Deductive Verification in Decidable Fragments with Ivy

Kenneth L. McMillan[1(✉)] and Oded Padon[2]

[1] Microsoft Research, Redmond, USA
kenmcmil@microsoft.com
[2] Tel Aviv University, Tel Aviv, Israel
odedp@mail.tau.ac.il

Abstract. This paper surveys the work to date on Ivy, a language and a tool for the formal specification and verification of distributed systems. Ivy supports deductive verification using automated provers, model checking, automated testing, manual theorem proving and generation of executable code. In order to achieve greater verification productivity, a key design goal for Ivy is to allow the engineer to apply automated provers in the realm in which their performance is relatively predictable, stable and transparent. In particular Ivy focuses on the use of *decidable* fragments of first-order logic. We consider the rationale or Ivy's design, the various capabilities of the tool, as well as case studies and applications.

Keywords: Deductive verification · Distributed systems
Safety verification · Liveness verification · Paxos · Decidable logics
Effectively propositional logic · Cache coherence
Model checking · Specification-based testing

1 Introduction

Ivy is a language and a tool for the formal specification and verification of distributed systems. The rationale underlying Ivy is that, to achieve a high degree of productivity in verification, the system, its representation and its proof must be designed in advance to take maximum advantage of automated provers while avoiding their weaknesses. Ivy is open-source software and is freely available under an MIT license [25].

The use of automated provers in program verification has a long history, going back to the work of Nelson and Oppen [28,29] and the Boyer-Moore prover [9]. More recent systems include ESC Java [12], Dafny [22] and F* [40]. Program proofs using such tools are typically more succinct than proofs using tactical theorem provers such as Coq [4] and Isabelle/HOL [30] by one or two orders of magnitude (e.g., compare the Ironfleet project [14] using Dafny to the Verdi [42,43] project using Coq). However, it is unclear that this succinctness leads to a proportionate improvement in verification productivity. In practice, users struggle with the unpredictability, instability and lack of transparency of the automated verifiers [11, Sect. 9.1]. Particularly problematic is the heuristic instantiation of quantifiers. This leads to unpredictable failures that are extremely hard to diagnose and may be triggered by small, seemingly irrelevant changes in the prover's input. By lack of transparency, we mean that no clear indication is given of the cause of failures. It is somewhat as if one were trying to develop a software system using a compiler that randomly failed to produce code, without producing any useful error message. In an iterative development process requiring frequent recompilations, this would be untenable.

To try to realize the verification productivity that automated provers promise, the design of Ivy starts with two basic choices: a prover and an application domain in which we wish to produce efficient verified systems. The chosen prover (at least initially) is Microsoft Z3 [27], a high-performance SMT solver [3] that supports satisfiability queries in full first-order logic modulo a variety of theories. The chosen application domain is distributed systems. The primary design goal of Ivy is to allow an engineer to quickly and intuitively reduce the proof of a distributed system to proof goals in a logical fragment for which Z3 is a *decision procedure*. We can think of Ivy as a test of the following three-part hypothesis:

1. Predictability, stability and transparency of proof automation lead to greater verification productivity,
2. Within the decidable fragments used by Ivy, the Z3 prover has these properties, and
3. With appropriate language and tool support, a we can reduce proofs of distributed systems to subgoals in this fragment.

2 Language Design

The use of decidable logics also has a long history in program verification. The choice of a logic generally depends on the application domain. For example, Klarlund proposed the use monadic second-order logic (MSO) for reasoning about manipulations of inductive data structures [15]. For distributed protocols, we posit that decidable fragments of first-order logic are more appropriate, since these protocols usually lack recursive structures, and uninterpreted relations and quantifiers can be used to reason about the multiple nodes or threads, as well as messages, values, and other objects of the system.

