Strengthened State Transitions for Invariant Verification in Practical Depth-Induction*

Péter Bokor†
Budapest University of Technology
Budapest, Hungary
petbokor@mit.bme.hu

Sandeep Shukla
Virginia Tech
Blacksburg, USA
shukla@vt.edu

András Pataricza
Budapest University of Technology
Budapest, Hungary
pataric@mit.bme.hu

Neeraj Suri
Technische Universität Darmstadt
Darmstadt, Germany
suri@cs.tu-darmstadt.de

ABSTRACT

Bounded Model Checking (BMC) is often able to handle thousands of system variables by encoding the system and its properties via symbolic formulas and using satisfiability (SAT) solvers for verification. To further ease the verification of state invariants, BMC is augmented with a general induction rule called k-induction; however, this sacrifices completeness. Invariant strengthening, a method proposed to overcome this problem, often requires user intervention which limits its general applicability.

This paper presents a systematic method which is able to prove every property that is provable with standard k-induction and, in addition, further properties that the standard technique is unable to prove might be provable as well. Our case studies demonstrate the benefit of our approach with respect to plain k-induction. The main idea is to constrain the state transition relation in a way that the space of reachable states remains unchanged and k-induction is more likely to succeed. We show an implementation of our technique where the user needs only to extend the guard conditions with invariants obtained from the system's specification. This is always possible if the schedule of the executed transitions is (partially) known a-priori.

1. INTRODUCTION

Model checking [6] has proven to be one of the most effective verification techniques. Over the last two decades, model checking has been applied to various fields including hardware, software and protocol verification. The growing demand for verifying large systems has made model checking of explicit state models infeasible, therefore, the system and its properties were proposed to be represented by symbolic formulas. In fact, Binary Decision Diagrams made it possible to verify systems with hundreds of variables [10], SAT-based methods have gone even beyond that [2]. The latter approach is called *Bounded Model Checking* because only execution paths of limited length are translated into input formulas of the SAT engine. The technique guarantees that, given a bound k, the SAT instance is satisfiable if and only if there exists a counterexample (i.e., a violating execution) whose maximum depth is k. The limitation of the BMC approach is that general completeness, i.e., the guarantee that correct properties can always be proven, comes at high price. The intuition is to find the diameter of the system which can be thought of as the longest depth to exhibit any counterexample. In practice, however, this value is often too large for effective analysis.

To mitigate the "depth-explosion" of BMC for the verification of state invariants, an adaptation of mathematical induction was proposed in which, under favorable circumstances, no exhaustive exploration of the state space is needed [17]. Despite the practical success of k-induction, the application of this approach is limited by the fact that the proof of valid invariants might fail due to *spurious counterexamples*. Such runs are false negatives because they lead to unreachable states and are thus invalid executions of the system. A possible solution is to increase the induction depth, which, however, does not scale since the best known SAT algorithms are exponential in the number of input variables. This solution, even if the complexity of SAT was manageable, might not work for systems with infinitely long paths if a spurious counterexample exists for any large k. As an alternative solution, *invariant strengthening* has been proposed which constrains the input formula of the SAT solver such that no spurious counterexamples are generated and soundness of the algorithm is maintained (e.g., [9]). *The contributions of this paper are to (1) present a new method of invariant strengthening, which can be used in conjunction with other*

*Research supported in part by DFG GRK 1362 (TUD GK MM), EC Genesys and ReSIST.

†Péter Bokor is also with Technische Universität Darmstadt.

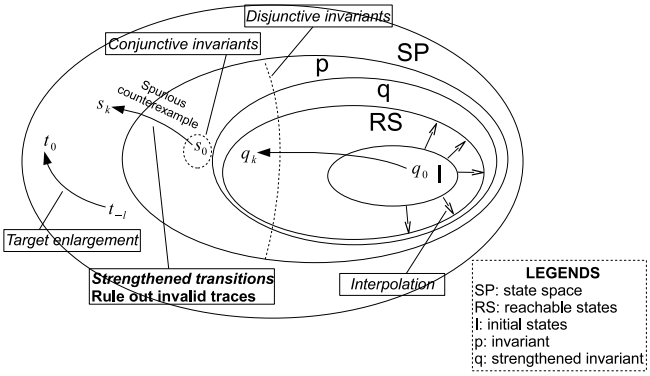

Figure 1: Comparing strengthened transitions with other methods

existing techniques, to (2) show an implementation of the proposed general technique, and to (3) demonstrate its practical usability in protocol verification.

