mypyvy: A Research Platform for Verification of Transition Systems in First-Order Logic

James R. Wilcox[1], Yotam M. Y. Feldman[2], Oded Padon[3], and Sharon Shoham[2(✉)]

[1] University of Washington, Seattle, USA
[2] Tel Aviv University, Tel Aviv-Yafo, Israel
sharon.shoham@gmail.com
[3] VMware Research, Palo Alto, USA

Abstract. mypyvy is an open-source tool for specifying transition systems in first-order logic and reasoning about them. mypyvy is particularly suitable for analyzing and verifying distributed algorithms. mypyvy implements key functionalities needed for safety verification and provides flexible interfaces that make it useful not only as a verification tool but also as a research platform for developing verification techniques, and in particular invariant inference algorithms. Moreover, the mypyvy input language is both simple and general, and the mypyvy repository includes several dozen benchmarks—transition systems that model a wide range of distributed and concurrent algorithms. mypyvy has supported several recent research efforts that benefited from its development framework and benchmark set.

1 Introduction

mypyvy is an open-source[1] research platform for automated reasoning about symbolic transition systems expressed in first-order logic. A chief design goal for mypyvy is to lower the barrier to entry for developing new techniques for solver-aided analysis and verification of transition systems. As a result, mypyvy's modeling language is simple and close to the underlying logical foundation, and the tool is designed as a collection of reusable components, making it easy to experiment with new verification techniques.

The main application domain of mypyvy is verification of complex distributed algorithms. Following prior work [32,33], transition systems in mypyvy are expressed in uninterpreted first-order logic (i.e., without theories). Using uninterpreted first-order logic is motivated by the experience that solvers often struggle when theories (e.g., arithmetic, arrays, or algebraic data types) are combined with quantifiers. Quantifiers are essential for describing distributed algorithms (e.g., to state properties about all messages in the network), but theories can often be avoided, yielding improved automation.

[1] https://github.com/wilcoxjay/mypyvy.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 71–85, 2024.
https://doi.org/10.1007/978-3-031-65630-9_4

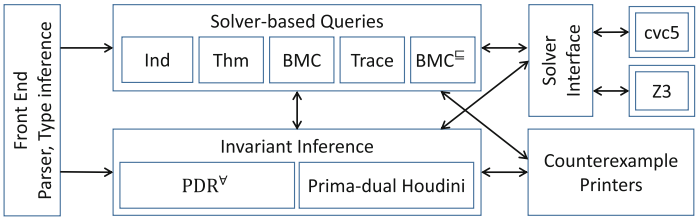

Fig. 1. Main components of `mypyvy`.

`mypyvy` consists of a language for expressing transition systems directly as logical formulas but in a convenient manner (Sect. 2), a tool for reasoning about such systems, and a collection of benchmarks accumulated over the last few years (Sect. 2.1). Figure 1 depicts `mypyvy`'s components, which are divided to solver-based queries (Sect. 3) and invariant inference algorithms (Sect. 4). Solver-based queries such as inductiveness checking and bounded model checking are answered by translating them into satisfiability checks that are sent to external first-order solvers. These queries are used as basic building blocks for developing invariant inference algorithms. `mypyvy` includes an implementation of two such algorithms: PDR$^{\forall}$ [21] and Primal-dual Houdini [34]. `mypyvy`'s internals are designed with the goal of making it easy to build on (Sect. 5). `mypyvy` interacts with multiple solvers, and currently supports Z3 [13] and cvc5 [2]. To present counterexamples (states, transitions, or traces) in a user-friendly way, `mypyvy` supports custom printers that simplify and improve readability of counterexamples.

`mypyvy` is not just the sum of the analyses currently available; it is a platform for doing research in automated verification. Several projects (including ongoing ones) use the `mypyvy` foundation and benchmark suite to build new invariant inference techniques, user interfaces for verification and exploration, and, most recently, liveness verification techniques (Sect. 6).

`mypyvy`'s first-order modeling is inspired by Ivy [30,33], which promoted the idea of modeling distributed systems in the EPR decidable fragment of first-order logic. Ivy includes a rich and modular high-level imperative specification language, as well as mechanisms for creating executable implementations, specification-based testing, liveness verification, and more. As a result, Ivy's syntax, semantics, and code base are more complicated than what would be ideal for enabling rapid exploration of new techniques. In contrast, `mypyvy`'s focus on transition systems, with a simple syntax and semantics, makes it especially suited for enabling verification research.[2] Moreover, `mypyvy`'s code base is intentionally designed, documented, and typed (using Python's support for type annotations), to make it easy to build on and extend.

