NASA/CR–2018-219834

Advanced Symbolic Analysis Tools for Fault-Tolerant Integrated Distributed Systems

Bruno Dutertre, Dejan Jonanovic, and Jorge Navas
SRI International, Menlo Park, California

May 2018

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/CR–2018-219834

Advanced Symbolic Analysis Tools for Fault-Tolerant Integrated Distributed Systems

Bruno Dutertre, Dejan Jovanovic, and Jorge Navas
SRI International, Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNX14AI05A

May 2018

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

The project aims to develop advanced model-checking algorithms and tools to automate the verification of fault-tolerant distributed systems for avionics. We present a new method called Property-Directed K-Induction (PD-KIND) for synthesizing K-inductive invariants of state-transition systems. PD-KIND builds upon Satifiability Modulo Theories (SMT) to generalize Bradley's IC3 method and its variants. This method is implemented in a new tool called SALLY. Case studies show that PD-KIND can automatically verify fault-tolerant algorithms under a variety of fault models and that SALLY is competitive with other SMT-based model checkers.

Contents

1 Goals

The project aims to advance the state of the art in verification of fault-tolerant distributed systems.

With the advent of powerful decision procedures and automated provers known as Satisfiability Modulo Theory (SMT) solvers, it is now possible to verify complex, fault-tolerant systems relevant to avionics in a semi-automated manner. One commonly starts with a state-machine model of the system or algorithm under investigation, and apply model checking tools to prove (or disprove) that the system satisfies critical properties under suitable assumptions about faults. Current proof techniques typically rely on bounded model checking [4] and induction. These techniques can be effective but they are usually not fully automatic, as a human expert must provide auxiliary lemmas that may be intricate and difficult to discover.

Our goal is to make analysis of fault-tolerant, distributed systems more effective and practical. Our focus is twofold: we want to automate verification as much as possible so that model-checking tools can be used by non-experts, and we want to support analysis of general system instances that consist of an arbitrary number of components.

2 Accomplishments

In this project, we have focused on IC3, a model-checking method invented by Aaron Bradley in 2011 [7]. IC3, also known as property-directed reachability [21] (PDR), has revolutionized model checking of finite-state systems. IC3 and its variants combine forward and backward reachability techniques to automatically verify invariants. These methods have been shown to be highly effective, and more scalable, than alternative approaches such as bounded model checking and verification techniques based on Craig interpolants (e.g. [30]). The original IC3 focuses on finite systems encoded as Boolean circuits and relies on powerful Boolean SAT solvers as the underlying reasoning engine. Since models of fault-tolerant systems are typically not finite, we have investigated extensions of IC3 to infinite models encoded in (fragments of) first-order logic, and with SAT solvers replaced by SMT solvers. IC3-based methods have the potential to significantly simplify the verification of infinite systems — they are fully automatic and do not require an expert to provide auxiliary invariants. IC3 discovers the "right" state invariants on its own.

Starting from previous work [8, 9, 24], we have developed an original extension of IC3 to infinite-state systems. We have implemented this algorithm in a new model-checking tool called SALLY. SALLY is more than just a prototype implementation. It is intended to be a general framework for prototyping and developing model-checking algorithms that depend on SMT solvers. SALLY supports several backend SMT solvers and implements verification algorithms others than IC3 (e.g., bounded model checking and k-induction). We evaluated our original algorithm and implementation on a set of case studies derived from existing fault-tolerant algorithms. These experiments showed that SALLY and our version of IC3 are more robust than other SMT-based model-checking algorithms. SALLY also improves over our older model checker (SAL [17]) as SALLY can automatically verify properties that are not k-inductive.

In the second year of the project, we have continued to develop SALLY and we have evaluated the algorithms on an extended set of more demanding case studies. We have applied SALLY to fault-tolerant systems and to examples coming from software verification. These case studies have identified scalability issues with our original algorithm. For valid properties, the invariant-generation process required an excessive amount of supporting facts, which resulted in high memory consumption and long runtimes. For invalid properties, this large amount of auxiliary facts could also stifle the counter-example search when the property is invalid at deep frames (many steps). After trying several remedies, including heuristic management of learned facts and optimistic trace extension, we attacked the problem in a more fundamental way. We replaced the induction core of our algorithm with a more powerful k-induction core, resulting in a novel verification algorithm that we call PD-KIND (Property-Directed k-Induction). PD-KIND combines the powers of both IC3

and k-induction. It is more powerful and effective than each of them, both theoretically and in practice. The additional reasoning power of k-induction allows the new algorithm to produce more concise invariants that are, at the same time, easier to find — in some cases reducing verification time from minutes to seconds.

In the third year, we have focused on improving the usability of SALLY by developing native support for (a subset of) the SAL language to MCMT, the specification language used by SALLY. This allows SALLY users to write state-machine models in the expressive SAL language, and use the advanced algorithms provided by SALLY for verification. We have also prototyped extensions of SALLY to handle fully parametric systems and added new features to SALLY, including techniques based on abstract interpretation to infer system invariants.

We have published results from the project in peer-reviewed international conferences, including the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD'2016) and the 18th Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'2017). Our FMCAD paper was awarded an Honorable Mention in the Best Paper category. We presented a tutorial on SALLY at the Automated Formal Methods workshop (AFM), which was held in connection with the NASA Formal Methods Conference in May 2017.

The software developed under this project is open source and distributed for free by SRI International. The software, documentation, and examples are available at `http://sri-csl.github.io/sally/`.

3 SALLY: An Extensible SMT-Based Model Checker

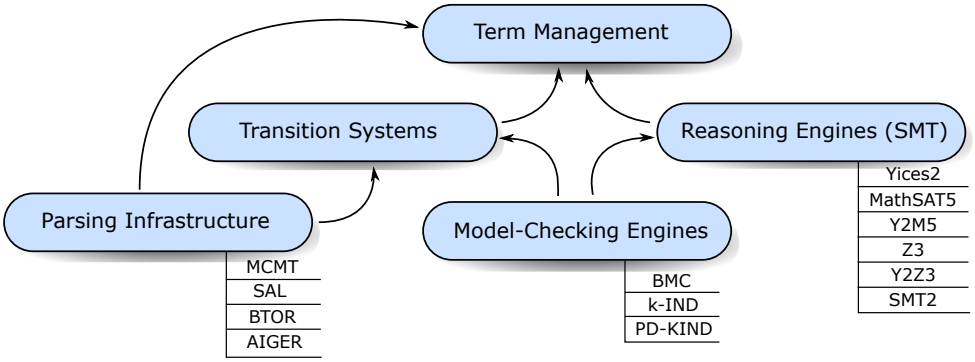

Figure 1. SALLY's architecture.

SALLY is the model checker we developed for this project. It relies on a modular and efficient software infrastructure that supports prototyping and development SMT-based model-checking algorithms. The architecture and the main modules of SALLY are sketched in Figure 1. Using this modular architecture, different backend SMT reasoning engines can be used, either individually or in combination. This allows complementary solver features to be used when needed. Currently, SALLY supports SRI's Yices 2 solver [19], the MathSAT 5 [10] solver, Z3 solver [15], or any other SMT2 compliant solver. For good balance of efficiency and support, SALLY is implemented in C++ (around 50000 LOC), and its main features include:

- An extensible, fast, and memory-efficient representation of terms and formulas. The term-management core supports the rest of the systems and takes care of low-level details such as memory management and garbage collection.
- A set of high-level classes that can model and operate on state transition systems. These classes provide a clean API allowing for a natural description of all relevant model-checking algorithms.

- A modular interface for importing SMT-based reasoning engines into the framework. The SMT-solver API abstracts the internals of the particular solver and makes a range of solvers available to the rest of the system. It also provides full access to the functionalities needed for model-checking purposes.
- An extensible parsing infrastructure that enables one to easily add new languages and extensions. Currently, SALLY can read state-transition systems specified in SALLY's own input format called MCMT (model checking modulo theories) or in the BTOR format (which is a standard input format for finite-state systems). A parser for the more general and expressive SAL language is in development.
- Implementation of several model-checking algorithms such as the IC3, developed in the first year, the PD-KIND procedure developed in the second year, and other algorithms such as SMT-based bounded model checking and k-induction. Any of these algorithms can be selected trough command-line options.

To illustrate the power and flexibility of the framework Figure 2 presents SALLY's simple implementation of a bounded checking engine in under 50 lines of code. The simplicity that the framework allows in describing model-checking algorithms is an enabling factor for SALLY to be an effective research platform but, more importantly, it improves the assurance that the resulting model-checking algorithms are correct.

SALLY is open source (GPL) and available at `http://sri-csl.github.io/sally/`, where further documentation and examples are available.

3.1 MCMT Language

The MCMT notation supported by SALLY is an extension of the SMT-LIB standard for SMT problems. SMT-LIB is a Lisp-like language used in the SMT community to represent terms, formulas, and SMT-relevant commands. MCMT builds on SMT-LIB as a term representation language, and extends it with syntax and commands for specifying states, initial conditions, transition relations, and properties. A simple example is shown in Figure 3. The language is designed to be easy for model checking tools to process, yet expressive enough to model the instances of fault-tolerant systems targeted by this project. The language contains several more advanced features beyond these basic commands, and will keep evolving as we target more complex fault-tolerant system.

3.2 Property-Directed k-Induction

The main new development in the second year of the project is a novel method for model checking of infinite state systems that we call PD-KIND (for Property-Directed k-induction). For infinite-state systems, IC3 and k-induction are the two commonly used methods, and in the first year we have implemented both (an original version) of IC3 and k-induction in SALLY.

At its core, the IC3 algorithm is based on induction. To show that a property is invariant, IC3 tries to incrementally produce an inductive strengthening of the property. Surprisingly, the relative power of k-induction and induction-based methods has not been studied in detail.[1] It is folklore knowledge that k-induction can be stronger than induction, but, to the best of our knowledge, this has never been formally accounted for. One of the goals of this project was to improve the automation of model-checking tools and understanding the relationship between different methods is crucial. We have analyzed the relationship between IC3 and k-induction and have shown that k-induction can be strictly more powerful than regular induction. For some classes of systems, k-induction can produce much more succinct invariants than regular induction. For other classes, there are properties that can be proved with k-induction but not with regular induction [25].