1.1 Positioning the Method and Related Work

The verification of a safety property p is to decide whether every reachable state satisfies p. The different methods of induction-based safety verification can be categorized based on how the verification problem is (re-)formulated (Figure 1).

Assume that k-induction returns a counterexample which starts in the non-reachable state s_0 and reaches s_k after k steps such that the goal invariant (denoted by p) is violated in s_k (see Figure 1). This is a spurious counterexample because s_k is not reachable. If such a run exists, we say the property p is non-inductive. As a general solution, invariant strengthening was proposed where q (the strengthened invariant) is assumed to be inductive and p can be proven if q implies p. The question is how to obtain q? The different approaches can be classified as bottom-up or top-down. The first class only considers the given program, the approaches in the second class are guided by the goal assertion p. System-independent solutions for both classes are given in [13]. Specific knowledge about the system can also be used to find the appropriate q (e.g., see the study about the analysis of protocols for low-level data transmission [5]). Another systematic approach, called conjunctive invariants [9], uses the spurious counterexample to rule out s_0 as a possible state and derives the strengthened invariant $q = p \wedge \neg s_0$. The execution of k-induction can be preceded by a BDD-based pre-image computation of the undesired states (target enlargement) [1]. The predicate describing the states that can reach the target t_0 in l steps can be used to strengthen the invariant[1]. Another form of strengthened invariants is to prove the invariants of auxiliary formulas (defined as a lemma) and discard spurious counterexamples that contradict any of the lemmas.
A different strategy is using disjunctive invariants where the invariant of the system is built incrementally such that it is weakened every time k-induction finds a valid counterexample [15]. For example, q_k is a reachable state from initial state q_0 and the fact that q_k violates p proves that p is not an invariant of the system. Property p is now extended such that all configurations in the set of reachable states are covered. Another approach is using interpolants which are over-approximations of the set of reachable states; approximations are computed based on the unsatisfiability of reaching a violating state after up to k steps [14]. Interpolants are inductively computed (starting with the initial set of states) such that the resulting set is an over-approximation of the set of states reachable in i steps ($i < k$). It is guaranteed that the invariant is true when a fixed-point is reached. The motivation of using interpolation instead of k-induction is that the depth k needed for the first one is, in general, significantly smaller. However, not all bounded model checkers implement the interpolation algorithm.

We propose to strengthen the state transition relation of the system such that, for a sufficiently large k, system executions starting from unreachable states (spurious counterexamples) are ignored during the induction step. Note that, although our method resembles invariant strengthening, it does not bound the state space (by ruling out unreachable states) but limit the space of trajectories instead. Our technique is bottom-up in the sense that it does not depend on the property under verification. The previous approaches (except of interpolation) do not contradict with strengthened transitions, hence, hybrid algorithms can be directly derived. For example, properties proven via strengthened state transitions can be used as lemmas in subsequent proofs, or disjunctive invariants can be proven based on a system with a strengthened transition relation. Interpolation, on the other hand, uses fixed-point calculation rather than induction as proof method; therefore, techniques towards inductive invariants are not directly applicable here. Note that our method, if successful, runs k-induction only once, whereas other induction-based approaches use failed proofs or pre-analysis before the property can be proven.

We present a general framework which uses k-induction as verification method and strengthens the state transition relation of the system. We prove that the framework, when used for verification, (i) might enhance the completeness of k-induction and (ii) preserves soundness. Analogous to invariant strengthening, the main challenge is to invent how to obtain the strengthened state transition relation. As a special case, we show how the Boolean guards of the system's transitions can be modified to strengthen the state transition relation. For this method to work, we make the assumption that the control flow of the system is (at least partially) known. This technique is demonstrated through the next simple example.

