Verification of Fault-Tolerant Protocols with Sally

Bruno Dutertre, Dejan Jovanović, and Jorge A. Navas

Computer Science Laboratory, SRI International

Abstract. Sally is a model checker for infinite-state systems that implements several verification algorithms, including a variant of IC3/PDR called Property-Directed K-induction. We present an application of Sally to automated verification of fault-tolerant distributed algorithms.

1 Introduction

Sally is a new model checker for infinite-state systems developed by SRI International. It is a successor of the Symbolic Analysis Laboratory (SAL) [6]. Sally supports bounded model checking and proof by k-induction, and it implements a novel model-checking algorithm based on IC3/PDR that can automatically discover k-inductive strengthening of a property of interest. Details of this *Property-Directed K-induction* (PD-KIND) algorithm are presented in [12].

We present an application of Sally to fault-tolerant distributed algorithms. We focus on a class of synchronous algorithms that consist of one or more rounds of communication between N processes—some of which may be faulty—followed by some form of averaging or voting to achieve agreement among processes. This type of algorithm is at the core of many fault-tolerant systems used in avionics or other control systems, including protocols for fault-tolerant sampling of sensor data and clock-synchronization protocols. Until the advent of PDR and relatives, such protocols could not be verified automatically by model checkers. The best technique available was k-induction, which is typically not fully automatic and requires expertise to discover auxiliary inductive invariants. We show that PD-KIND can automatically verify complex fault-tolerant algorithms, under a variety of fault assumptions.

2 Sally

Sally is a modular and extensible framework for prototyping and development of model-checking algorithms. Currently, Sally implements several algorithms based on satisfiability modulo theories (SMT), including bounded model checking and k-induction, and the novel PD-KIND algorithm [12]. PD-KIND generalizes IC3/PDR [4, 10] by relying of k-induction and k-step reachability as subprocedures, rather than ordinary one-step induction and reachability. The PD-KIND procedure relies on backend SMT solvers to provide features such as model-based generalization [13] and interpolants. The current implementation combines Yices 2 [8] and MathSAT 5 [5]. Other backend solvers can also be used for k-induction and bounded model checking. Sally is open source software available at `https://github.com/SRI-CSL/sally`.

The primary input language of Sally is called MCMT (for Model Checking Modulo Theories). This language extends the SMT-LIB 2 standard [1] with commands for defining transition systems. SMT-LIB 2 is used to represent terms and formulas. Transition systems are defined by specifying a state space, a set of initial states, and a transition relation. MCMT also allows one to specify invariant properties. Sally can parse other input languages than MCMT and internally convert them to MCMT. All our examples are written in a subset of the SAL language [7], converted to MCMT, then analyzed using Sally.

3 Modeling Fault-Tolerant Protocols

We use a simple modeling approach that is generally applicable to synchronous algorithms. The system state is a finite set of arrays indexed by process identities. Communication channels are also modeled using arrays (e.g., a channel from process i to process j is represented as an array element $c[i][j]$). Each transition of the system corresponds to one round of the algorithm: a process i updates its local variables then send data on one or more communication channel.

To model faults, we assign a status to each process and we specify faulty behavior as assumptions on the data transmitted by processes. A faulty process is then assumed to execute the algorithm correctly, except when it sends data. This approach simplifies process specifications and is sufficient for all types of process faults [20].

3.1 Approximate Agreement

We illustrate our approach using a protocol based on the unified fault-tolerant protocol of Miner et al. [16]. The protocol ensures approximate agreement. It assumes inexact communication, which models errors in sensor sampling or clock drifts. The fault model distinguishes between omissive and transmissive faults, and between symmetric and asymmetric behavior. A symmetric omissive process either fails to send data (on all its channels) or sends correct data. An asymmetric omissive process may send nothing to some and correct data to other processes. A symmetric transmissive process sends possibly incorrect data, but it sends the same data on all its channels. An asymmetric transmissive process behaves in an arbitrary way.