[1] Some IC3-variants, such as PDR [24], are not guaranteed to terminate even if the property to prove is already inductive.

```
result bmc_engine::query(const transition_system* ts, const state_formula* sf) {

  // Scope for push/pop on the solver
  solver_scope scope(d_solver);
  scope.push();

  // Initial states
  term_ref init = ts->get_initial_states();
  d_solver->add(d_trace->get_state_formula(init, 0), solver::CLASS_A);
  // Transition formula
  term_ref trans = ts->get_transition_relation();
  // The property
  term_ref p = sf->get_formula();
  term_ref p_not = tm().mk_term(TERM_NOT, p);

  // BMC loop
  size_t bmc_min = ctx().get_options().get_unsigned("bmc-min");
  size_t bmc_max = ctx().get_options().get_unsigned("bmc-max");
  for (size_t k = 0; k <= bmc_max; ++ k) {
    if (k >= bmc_min) {
      // Check the current unrolling
      scope.push();
      d_solver->add(d_trace->get_state_formula(p_not, k), solver::CLASS_A);
      solver::result r = d_solver->check();
      // See what happened
      switch (r) {
      case solver::SAT:
        return INVALID;
      case solver::UNSAT:
        break;
      default:
        return UNKNOWN;
      }
      // Pop the solver
      scope.pop();
    }
    // Unroll once more
    term_ref trans_k = d_trace->get_transition_formula(trans, k);
    d_solver->add(trans_k, solver::CLASS_A);
  }

  return UNKNOWN;
}
```

Figure 2. Example BMC implementation.

These results argue for an IC3-style method that is based on (or integrated with) k-induction. The additional reasoning power is particularly important when one works within an expressive logical theory such as the theory of arrays [22, 27], which is useful for modeling parameterized systmes. Our new method PD-KIND builds on the modular nature of IC3 we have developed in SALLY: satisfiability checking, reachability checking, and generation of inductive invariants are independent layers as depicted in Figure 4. Isolating these functionally independent modules allowed us to replace the inductive core with k-induction, producing a method that is a natural combination of IC3 and k-induction. This method, in addition to being effective in practice, can be shown to be at least as powerful than k-induction (provided the interpolation procedure satisfies a natural property that we call finite-covering).

We have implemented the new procedure in the SALLY tool, and our experimental evaluation shows that our prototype is at least as effective as the state-of-the-art infinite-state model checkers. An interesting aspect of the PD-KIND method is that it is parametric in the maximal k-induction depth k_m that is to be considered. This k-induction "knob" gives rise to different variants IC3 $=$ PD-KIND$_1$, PD-KIND$_2$, $\ldots$, PD-KIND$_\infty$.

```
;; State variables bundled into state_type: one variable x of type Real
(define-state-type state_type ((x Real)))

;; Initial states (a formula over state_type variables): x is zero
(define-states initial_states state_type
  (= x 0)
)

;; One transition: increase x by one
(define-transition transition state_type
  ;; Implicit variables next, state of state_type
  (= next.x (+ state.x 1))
)

;; The system definition using above
(define-transition-system T
  state_type
  initial_states
  transition
)

;; Assume that x is always positive
(assume T (>= x 0))

;; Query (any state formula over state_type)
(query T (not (= x (/ 1 2))))
```

Figure 3. Example MCMT Specification

These variants have different strengths and give the user the flexibility of tuning the performance of the tool to the particular problem at hand. Since unrolling of the transition relation can negatively effect solver performance, lower values of k_m tend to work better for problems that have a complex transition relation. On the other hand, for complex problems that do not have simple inductive invariants, a larger k_m is better since k-induction and can produce much simpler invariants.

The PD-KIND algorithm is presented in details in the appendix.

3.3 Support for the SAL Language and Parameterized Systems

Although the MCMT language is sufficient for modeling infinite state systems, it is designed for easy processing by tools and can be too low-level for users. Parametric systems are more conveniently specified using a richer language with more features than MCMT. Writing specifications in a low-level language such as MCMT can be cumbersome, time-consuming, and error prone. In the first years of the project, we have relied on SAL for modeling the case studies, and we have used SAL tools for translating models to MCMT. This translation is not fully automated and requires manual editing. In order to streamline this process, and to support reasoning about (non-instantiated) parametric systems, we have developed support for the SAL language [18] directly in SALLY. Although not all of the many SAL features are finalized yet, SALLY now supports the most important SAL features that are used in our case studies: arrays, records, tuples, enumerated types, predicate subtypes, quantifiers, functions (including recursive functions), and parametric module composition (both synchronous and asynchronous).

3.4 Counting Constraints and Parameterized Systems

The main goal of SALLY (and this project) is verification of fault-tolerant parameterized protocols. Most automated methods, including PD-KIND, are applicable only to systems with a fixed number of components. In order to support freely parametric systems, where the number of components is not fixed a priori, we have

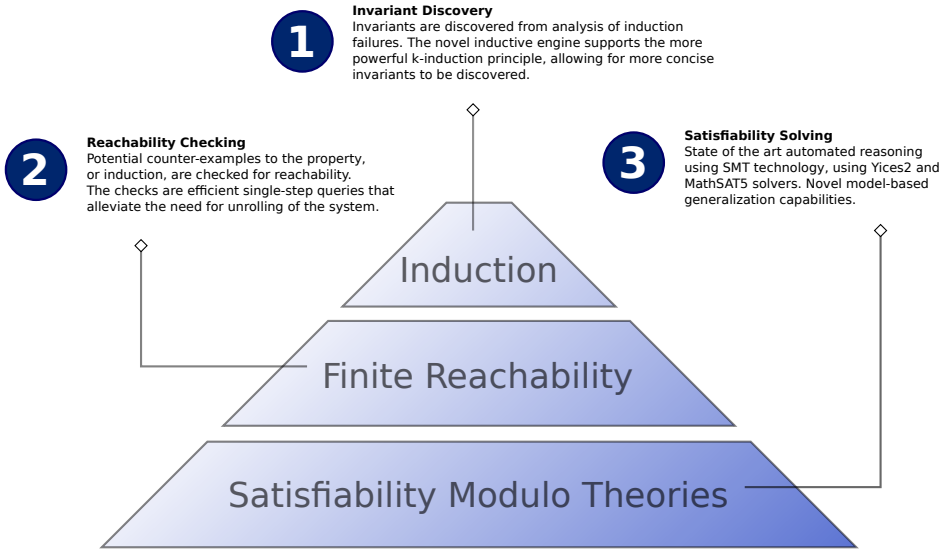

Figure 4. Main idea of PD-KIND.

developed decision procedures that can reason symbolically about constraints that can express "counting". The ability to reason about number of solutions is crucial in modeling of fault-tolerant protocols. A typical example of counting constraints is an assumption of the form "less than one third of the processes are faulty" appearing, e.g., in the Byzantine Generals' Problem [28]. Although several methods and tools have been proposed for analysis of parametric protocols, they are not well suited to fault-tolerant protocols. For instance, the CUBICLE model checker [12] supports analysis of parameterized systems such as cache coherence protocols, but it cannot express counting. Recent developments based on the MCMT framework do support counting constraints [1], but the proposed decision procedures are mostly theoretical and cannot be integrated well into an invariant-discovery algorithm such as IC3. As a step toward parametric-system verification, we have developed a decision procedure that can decide counting constraints in arithmetic, and extended it to counting constraints with arrays. The prototype implementation is available at `https://github.com/xapantu/counting-smt/`. This decision procedure can be used to check properties of parametric fault-tolerant protocols. Both decision procedures are implemented as overlays to an existing SMT solver making them readily applicable on top of existing technology.

3.5 Learning Invariants Using Abstract Interpretation

Abstract interpretation [13] (AI) is a theory of sound approximation of the semantics of transition systems. It has been successfully used to produce inductive invariants in software. The use of AI techniques in this project is appealing for its scalability and as a complement to the invariant-generation techniques used in PD-KIND. PD-KIND relies on interpolation to find an inductive strengthening of the property and, although interpolants have been relatively effective they also have significant limitations: (a) different proofs can generate different interpolants making the invariant generation unpredictable, and (b) interpolants are syntactic in nature since they are extracted from proofs generated by SMT solvers. Instead, AI is semantic in nature, much more predictable, and can discover invariants that are out of reach of interpolation.

The MCMT specification in Figure 5 exemplifies the benefits of combining AI and PD-KIND. The goal is to prove that the simple linear system satisfies the property $(x < 1)$. But PD-KIND cannot find an inductive

```
(define-state-type state_type ((x Real) (y Real)))
(define-states initial_states state_type
  (and (= x 0) (= y (/ 1 2)))
)
(define-transition transition state_type
  (and (= next.x (+ state.x state.y))
       (= next.y (/ state.y 2)))
)
(define-transition-system T
  state_type initial_states transition
)
(query T (< x 1))
```

Figure 5. Example where PD-KIND benefits from a semantic invariant inference method such as AI.

strengthening for this property. Instead, it learns a series of inequalities of the form

$$2^k x + (2^{k+1} - 1)y \leq (2^{k+1} - 1)/2$$

for increasing values of k. These inequalities hold up to a certain number of states, but they do not make an inductive invariant. Due to its reliance on interpolant-based (syntactic) invariant inference, the PD-KIND method can not infer an appropriate strengthening, and it generates an ever-increasing set of auxiliary lemmas that fails to converge in a finite number of steps.

On the other hand, AI with the Polyhedra domain [14] infers two inductive invariants for this system: $(x + 2y = 1)$ and $(0 \leq x \leq 1)$. While these two invariants do not directly imply $(x < 1)$, they are helpful in completing the proof. The learned invariant $(x + 2y = 1)$ is enough to strengthen the property and both PD-KIND and k-induction can immediately prove that $(x < 1)$ is invariant.

