinus and Low. Regarding **RQ3**, the results of the full HyperPardinus pipeline show an expected overhead, but remain well competitive with the low-level backends. The overhead is near-constant, and mostly visible on the sub-second entries; in the worst case, $\text{SNARK}_{\text{lin}}$, it becomes 4× slower but still reports in a few seconds. Recall that these executions start from a relational model translated from SMVs; HyperPardinus then translates them back into SMV and calls HyperSMV; while this has the overhead of the RL2LTL translation, it also benefits by the optimizations implemented in HyperPardinus. On the other hand, the translations break the structure of the original formulas, which may affect performance. Explicit backends are more sensitive to this because the original benchmarks are annotated with APs, but there are no such annotations at the level of HyperPardinus, and HyperSMV must infer APs from the translated formula.

Optimizations in the pipeline. Focusing on Table 3, we can support a more detailed discussion of **RQ3** and reveal how the optimizations have proven crucial for High examples[13]. For the smaller Mutant examples, the optimizations were naturally not significant. For Robot examples, we can see that M- and F- optimizations were key for the viability of Exp. The same examples are inconclusive for Sym, with the exception of $\text{Robot}_{\text{two}}$ for which M- is also crucial. For CMS examples, we can attest that all optimizations had a significant impact on the result for the Exp backend. For instance, for $\text{CMSmax}_{\text{GNI}}$ only S- and I- still produce a result, at respectively 34× and 3× the baseline cost. For the Sym backend, the most interesting result is that without the C- optimization all CMS results are inconclusive. For the ∀ properties that are valid, and thus require an exploration of all possible models, we can see that S- trades slightly larger models for a narrowing of the search space. This can be evidenced clearly for $\text{CMSmax}_{\text{GNI}}$ under Exp, where S- induces a 34× larger time, and for $\text{SNARK}_{\text{bug}}$ under Sym, where S- concludes above the timeout.

[13]We don't show the same results for Low examples, as optimizations are only significant for $\text{SNARK}_{\text{lin}}$.

8.4 Threats to validity

Benchmark selection and construction. The High idiomatic models were developed by the authors who are well-versed in Alloy and HyperPardinus, which may introduce a bias. Nonetheless, these examples and hyperproperties were selected from the literature: CMS is a popular example in the security verification community, and some of the Robot and Mutant examples are higher-level re-implementations from Low.

Model translation and optimization. The Low Alloy models were generated automatically from SMV files. The outcome of all executions was consistent with the expected result, giving us confidence in this translation. However, note that we do not aim to validate this SMV-to-Alloy translation, but only to show that HyperPardinus is able to deal with models at a low level of abstraction with little performance overhead.

In BMC semantics, moving expressions from the formula to the models can impact the result. Note that in high-level Alloy the same declarative specification defines models and properties, and there is no separation between hyper formula and non-hyper state machines. For the Exp backend, there is no concern on this regard, and we effectively try to push most of the formula to the state machines (and eventually do all the solving in this phase, what happens for examples whose model size becomes 1 after optimizations). For the Sym backend, however, we must exercise additional caution to ensure that our translation and optimizations are sound. This is because, for the particular cases when the finite semantics is conclusive, sound infinite inference assumes that the state machines are total, while moving restrictions from the formula to the state machines may break totality. All our High examples have stuttering and are thus also total; in our examples, we can safety push invariants from the formula, preserving stuttering and totality. The state machines of all Low examples are total, as can be verified with a simple syntactic check; we have verified that all HyperPardinus-generated SMV models preserve such totality, and have disabled further formula optimizations in HyperSMV for these examples. As a result, in all our benchmarks, moving expressions from the formula to the models can only improve the conclusiveness of BMC.

Toolchain and implementation. Regarding the trusted computing base, our pipeline is long and contains many non-validated components, both developed by us and by third-parties. At the bottom of the pipeline, both HyperSMV and state-of-the-art tools either rely on robust QBF solvers (which support certificate generation) or mature libraries for automata manipulation. The fact that 4 different backends provided consistent results (except for some inconclusive symbolic instances) gives us further confidence.

9 Conclusion

This paper proposes a model finding approach for hyper relational temporal logic, backed by automated analysis procedures. Through a minimal extension to Alloy, we show that the approach allows model checking hyperproperties for design models written in a high-level specification language. This makes it more expressive than state-of-the-art model checkers for HyperLTL, and more suitable for end users as models are specified at a higher-level of abstraction and feedback provided in the Alloy Analyzer. Models are automatically analyzed in a two-phased backend that has showed to be competitive with the state of the art for low-level models, and outperform it in high-level ones.

There are many open challenges in the verification of HyperLTL properties, such as finding looping traces in BMC. By relying on standard formats, our approach will benefit from improvements to low-level model checkers for HyperLTL. At the modeling language level, we relied on a simple

extension to Alloy to demonstrate the feasibility of the approach, but further studies are needed to evaluate it from the end-user perspective.

Acknowledgments

The authors would like to thank David Chemouil for support on the Alloy 6 to SMV translation, and Rui Gonçalves and Miguel Montes for their implementation efforts on previous prototypes.