The protocol involves N processes. Process i holds a real value $v[i]$. In each round, this process broadcasts its value to the all processes,[1] computes a fault-tolerant average of the values it receives, and updates $v[i]$ using this average. The protocol is intended to ensure *convergence*: the absolute difference between $v[i]$ and $v[j]$ is approximately reduced by half with each protocol round.

[1] To avoid special cases, we assume that i is included in the set of recipients.

We model this protocol as a single state-transition system that operates on arrays. The main state variables include an array `v` that stores process values, and arrays `m` and `c` that model communication channels:

```
v: ARRAY PID OF DATA,
m: ARRAY PID OF ARRAY PID OF BOOLEAN,
c: ARRAY PID OF ARRAY PID OF DATA,
```

Variable `m[j][i]` indicates that the message from i to j is missing. If `m[j][i]` is true, then `c[j][i]` is ignored, otherwise `c[j][i]` is the value that j receives from i. We formalize the fault model as constraints on `m’[j][i]` and `c’[j][i]` based on the status of process i. These constraints are as follows:[2]

```
(FORALL (i: PID): status[i] = Good =>
  (FORALL (j: PID): NOT m’[j][i] AND received(v[i], c’[j][i])))

(FORALL (i: PID): status[i] = SymmetricOmissive =>
  (FORALL (j: PID): m’[j][i]) OR (FORALL (j: PID): received(v[i], c’[j][i])))

(FORALL (i: PID): status[i] = AsymmetricOmissive =>
  (FORALL (j: PID): m’[j][i] OR received(v[i], c’[j][i])))

(FORALL (i: PID): status[i] = SymmetricTransmissive =>
  (FORALL (j: PID): m’[j][i])
      OR (FORALL (j, k: PID): c’[j][i] - c’[k][i] <= 2 * epsilon))
```

The parameter ϵ is a bound on communication error; if a process sends a value x then the recipient reads a value in the interval $[x-\epsilon, x+\epsilon]$. In the above rules, this communication error is specified by predicate `received`:

```
received(x: DATA, y: DATA): BOOLEAN = x - epsilon <= y AND y <= x + epsilon;
```

A non-trivial part of the model is the definition of the fault-tolerant average. We use a form of mid-value select, parameterized by an constant τ: when a process i receives $n \leq N$ values in round k, it sorts these values in increasing order to form a sequence of reals $x_1, \ldots, x_n$. The mid-value select is the average of $x_{\tau+1}$ and $x_{n-\tau}$. (If $n < \tau$, a default value is chosen.) In practice, the parameter τ is equal to the number of asymmetric faults to tolerate, and must be chosen so that $n > 2\tau$.

We do not want to write a sorting algorithm in SAL, as translation to MCMT requires all functions applications to be inlined. For any sorting algorithm, this unrolling inevitably would cause an exponential blowup. Instead, we use a specification trick. We introduce two auxiliary state variables `p` and `n`:

```
p: ARRAY PID OF ARRAY PID OF PID,
n: ARRAY PID OF [0 .. N],
```

For a process i, `n[i]` denotes the number of messages received by i, and `p[i]` is a permutation of the indices in $\{1, \ldots, N\}$ that enumerates the n received values in increasing order. We specify these relations as shown in Figure 1. This essentially states the post-condition of the sorting algorithm we need. The input is an array v of N values and an array m of Boolean flags; where $m[i]$ true means that $v[i]$ is missing. The output includes a variable n that counts the number of

[2] The actual SAL syntax is less readable but equivalent.

non-missing elements, and a permutation p that sorts the non-missing elements in increasing order. From p, n, and v, we can easily define the mid-value select.

```
sort_and_filter(v: ARRAY PID OF DATA,
                m: ARRAY PID OF BOOLEAN,
                n: [0 .. N],
                p: ARRAY PID OF PID): BOOLEAN =
      (FORALL (i: PID): (i>n <=> m[p[i]]))
  AND (FORALL (i: PID): i<n => v[p[i]] <= v[p[i+1]])
  AND (FORALL (i, j: PID): p[i] = p[j] => i = j);
```