In the third year of this project, we have prototyped a method that combines AI techniques with PD-KIND. For doing this, we extended SALLY with the capability to perform abstract interpretation of MCMT specifications. Conceptually, this could be seen as a new engine in Figure 1. However, building abstract interpreters from scratch is a difficult and very time-consuming engineering task. Instead, we took a more pragmatic approach and integrated an existing abstract interpreter (one provided by the SeaHorn [23] verification framework) into SALLY. The SeaHorn abstract interpreter provides efficient fixpoint iterators and numerical abstract domains with different level of precision/efficiency trade-offs. However, SeaHorn focuses on verification of LLVM programs and therefore it takes as input a control-flow graph (CFG)-based language. To circumvent this, we implemented a translation from MCMT specifications to the CFG-based language understood by the abstract interpreter.

4 Case Studies

We have evaluated our algorithms on a range of existing fault-tolerant systems and on additional problems from software verification. We have modeled all the fault tolerant systems in the SAL language. Using the existing SAL tools, we have translated the models to a so-called *flat module* representation. Since the SAL notion of a flat module is very close to MCMT, this representation can then be translated to SALLY's MCMT with minimal manual edits. All SAL and MCMT models are available as examples in SALLY's repository.

4.1 Fault-Tolerant Systems

Oral Message Protocol (OM1). OM1 is the simplest version of Oral Message protocol for Byzantine Agreement due to Lamport, Shostak, and Pease [28]. The protocol has N participants. One of them, the

source, attempts to reliably distribute a message m to the others. The protocol has two goals: all non-faulty participants receive the same message (*agreement*), and, if the source is non-faulty then message m is received (*validity*).

These two properties are achieved if at most one of these participants is Byzantine-faulty, provided N is at least 4. We have modeled a variant of this protocol that consists of four main processors and four relays in SAL and converted the model to MCMT.

Clock Synchronization in Timed-Triggered Ethernet. Our second case study is based on a earlier study of Timed-Triggered Ethernet (TTE) that was part of a previous NASA-funded project [20]. In this previous project, we used SAL to build a model of the TTE fault-tolerant clock-synchronization algorithm and computed bounds on the synchronization guarantees. At the time, we performed the analysis using SAL's SMT-based bounded model checker. The proofs were not automatic, as the properties of interest are not k-inductive, and we had to provide several auxiliary invariants.

We used the same SAL model to investigate whether PD-KIND in SALLY and other solvers could prove the same properties in a fully automated manner (that is, without the auxiliary invariants).

Simple Approximate Agreement. We have modeled and verified a variant of the approximate agreement protocol described in Chapter 7.2 of Nancy Lynch's book on distributed algorithms [29]. The modeling approach and verification procedure is the same as described previously. The main difference is that we use the median instead of the fault-tolerant midpoint as voting function. We verified the protocol under two fault models: we first looked at the case of a single Byzantine fault then we used a hybrid fault model with $N = 6$ processes, one of which is symmetric faulty and another suffers a benign fault.

Unified Approximate Agreement. We examined an approximate-agreement algorithm based on the unified protocol of Miner et al. [32]. Our goal was to check the applicability of our new model-checking techniques to proving properties under different fault scenarios, including hybrid fault models. As before, we built a model in SAL then converted it to MCMT for analysis.

The protocol consists of N processes that each own a value $v[i]$. In each round of the protocol, each process broadcasts its value to the others then update $v[i]$ by computing the mid-value of all the values it receives. The protocol satisfies the following agreement property:

$$\forall i, j : |v[i] - v[j]| < \epsilon \ ,$$

where i and j are two arbitrary non-faulty processes. We checked whether this property is satisfied in our state-transition model for different fault assumptions. As in the previous example, this property is not inductive and cannot be proved with SAL unless the user provides auxiliary invariants, but it can be proved automatically with SALLY.

Azadmanesh & Kieckhafer We examined the approximate agreement protocol of Azadmanesh & Kieckhafer [2] in synchronous systems under a hybrid fault model. We considered a system of N processes with b benign faults, w_s symmetric ommissive faults, w_a asymmetric omissive faults, s symmetric transmissive faults, and a asymmetric transmissive faults. Assuming that

$$N \geq 3a + 2s + w_a + w_s + b + 1 \ .$$

the *convergence property* is an invariant of the system. The convergence property states that $|v_i - v_j| \leq \Delta_o C^k$ where

- v_i and v_j are the values of two processes i and j

- Δ_0 is a bound on the initial difference between values of all processes
- C is the convergence rate ($0 \leq C < 1$)
- k is the number of rounds.

The voting was modeled using the fault-tolerant midpoint by sorting n received values in increasing order: $x_1, \ldots, x_n$, then removing τ largest and τ smallest values (in our case $\tau = 1$), and finally computing the midpoint as $\frac{x_{\tau+1}+x_{n-\tau}}{2}$.

Using SALLY we have automatically proved convergence for $C = 1/2$ and 5 processes, and showed that the convergence rate $C = 1/2$ is optimal for our voting function.

4.2 Evaluation

In addition to the case studies described above, we have also evaluated the new procedure on additional problems from other sources. We have used publicly-available benchmarks from software model checking (cav12 [8], ctigar [5]). The lustre benchmarks are from the benchmark suite of the KIND model-checker, and cons are simple concurrent programs.

To put the evaluation of the new method in context, we first compare PD-KIND$_5$ with other state-of-the-art, infinite-state model checkers, namely, Z3 [24], NUXMV [9], and SPACER [26]. The results are presented in Table 1. Each solver is executed with a timeout of 20 minutes. Each column of the table corresponds to a different model-checking engine, and each row corresponds to the different problem sets. For each problem set and tool combination we show the number of problems that the tool has solved and the total time (in seconds) that the tool took to solve those problems. In each problen, the goal is to prove or disprove that a postulated invariant property P holds. The column valid/invalid reports the number of instances for which the property is invariant (i.e., P is valid) and the number of instances for which the tool found a counterexample (i.e., P is invalid).

Table 1. Experimental evaluation

	Z3			SPACER			NUXMV			PD-KIND$_5$		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
approx-agree (9)	**9**	8/1	213	7	6/1	1150	9	8/1	2174	9	8/1	335
azadmanesh (20)	20	17/3	3404	20	17/3	4678	**20**	17/3	294	16	13/3	1694
oral-messages (9)	9	7/2	16	9	7/2	44	9	7/2	161	**9**	7/2	3
tta (3)	1	1/0	9	1	1/0	8	1	1/0	17	**1**	1/0	3
tte (6)	6	3/3	969	6	3/3	445	5	2/3	405	**6**	3/3	13
unified-approx (11)	8	5/3	2928	11	8/3	589	**11**	8/3	139	11	8/3	141
cav12 (99)	69	48/21	2102	71	49/22	3529	72	50/22	7443	**72**	51/21	6053
conc (6)	**4**	4/0	128	4	4/0	655	6	6/0	421	3	3/0	8
ctigar (110)	64	44/20	1683	72	52/20	4249	**76**	56/20	1342	75	55/20	2460
hacms (5)	1	1/0	11	1	1/0	4	**4**	3/1	388	1	1/0	18
lustre (790)	757	421/336	1888	**763**	427/336	2263	760	424/336	7660	763	430/333	8641

A detailed inspection of the results reveals the expected deductive strengths of the PD-KIND method. The most prominent example is the `cm-clock-distance` problem. It is a valid problem of the tte set. On this problem, SALLY with PD-KIND$_1$ (the inductive version) produces an inductive invariant containing 3692 facts. Other tools also struggle on this problem, either timing out or taking more than 250s. On the other hand, PD-KIND$_5$ produces a 4-inductive invariant of only 25 facts in seconds. This shows the advantage of the new method in terms of conciseness.

We have also run SALLY's k-induction checker on the examples. By itself, k-induction can solve 287 properties (i.e., the properties are k-inductive for some k and can be proved without searching for an inductive strengthening). With the new PD-KIND engine, SALLY *is the only model checker that can also prove all these k-inductive properties*.

On the other hand, regular induction is often enough for practical purposes. For example, PD-KIND$_1$ fares well on the cav12 benchmarks. It solves 74 problems, and as the induction depth increases the results degrade. A comparison of different PD-KIND variants is presented in Table 2. As shown in Table 1, PD-KIND struggles on problems in the azadmanesh class. On these problems, the procedure struggles for two reason. First, the transition relation has a complex control-flow that makes the k-inductive queries difficult to solve. Second, because the interpolation procedure provided in MathSAT is not finite covering the invariant generation diverges into an endless sequence of trivial invariants.[2]

Table 2. Comparison of different variants

	PD-KIND$_1$			PD-KIND$_3$			PD-KIND$_5$			PD-KIND$_\infty$		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
approx-agree (9)	9	8/1	348	9	8/1	341	**9**	8/1	335	9	8/1	342
azadmanesh (20)	**19**	16/3	3193	16	13/3	1651	16	13/3	1694	16	13/3	1671
oral-messages (9)	9	7/2	10	**9**	7/2	3	9	7/2	3	9	7/2	3
tta (3)	1	1/0	71	**1**	1/0	3	1	1/0	3	1	1/0	6
tte (6)	6	3/3	1018	6	3/3	17	**6**	3/3	13	6	3/3	14
unified-approx (11)	**11**	8/3	88	11	8/3	139	11	8/3	141	11	8/3	140
cav12 (99)	**74**	51/23	4983	73	51/22	5196	72	51/21	6053	65	47/18	1052
conc (6)	**4**	4/0	58	3	3/0	11	3	3/0	8	3	3/0	10
ctigar (110)	74	54/20	3358	74	54/20	2012	**75**	55/20	2460	74	54/20	2532
hacms (5)	**1**	1/0	11	1	1/0	14	1	1/0	18	1	1/0	27
lustre (790)	746	413/333	9998	759	430/329	4390	**763**	430/333	8641	760	431/329	9666

5 Publications

We have published several papers describing the project results. The following papers acknowledge the support of NASA cooperative agreement NNX14AI05A:

1. Dejan Jovanović. Solving nonlinear integer arithmetic with MCSAT. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking and Abstract Interpretation (VMCAI 2017)*, pages 330–346. Springer, 2017.