The classical example of a decidable fragment of first-order logic is the Bernays-Shönfinkel-Ramsey fragment (also known as EPR, for "effectively propositional"). This consists of formulas without function symbols, whose quantifier structure is $\exists\forall$ in prenex normal form. We can extend this in various ways. For example, in a many-sorted setting, we can allow function symbols that are stratified (*i.e.,* there are no cycles in the graph that the function symbols and the $\forall\exists$ quantifier alternations induce on the sorts). From this we observe that (1) quantifier structure is critical, and (2) function symbols and quantifier alternations should be used cautiously.

2.1 Ivy's Procedural Language

These considerations motivate several important design decisions in Ivy. First, the programming language is imperative rather than functional. Partly this is motivated by decidability. Often, though we are computing a total function, we do not wish to specify it as such, since this could contribute to a function cycle. Instead, we use a procedure and specify only partial correctness. Generally speaking, we avoid any unnecessary assumptions of totality or termination for decidability reasons. Second, the only primitive data type is the Boolean type. This is because primitive data types would introduce both total functions and axioms whose quantifier structure could be problematic. Instead, variables in Ivy hold first-order relations and functions over uninterpreted sorts as their values. This gives the user control of the use of function symbols. By preference, when reasoning about concrete data, we use relational abstractions whose axioms are expressible in the decidable fragment.

Finally, Ivy generates verification conditions (VC's) using the weakest precondition calculus, much like other tools, such as Dafny. The primitive constructs of the language have been chosen so that, not only are the VC's always expressed in first-order logic, but their quantifier structure is apparent from the program source. These considerations are discussed in more detail in [38], which describes the basic procedural language, which has since been extended.

Within these constraints, Ivy's programming language is designed to be as expressive as possible. We can express in Ivy any update to the variables whose transition relation is expressible in first-order logic, using relational and functional updates with free parameters. For example, the following assignment statement removes the pairs (x, Y) from relation r, for all Y:

```
r(x,Y) := false
```

The capitalized symbol Y is treated implicitly as a free parameter. The transition relation of this statement can be expressed as

$$\forall X, Y.\;\; r'(X,Y) = \text{false if } X = x \text{ else } r(X,Y)$$

We can also create pure first-order function closures. For example, this assignment computes a function f from the current value of the function g and a variable v:

```
f(X) := g(X,v)
```

While the semantics of this is easily expressed in first-order logic, compiling it is a bit more subtle. The compiler creates a closure that captures the values of g and v, allowing function f to be evaluated on demand.

Other first-order expressible updates, such as relational joins, are also possible. With quantifiers and parameterized updates, it is possible to describe procedures that are not actually computable. The compiler handles only a subset of the language in which finite bounds on quantifiers can be statically inferred. Uncomputable updates are still useful, however, in writing specifications.

2.2 Modularity

Another important generalization we can make about decidability is that mixing theories and quantifiers is problematic. For example, quantifier-free integer linear arithmetic with function symbols is decidable, but adding quantifiers makes it undecidable. Moreover, by mixing procedures that use function symbols, we might create a cycle in the function graph and thus also lose decidability.

Ivy's answer to this conundrum is modularity. That is, we hide problematic theories or functions inside modules, to prevent their combinations from taking us outside the decidable fragment. As a very simple example, suppose we require an index type t that forms a discrete total order. We implement this type in a module I which provides an interface with certain operations, such as incrementation (*i.e.,* computing the successor of a value), and guarantees certain properties, such as the axioms of total order. Type t is *interpreted* as the integers, that is, we instantiate the integer arithmetic theory for type t, giving us interpretations for the signature $\{0, 1, +, <, \ldots\}$ over this type. However, only module I sees this interpretation. Since its VC's are quantifier free, Z3 can decide them using its integer arithmetic theory. An application module A using the index type t sees only its abstract specification, and not the integer theory. Thus, it can use quantifiers safely. As we will observe in Sect. 4, this principle can be applied in more complex situations, for example in refining an abstract protocol model to an implementation.

To enforce the separation of theories, modularity in Ivy is quite strict. The specification of a module is never allowed to reference internal state of the module. Rather, the specification of a module provides an abstract notion of state. This consists of a collection of *monitors*: procedures that synchronize with call and return events at the interface, updating the abstract state. The monitors contain assertions that act as either assumptions or guarantees for the modules. Stateful monitors can be used, for example, to specify the interfaces of concrete services, such as networking layers, or abstract models that are used only in the proof.