[2] There are current open-source efforts to automatically translate Ivy to `mypyvy` [9,36], which would allow Ivy users to benefit from `mypyvy`'s algorithms.

Broadly, `mypyvy` has three target audiences:

1. Researchers interested in modeling and verifying distributed algorithms. `mypyvy` offers a user-friendly input language, several queries that assist in developing models of distributed algorithms, readable counterexamples, and access to a variety of automatic verification algorithms.
2. Researchers developing verification techniques, and invariant inference in particular. `mypyvy` offers a starting point for implementing new algorithms on top of a developer-friendly code base. `mypyvy` includes many useful building blocks, and has already been successfully used in several research projects.
3. Researchers looking for benchmarks for various verification tasks. `mypyvy` includes a significant set of transition systems (and their invariants), which can serve as benchmarks for invariant inference or other verification tasks.

2 Modeling Language

We present `mypyvy` through a simple example of modeling and analyzing a toy consensus protocol.[3] To get started, the user first expresses a transition system in `mypyvy`'s input language, which is a convenient syntax for (many-sorted) uninterpreted first-order logic. A `mypyvy` model of the toy consensus protocol is shown in Fig. 2. In this protocol, each node *votes* for a single value, and once a majority or *quorum* of nodes vote for the same value a *decision* takes place. Because majorities intersect, the protocol ensures that at most one value is decided on. Modeling an algorithm or system of interest as a transition system in first-order logic may involve some abstraction, e.g., modeling majorities as abstract quorums such that every two quorums intersect [31].

States. The first step is to choose the types over which the transition system is defined. In the fashion of first-order logic, the basic types are *uninterpreted sorts* (`mypyvy` does not use SMT theories). In the example, we use the sorts `node`, `value`, and `quorum` to represent the nodes that participate in the distributed system, the values they choose from, and the sets of nodes that suffice for a decision (we abstract majorities following [4,32]). The state of the system is modeled by variables which can be *constants* (individuals), *relations*, or *functions*, whose domains are constructed from the aforementioned sorts. Each state variable is either `immutable`, which means it does not change throughout an execution of the system, or `mutable`, which means it may change with each transition. In the example, all state variables are relations. An immutable relation `member` denotes membership of a node in a quorum. The other relations are mutable: `v` records votes of nodes for values, `b` tracks which nodes already voted, and `d` records decisions.

[3] While not useful as a consensus protocol, this example does illustrate important aspects from proofs of complex, widely used consensus protocols like Paxos [25].

```
1  sort node
2  sort value
3  sort quorum
4
5  immutable relation member(node, quorum)
6  axiom forall Q1, Q2. exists N.
7           member(N, Q1) & member(N, Q2)
8
9  mutable relation v(node, value)
10 mutable relation b(node)
11 mutable relation d(value)
12
13 init forall N, V. !v(N,V)
14 init forall N. !b(N)
15 init forall V. !d(V)
16
17 transition vote(n: node, x: value)
18   modifies v, b
19   !b(n) &
20   (forall N, V.
21     v'(N, V) <-> v(N, V) | (N = n & V = x)) &
22   (forall N. b'(N) <-> b(N) | N = n)
23
24 transition decide(x: value)
25   modifies d
26   (exists Q. forall N. member(N, Q) -> v(N, x)) &
27   (forall V. d'(V) <-> d(V) | V = x)
28
29 safety [agreement] forall X, Y. d(X) & d(Y) -> X = Y
30 invariant [decision_quorums] forall X. d(X) ->
31   exists Q. forall N. member(N, Q) -> v(N, X)
32 invariant [unique_votes] forall N, X, Y.
33       v(N, X) & v(N, Y) -> X = Y
34 invariant [voting_bit] forall N, X. v(N, X) -> b(N)
35
36 zerostate theorem forall Q. exists N. member(N, Q)
37 onestate theorem unique_votes & decision_quorums -> agreement
38 twostate theorem forall N, X.
39       voting_bit & vote(N, X) -> voting_bit'
40
41 unsat trace {
42   vote
43   vote
44   vote
45   decide
46   decide
47   assert !safety
48 }
49
50 sat trace {
51   any transition
52   assert exists N, V. v(N,V)
53   decide
54   assert exists V. d(V)
55 }
```