2. Dejan Jovanović and Bruno Dutertre. Property directed k-induction. In Ruzica Piskac and Muralidhar Talupur, editors, *Formal Methods in Computer-Aided Design (FMCAD 2016)*, pages 85–92, 2016.

3. Dejan Jovanović and Bruno Dutertre. LibPoly: A library for reasoning about polynomials. In *SMT Workshop*. 2017.

4. Temesghen Kahsai, Jorge A Navas, Dejan Jovanović, and Martin Schäf. Finding inconsistencies in programs with loops. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 499–514. Springer, 2015.

[2] We are planning to replace MathSAT with our own interpolation procedure based on MCSat [16].

5. Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanović, Philipp Rümmer, and Thomas Wies. Conflict-directed graph coverage. In *NASA Formal Methods*, pages 327–342. Springer, 2015.

6 Conclusion

At the start of the project, our main goal was to develop algorithms and tools to facilitate verification of fault-tolerant distributed systems. We have made signifcant progress toward this goal by developing a new variant of IC3 that uses k-induction and implementing the algorithm in a new model checker called SALLY. Experimental evaluation shows that SALLY is better than SAL, our previous generation of model checker. Systems that could not be verified automatically with SAL[3] can now be proved automatically with SALLY. On these fault-tolerant examples and others, SALLY is also competitive and often better than alternative SMT-based model checkers.

We did not fully achieve our other goal of supporting verification of truly parametric systems (i.e., without specifying in advance the number of components in the system). We have made progress in this direction by researching decision procedures that can handle counting constraints and array-based models. but these procedures remain to be developed and integrated into SALLY.

Another extension of this project might be generalization of SALLY and PD-KIND to richer logical theories. Currently, SALLY relies on relatively simple theories (i.e., linear arithmetic) because these are the theories supported by the backend SMT solvers. Extensions to nonlinear arithmetic would make SALLY more generally applicable and useful. To work in this direction, we would need to extend Yices to provide features such as model generalization and interpolants in nonlinear arithmetic.

[3]The SAL proofs require the user to find auxiliary lemmas by hand.

References

1. Francesco Alberti, Silvio Ghilardi, and Elena Pagani. Counting constraints in flat array fragments. In *Proceedings of the 8th International Joint Conference on Automated Reasoning-Volume 9706*, pages 65–81. Springer-Verlag New York, Inc., 2016.

2. Mohammad H Azadmanesh and Roger M Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.

3. Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

4. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

5. Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification*, pages 831–848. Springer, 2014.

6. Nikolaj Bjørner and Mikoláš Janota. Playing with quantified satisfaction. Logic for Programming, Artificial Intelligence and Reasoning, 2015.

7. Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

8. Alessandro Cimatti and Alberto Griggio. Software model checking via ic3. In *Computer Aided Verification*, pages 277–293. Springer, 2012.

9. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 46–61. Springer, 2014.

10. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

11. Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Transactions on Computational Logic (TOCL)*, 12(1):7, 2010.

12. Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *Computer Aided Verification*, pages 718–724. Springer, 2012.

13. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252, 1977.

14. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM, 1978.

15. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

16. Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

17. Leonardo De Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In *Computer aided verification*, pages 496–500. Springer, 2004.

18. Leonardo De Moura, Sam Owre, and Natarajan Shankar. The sal language manual. *Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01*, 2003.

19. Bruno Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer, 2014.

20. Bruno Dutertre, Arvind Easwaran, Brendan Hall, and Wilfried Steiner. Model-based analysis of timed-triggered ethernet. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 9D2–1. IEEE, 2012.

21. Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 125–134. IEEE, 2011.

22. Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *Automated Reasoning*, pages 22–29. Springer, 2010.

23. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015*, pages 343–361, 2015.

24. Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 157–171. Springer, 2012.

25. Dejan Jovanović and Bruno Dutertre. Property directed k-induction. In Ruzica Piskac and Muralidhar Talupur, editors, *Formal Methods in Computer-Aided Design (FMCAD 2016)*, pages 85–92, 2016.

26. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Computer Aided Verification*, pages 17–34. Springer, 2014.

27. Nikolaj Komuravelli, Anvesh adn Bjørner, Arie Gurfinkel, and Kenneth L McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Formal Methods in Computer-Aided Design*, pages 89–96, 2015.

28. Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

29. Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

30. Kenneth L McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003.

31. Kenneth L McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

32. Paul Miner, Alfons Geser, Lee Pike, and Jeffrey Maddalon. A unified fault-tolerance protocol. In *FORMATS/FTRTFT*, pages 167–182. Springer, 2004.

33. Andrew M Pitts. On an interpretation of second order quantification in first order intuitionistic propositional logic. *The Journal of Symbolic Logic*, 57(01):33–52, 1992.

Appendix A

The PD-KIND Algorithm

We reason about transition systems in the satisfiability modulo theories (SMT) framework [3]. Specifically, we assume that the transition system is described in a theory where quantifier-free satisfiability is decidable.

A.1 Notations and Background

We assume a finite set of typed variables $\vec{x}$ called *state variables*. To each variable $x \in \vec{x}$, we associate its primed version x' of the same type. We call any quantifier-free formula $F(\vec{x})$ over the state variables a *state formula*, and any quantifier-free formula $T(\vec{x}, \vec{x}')$ a *state-transition formula*. A *state* s is a type-consistent interpretation of $\vec{x}$ that assigns to each variable $x \in \vec{x}$ a value $s(x)$ over its domain. A state formula $F(\vec{x})$ holds in a state s (written $s \vDash F$) if the formula evaluates to true under the state's assignment.

A *state-transition system* is a pair $\mathfrak{S} = \langle I, T \rangle$, where $I(\vec{x})$ is a state formula describing the initial states and $T(\vec{x}, \vec{x}')$ is a state-transition formula describing the system's evolution. A state s' is a successor of a state s in $\mathfrak{S}$ if the formula $T(\vec{x}, \vec{x}')$ evaluates to true when we interpret each $x \in \vec{x}$ as $s(x)$ and each $x' \in \vec{x}'$ as $s'(x)$. A state s is *k-reachable* if there exists a sequence of states $\sigma = \langle s_0, \ldots s_k \rangle$ such that, $s = s_k$, the state s_0 satisfies I, and each s_{i+1} is a successor of s_i. We call σ a *concrete trace* of the system. We also say that a state formula F is reachable in k steps if there is a k-reachable state s such that $s \vDash F$.

Given a state formula P (*the property*), we want to determine whether all the reachable states of $\mathfrak{S}$ satisfy P. If this is the case, P is an *invariant* of $\mathfrak{S}$, which we denote by $\mathfrak{S} \vDash P$. We also write $\mathfrak{S} \vDash_a^b P$ to denote that P is true in all k-reachable states for $a \leq k \leq b$. If P is not invariant, there is a concrete trace, called a *counter-example*, that reaches $\neg P$.

Definition A.1 ($\mathcal{F}$-Induction)**.** *Given a set $\mathcal{F}$ of state formulas such that $P \in \mathcal{F}$, P is* $\mathcal{F}$-inductive[A1] *with respect to $\mathfrak{S}$ if*

$$I(\vec{x}) \Rightarrow \mathcal{F}(\vec{x}) \ , \tag{init}$$

$$\mathcal{F}(\vec{x}) \wedge T(\vec{x}, \vec{x}') \Rightarrow P(\vec{x}') \ . \tag{cons}$$

If $\mathcal{F} = \{P\}$, we say that P is inductive*.*

If P is inductive then it is also invariant. Since invariants are in general not inductive, a common approach to prove that P is invariant is to find a *strengthening* of P. Such a strengthening is a set of formulas $\mathcal{F}$ such that $P \in \mathcal{F}$ and $\mathcal{F}$ is inductive. If such a strengthening exists, then P is invariant.

Definition A.2 ($\mathcal{F}^k$-Induction)**.** *Given a set $\mathcal{F}$ of state formulas such that $P \in \mathcal{F}$, P is* $\mathcal{F}^k$-inductive *with respect to $\mathfrak{S}$ if*

$$I(\vec{x_0}) \wedge \bigwedge_{i=0}^{l-1} T(\vec{x}_i, \vec{x}_{i+1}) \Rightarrow \mathcal{F}(\vec{x}_l) \ , \textit{ for } 0 \leq l < k \ , \tag{k-init}$$

$$\bigwedge_{i=0}^{k-1} (\mathcal{F}(\vec{x}_i) \wedge T(\vec{x}_i, \vec{x}_{i+1})) \Rightarrow P(\vec{x}_k) \ . \tag{k-cons}$$

When $\mathcal{F} = \{P\}$, we say that P is k-inductive*.*

[A1] This is the same idea as induction relative to $\mathcal{F}$ used in [7].

A property that is inductive is 1-inductive by definition. It is also k-inductive for any k. In the other direction, if a property P is k-inductive and the logical theory underlying the system admits quantifier elimination, then we can construct an inductive strengthening of P by eliminating quantifiers.[A2] For such theories, induction and k-induction have the same deductive power but k-induction may give more succinct strengthenings. If the base theory does not admit quantifier elimination then k-induction can be more powerful than induction.