1.2 A Short Example: the Bakery Protocol

We show how strengthened guards can be used to model check a simplified version of Lamport's Bakery protocol [11]. We use SAL's implementation of k-induction [8] which applies none of the techniques mentioned above. As the Bakery algorithm works with infinite domains (the ticket number of a process can be any natural number), we use SAL's infinite bounded model checker (sal-inf-bmc) for verification[2]. The

[1]The parameter l can be tuned such that the BDD computation is still effective.

[2]sal-inf-bmc makes calls to the Yices [8] Satisfiability Modulo Theories (SMT) solver which can also handle infinite

```
1 bakery: CONTEXT = BEGIN
2  PC: TYPE = {sleeping, trying, critical};
3  job: MODULE =
4    BEGIN
5      INPUT y2 : NATURAL
6      OUTPUT y1 : NATURAL
7      LOCAL pc : PC
8      INITIALIZATION
9                  pc = sleeping;
10                 y1 = 0
11    TRANSITION
12      [ pc = sleeping --> y1' = y2 + 1;
13                          pc' = trying
14        []
15        pc = trying AND (y2 = 0 OR y1 < y2)
16 AND y1 > 0          %STRENGTHENED GUARD
17             --> pc' = critical
18        []
19          pc = critical --> y1' = 0;
20                          pc' = sleeping ]
21  END;

22  system: MODULE =
23   job
24   []
25   RENAME y2 TO y1, y1 TO y2 IN job

26  mutex:THEOREM system
27   |- G(NOT(pc.1 = critical AND pc.2 = critical));
28 END
```

Figure 2: The SAL model of the Bakery protocol

SAL code of the protocol with two processes is depicted in Figure 2. The protocol's main property, mutual exclusion (`mutex`), is defined as an invariant saying that it is never true that both processes are in the critical section at the same time.

SAL cannot prove `mutex` using the default depth 10. A spurious counterexample of length 10 is produced starting from a state where the 1^{st} process has `pc=trying`, `y1=0`. This is not a reachable state because every process increments its ticket when changing from `sleeping` to `trying` (line 12-13). As a result, the system reaches a state which violates mutual exclusion. Note that the original model cannot be proven with any depth. Therefore, it does not help increasing k to any large number. This can be seen by considering that the state where the two processes have respectively `pc.1=trying`, `y1=0` and `pc.2=trying`, `y2=1` is a recurring one because the 2^{nd} process returns to the same state if it first enters the critical section. As a result, a counterexample similar to the previous one can be produced for any value of k. We propose to strengthen the state transition relation such that the spurious counterexample is not a possible run of the system. We do it by simply adding a new clause to the guard of the transition that drives the process into the critical section (line 16). The new condition requires that `y1` is always positive when the process is at `pc=trying`. The theorem can now be proven.

Our observation is that the previous solution gives rise to a general approach if the system follows a strict control flow. The idea is to use the updates in the previously executed transitions for strengthening the guards. In particular, each Bakery process periodically alternates between `sleeping`, `trying` and `critical`. Therefore, we can safely state (without changing the specified behavior) that the assignment of `y1` at line 12 is still valid at line 16.

Paper structure. First we establish a general framework of transition strengthening (Section 2), then we present strengthened guards as a special case (Section 3). Finally, we use strengthened guards to formally verify a diagnosis protocol with more than 150 lines of SAL code and compare our technique with other approaches (Section 4).

2. A SIMPLE FRAMEWORK

We first formally define the system, its properties and the applied proof method (Section 2.1), then a general technique is presented and its properties are formally proven (Section 2.2).

2.1 Preliminaries

Assume that the *system* is defined as a general state transition graph with tuple $M = (S, I, R, L)$. As usual, S denotes the set of states, $I \subseteq S$ the set of initial states, $R \subseteq S \times S$ the state transition relation and $L : S \rightarrow 2^{AP}$ the labeling function (AP is the set of atomic propositions). For simplicity, we use the notations $I(s)$ iff $s \in I$, $R(s, s')$ iff $(s, s') \in R$. A *path* in M is a sequence of states $s_0, ..., s_k$ iff $R(s_i, s_{i+1})$ for all $0 \leq i < k$. Note that this definition of path does not require the first state to be an initial state. We use the predicate $path(s_0, ..., s_k)$ to designate paths. Furthermore, we define the set of *reachable states* in M as $Reach_M = \{s | I(s) \vee \exists s_0, s_1, ... : I(s_0) \wedge path(s_0, ..., s)\}$.