Fig. 2. The toy consensus example in `mypyvy`.

```
> mypyvy verify consensus.pyv

checking init:
 implies invariant agreement..ok.
checking transition vote:
 preserves invariant agreement..ok.
checking transition decide:
 preserves invariant agreement..no!

counterexample:
  universes:
    sort node (1): node0
    sort quorum (1): quorum0
    sort value (2): value0 value1

  immutable:
    member(node0,quorum0)

  state 0:
    d(value1)
    v(node0,value0)

  state 1:
    d(value0)
    d(value1)
    v(node0,value0)

error consensus.pyv: invariant
agreement is not preserved by
transition decide
```

Fig. 3. A counterexample to induction (CTI) for the toy consensus protocol's safety property without additional invariants.

Axioms. `mypyvy` allows the user to define a "background theory" over the immutable symbols, which restricts the state space, via **`axiom`** declarations. In the example, the property that any two quorums intersect (abstracting majorities) is expressed as an axiom for the `member` relation (line 6). (The sorts of quantified variables are omitted in formulas since `mypyvy` infers them automatically.) Another common background theory that is useful when modeling distributed protocols in `mypyvy` is a total order, which can be used to abstract the natural numbers in first-order logic (e.g., to model rounds or indices).

Initial States. The initial states are defined as those that satisfy all `init` declarations. In the example, these declare that all mutable relations are initially empty (lines 13 to 15).

Transitions. The transitions of the system are expressed by `transition` declarations. The semantics is that each transition executes atomically and can modify the system's state. Transitions can have parameters, which are local variables that are assigned nondeterministically whenever the transition is executed. The example has two transitions: vote(n, x) and decide(x) (lines 17 to 27). An important design choice of `mypyvy` is that the user specifies transitions by explicitly writing logical formulas. Each transition is defined over two states: variables in the usual notation refer to the state *before* the transition is applied (*pre-state*), and primed variables refer to the state *after* the transition (*post-state*). Pre-conditions are encoded as conjuncts in the formula about the pre-state; for example, vote requires that the node has not already voted by specifying !b(n). Post-conditions are encoded as conjuncts about the post-state, relating it to the pre-state; for example, vote specifies that the relation b is updated to include exactly the same nodes as before in addition to n. Writing transitions directly through formulas offers great flexibility, but in order to write these formulas succinctly, a transition starts with a `modifies` clause that declares which mutable state variables are changed by it. For any mutable state component *not* in the modifies clause, `mypyvy` implicitly adds a conjunct encoding that the component does not change. Formally, the transition relation is the disjunction of the formulas from each of the transitions, where parameters are existentially quantified.

Safety. Finally, the user may specify safety properties using first-order formulas in `safety` declarations. The agreement safety property in the example (line 29) states that at most one value is decided. A safety property holds if it is satisfied by every state that is reachable from an initial state via a sequence of transitions.

2.1 Benchmarks

The `mypyvy` repository includes over 30 transition systems collected over the years. Some of these were translated from Ivy, while others were directly modeled in `mypyvy`. The benchmarks model a variety of distributed and concurrent algorithms, including consensus algorithms, networking algorithms, and cache coherence protocols. The variety of benchmarks, which also vary in complexity, is useful for evaluating and experimenting with new verification techniques. Additional details can be found in the paper's artifact [39].

3 Satisfiability-Based Queries

Once a transition system is specified, `mypyvy` supports several satisfiability-based queries over it, which are directly translated to satisfiability checks and handed off to solvers (currently Z3 [13] and cvc5 [2] are supported). These queries are

useful building blocks for developing more advanced solver-aided algorithms, and for users who are interested in analyzing specific systems (especially during the model development process). For most queries, `mypyvy` provides counterexamples based on satisfying models obtained from solvers. And while solvers are not guaranteed to terminate, `mypyvy` makes it easy to follow the EPR fragment restrictions, which ensures termination.