A.2 Satisfiability Checking

Given a state formula F, we denote with $T[F]^k$ the unrolling of T to length k where F holds in the intermediate states. For $k > 1$, $T[F]^k(\vec{x}, \vec{x}')$ is then defined as

$$T(\vec{x}, \vec{w}_1) \wedge \bigwedge_{i=1}^{k-1} (F(\vec{w}_i) \wedge T(\vec{w}_i, \vec{w}_{i+1})) \wedge T(\vec{w}_{k-1}, \vec{x}')$$

where $\vec{w}$ are the state variables in the intermediate states. For $k = 0$ and $k = 1$, we set $T^0[F](\vec{x}, \vec{x}') \equiv (\vec{x} = \vec{x}')$ and $T^1[F](\vec{x}, \vec{x}') \equiv T(\vec{x}, \vec{x}')$. When $F \equiv \mathbf{true}$, we omit it and simply write T^k.

A basic step in our algorithms is to check the satisfiability of formulas of the form

$$A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y}) \ , \tag{A1}$$

where A, B, and C are state formulas. We denote by CHECK-SAT$(A, T[B]^k, C)$ an (SMT-based) procedure that checks satisfiability of formula (A1), and returns a model if the formula is satisfiable. In addition, we require two artifacts from the SMT solver: *interpolants* and *generalizations*.

Definition A.3 (Interpolant)**.** *If the formula* (A1) *is unsatisfiable, a formula $J(\vec{y})$ is a* state interpolant *if*

1. $A(\vec{x}) \wedge T[B]^k(\vec{x}, \vec{w}, \vec{y}) \Rightarrow J(\vec{y})$*, and*
2. $J(\vec{y})$ *and* $C(\vec{y})$ *are inconsistent.*

Definition A.4 (Generalization)**.** *If the formula* (A1) *is satisfiable, we call a formula $G(\vec{x})$ a* state generalization *if*

1. $A(\vec{x})$ *and* $G(\vec{x})$ *are consistent, and*
2. $G(\vec{x}) \Rightarrow \exists \vec{w}, \vec{y} \,.\, T[B]^k(\vec{x}, \vec{w}, \vec{y}) \wedge C(\vec{y})$.

Interpolation provides forward learning. An interpolant over-approximates the set of states reachable from A via $T[B]^k$ and is enough to refute C. Generalization is the dual and provides backward learning. A generalization G is consistent with A and under-approximates the set of states that can reach C via $T[B]^k$.

Our notion of a state interpolant is more specific than the one usually considered in general interpolation, and our definition can be easily satisfied: formula $\neg C$ is always an interpolant (so interpolants exists in our case even if the underlying theory does not support general interpolation). Similarly, one can construct a generalization G from a model v of formula (A1) by substitution (i.e., the formula $(A \wedge T[B]^k)[\vec{w}/v(\vec{w}), \vec{y}/v(\vec{y})]$ is a trivial generalization). Although correct, trivial interpolants and generalizations are not ideal for practical applications. In particular, they do not satisfy the following property.

[A2] If P is k-inductive then $P \wedge \circ P \wedge \circ \circ P \wedge \ldots \wedge \overbrace{\circ \circ \cdots \circ}^{k-1} P$ is inductive, where $\circ$ stands for "next state".

Definition A.5 (Finite Cover Property)**.** *An interpolation (resp. generalization) procedure has the* finite cover property *(is finite-covering) when, for a fixed $T[B]^k$ and A (resp C), it can only produce a finite number of distinct interpolants (resp. generalizations).*

Interpolation is a well-studied topic [11,31] and it is available in several SMT solvers. Effective generalization in SMT was introduced in [26] (as model-based projection) for specific use in a PDR engine. There are known generalization methods for the theories of linear arithmetic [26], arrays [27], and algebraic datatypes [6]. These methods have the finite cover property. On the other hand, most interpolation procedures are proof-based and do not ensure finite covering.

Both interpolation and generalization approximate quantifier elimination. For theories that admit quantifier elimination, one can construct precise interpolants by eliminating $\vec{x}$ and $\vec{w}$ from $A \wedge T[B]^k$ and precise generalizations by eliminating $\vec{w}$ and $\vec{y}$ from $T[B]^k \wedge B$. Such precise procedures have the finite cover property, and it is not unreasonable to expect the same from interpolation and generalization. This is the case for pure SAT problems. In SAT, interpolants can always be expressed as clauses, while generalizations can be expressed as prime implicants, both of which guarantee the finite cover property.[A3] The finite-cover property for interpolants is an incremental form of the related notion of uniform interpolation [33]: uniform interpolation requires a single interpolant instead of a finite set.

Example A.1. *Consider the system $\mathfrak{S} = \langle I, T\rangle$ defined as $I \equiv (x = 0)$, $T \equiv (x' = x + 1)$ where x is a real-valued variable. Let P be the formula $0 \leq x \vee x \geq 1$. To check whether P is inductive, we can ask the following satisfiability query to the SMT solver*

$$\overbrace{(0 \leq x \vee x \geq 1)}^{A_1} \wedge \overbrace{(x' = x + 1)}^{T^1} \wedge \overbrace{\neg(0 \leq x' \vee x' \geq 1)}^{B_1 \equiv \neg A_1} \ .$$

This formula is satisfiable in a model $x \mapsto -0.5$, $x' \mapsto 0.5$. We can generalize this model to $G \equiv (-1 < x < 0)$; any state that satisfies G is a counterexample to induction of P. We can then check whether G intersects with the initial states and whether G is reachable in one step, by making two separate queries

$$\begin{array}{c} \overbrace{(x = 0)}^{A_2} \wedge \overbrace{(x' = x)}^{T^0} \wedge \overbrace{(-1 < x' < 0)}^{B_2} \ , \\ \underbrace{(x = 0)}_{A_3} \wedge \underbrace{(x' = x + 1)}_{T^1} \wedge \underbrace{(-1 < x' < 0)}_{B_3} \ . \end{array}$$

Both queries are unsatisfiable. From the first query, we can get an interpolant $J_0(x') \equiv (x' \geq 0)$ that refutes G in the initial states. From the second query, we can get an interpolant $J_1(x') \equiv (x' \geq 1)$ that refutes G after one transition. Although P itself is not inductive, the two interpolants give us a strengthening: the formula $P' \equiv P \wedge (J_0(x) \vee J_1(x))$ is inductive.

A.3 Reachability

Problem A.1 (k-reachability)**.** *Given a state formula F that is not reachable in fewer than k steps, check whether F is reachable in k steps.*

The reachability problem can be solved by bounded model checking [4], but we discuss an alternative method that does not require unrolling the transition relation. We introduce the concept of k-interpolation as a way to learn from failures of k-reachability.

[A3] For arithmetic theories, finite-covering interpolants can be generated using model-based procedures such as MCSat [16].

Definition A.6 (k-interpolant)**.** *Given a system $\mathfrak{S}$ and state formula F that is unreachable in $\leq k$ steps ($system$ is k-inconsistent with F), a state formula J is a state k-interpolant for F if*

$$\mathfrak{S} \models_0^k J \ , \qquad\qquad J \text{ and } F \text{ are inconsistent} \ .$$

As with regular interpolation, the formula $\neg F$ itself is a trivial k-interpolant. We can also construct a k-interpolant by calling a standard interpolation procedure $k+1$ times: If $\mathfrak{S}$ and F are k-inconsistent, then $I(\vec{x}) \wedge T^i(\vec{x}, \vec{w}, \vec{x}') \wedge F(\vec{x}')$ is unsatisfiable for $0 \leq i \leq k$. From these inconsistencies we can obtain interpolants $J_0, \ldots, J_k$, and the formula $J \equiv (J_0 \vee \ldots \vee J_k)$ is a k-interpolant for F. Moreover, if the interpolation procedure has the finite-cover property then so does the k-interpolation procedure.

To check k-reachability, we adopt the incremental depth-first reachability method of IC3, which relies on local reasoning. The procedure maintains a sequence $\mathcal{R}_0, \mathcal{R}_1, \ldots$ of *reachability frames*. Frame $\mathcal{R}_i$ is a set of state formulas that over-approximates the set of states reachable in i steps or less. This implies that $\mathfrak{S} \models_0^i \mathcal{R}_i$. Unlike IC3/PDR, we *do not require the frames to be monotonic*; we may have $\mathcal{R}_{i+1} \not\subseteq \mathcal{R}_i$.[A4]

This setup allows us to build k-interpolants efficiently provided an extra local condition holds. If $k = 0$, we just take the interpolant of $I \wedge T^0 \wedge F$. If $k > 0$ and the formula F is not reachable in up to k steps, and if, in addition, F is not reachable in one step from $\mathcal{R}_{k-1}$, then both $I \wedge T^0 \wedge F$ and $\mathcal{R}_{k-1} \wedge T \wedge F$ are inconsistent. We can then obtain interpolants J_1 and J_2 for these two formulas and $(J_1 \vee J_2)$ is a k-interpolant for F. This k-interpolant, which we denote by EXPLAIN$(\mathfrak{S}, k, F)$, is potentially more concise than the one described before and it is obtained by local reasoning only. Although EXPLAIN has an additional precondition, our algorithm ensures that this holds whenever EXPLAIN is called.

Lemma A.1. *Starting from a fixed finite frame sequence $\mathcal{R}_0, \mathcal{R}_1, \ldots$, if the only formulas we add to the frames are obtained through the* EXPLAIN *procedure, and the interpolation procedure is finite-covering, then the* EXPLAIN *procedure is also finite-covering.*

Algorithm 1 Check k-reachability of F.