2.3 The Fragment Checker

The decidable fragment used by Ivy is called the Finite Almost Uninterpreted fragment or FAU [13], which is supported by Z3. FAU generalized the many-sorted extension of EPR, and also allows restricted combination of quantifiers and linear arithmetic. The Ivy verifier generates the verification conditions for a program and checks syntactically that they fall into the FAU fragment. If not, it provides a diagnostic message that explains the failure (for example, it presents an illegal cycle of function symbols). This is important from the point of view of transparency. That is, if a VC cannot be verified, some feedback must be provided to help the user correct the situation and continue developing the proof.

3 Expressiveness of Decidable Fragments of First-Order Logic

Much of the verification using Ivy is done in a many-sorted extension of the EPR fragment of first order logic that allows only allows *stratified* or *acyclic* quantifier alternations and function symbols. Since it is a fragment of first-order logic, it may seem to be too restricted for challenging verification tasks. For example, first-order logic cannot express properties of arithmetic, graph reachability, or inductive data structures. Quite surprisingly, the work on Ivy shows that pure first-order logic, and even a decidable fragment thereof, is powerful enough to capture everything that is needed to verify several complex distributed protocols.

Transitive closure of deterministic paths can be expressed in EPR. This was used in [16–18] for linked data structures, and these ideas can also be used to represent tree topologies, such as forwarding trees of routing algorithms [35]. This also generalizes to other topologies, including rings [38], that can be similarly axiomatized in EPR, and more recently general graphs of out-degree one [32]. The key idea is to take the transitive closure as a primitive relation, and use a formula to represent edges. This allows for a sound axiomatization, which is also complete for finite models. That is, every finite model of the axioms corresponds to a graph of the suitable class. Due to the finite model property of EPR, this completeness ensures that counterexamples obtained in the verification process are never spurious.

Another useful axiomatization is that of *quorums*. Many distributed protocols employ quorums that are defined by thresholds on set cardinalities. For example, a protocol may wait for at least $\frac{N}{2}$ nodes to confirm a proposal before committing a value, where N is the total number of nodes. This is often used to ensure consistency. In Byzantine failure models, a common threshold is $\frac{2N}{3}$, where at most a third of the nodes may be Byzantine. First-order logic cannot completely capture set cardinalities and thresholds. However, we can exploit the fact that protocol correctness relies on rather simple properties that are implied by the cardinality threshold, and that these properties can be encoded in first-order logic.

The idea is to use a variant of the standard encoding of second-order logic in first-order logic. We introduce a sort for quorums, that is sets of nodes with the appropriate cardinality, and use a binary relation *member* to capture set membership. (Alternatively, we can add a sort that represents general sets of nodes, with a unary predicate over it that represents "being a quorum".) Then, properties that are needed for protocol correctness can be axiomatized in first-order logic.

For example, the fact that any two sets of at least $\frac{N}{2}$ nodes intersect is crucial for many consensus protocols. This property can be expressed in first order logic:

$$\forall q_1, q_2 : \mathrm{quorum_i}.\ \exists n : \mathrm{node}.\ \mathit{member}(n, q_1) \land \mathit{member}(n, q_2)$$

For Byzantine consensus algorithms that use sets of at least $\frac{2N}{3}$ nodes, they key property is that any two quorums intersect at a non-Byzantine node. This can also be expressed in first-order logic:

$$\forall q_1, q_2 : \text{quorum}_{\text{ii}}.\ \exists n : \text{node}.\ \neg \mathit{byz}(n) \wedge \mathit{member}(n, q_1) \wedge \mathit{member}(n, q_2)$$

These ideas are used in [36,37] to verify multiple consensus protocols from the Paxos [19,20] family, showing that properties that are expressible in first-order logic can be used to prove challenging protocols. For several variants, this provided the first mechanical safety proof.