3.1 Queries

Inductiveness Checking. `mypyvy` allows the user to add `invariant` declarations to prove safety by induction. These are first-order formulas, whose conjunction (together with the safety properties) forms a candidate inductive invariant. Figure 2 lists three supporting invariants (lines 30 to 34). The most common query in `mypyvy` is to check if the candidate invariant is inductive. When translating an inductiveness check to the solver, `mypyvy` splits it into one solver query per (transition, invariant) pair. In our experience, splitting the disjunction outside the solver improves performance and reliability, and, best of all, improves transparency for the user when one of the cases is more problematic (e.g., takes a long time).

Theorems. In addition to invariants, which are meant to hold in all reachable states of the transition system, `mypyvy` supports checking `theorem` declarations, which specify first-order formulas that are expected to be valid modulo the background theory (i.e., axioms). `zerostate` theorems refer to immutable state variables only, `onestate` theorems may refer to the mutable state variables as well, and `twostate` theorems involve two states, similarly to `transition` declarations. In the toy consensus example, a `zerostate` theorem (line 36) is used to state that quorums cannot be empty (follows from the quorum intersection axiom); a `onestate` theorem (line 37) is used to state that, given the background theory, the `unique_votes` and `decision_quorums` invariants imply the `agreement` safety property; and a `twostate` theorem (line 39) is used to check that the `voting_bit` invariant is preserved by the vote transition.

Bounded Model Checking (BMC). It is often useful to explore (un)reachability of a safety violation via BMC. Given a transition system and a safety property, BMC asks, "Is there a counterexample trace with $\leq k$ transitions?" BMC is implemented in the usual way, by unrolling the transition relation.

Trace Queries. Trace queries allow the user to explore the possible executions of the system in a more targeted way than BMC. This is useful both when the user is interested only in specific scenarios, and when BMC does not scale to sufficient depth. As an illustration, in a model of a distributed system with many protocol steps, BMC may only reasonably scale to a small depth, say 5 transitions, but many interesting behaviors of the system may not occur until at least 10 or 15 transitions. In Fig. 2, lines 41 to 48 show a query for the nonexistence of an execution trace that starts with three vote transitions, followed by two decide

transitions, and then reaches a safety violation. `mypyvy` translates such a query to a first-order formula that is checked for unsatisfiability.

As a complement of trace queries that are expected to be unsatisfiable (specified by the **`unsat`** keyword), it is also useful to make **`sat`** trace queries that are expected to be satisfiable, demonstrating that some behaviors are indeed possible.[4] For example, lines 50 to 54 show a query expecting the existence of a trace that starts with any transition after which there exists a vote, followed by a decide transition after which there exists a decision. (That is possible when the number of nodes is 1.) Such satisfiable trace queries are especially useful for detecting *vacuity bugs*, where, due to a modeling error, some transitions mistakenly cannot execute, potentially making the system erroneously safe.

Relaxed Bounded Model Checking ($BMC^{\sqsubseteq}$). So far we discussed *concrete* traces. `mypyvy` can also search for *relaxed* counterexample traces of a bounded depth. A relaxed trace consists of a sequence of interleaved transitions and "relaxation steps", where some elements get deleted from the structure. As shown in [21], a relaxed counterexample trace that starts at an initial state and ends in a safety violation *proves* that there is no universally quantified inductive invariant that implies safety. This is the case in the toy consensus example—a relaxed counterexample trace found by `mypyvy` for this example is provided in the paper's artifact [39]. The key to implementing relaxed BMC queries is encoding universe reduction between states. `mypyvy` does so by introducing a mutable unary relation active for each sort and using it as a guard in every quantifier, effectively restricting the universe in each state to the "active" part. Relaxation steps are then modeled by adding a relax transition where each active relation in the post-state is a subset of the corresponding one in the pre-state (expressed as a universally quantified formula); all other state variables are unmodified over the active part. Finally, a relaxed BMC query is encoded similarly to a BMC query (with the added relax transitions), except that, due to the use of different active universes, the axioms are asserted not only at the beginning of the trace but also after every (relaxation) step, together with assertions requiring that the active universe contains the constants and is closed under functions.