Require: $\mathfrak{S} \models_0^i \mathcal{R}_i$ for $0 \leq i \leq k$, $\mathfrak{S} \models_0^{k-1} \neg F$.
Ensure: $\mathfrak{S} \models_0^i \mathcal{R}_i$ for $0 \leq i \leq k$. If not reachable, $\mathcal{R}_{k-1} \wedge T \wedge F$ is unsatisfiable.

```
1:  function REACHABLE(𝔖, k, F)
2:      if k = 0 then return CHECK-SAT(I, T^0, F)
3:      loop
4:          if CHECK-SAT(R_{k-1}, T, F) then
5:              G ← GENERALIZE(R_{k-1}, T, F)
6:              if REACHABLE(𝔖, k − 1, G) then
7:                  return true
8:              else
9:                  E ← EXPLAIN(𝔖, k − 1, G)
10:                 R_{k-1} ← R_{k-1} ∪ {E}
11:         else return false
```

Finally, our reachability routine REACHABLE$(\mathfrak{S}, k, F)$ performs a step-wise search for a concrete trace by using a depth-first search strategy. It tries to reach the initial states backwards. To reach F at frame k, we check first whether F can be reached in one transition from the previous frame $\mathcal{R}_{k-1}$. If no such transition is possible, then F is not reachable. Otherwise, we get a state s that satisfies $\mathcal{R}_{k-1}$ and from which F is reachable in one step. The generalization procedure gives us a formula G that generalizes s: every state that satisfies G has a successor that satisfies F. We then recursively check whether G is reachable. The recursive

[A4] From an implementation perspective, this gives flexibility in garbage collection. We can remove any subset of formulas from any frame $\mathcal{R}_i$ without compromising correctness.

call will either find a path from the initial states to G, in which case F is reachable, or determine that G is not reachable, in which case we can learn an explanation E of the reachability failure and eliminate G. Learning E eliminates G as a potential step backward, and we continue.

Lemma A.2. *Algorithm 1 solves the k-reachability problem. If F is not reachable then, upon completion, either $k = 0$ and F is inconsistent with I, or $k > 0$ and F is not reachable in one step from $\mathcal{R}_{k-1}$. In addition, if the interpolation or the generalization procedure is finite-covering, then the procedure always terminates.*

We use a variant of the REACHABLE procedure to check whether F is reachable in steps k_1 to k_2. We denote this by $(r, l) = \text{REACHABLE}(\mathfrak{S}, k_1, k_2, F)$. This extension of the REACHABLE procedure is a straightforward loop from k_1 to k_2, and has the same precondition as the single check version (on k_1). In the return value, r denotes the reachability result (true/false), and, if r is true, l is the length of the shortest trace that can reach F. The postcondition (and hence Lemma A.2) of the iterative extension is also the same (on k_2).

A.4 Property-Directed k-Induction

We now present the main procedure of PD-KIND. This procedure checks whether a property P is invariant for a system $\mathfrak{S} = \langle I, T \rangle$. It does so by iteratively trying to construct a k-inductive strengthening of P for some $k > 0$. The overall idea behind the procedure is simple. Assume a set of formulas $\mathcal{F}_{\text{ABS}}$ that is a strengthening of P and is valid in $\mathfrak{S}$ for up to n steps. In other words, $\mathcal{F}_{\text{ABS}}$ is an over-approximation of states reachable in n steps or less. Then, the set $\mathcal{F}_{\text{ABS}}$ satisfies (k-init) for all $1 \leq k \leq n + 1$. We can pick any such k and try to show that $\mathcal{F}_{\text{ABS}}$ is k-inductive by checking whether it also satisfies (k-cons). Each iteration of the procedure PD-KIND does this check. The core of our algorithm is procedure PUSH that either finds a counter-example to P or produces a new strengthening $\mathcal{G}_{\text{ABS}} \subseteq \mathcal{F}_{\text{ABS}}$. This new set $\mathcal{G}_{\text{ABS}}$ satisfies (k-cons) with respect to $\mathcal{F}_{\text{ABS}}$. The set $\mathcal{G}_{\text{ABS}}$ is then $\mathcal{F}^k_{\text{ABS}}$-inductive. If $\mathcal{G}_{\text{ABS}} = \mathcal{F}_{\text{ABS}}$, we can conclude that P is invariant. Otherwise, we know that $\mathcal{G}_{\text{ABS}}$ is valid (at least) up to index $n + 1$. Procedure PUSH actually returns an integer n_p such that $\mathcal{G}_{\text{ABS}}$ is valid up to n_p. This index n_p is the length of the shortest trace of $\mathfrak{S}$ that reaches $\neg\mathcal{F}_{\text{ABS}}$; it is guaranteed to be at least $n + 1$ but it may be larger. At this point, we repeat the loop with $\mathcal{G}_{\text{ABS}}$ as our current strengthening and n_p as our new index.

Algorithm 2 Main PD-KIND procedure.

Require: $\mathfrak{S} = \langle I, T \rangle$ and $I \Rightarrow P$
1: **function** PD-KIND($\mathfrak{S}, P$)
2: $n \leftarrow 0$
3: $\mathcal{F} \leftarrow \{(P, \neg P)\}$
4: **loop**
5: pick k-induction depth $1 \leq k \leq n + 1$
6: $\langle \mathcal{F}, \mathcal{G}, n_p \rangle \leftarrow \text{PUSH}(\mathfrak{S}, \mathcal{F}, P, n, k)$
7: **if** P marked invalid **then return invalid**
8: **if** $\mathcal{F} = \mathcal{G}$ **then return valid**
9: $n \leftarrow n_p$
10: $\mathcal{F} \leftarrow \mathcal{G}$

In addition to the set of formulas $\mathcal{F}_{\text{ABS}}$, procedure PD-KIND associates with each each $F_{\text{ABS}} \in \mathcal{F}_{\text{ABS}}$ information about a potential counter-example to P that the formula F_{ABS} eliminates. The set $\mathcal{F}_{\text{ABS}}$ and this additional information is represented in the form of an induction frame. Let $\mathbb{F}$ denote the set of all state formulas in our theory.

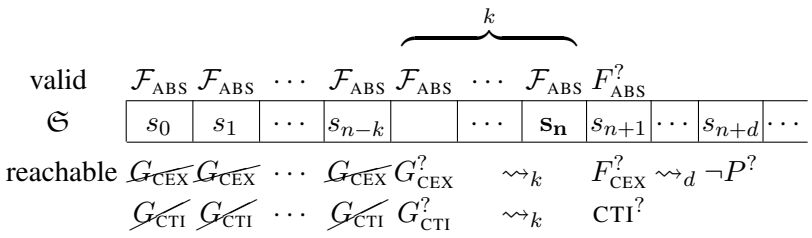

Figure A1. Illustration of the formulas and frame indices over which PUSH operates.

Definition A.7 (Induction Frame)**.** *A set of tuples $\mathcal{F} \subset \mathbb{F} \times \mathbb{F}$ is an* induction frame at index n *if $(P, \neg P) \in \mathcal{F}$ and the following holds for all $(F_{\text{ABS}}, F_{\text{CEX}}) \in \mathcal{F}$:*

1. *F_{ABS} is valid up to n steps and refutes F_{CEX}, and*
2. *F_{CEX}-states can be extended to a counter-example to P.*

If $I \Rightarrow P$, then the set $\mathcal{F} = \{(P, \neg P)\}$ is an induction frame at index 0. Given an induction frame $\mathcal{F}$, we denote by $\mathcal{F}_{\text{ABS}}$ the strengthening represented by $\mathcal{F}$, i.e., $\mathcal{F}_{\text{ABS}} = \{F_{\text{ABS}} \mid (F_{\text{ABS}}, F_{\text{CEX}}) \in \mathcal{F}\}$. With this in mind, the procedure PD-KIND is presented in Algorithm 2.

A.5 The PUSH Procedure

The core of the PD-KIND algorithm is the PUSH procedure (Algorithm 3). This procedure takes as input an induction frame $\mathcal{F}$ at index n, and tries to push formulas of the frame using k-induction where $1 \leq k \leq n+1$. Figure A1 illustrates the formulas and frame indices over which PUSH operates.

Since $\mathcal{F}$ is an induction frame at n, we know that $\mathcal{F}_{\text{ABS}}$ is valid up to index n. In each iteration, the procedure picks one yet unprocessed $(F_{\text{ABS}}, F_{\text{CEX}})$ from $\mathcal{F}$. Both F_{ABS} and $\neg F_{\text{CEX}}$ hold up to index n in $\mathfrak{S}$.

First, the procedure checks whether F_{ABS} is $\mathcal{F}^k_{\text{ABS}}$-inductive (lines 9-12). If so, then we know that F_{ABS} is valid at least up to position $n+1$. We call this a successful push and we add $(F_{\text{ABS}}, F_{\text{CEX}})$ to the set of pushed obligations $\mathcal{G}$, and continue with the next obligation. If the k-induction check fails, then we have a model (counterexample to induction) m_{CTI}. This is a trace of length $k+1$ in which $\mathcal{F}_{\text{ABS}}$ holds for the first k states but F_{ABS} is false in the last state.

The procedure does not use m_{CTI} yet. Instead, it checks whether the counterexample formula F_{CEX} is reachable from $\mathcal{F}_{\text{ABS}}$ (lines 15-24). If the query at line 15 is satisfiable, it has a model m_{CEX}. Like m_{CTI}, this model is a trace of length $k+1$; it starts with k states that satisfy $\mathcal{F}_{\text{ABS}}$ and ends with a state that satisfies F_{CEX} (thus, from the first state of m_{CEX} we can reach $\neg P$). At this point, we generalize m_{CEX} to a formula G_{CEX}. From any state that satisfies G_{CEX}, one can reach $\neg P$. Formula G_{CEX} is then a potential counterexample for P. We check whether G_{CEX} is reachable from the initial states of $\mathfrak{S}$. Because we know that F_{CEX} is not n-reachable, G_{CEX} can't be reached in less than $n-k+1$ steps. So we check reachability of G_{CEX} at positions $n-k+1 \ldots n$. If G_{CEX} is reachable, then so is $\neg P$ and we mark P as invalid. Otherwise, we call the EXPLAIN procedure, which returns a new fact G_{ABS} that eliminates G_{CEX}. The new fact G_{ABS} is true up to position n, and refutes G_{CEX}, so we can add the new induction obligation $(G_{\text{ABS}}, G_{\text{CEX}})$ to $\mathcal{F}$, strengthening $\mathcal{F}$, and try again with a potential counter-example eliminated.