Fig. 1. Sorting and filter predicate

The final step is to specify the convergence property. The values $v[i]$s are initially within some distance Δ of each other and get closer and closer with each protocol round. Because of the communication error, the best bound one can achieve is 2ϵ. The protocol converges towards this bound at an exponential rate. A more precise specification is shown in Figure 2. We add a state variable `delta` to our state-transition system to store the bound. The variable is initialized to an arbitrary bound larger than 2ϵ, then it is updated with every protocol round as shown in the figure. Our goal is to show that the `convergence` property is invariant: the difference between `v[i]` and `v[j]` is bounded by `delta`.

```
% Initial precision: maximum difference between the initial values
initial_delta: { x: REAL| x > 2 * epsilon };

% Convergence function: if all values are with some delta
% at round k then they are within next(delta) at round k+1.
next(x: REAL): REAL = x/2 + 2 * epsilon;

% Precision improvement with each round
delta' = next(delta);

% Convergence property for the approx system
convergence: LEMMA approx |- G(FORALL (i, j: PID): v[i] - v[j] <= delta);
```

Fig. 2. Convergence and approximate agreement property

3.2 Verification Results

We have analyzed the approximate agreement protocol under four scenarios, and for different values of N. Each scenario makes different fault assumptions: no faults (Scenario 0); one symmetric transmissive and one asymmetric omissive faults (Scenario 1); one asymmetric transmissive and one asymmetric omissive faults (Scenario 2); and one asymmetric transmissive, one asymmetric omissive, and one symmetric omissive faults (Scenario 3). The results are summarized in Table 1. The left part shows the results and runtime of Sally's k-induction engine. The right-hand side shows results and runtimes of Sally's PD-KIND. The k-induction engine iteratively tries k-induction for k from 1 to 10.

	K-induction				**PD-Kind**			
N	Scen. 0	Scen. 1	Scen. 2	Scen. 3	Scen. 0	Scen. 1	Scen. 2	Scen. 3
4	u 88	i 0	i 0	i 0	v 3	i 0	i 0	i 0
5	t _	t _	t _	i 0	v 57	v 84	v 47	i 1
6	t _	t _	t _	t _	v 397	v 1742	v 1771	v 461
7	t _	t _	t _	t _	v 3581	v 3935	v 4838	v 3124

Table 1. Analysis results. Each entry reports the result and runtime of an experiment. v means *valid* (the property was proved), i means *invalid* (a counterexample was produced), u means *unknown* (*k*-induction was inconclusive), t means timeout. Runtimes are CPU time in seconds. The timeout is 5000 seconds.

		M															
	N	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	3	0.04	0.05	0.06	0.08	0.09	0.11	0.21	0.18	0.24	0.32	0.46	0.43	0.53	0.85	0.85	0.91
P1	4	0.06	0.08	0.13	0.16	0.21	0.22	0.30	0.41	0.56	0.61	0.74	0.67	1.04	0.94	1.47	1.66
	5	0.09	0.15	0.21	0.27	0.37	0.54	0.79	0.77	0.86	1.11	1.08	1.31	1.87	2.55	3.11	3.36
	3	0.03	0.04	0.06	0.08	0.11	0.13	0.14	0.18	0.30	0.36	0.44	0.48	0.59	0.85	0.94	1.04
P2	4	0.06	0.10	0.13	0.16	0.30	0.28	0.40	0.35	0.47	0.78	1.02	0.83	1.02	1.17	1.77	1.61
	5	0.09	0.13	0.20	0.38	0.37	0.47	0.49	0.70	0.88	1.20	1.14	1.65	2.02	2.28	2.73	3.38

Table 2. Runtimes on variants of the OM1 protocol.