3.2 Counterexamples

When a query fails (except for a **`sat trace`** query), it is because the formula sent to the solver was satisfiable. In such cases, `mypyvy` obtains a model from the solver and displays a *counterexample*—which can be a state, a transition, or a trace, depending on the failing query. For example, when inductiveness checking fails, it returns either a 1-state model demonstrating a violation of safety at an initial state, or a 2-state model demonstrating a counterexample to induction (CTI). As

[4] `mypyvy` uses solver queries to generate executions of the transition system. A solver is needed due to `mypyvy`'s flexible and abstract modeling language. More imperative modeling languages, e.g. that of Ivy, admit execution/simulation without solvers, which can be useful for invariant inference as well [40,42]. Such simulation can also be implemented for a fragment of `mypyvy`'s language.

another example, when BMC finds an execution that violates safety, it returns a k-state model providing a counterexample trace. Figure 3 shows a CTI (2-state model) for the toy consensus protocol when the invariants supporting the safety property are omitted. In general, `mypyvy` displays a k-state model by first listing the universe of each sort and the interpretations of the immutable symbols (member in our example). Then, for each of the k states, the interpretations of the mutable symbols in that state are printed. For relations, by default `mypyvy` only prints positive literals, i.e., the tuples that are in the relation.

Annotations, Plugins, and Custom Printers. In some cases, the default counterexample printing of `mypyvy` is not as readable as it could be. For example, if one of the sorts in the transition system is totally ordered (using a binary relation and suitable axioms), it would make sense to name the elements of that sort according to the total order. To improve the readability of counterexamples, `mypyvy` supports custom formatting via *printer plugins* and *annotations*. Every declaration in `mypyvy` can be tagged with *annotations*, which have no inherent meaning, but can be detected by plugins, e.g., to cause things to be printed differently. For example, the declaration `sort round @printed_by(ordered_by_printer, le)` invokes the `ordered_by_printer` plugin and tells `mypyvy` that the sort `round` should be printed in the order given by the `le` relation. `mypyvy` provides several other custom printers, including one for printing sorts that represent sets of elements coming from another sort. Users can also implement their own custom printing plugins in Python.

`mypyvy` also supports a handful of other annotations. `@no_print` instructs `mypyvy` not to print a sort, relation, constant, or function at all, which can be useful either because of a custom printer for another symbol, or temporarily because the model is large and the symbol is irrelevant to the current debugging session. `@no_minimize` is used to instruct `mypyvy`'s model minimizer not to minimize elements of a certain sort or relation. The annotation framework is extensible, and we expect more uses for it to come up.

3.3 Decidability and Finite Counterexamples via EPR

In general, `mypyvy` does not restrict the quantifier structure used in formulas, nor the signatures of state variables. As a result, the first-order formulas that encode different queries in `mypyvy` are not guaranteed to reside in any decidable fragment and solvers may diverge. However, a common practice when working with `mypyvy` is to use the effectively propositional (EPR) [35,37] fragment of first-order logic, which imposes certain restrictions on functions and quantifier alternations. To encode a system in EPR (i.e., ensure that formulas generated for all queries are in EPR), the user can rely on recently developed methodologies [32,38]. For example, the toy consensus example of Fig. 2 is in EPR. Satisfiability of EPR is decidable, and reliably checked by solvers. EPR enjoys a small-model property, which implies queries have finite counterexamples (if any). Solver reliability and finite counterexamples are key enablers for more advanced algorithms (e.g., invariant inference) that make thousands of solver queries and

employ model-based techniques. `mypyvy`'s language is close to the underlying logic used in queries, making it relatively easy to follow the EPR restrictions.

4 Invariant Inference

`mypyvy`'s design aims to make it easy to implement complex solver-aided analysis algorithms on top of the simpler queries. Two such algorithms, for automatically finding inductive invariants, are included in `mypyvy`: PDR$^\forall$ and Primal-dual Houdini.

*Universal Property-Directed Reachability (*PDR$^\forall$ *).* `mypyvy` includes an implementation of PDR$^\forall$ [21], which infers universally quantified inductive invariants in first-order logic. Like IC3/PDR [7], PDR$^\forall$ constructs invariants incrementally by finding backwards reachable states and "blocking" them relative to a "frame". To block a state, PDR$^\forall$ computes a "forbidden sub-state" that rules out all states containing a certain pattern. If PDR$^\forall$ succeeds, it returns the inductive invariant in the form of a conjunction of universally quantified clauses. Otherwise, it either loops forever or returns a relaxed trace, proving that no universally quantified inductive invariant exists for the property. On the toy consensus example, PDR$^\forall$ returns a relaxed trace similar to the one obtained by BMC$^\sqsubseteq$. `mypyvy`'s implementation is the state-of-the-art implementation of PDR$^\forall$, and was used for comparison with PDR$^\forall$ in various papers [23,34,40]. The results demonstrate the success of `mypyvy`'s PDR$^\forall$ implementation in solving benchmarks that only require universally quantified invariants.