4 Using Modularity to Verify Implementations

While the techniques outlined in Sect. 3 allow one to verify distributed protocols at the abstract protocol level, they do not suffice to verify an executable implementation. For verified executable implementations, we want to replace the notion of *axioms* with a notion of *interface specification* in a modular, assume/guarantee style. That is, we would like most of the proof to rely on first-order properties such as total order or quorum intersection, but then we would like to produce a concrete implementation, and prove that it satisfies these properties.

Concrete implementations rely on concrete data types such as integers, and data structures such as arrays. Ivy includes a built-in library of several concrete types with their specifications, and allows users to create user defined data types via a module system. Verification of concrete data type implementations is carried out in decidable theories, most commonly the FAU fragment mentioned earlier. This fragment allows restricted combination of quantifiers and arithmetic, and is supported by Z3. In [41], well-known modular verification techniques are applied to separate such theory reasoning, allowing the global protocol verification to be done in pure first-order logic (and EPR), while theories are isolated to particular implementation modules (for example, a module implementing finite sets with a quorum predicate).

An important tactic in that work is to use modularity to break cycles of function symbols or quantifier alternations. For example, if verification requires both a function (or $\forall\exists$ quantifier alternation) from sort A to sort B, and a function from sort B to sort A, then a possible solution is to break the problem into two modules, where each module can be verified with only one of the functions, thus avoiding cycles.

A typical approach is to introduce a "ghost" module that formalizes an abstract model of the protocol. The state of the ghost module is usually encoded using relations, allowing us to verify global properties of the protocol using EPR. The interface of the ghost module is called in the implementation module at the "commit points" of abstract operations. Thus, by assume/guarantee reasoning, we can use the proved properties of the ghost module as lemmas in the proof the implementation module. This allows us to use some quantifier alternations when proving the implementation module, and other quantifier alternations when proving the abstract protocol module, even though combining them would create cycles.

Although the approach uses only modular assume/guarantee reasoning in the proof, this method is still related to approaches based on refinement mappings [2]. In this case, an inductive invariant relating the ghost module's interface state and the implementation state takes the role of the refinement mapping. Although prophecy variables could in principle be used, this was found in [41] to be unnecessary in practice, as we have the flexibility to make the ghost module deterministic.

In [41], these principles are applied to obtain verified implementations of both Multi-Paxos and Raft [31]. The obtained implementations have performance that is on par with other verified and unverified implementations, and the proof burden is much lower compared to other verified implementations such as Verdi [42,43] (using Coq) and IronFleet [14] (using Dafny and Z3). Applying this methodology requires us to carefully consider the functional dependencies in the system while planning the specification and proofs. This effort was more than repaid, however, by the predictability, stability and transparency of Z3 when applied to proof goals in the decidable fragments. Overall, restricting the proof automation to the decidable fragments did not appear to be an insuperable obstacle and in practice resulted in more concise proofs.

5 Liveness and Temporal Verification

Safety properties can be proven using inductive invariants. However, liveness properties of infinite-state systems are usually proven using ranking functions or well-founded relations. Unfortunately, pure first-order logic (without theories) cannot express the required rankings or the notion of a well-founded relation or well-ordered set. Therefore, it may seem that liveness verification cannot be done in pure first-order logic. However, a new technique [33] integrated into Ivy shows that on the contrary, the formalism of first-order logic provides a unique opportunity for proving liveness and temporal properties.

The technique exploits the flexibility of representing states as first-order structures, and uses first-order temporal logic (FO-LTL) (e.g., [1,23]) for temporal specification. This general formalism provides a powerful and natural way to model temporal properties of infinite-state systems. It naturally supports both *unbounded parallelism*, where the system is allowed to dynamically create processes, and *infinite-state per process*. Unbounded-parallelism usually requires infinitely many (or quantified) fairness assumptions (e.g., that every thread is scheduled infinitely often in a program with dynamic thread creation, where an infinite trace can have infinitely many threads). This is fully supported by the formalism and the developed proof technique.