References

[1] Raven Beutner. 2024. Automated Software Verification of Hyperliveness. In *TACAS (2) (LNCS, Vol. 14571)*. Springer, 196–216.
[2] Raven Beutner and Bernd Finkbeiner. 2022. Prophecy variables for hyperproperty verification. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. IEEE, 471–485.
[3] Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 341–362.
[4] Raven Beutner and Bernd Finkbeiner. 2023. AutoHyper: Explicit-State Model Checking for HyperLTL. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, Cham, 145–163.
[5] Raven Beutner and Bernd Finkbeiner. 2023. Model Checking Omega-Regular Hyperproperties with AutoHyperQ. In *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023 (EPiC Series in Computing, Vol. 94)*. EasyChair, 23–35. doi:10.29007/1XJT
[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS (LNCS, Vol. 1579)*. Springer, 193–207.
[7] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *Principles of Security and Trust: Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 3*. Springer, 265–284.
[8] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.
[9] Norine Coenen, Raimund Dachselt, Bernd Finkbeiner, Hadar Frenkel, Christopher Hahn, Tom Horak, Niklas Metzger, and Julian Siber. 2022. Explaining Hyperproperty Violations. In *CAV (1) (LNCS, Vol. 13371)*. Springer, 407–429.
[10] Arthur Correnson, Tobias Nießen, Bernd Finkbeiner, and Georg Weissenbacher. 2024. Finding ∀∃ Hyperbugs using Symbolic Execution. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1420–1445.
[11] Thibault Dardinier, Anqi Li, and Peter Müller. 2024. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 316 (Oct. 2024), 30 pages. doi:10.1145/3689756
[12] Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. 2004. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA*. ACM, 216–224.
[13] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. 2022. From Spot 2.0 to Spot 2.10: What's New?. In *CAV (2) (LNCS, Vol. 13372)*. Springer, 174–187.
[14] Andreas Fellner, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2021. Mutation testing with hyperproperties. *Softw. Syst. Model.* 20, 2 (2021), 405–427.
[15] Bernd Finkbeiner. 2023. Logics and Algorithms for Hyperproperties. *ACM SIGLOG News* 10, 2 (2023), 4–23. doi:10.1145/3610392.3610394
[16] Bernd Finkbeiner, Christian Müller, Helmut Seidl, and Eugen Zălinescu. 2017. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 633–645. doi:10.1145/3133956.3134080
[17] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL. In *Computer Aided Verification*. Springer International Publishing, Cham, 30–48.
[18] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. 2011. Swarm Verification Techniques. *IEEE Transactions on Software Engineering* 37, 6 (2011), 845–857. doi:10.1109/TSE.2010.110
[19] Tom Horak, Norine Coenen, Niklas Metzger, Christopher Hahn, Tamara Flemisch, Julián Méndez, Dennis Dimov, Bernd Finkbeiner, and Raimund Dachselt. 2022. Visual Analysis of Hyperproperties for Understanding Model Checking Results. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2022), 357–367. doi:10.1109/TVCG.2021.3114866
[20] Tzu-Han Hsu, Borzoo Bonakdarpour, and César Sánchez. 2024. HyperQB: A QBF-Based Bounded Model Checker for Hyperproperties. arXiv:2109.12989 [cs.LO] https://arxiv.org/abs/2109.12989
[21] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, Cham, 94–112.

[22] Tzu-Han Hsu, César Sánchez, Sarai Sheinvald, and Borzoo Bonakdarpour. 2023. Efficient Loop Conditions for Bounded Model Checking Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Nature Switzerland, Cham, 66–84.
[23] Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 2024. Hyperproperty Verification as CHC Satisfiability. In *ESOP (2) (LNCS, Vol. 14577)*. Springer, 212–241.
[24] Daniel Jackson. 2006. *Software abstractions: Logic, language, and analysis* (revised ed.). MIT Press.
[25] Daniel Jackson. 2019. Alloy: A language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76.
[26] Jure Kukovec, Thanh-Hai Tran, and Igor Konnov. 2020. Extracting symbolic transitions from TLA+ specifications. *Science of Computer Programming* 187 (2020), 102361. doi:10.1016/j.scico.2019.102361
[27] Tomas Kulik, Brijesh Dongol, Peter Gorm Larsen, Hugo Daniel Macedo, Steve Schneider, Peter W. V. Tran-Jørgensen, and James Woodcock. 2022. A Survey of Practical Formal Methods for Security. *Form. Asp. Comput.* 34, 1, Article 5 (2022), 39 pages. doi:10.1145/3522582
[28] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923.
[29] Leslie Lamport and Fred B. Schneider. 2021. Verifying Hyperproperties With TLA. In *CSF*. IEEE, 1–16.
[30] Flavio Lerda and Riccardo Sisto. 1999. Distributed-Memory Model Checking with SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking (LNCS, Vol. 1680)*. Springer, 22–39. doi:10.1007/3-540-48234-2_3
[31] Nuno Macedo, Julien Brunel, David Chemouil, and Alcino Cunha. 2022. Pardinus: A temporal relational model finder. *Journal of Automated Reasoning* 66, 4 (2022), 861–904.
[32] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. 2016. Lightweight specification and analysis of dynamic systems with rich configurations. In *SIGSOFT FSE*. ACM, 373–383.
[33] Kenneth L. McMillan. 1993. *Symbolic model checking*. Kluwer.
[34] Christian Müller, Helmut Seidl, and Eugen Zălinescu. 2018. Inductive Invariants for Noninterference in Multi-agent Workflows. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 247–261.
[35] Andrei Popescu, Peter Lammich, and Ping Hou. 2021. CoCon: A conference management system with formally verified document confidentiality. *Journal of Automated Reasoning* 65 (2021), 321–356.
[36] Emina Torlak and Daniel Jackson. 2007. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 632–647.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009