Primal-Dual Houdini. Primal-dual Houdini [34] is a recent invariant inference algorithm based on a formal duality between reachability in transition systems and a notion of incremental induction proofs. `mypyvy` includes an implementation of Primal-dual Houdini for universally quantified invariants. Primal-dual Houdini works best for transition systems where the inductive invariant can be constructed incrementally, adding one universally quantified clause at a time. Several complex distributed algorithms have this feature. In cases where the invariant cannot be constructed incrementally, Primal-dual Houdini can find a witness for that fact. See [34] for more details and an empirical evaluation. Primal-dual Houdini was prototyped using `mypyvy`'s infrastructure, and its development is an example of the usefulness of `mypyvy` for research in invariant inference.

5 Designing `mypyvy`'s Internals

We designed `mypyvy`'s internals with the goal of making it easy to build on. The most important aspects of the internals from the developer's perspective are (1) using typed Python,(2) the design of the abstract syntax trees (ASTs), and (3) the interface to the underlying first-order solver. `mypyvy` is written in statically typed Python using the `mypy` type checker. Types not only help catch bugs, but also document the interfaces available to the developer. In our experience, types

allow developers to get up to speed more quickly on the code base and facilitate communication.

The ASTs for representing logical formulas in `mypyvy` were designed to support symbolic manipulation, as is common in solver-aided algorithms. This led us to avoid any additional intermediate representations between the ASTs representing the user-level formulas and the ASTs representing the input to solvers. We also structured the ASTs so that it is easy to (re)compute any analysis performed. For example, instead of using a traditional (mutable, long-lived) symbol table to resolve names, `mypyvy` uses a purely functional context to track scopes during AST traversals. The context is thrown away and recomputed every time the AST is traversed. This makes it easy to traverse programmatically generated ASTs, without needing to update any symbol tables or other global data structures, and the extra run time overhead is negligible.

Developers who use `mypyvy` often want to make many queries to the underlying solvers (currently Z3 and cvc5). We expose two interfaces for this. First, many common primitives, such as those discussed in Sect. 3.1, are exposed as a library. Second, `mypyvy` has a lower-level solver interface, where developers can issue their own satisfiability queries, and also gain access to minimized models and minimized unsat cores. Furthermore, developers of sophisticated invariant inference algorithms may have many thousands of queries to run, so `mypyvy` supports running many solvers in parallel.

6 Works Using `mypyvy`

One of `mypyvy`'s goals is to serve the research community and enable research on verification, and invariant inference in particular. Indeed, in recent years several works have built on `mypyvy` or used it to various extents.

Phase-PDR$^\forall$ [14] is a user-guided invariant inference technique. The user provides a *phase structure* to convey temporal intuition, and suitable *phase invariants* are found using an adaptation of PDR$^\forall$. Phase-PDR$^\forall$ was developed on top of the `mypyvy` code base and `mypyvy`'s PDR$^\forall$ implementation, and its evaluation uses benchmarks available from `mypyvy` augmented with phase structures.

SWISS [18] is an invariant inference algorithm that finds quantified invariants, including quantifier alternations, using explicit search. While SWISS does not use the `mypyvy` code base (it is implemented in C++), it accepts `mypyvy`'s input files and its evaluation uses benchmarks available from `mypyvy`.

P-FOL-IC3 [23] is a variant of IC3/PDR that can find invariants with arbitrary quantification using *quantified separation* [22]. P-FOL-IC3 was implemented using `mypyvy`'s code, and also benefited from `mypyvy`'s benchmark set.

IC3PO [15,16] is an IC3/PDR variant that finds quantified invariants for protocols by analyzing finite instances. It does not use `mypyvy`'s code, but is evaluated on some of `mypyvy`'s benchmarks, manually translated to its input format.