The technique developed in [33] and implemented in Ivy is based on a novel liveness-to-safety reduction, that reduces temporal verification (expressed in FO-LTL) to safety verification of an infinite-state system expressed in first-order logic without temporal operators. This allows us to leverage existing safety verification techniques, and the other techniques implemented in Ivy, to verify liveness and temporal properties. While such a reduction cannot be complete for complexity reasons[1], it is sound, and it was successful in proving liveness of several challenging protocols, including the first mechanized liveness proofs of Stoppable Paxos [21], and the TLB Shootdown algorithm [7].

The liveness-to-safety reduction is based on an abstract notion of acyclicity, using *dynamic abstraction*. For finite-state systems, liveness can be proven through acyclicity (the absence of fair cycles). This is the classical liveness-to-safety reduction of [5]. This also works for parameterized systems, where the state-space is finite (albeit unbounded) for every system instance [39]. For infinite-state systems, the acyclicity condition is unsound (an infinite-state system can be acyclic but non-terminating). The liveness-to-safety reduction with dynamic abstraction defines a finite abstraction that is fine-tuned for each execution trace, while abstracting only the cycle detection aspect (rather than the actual transitions of the system). Such fine-tuned abstraction is made possible by exploiting the symbolic representation of the transition relation in first-order logic, as well as the first-order formulation of the fairness constraints. The full details are explained in [33].

An additional novel mechanism [34] implemented in Ivy that enhances the proof power of the liveness-to-safety reduction is *temporal prophecy* and *temporal witnesses*. Here, the idea is to augment the system with additional temporal formulas that are not part of the specification, and also with additional constant symbols that are essentially Skolem witnesses for temporal formulas. In addition to increasing the proof power, temporal witnesses also facilitate verification of the resulting safety problem using EPR. By introducing a temporal witnesses, one can often eliminate quantifier alternations in the resulting verification conditions. The idea is that a temporal witness is used to name a particular element (e.g., the thread that is eventually starved), and then the inductive invariant can be specified for this particular constant, rather than with a quantifier. In several cases we considered, this allowed to eliminate quantifier alternation cycles. In Ivy, temporal prophecy formulas are derived from an inductive invariant provided by the user (for proving the safety property resulting induced by the liveness-to-safety reduction), which provides a seamless way to prove temporal properties.

6 Additional Topics

6.1 Compositional Simulation

As described in Sect. 2, module specifications in Ivy are stateful monitors. An additional use for these monitors is to generate tests for the module using a compositional testing approach [8]. That is, by symbolically executing the monitor in a given state, we can derive a predicate that represents all of the legal input values for a given procedure in that state. By sampling randomly from the satisfying assignment of this predicate, we can generate sequences of a test inputs.

[1] The temporal verification problem in this setting is Π^1_1-complete [1], while safety verification is in the arithmetical hierarchy.

For example, in a client/server protocol, Ivy can take the role of the client in testing the server, or the server in testing the client.

This modular approach to test generation has several advantages. First, compared to traditional unit testing, it has the advantage that it is in a limited sense complete. That is, we have a formal assume/guarantee proof that correctness of the modules implies correctness of the system. This means that if the system does not actually satisfy its specification, there is *some* unit test that exposes this (though this test might be generated with low probability). Compared to integration testing, the advantage is that it is easier to cover the behaviors of a module by stimulating its inputs directly rather than the system-level inputs. This is particularly important in the case of concurrent systems, which suffer from an explosion of interleavings. Because the module has less concurrency than the system, its possible interleavings are more easily explored.

In [26], this method is used to verify the hardware building blocks of a modular cache coherence system for the RISC-V processor architecture, based on a formal specification of the coherent interface. The approach was able to find subtle timing bugs in the RTL-level implementations, and also provides a limited guarantee that, if every block passes all possible tests, then the system as a whole provides the required memory coherence properties.

Specification-based testing also gives a way to check parts of the "trusted base" of Ivy, for example the networking interface, which is based on system services that cannot be formally verified.