In the remaining case, we have a counterexample m_{CTI} to the k-inductiveness of F_{ABS}. Since the query at line 15 is not satisfiable and $\mathcal{F}_{\text{ABS}} \Rightarrow \neg F_{\text{CEX}}$, we know that $\neg F_{\text{CEX}}$ is $\mathcal{F}^k_{\text{ABS}}$ inductive. We first apply generalization to m_{CTI} to construct a formula G_{CTI}. From any state that satisfies G_{CTI}, we can reach $\neg F_{\text{ABS}}$ in k steps. If G_{CTI} is reachable in $\mathfrak{S}$ then $\neg F_{\text{ABS}}$ is also reachable, so F_{ABS} can't be part of a valid strengthening of P. This check is performed at line 28; as previously, it is enough to check reachability of G_{CTI} at positions

$n-k+1, \ldots, n$. If G_{CTI} is reachable, we can't push F_{ABS}. Instead, we replace the triple $(F_{\text{ABS}}, F_{\text{CEX}})$ by the weaker obligation $(\neg F_{\text{CEX}}, F_{\text{CEX}})$. This new obligation can be immediately pushed to $\mathcal{G}$. On the other hand, if G_{CTI} is not reachable then we strengthen F_{ABS} with a new fact G_{ABS} learned from procedure EXPLAIN. This eliminates the counterexample to k-induction and the procedure continues.

At lines 30-31 of the procedure, we know that $\neg F_{\text{ABS}}$ is reachable in $\mathfrak{S}$ and that this requires at least $n+1$ transitions. It is useful to make this more precise by computing the actual length of the shortest path to $\neg F_{\text{ABS}}$ (line 30). This length is stored in variable n_p (if it's smaller than n_p's current value).

After the loop terminates, PUSH returns the set of successfully pushed induction obligations $\mathcal{G}$, the modified set $\mathcal{F}$ of k-induction assumptions for $\mathcal{G}$, and the shortest refutation length n_p for any $F_{\text{ABS}} \in \mathcal{F}_{\text{ABS}}$ that was not successfully pushed. The procedure does not only add to the original set $\mathcal{F}$, it also actively modifies it (line 37). Unlike existing IC3-based procedures where frames are explored in succession, keeping track of n_p allows us to perform "jumps" that move to deeper frames faster. This is because $\mathcal{F}_{\text{ABS}}$ is valid up to position $n_p - 1 \geq n$, and the facts in $\mathcal{G}_{\text{ABS}}$ are valid up to position $n_p \geq n+1$.[A5]

Assuming that PD-KIND terminates, it is not hard to show that it returns the correct result. If PD-KIND terminates with P marked invalid, then we have found a counter-example to the property. On the other hand, if PD-KIND terminates when the inductive frames become equal, i.e. $\mathcal{F} = \mathcal{G}$, then $\mathcal{F}_{\text{ABS}}$ is a k-inductive strengthening of P and P is therefore valid. In general, for infinite domains, even termination of the PUSH procedure is not guaranteed. But, a finite-covering interpolation procedure ensures termination: the number of new facts that PUSH can learn is finite, and this bounds both the number of possible refinement steps, and the number of new counter-examples that can be found in line 15.

The PD-KIND procedure, as presented, has the freedom to choose the induction depth k in each iteration (line 5). We call a strategy for picking the depth increasing if it guarantees that, for every k, PD-KIND eventually picks induction depths k' larger than k.

Lemma A.3. *If the interpolation procedure is finite-covering, then the* PUSH *procedure terminates. If the property P is k-inductive for some $k > 0$, and* PD-KIND *uses an increasing strategy for k, then the* PD-KIND *procedure terminates.*

Proof. (Sketch) If the property P is k inductive, then PD-KIND will eventually pick only depths $k' \geq k$. For any such k' no counterexamples can be found at line 15, because any m_{CEX} could be extended to a counter-example of P, violating the assumption that P is k-inductive. If no new counter-examples can be found then, as PD-KIND goes from frame to frame, the only new facts that can be added to the frame are either obtained from refinement on line 35, where existing facts are replaced with stronger facts, or line 36, where facts are weakened to a counter-example refutation. Since we know that no new counter-examples can be found, the latter can only happen a finite number of times. Therefore, the size of the frame can not increase indefinitely, and will eventually converge to a state where $\mathcal{F} = \mathcal{G}$. □

[A5] We have observed significant frame jumps in practice although this is problem-specific.

Algorithm 3 Push $\mathcal{F}$ with k-induction.

Require: $\mathcal{F}$ is a valid frame for P at position n, $1 \leq k \leq n+1$, $(P, \neg P) \in \mathcal{F}$.

Ensure: $\mathcal{F}$ is a valid frame for P at position $n_p - 1 \geq n$, $\mathcal{G} \subseteq \mathcal{F}$ is $\mathcal{F}^k$-inductive, and P marked invalid or $(P, \neg P, 0) \in \mathcal{F}_p$.

1: **function** PUSH($\mathfrak{S}$, $\mathcal{F}$, P, n, k)
2: push elements of $\mathcal{F}$ to $\mathcal{Q}$ ▷ $\mathcal{Q}$ is a priority queue.
3: $\mathcal{G} \leftarrow \{\}$ ▷ Pushed facts, i.e. $\mathcal{G}_{\text{ABS}}$ is $\mathcal{F}^k_{\text{ABS}}$-inductive.
4: $n_p \leftarrow n + k$ ▷ Keeps track of the shortest CTI position.
5: **while** P not marked invalid, $\mathcal{Q}$ not empty **do**
6: pop $(F_{\text{ABS}}, F_{\text{CEX}})$ from $\mathcal{Q}$
7:
8: ——— ▷ Is F_{ABS} $\mathcal{F}^k_{\text{ABS}}$-inductive?
9: $(sat_{\text{CTI}}, m_{\text{CTI}}) \leftarrow$ CHECK-SAT$(\mathcal{F}_{\text{ABS}}, T[\mathcal{F}_{\text{ABS}}]^k, \neg F_{\text{ABS}})$
10: **if not** sat_{CTI} **then**
11: $\mathcal{G} \leftarrow \mathcal{G} \cup \{(F_{\text{ABS}}, F_{\text{CEX}})\}$ ▷ $\mathcal{G}_{\text{ABS}}$ is $\mathcal{F}^k_{\text{ABS}}$-inductive.
12: **continue**
13:
14: ——— ▷ Is F_{CEX} reachable?
15: $(sat_{\text{CEX}}, m_{\text{CEX}}) \leftarrow$ CHECK-SAT$(\mathcal{F}_{\text{ABS}}, T[\mathcal{F}_{\text{ABS}}]^k, F_{\text{CEX}})$
16: **if** sat_{CEX} **then**
17: $G_{\text{CEX}} \leftarrow$ GENERALIZE$(m_{\text{CEX}}, T^k, F_{\text{CEX}})$
18: **if** REACHABLE$(\mathfrak{S}, n-k+1, n, G_{\text{CEX}})$ **then**
19: mark P invalid ▷ $I \rightsquigarrow G_{\text{CEX}} \rightsquigarrow_k F_{\text{CEX}} \rightsquigarrow \neg P$.
20: **else**
21: $G_{\text{ABS}} \leftarrow$ EXPLAIN$(\mathfrak{S}, n, G_{\text{CEX}})$
22: $\mathcal{F} \leftarrow \mathcal{F} \cup \{(G_{\text{ABS}}, G_{\text{CEX}})\}$ ▷ Eliminate CEX.
23: push $(G_{\text{ABS}}, G_{\text{CEX}})$, $(F_{\text{ABS}}, F_{\text{CEX}})$ to $\mathcal{Q}$
24: **continue**
25:
26: ——— ▷ Analyze the induction failure.
27: $G_{\text{CTI}} \leftarrow$ GENERALIZE$(m_{\text{CTI}}, T^k, \neg F_{\text{ABS}})$
28: $(r_{\text{CTI}}, n_{\text{CTI}}) \leftarrow$ REACHABLE$(\mathfrak{S}, n-k+1, n, G_{\text{CTI}})$
29: **if** r_{CTI} **then** ▷ $I \rightsquigarrow_{n_{\text{CTI}}} G_{\text{CTI}} \rightsquigarrow_k \neg F_{\text{ABS}}$.
30: $(r_{\text{CTI}}, n_{\text{CTI}}) \leftarrow$ REACHABLE$(\mathfrak{S}, n+1, n_{\text{CTI}}+k, \neg F_{\text{ABS}})$
31: $n_p \leftarrow \min(n_p, n_{\text{CTI}})$
32: $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\neg F_{\text{CEX}}, F_{\text{CEX}})\}$
33: $\mathcal{G} \leftarrow \mathcal{G} \cup \{(\neg F_{\text{CEX}}, F_{\text{CEX}})\}$
34: **else** ▷ $I \not\rightsquigarrow_{\leq n} G_{\text{CTI}} \rightsquigarrow_k \neg F_{\text{ABS}}$.
35: $G_{\text{ABS}} \leftarrow$ EXPLAIN$(\mathfrak{S}, n, G_{\text{CTI}})$ ▷ $G_{\text{ABS}} \Rightarrow \neg G_{\text{CTI}}$.
36: $G_{\text{ABS}} \leftarrow F_{\text{ABS}} \wedge G_{\text{ABS}}$ ▷ $G_{\text{ABS}} \Rightarrow \neg F_{\text{CEX}}$.
37: $\mathcal{F} \leftarrow \mathcal{F} \cup \{(G_{\text{ABS}}, F_{\text{CEX}})\} \setminus \{(F_{\text{ABS}}, F_{\text{CEX}})\}$
38: push $(G_{\text{ABS}}, F_{\text{CEX}})$ to $\mathcal{Q}$.
39:
40: **return** $\langle \mathcal{F}, \mathcal{G}, n_p \rangle$

Appendix B

Example Model

We present the SAL specification of an approximate agreement protocol due to Azadmanesh and Kieckhaefer [2]. This example and some variations are included in the SALLY repository at `http://sri-csl.github.io/sally`, which also gives translations of the protocol to the MCMT language. This model illustrates the approach we have used in the project to model faults and fault-tolerant protocols.