LVR [41] develops a methodology for proving liveness properties. It uses `mypyvy` "twice": first, as a modeling language and a source of benchmarks, and second, as an invariant inference engine (using P-FOL-IC3) to find invariants that are required to support a liveness proof based on ranking functions.

7 Related Work

Several tools promote specification and verification of systems and algorithms using first-order logic, dating back to Abstract State Machines [6,17]. Alloy [20] is a relational modeling language and a tool that performs bounded verification, i.e., bounding the size of the universe of each sort. Alloy goes beyond first-order logic and has concepts such as transitive closure, but it shares `mypyvy`'s emphasis on using uninterpreted relations and quantifiers, rather than SMT theories. Electrum [8,29] is an extension of Alloy that was recently integrated into Alloy 6 [1]; it essentially turns Alloy into a modeling language for transition systems. When universe sizes are bounded, Electrum/Alloy 6 can use finite-state model checkers to verify safety as well as liveness properties.

Ivy [30,33] is a multi-modal verification tool that supports modeling using first-order logic and EPR as well as some decidable SMT theories, modular reasoning, extracting executable implementations, liveness verification, specification-based testing, and more. Unlike Alloy, Ivy is not restricted to bounded verification; instead, it relies on user-provided inductive invariants and restricts the quantifier-alternation structure of verification conditions to ensure decidability of unbounded verification queries.

Verification of transition systems is also the focus of the TLA$^+$ toolbox [26], where transition systems are expressed in a very rich logic (based on set theory). As a result, verification is restricted to model checking bounded instances [24,43] similar to Alloy, or manually writing detailed machine-checked proofs [10].

The IronFleet project [19] verifies distributed systems by formalizing transition systems and refinement in Dafny [27], a general-purpose deductive verification language. In IronFleet, transition systems are expressed using the rich Dafny type system, which is based on SMT combined with quantifiers. But as a result, queries to Z3, the underlying SMT solver, suffer from instability, especially when quantifiers—which are handled using triggers—are involved [28].

Compared to the aforementioned systems, `mypyvy` takes a similar approach to Ivy in using first-order logic without theories and aiming for unbounded verification, but unlike Ivy it focuses on automatically finding inductive invariants, and enabling research in that direction. We note that automated invariant inference depends on the reliability of invariant checking and related queries, which is absent from Dafny, TLA+, or Alloy (for the unbounded case), and obtained in `mypyvy` by using EPR in the style of [32].

Another related line of research is developing intermediate representation languages for invariant inference. VMT [11] is a format that extends SMT-LIB [3] to a transition system semantics. Constrained Horn Clauses (CHCs) [5,12] is another SMT-LIB extension that is similar to transition systems but more general (it captures, e.g., recursive programs). Both VMT and CHCs are typically

used with rich SMT theories, whereas `mypyvy`'s logic is centered around uninterpreted first-order logic and quantifiers.

Acknowledgements. The research leading to these results has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). This research was partially supported by the Israeli Science Foundation (ISF) grant No. 2117/23.

References

1. Alloy 6 announcement (2021). https://alloytools.org/alloy6.html. Accessed 03 Feb 2023
2. Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) ETAPS 2022, Part I. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Berkovits, I., Lazić, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 245–266. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_15
5. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
6. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). http://www.springer.com/computer/swe/book/978-3-540-00702-9
7. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
8. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The electrum analyzer: model checking relational first-order temporal specifications. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018, Montpellier, France, 3–7 September 2018, pp. 884–887. ACM (2018). https://doi.org/10.1145/3238147.3240475
9. Chajed, T.: Ivy to mypyvy translator (2023). https://github.com/tchajed/ivy-to-mypyvy
10. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ proof system: building a heterogeneous verification platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 44–44. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14808-8_3
11. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. CoRR abs/2109.12821 (2021). https://arxiv.org/abs/2109.12821