6.2 Abstract Model Checking

Propositional LTL is another example of a decidable logic. Satisfiability problems in this logic can be reduced to circuit representations in a standard format [6] that can be checked by highly efficient hardware model checkers such as ABC [10]. Ivy can exploit such model checkers by means of an abstraction. As in an SMT solver, the first-order transition relation is reduced to its "propositional skeleton" by replacing each atomic formula with a free Boolean variable. Though all of the theory information is lost by this transformation, some can be regained by a process of "eager instantiation" of the theory axioms. This process can be controlled by the user by providing a collection of axiom schemata to be instantiated or by applying standard libraries of such schemata. The user can also increase precision by adding history and prophecy variables.

In [24] this approach is tested on a collection of distributed protocols. The ability of the model checker to automatically synthesize part of the system's inductive invariant is seen to substantially reduce the complexity of the invariants that must be provided manually.

6.3 Manual Theorem Proving

It some cases, it may be necessary to fall back on detailed manual proof. For this purpose, Ivy provides a collection of proof tactics that can be used to manually transform proof goals. A standard library provides complete proof rules for first-order logic in the natural deduction style. These can be used where needed for reasoning about specifications that are outside the decidable fragment, for example, to apply induction over the natural numbers using the Peano induction axiom. That is, while Ivy restricts *automated* proof generation to decidable fragments, manual proof is not restricted in this way.

7 Conclusion

A key design goal for Ivy is to allow the engineer to apply automated provers in a realm in which their performance is relatively predictable, stable and transparent. Ivy differs from other program verification tools, such as Dafny and F*, primarily in that its language and features have been designed based on the capabilities of a particular automated prover and the needs of a particular application domain. Ivy's design allows users to structure specifications, implementations and proofs to make maximum use of the capabilities of the prover while avoiding its weaknesses, particularly in the area of heuristic quantifier instantiation.

Case studies have provided preliminary evidence that such a methodology is practical, and that the resulting predictability, stability and transparency of the prover improves overall verification productivity. To some degree, this confirms the three-part hypothesis of the introduction. In particular, it appears that the performance of Z3 is substantially more stable within the decidable fragments, and that, with appropriate language and tool support, the restriction of automation to the decidable fragment is not unduly burdensome. Still, more experience is needed to say with certainty that this trade-off is the right one within the chosen domain and to validate the various design decisions.

Liveness proofs are yet to be integrated with Ivy's modular assume/guarantee reasoning. This is needed to verify liveness of system implementations, rather than abstract protocols. For this, module interfaces may need to be expressed in temporal logic, such that one module's liveness property becomes another module's fairness assumption. Other important issues have yet to be addressed, for example the verification of security or privacy properties. In the long run, the large size of the trusted computing base in Ivy must also be addressed.

Ultimately, the goal of the project is to realize in practice the promise of greater verification productivity inherent in powerful proof tools such as Z3.

Acknowledgements. We thank the many researchers that have contributed to the research agenda reviewed in this article, both as co-authors and via insightful discussions, including: Thomas Ball, Amir Ben-Amram, Nikolaj Bjørner, Tej Chajed, Constantin Enea, Yotam M. Y. Feldman, Jochen Hoenicke, Neil Immerman, Shachar Itzhaky, Ranjit Jhala, K. Rustan M. Leino, Giuliano Losa, Yuri Meshman, Leonardo de Moura, Alexander Nutz, Aurojit Panda, Bryan Parno, Andreas Podelski, Shaz Qadeer, Alexander Rabinovich, Mooly Sagiv, Sharon Shoham, Or Tamir, Zachary Tatlock, Marcelo Taube, James R. Wilcox, Doug Woos, and the anonymous referees and artifact evaluation referees of POPL, PLDI, OOPSLA, CAV, and FMCAD.

Padon was supported by Google under a PhD fellowship and by the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013)/ERC grant agreement no. [321174-VSSC].