In the protocol instance included here, the model includes five processes, two of which are faulty. This scenario is described by predicate `scenario2`:

- Process 1 is asymmetric transmissive (i.e., it suffers a Byzantine fault). This means that it fails in an arbitrary asymmetric fashion.
- Process 2 is asymmetric omissive: it may fail to send anything to some other processes and send correct data to others. It does not send arbitrary incorrect data.

The fault-tolerant averaging function is the midpoint of the values received. This averaging is defined by two functions in SAL:

- Function `sort_and_filter` specifies sorting of an array v of $N = 5$ values. It excludes the special value `missing` from the sorting.
- Function `vote` picks the mid-value of a sorted array of $n \leq N$ values.

The protocol itself is modeled as a single flat SAL module. The key state variables include an array `c` that models the communication channels. The value `c[i][j]` is the value transmitted by process `j` to `i`. This value is updated at each round of execution depending on the status of process `j`. For example, if `j` is symmetric transmissive, we require that it sends the same value to all processes, that is, for every process `i` and `k`, we must have `c[k][j] = c[i][j]`.

The main property of this protocol is convergence:

```
convergence: LEMMA
  approx |- G(FORALL (i, j: PID): v[i] - v[j] <= delta);
```

This states that the values of two processes `i` and `j` differ by no more than the value `delta`, where `delta` decreases geometrically with each round of the algorithm.

Verification of this protocol is automatic with SALLY. This requires first translating the SAL model into MCMT, then invoking SALLY with the following command:

```
sally --engine pdkind --solver y2m5 --lsal-extensions \
  scenario2_revised_convergence.mcmt
```

The full SAL specification is given below.

```
%
% Reference:
%  M.H. Azadmanesh and R.M. Kieckhafer
%  Exploiting Omissive Faults in Synchronous
%  Approximate Agreement
%  IEEE Transactions on Computers, Vol 48, No 10,
%  October 2000.
```

```
%
% This differs from approx.sal in ../approximate_agreement
% by using a different fault-tolerant averaging function
% and fault model.
%

approx_scenario2: CONTEXT =

BEGIN

  %
  % Fault mode: status of process i
  % - good: not faulty
  % - benign: faulty and known to be
  %   by all good processes (we ignore them for now)
  % - symmetric ommissive: sends nothing
  % - asymmetric ommissive: sends a correct value
  %   to some, nothing to others
  % - symmetric transmissive: sends the same value
  %   to all (possibly incorrect)
  % - asymmetric transmissive: can do anything
  %
  STATUS: TYPE = { Good,
                   SymmetricOmissive,
                   AsymmetricOmissive,
                   SymmetricTransmissive,
                   AsymmetricTransmissive };


  %
  % The protocol requires
  %    N >= 3a + 2s + wa + ws + b + 1
  % and tau = a + s
  % where
  %   a = number of asymmetric transmissive faults
  %   s = number of symmetric transmissive faults
  %  wa = number of asymmetric omissive faults
  %  ws = number of summetric omissive faults
  %   b = number of benign faults.
  %

  N: NATURAL = 5;  %% number of processes

  TAU: NATURAL = 1; %% Maximal number of faults (non-benign)


  PID: TYPE = [1 .. N];

  %
  % Data = real values
  %
  DATA: TYPE = REAL;

  %
```

```
% Special value: 0 is interpreted
% as nothing sent
%
missing: DATA = 0;


%
% Sort and filter function
% - input: array v of N values
% - output:
%   n = number of values in v that are different from missing
%   p = a permutation p of the N indices such that
%   p[1] ... p[n] enumerate the n non-missing values of v in
%   increasing order
%
sort_and_filter(v: ARRAY PID OF DATA,
                n: [0 .. N],
                p: ARRAY PID OF PID): BOOLEAN =
      (FORALL (i: PID): i>n <=> v[p[i]] = missing)
  AND (FORALL (i: PID): i<n AND i<N => v[p[i]] <= v[p[i+1]])
     %% We need i<N to convince SAL that this is type correct
  AND (FORALL (i, j: PID): p[i] = p[j] => i = j);



%
% Voting: midvalue select
% - if we have n values, then we sort them in increasing
%   order and return (x[tau+1] + x[n-tau])/2
%
% If n is smaller than tau, we return missing.
%
% - input:
%    v = array of N values
%    n = number of non-missing values
%    p = permutation as defined above
%
midval(a: DATA, b: DATA): DATA = (a + b)/2;

vote(v: ARRAY PID OF DATA,
     n: [0 .. N],
     p: ARRAY PID OF PID): DATA =
  IF n > TAU THEN midval(v[p[TAU+1]], v[p[n-TAU]])
             ELSE missing ENDIF;




%
% Fault scenarios
% - all_good
% - scenario1: one symmetric transmissive
%            + one asymmetric omissive fault
% - scenario2: one asymmetric transmissive
```

```
%            + one asymmetric omissive fault
%
all_good(s: ARRAY PID OF STATUS): BOOLEAN =
   (FORALL (i: PID): s[i] = Good);

scenario1(s: ARRAY PID OF STATUS): BOOLEAN =
   s[1] = SymmetricTransmissive AND
   s[2] = AsymmetricOmissive AND
   (FORALL (i: PID): i > 2 => s[i] = Good);

scenario2(s: ARRAY PID OF STATUS): BOOLEAN =
   s[1] = AsymmetricTransmissive AND
   s[2] = AsymmetricOmissive AND
   (FORALL (i: PID): i > 2 => s[i] = Good);

%
% Initial precision:
% maximum difference between the values
%
initial_delta: { x: REAL| x > 0 };

%
% Approximate agreement: flat representation
%   status[i] = status of process i
%   v[i] = value of process i
%   c[j][i] = channel from process i to process j
%
% So c[i][1], ..., c[i][N] = values received by process i
%   n[i] = number of values in this list that
%          are not missing
%   p[i] = permutation used by process i
%
% To look at convergence properties, we add a
% variable delta such that all | v[i] - v[j] | <= delta
% for all good processes i and j.
%
% We want delta to decrease exponentially
% (with the number of rounds).
%
approx: MODULE =
BEGIN
  OUTPUT
    v: ARRAY PID OF DATA,
    c: ARRAY PID OF ARRAY PID OF DATA,
    p: ARRAY PID OF ARRAY PID OF PID,
    n: ARRAY PID OF [0 .. N],
    status: ARRAY PID OF STATUS,
    round: INTEGER,
    delta: REAL

INITIALIZATION
   %%
   %% Initial value: v[i] must not be missing
   %%
```

```
   delta = initial_delta;

   v IN { a: ARRAY PID OF DATA |
           (FORALL (i: PID): a[i] > 0)
       AND (FORALL (i, j: PID):
              a[i] - a[j] <= initial_delta) };

   p = [[i: PID] [[j: PID] j]];

   round = 0;

   status IN { s: ARRAY PID OF STATUS | scenario2(s) };

TRANSITION
   round' = round + 1;

   status' = status;

   %%
   %% communication and fault model
   %%
   c' IN { x: ARRAY PID OF ARRAY PID OF DATA |

            (FORALL (i: PID): status[i] = Good =>
              (FORALL (j: PID): x[j][i] = v[i]))

        AND (FORALL (i: PID): status[i] = SymmetricOmissive =>
               (FORALL (j: PID): x[j][i] = missing)
            OR (FORALL (j: PID): x[j][i] = v[i]))

        AND (FORALL (i: PID): status[i] = AsymmetricOmissive =>
              (FORALL (j: PID): x[j][i] = missing OR x[j][i] = v[i]))

        AND (FORALL (i: PID): status[i] = SymmetricTransmissive =>
              (FORALL (j, k: PID): x[j][i] = x[k][i]))
     };


   %%
   %% update rule:
   %% - process i receives c'[1][i] ... c'[N][i]
   %% - it removes all the missing values
   %% - it sorts the rest in increasing order
   %% - then it picks the median as its new value
   %%
   n' IN { x: ARRAY PID OF [0 .. N] | true };

   p' IN { x: ARRAY PID OF ARRAY PID OF PID |
             FORALL (i: PID):
               sort_and_filter(c'[i], n'[i], x[i]) };

   %%
   %% voting
   %%
```

```
    v' IN { x: ARRAY PID OF DATA |
              FORALL (i: PID):
               x[i] = vote(c'[i], n'[i], p'[i]) };

    %%
    %% convergence rate
    %%
    delta' = delta/3;

  END;


  %%
  %% Sanity check: v[i] is positive
  %%
  sanity: LEMMA
    approx |- G(FORALL (i: PID): v[i] > 0 );


  %%
  %% Sanity check2: n[i] is at least 3
  %% (except for round 0 since we don't initialize n[i]).
  %%
  min_received: LEMMA
    approx |- G(round = 0 OR (FORALL (i:PID): n[i] >= 3));

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

END
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-05-2018	Contractor Report	05/2014–05/2017

4. TITLE AND SUBTITLE
Advanced Symbolic Analysis Tools for Fault-Tolerant Integrated Distributed Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER
NNX14AI05A

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Bruno Dutertre, Dejan Jovanović, and Jorge Navas

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR–2018–219834

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 62
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Wilfredo Torress-Pomales

14. ABSTRACT

The project aims to develop advanced model-checking algorithms and tools to automate the verification of fault-tolerant distributed systems for avionics. We present a new method called Property-Directed K-Induction (PD-KIND) for synthesizing K-inductive invariants of state-transition systems. PD-KIND builds upon Satifiability Modulo Theories (SMT) to generalize Bradley's IC3 method and its variants. This method is implemented in a new tool called SALLY. Case studies show that PD-KIND can automatically verify fault-tolerant algorithms under a variety of fault models and that SALLY is competitive with other SMT-based model checkers.

15. SUBJECT TERMS

Fault tolerance; Model-Checking; Verification; distributed systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	36	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658