The convergence property is not *k*-inductive, so *k*-induction cannot prove it. On the other hand, for instances where the property does not hold, then the *K*-induction engine finds a counterexample very quickly. PD-KIND works much better. On all instances where the property holds, it can automatically prove it. On all instances where the property is false, it can find a counterexample. The verification cost tends to be higher with more complex fault models. Because the algorithm is quite complex, scalability is an issue. The runtime grows very quickly as N increases, both for the PD-KIND and *k*-induction engines. We believe the complexity of the averaging function is the main bottleneck for this example.

We have verified a simpler fault-tolerant algorithm such as OM1, which uses majority voting. Our formalization is based on the Boyer-Moore algorithm [3]. Table 2 shows runtimes for a variant of OM1 that uses N processes and M relays. We used Sally to prove the two classic properties of OM1—agreement (**P1**) and validity (**P2**)—in a scenario with one Byzantine-faulty relay. It turns out that both properties are *k*-inductive. As shown in the table, the *k*-induction engine proves the properties in a few seconds at most. PD-KIND also works for these examples, but it is much slower. For example, proving agreement for $N = 5$ and $M = 20$ takes about 4 sec for *k*-induction and more than 20 min for PD-KIND.

4 Related Work

Developing correct distributed algorithms is notoriously hard; making sure that these algorithms tolerate failures is even harder. Since the 1980s, formal methods have been used to precisely model and mathematically prove the correctness of such fault-tolerant algorithms. Most of this work use interactive theorem provers (e.g., [18,17,15,21]). More recently, Padon et al. use a semi-automated proof method based encoding protocol rules into a decidable logic [19].

Model checking using abstraction technique has also been applied to this domain. Konnov and his colleagues show how a threshold-based algorithms can be modeled using counter systems, and develop verification algorithms for these systems [11,14]. An earlier example by Fisman et al. [9] uses another abstraction technique and applies regular model checking [2]. These abstraction methods are limited to special classes of protocols. The type of algorithms that we have presented manipulate numerical data and rely on non-trivial computation, and do not belong to these classes. When abstractions are not applicable, proofs using *k*-induction are possible but such proofs can be difficult, and require expertise to identify key auxiliary invariants.

5 Conclusion

Until recently, automated verification of complex fault-tolerant algorithms was impossible. One either had to resort to interactive theorem proving—which is slow and requires expertise—or rely on semi-automated method such as *k*-induction. New model-checking algorithms based on IC3/PDR have changed the picture; it is now feasible to verify a rich class of fault-tolerant protocols in a fully automated manner. Remaining challenges include improving scalability of these methods and extending them to richer logical theories.

References

1. C. Barrett, A. Stump, C. Tinelli, et al. The SMT-LIB standard: Version 2.0.
2. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.
3. R. S. Boyer and J. S. Moore. MJRTY—a fast majority vote algorithm. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Blesdoe*, pages 105–117, 1991.
4. A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.
5. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, pages 93–107. 2013.
6. L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *CAV*, pages 496–500, 2004.
7. L. de Moura, S. Owre, and N. Shankar. The SAL Language Manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, 2003.
8. B. Dutertre. Yices 2.2. In *CAV*, pages 737–744, 2014.

9. D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, pages 315–331, 2008.
10. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171. 2012.
11. A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.
12. D. Jovanović and B. Dutertre. Property-directed k-induction. In *FMCAD*, pages 85–92, 2016.
13. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.
14. I. Konnov, H. Veith, and J. Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV*, pages 85–102, 2015.
15. P. Lincoln and J. Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS*, pages 107–120, 1994.
16. P. Miner, A. Geser, L. Pike, and J. Maddalon. A unified fault-tolerance protocol. In *FORMATS/FTRTFT*, pages 167–182, 2004.
17. P. S. Miner. Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349, 1993.
18. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.
19. O. Padon, G. Losa, M. Sagiv, and S. Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *OOPSLA*, 1:108:1–108:31, 2017.
20. L. Pike, J. Maddalon, P. Miner, and A. Geser. Abstractions for fault-tolerant distributed system verification. In *TPHOL*, pages 257–270, 2004.
21. J. R. Wilcox, D. Woos, P. Pancheckha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.