References

1. Abadi, M.: The power of temporal proofs. Theor. Comput. Sci. **65**(1), 35–83 (1989). https://doi.org/10.1016/0304-3975(89)90138-2
2. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
3. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-825
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
5. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002)
6. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, July 2011
7. Black, D.L., Rashid, R.F., Golub, D.B., Hill, C.R.: Translation lookaside buffer consistency: a software approach. In: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS III, pp. 113–122. ACM, New York (1989). https://doi.org/10.1145/70082.68193
8. Blundell, C., Giannakopoulou, D., Pasareanu, C.S.: Assume-guarantee testing. ACM SIGSOFT Softw. Eng. Notes **31**(2) (2006). https://doi.org/10.1145/1118537.1123060
9. Boyer, R., Moore, J.: A Computational Logic. Academic Press, New York (1979)
10. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
11. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: using verification to disentangle secure-enclave hardware from software. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017, pp. 287–305 (2017)
12. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, pp. 234–245. ACM (2002). https://doi.org/10.1145/512529.512558
13. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25

14. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP, pp. 1–17 (2015)
15. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5
16. Itzhaky, S.: Automatic reasoning for pointer programs using decidable logics. Ph.D. thesis, Tel Aviv University (2014)
17. Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A., Sagiv, M.: Modular reasoning about heap paths via effectively propositional formulas. In: the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 385–396 (2014)
18. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 756–772. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_53
19. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229
20. Lamport, L.: Paxos made simple. ACM SIGACT News **32**(4), 51–58 (2001). https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
21. Lamport, L., Malkhi, D., Zhou, L.: Stoppable paxos. Technical report, Microsoft Research (2008). https://www.microsoft.com/en-us/research/publication/stoppable-paxos/
22. 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
23. Manna, Z., Pnueli, A.: Verification of concurrent programs: a temporal proof system. In: de Bakker, J.W., van Leeuwen, J. (eds.) Foundations of Computer Science: Distributed Systems, pp. 163–255. Mathematisch Centrum, Amsterdam (1983)
24. McMillan, K.L.: Eager abstraction for symbolic model checking. In: Conference on Computer-Aided Verification (CAV 2018). Springer (2018, to appear)
25. McMillan, K.L.: Ivy. http://microsoft.github.io/ivy/. Accessed 7 May 2018
26. McMillan, K.L.: Modular specification and verification of a cache-coherent interface. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, 3–6 October 2016, pp. 109–116. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886668
27. 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
28. Nelson, C.G.: Techniques for program verification. Ph.D. thesis, Stanford, CA, USA (1980). aAI8011683
29. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
30. Nipkow, T., Wenzel, M., Paulson, L.C.: A Proof Assistant for Higher-Order Logic Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9

31. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, 19–20 June 2014, pp. 305–319 (2014). https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
32. Padon, O.: Deductive verification of distributed protocols in first-order logic. Ph.D. thesis, Tel Aviv University (2018)
33. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. PACMPL **2**(POPL), 26:1–26:33 (2018). https://doi.org/10.1145/3158114
34. Padon, O., Hoenicke, J., McMillan, K.L., Podelski, A., Sagiv, M., Shoham, S.: Temporal prophecy for proving temporal properties of infinite-state systems (in preparation)
35. Padon, O., Immerman, N., Shoham, S., Karbyshev, A., Sagiv, M.: Decidability of inferring inductive invariants. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 217–231 (2016). https://doi.org/10.1145/2837614.2837640
36. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made epr: Decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017). https://doi.org/10.1145/3140568
37. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. CoRR abs/1710.07191 (2017). http://arxiv.org/abs/1710.07191
38. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 614–630 (2016)
39. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 328–343. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_26
40. Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. J. Funct. Program. **4**(23), 402–451 (2013)
41. Taube, M., Losa, G., McMillan, K.L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J.R., Woos, D.: Modularity for decidability of deductive verification with applications to distributed systems. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 662–677. ACM (2018). https://doi.org/10.1145/3192366.3192414
42. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 357–368 (2015)
43. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the raft consensus protocol. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, 20–22 January 2016, pp. 154–165. ACM (2016). https://doi.org/10.1145/2854065.2854081