12. De Angelis, E., Hari Govind, V.K.: CHC-COMP 2022: competition report. In: Hamilton, G.W., Kahsai, T., Proietti, M. (eds.) Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation. HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3 April 2022. EPTCS, vol. 373, pp. 44–62 (2022). https://doi.org/10.4204/EPTCS.373.5
13. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. Feldman, Y.M.Y., Wilcox, J.R., Shoham, S., Sagiv, M.: Inferring inductive invariants from phase structures. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 405–425. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_23
15. Goel, A., Sakallah, K.: On symmetry and quantification: a new approach to verify distributed protocols. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 131–150. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_9
16. Goel, A., Sakallah, K.A.: Towards an automatic proof of Lamport's paxos. In: Formal Methods in Computer Aided Design. FMCAD 2021, New Haven, CT, USA, 19–22 October 2021, pp. 112–122. IEEE (2021). https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_20
17. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide, pp. 9–36. Oxford University Press, Specification and Vgalidation Methods edn. (1995). https://arxiv.org/pdf/1808.06255.pdf
18. Hance, T., Heule, M., Martins, R., Parno, B.: Finding invariants of distributed systems: it's a small (enough) world after all. In: Mickens, J., Teixeira, R. (eds.) 18th USENIX Symposium on Networked Systems Design and Implementation. NSDI 2021, 12–14 April 2021, pp. 115–131. USENIX Association (2021). https://www.usenix.org/conference/nsdi21/presentation/hance
19. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP), pp. 1–17. Monterey, CA (2015)
20. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2012)
21. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. J. ACM **64**(1), 7:1–7:33 (2017)
22. Koenig, J.R., Padon, O., Immerman, N., Aiken, A.: First-order quantified separators. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2020, London, UK, 15–20 June 2020, pp. 703–717. ACM (2020). https://doi.org/10.1145/3385412.3386018
23. Koenig, J.R., Padon, O., Shoham, S., Aiken, A.: Inferring invariants with quantifier alternations: taming the search space explosion. In: TACAS 2022. LNCS, vol. 13243, pp. 338–356. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_18
24. Konnov, I., Kukovec, J., Tran, T.: TLA+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA), 123:1–123:30 (2019). https://doi.org/10.1145/3360549

25. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
26. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional, Boston (2002)
27. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
28. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 361–381. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_20
29. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 373–383. ACM (2016). https://doi.org/10.1145/2950290.2950318
30. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020, Part II. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12
31. Padon, O.: Deductive verification of distributed protocols in first-order logic. Ph.D. thesis, Tel Aviv University (2018)
32. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. PACMPL **1**(OOPSLA), 108:1–108:31 (2017)
33. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 614–630. Santa Barbara, CA (2016)
34. Padon, O., Wilcox, J.R., Koenig, J.R., McMillan, K.L., Aiken, A.: Induction duality: Primal-dual search for invariants. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022).https://doi.org/10.1145/3498712
35. Piskac, R., de Moura, L.M., Bjørner, N.S.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reason. **44**(4), 401–424 (2010)
36. Pîrlea, G.: Translation from ivy to mypyvy (2024). https://github.com/kenmcmil/ivy/pull/76
37. Ramsey, F.P.: On a problem of formal logic. Proc. Lond. Math. Soc. **s2–30**(1), 264–286 (1930). https://doi.org/10.1112/plms/s2-30.1.264, https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-30.1.264
38. Taube, M., et al.: Modularity for decidability of deductive verification with applications to distributed systems. In: Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Philadelphia, PA (2018)
39. Wilcox, J.R., Feldman, Y.M.Y., Padon, O., Shoham, S.: mypyvy: A Research Platform for Verification of Transition Systems in First-Order Logic (Artifact) (2024). https://doi.org/10.5281/zenodo.10948110
40. Yao, J., Tao, R., Gu, R., Nieh, J.: Duoai: fast, automated inference of inductive invariants for verifying distributed protocols. In: Aguilera, M.K., Weatherspoon, H. (eds.) 16th USENIX Symposium on Operating Systems Design and Implementation. OSDI 2022, Carlsbad, CA, USA, 11–13 July 2022, pp. 485–501. USENIX Association (2022). https://www.usenix.org/conference/osdi22/presentation/yao

41. Yao, J., Tao, R., Gu, R., Nieh, J.: Mostly automated verification of liveness properties for distributed protocols with ranking functions. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3632877
42. Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., Ryan, G.: Distai: data-driven automated invariant learning for distributed protocols. In: Brown, A.D., Lorch, J.R. (eds.) 15th USENIX Symposium on Operating Systems Design and Implementation. OSDI 2021, 14–16 July 2021, pp. 405–421. USENIX Association (2021). https://www.usenix.org/conference/osdi21/presentation/yao
43. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6