Properties are formulas that are defined based on the formal definition of the system. We restrict to *state invariants* (or simply invariants) which are true in all reachable states. Formally, invariants can be defined over AP using the standard Boolean operators. We use the shorthand $P(s)$ such that the predicate is true iff p holds in s, i.e., the atomic propositions in $L(s)$ satisfy p. A state s is called *P-state* iff $P(s)$ holds.

We assume that k-induction is used to verify invariant p in system M. Before giving the formal definition of k-induction, we describe it informally. The intuition is to check that paths of length k starting from an initial state visit only P-states. If it is not always the case, a counterexample of length k is found. The dilemma is to determine the value of k such that the invariance of the property can be safely established. The strategy is to check whether there exists a $(k+1)$-long path starting from an arbitrary state and leading to a non-P-state. If not, we can safely stop since all shorter paths have already been checked. Otherwise, the value of k needs to be increased [17]. Note that this method corresponds to the generalization of the simple induction rule which first states that the invariant includes all initial states (base case), then it proves that the invariant is closed on the transitions (inductive step). For the formal definition of k-induction we use the one taken from [9]. The proof is parameterized with a system M, depth k and property p. A

domains. We note that in the context of this paper, SAT and SMT solvers are conceptually identical in that they both decide satisfiability of logical formulas.

k-induction proof instance is denoted by $IND_M(k)(p)$ and is true if the following two predicates hold for all $s_0, ..., s_k$.

Base case: $I(s_0) \wedge path(s_0, ..., s_{k-1}) \rightarrow P(s_0) \wedge ... \wedge P(s_{k-1})$

Induction: $P(s_0) \wedge ... \wedge P(s_{k-1}) \wedge path(s_0, ..., s_k) \rightarrow P(s_k)$

Note that there is no need to explicitly search for paths shorter than k if we can assume that the system is live. In this case, violating paths of length $< k$ are also checked by $IND_M(k)$.

Finally, a proof method is called *sound*, if the fact that it proves p in M implies that p is indeed true in M. It has been shown that k-induction is sound with respect to state invariants [17, 9]. Furthermore, a proof method is *complete* if every property p which is true in M can be proven by the method. There have been several attempts towards the completeness of k-induction. In worst case, exhaustive exploration can be used if there is an upper bound on the length of the possible trajectories of the system. It is often possible to find such an upper bound even in systems with infinite paths. For example, path compression can be used to make sure that a trajectory only visits "new" states [9]. As the most common special case, only loop-free paths are considered. In this case, an alternative induction step is to check that no $(k+1)$-long loop-free path starts form an initial state [17]. Consequently, we can conclude that all loop-free paths are covered by the base case and the entire reachable state has been explored. Formally, the definition of *path* predicate needs to be modified if compressed paths are used. Although our technique works with all known optimizations, it is not dependent of any of them, therefore, they are omitted in the formal discussion.

2.2 k-Induction with Strengthened Transitions

We define a new system based on the original specification which will be used during the verification. The intuition is to constrain the state transition relation such that the observable behavior of the system remains the same. In this paper, we concentrate on state invariants, therefore, the preservation of the behavior corresponds to having the same set of reachable states in both systems. Next, we define systems with strengthened transitions in a declarative way, i.e., without discussing how the conditions can actually be fulfilled. We will show an implementation of such systems in Section 3.

Definition 1. $M' = (S, I, R', L)$ is a *system with strengthened transitions* with respect to $M = (S, I, R, L)$ if $R' \subseteq R$ and $Reach_M = Reach_{M'}$.

The next simple theorem claims that a system with strengthened transitions might improve but never weakens completeness of k-induction. This is equivalent with showing that the set of provable invariants in M is never greater than that in M'. In general, full completeness is not reached, therefore, both sets are a subset of the set of all invariants in M.

THEOREM 1. *The completeness of k-induction in a system with strengthened transitions can be characterized as follows.*

$$\{p|IND_M(k)(p)\} \subseteq \{p|IND_{M'}(k)(p)\} \subseteq \{p|M \models \mathsf{G}p\}$$

PROOF. Assume that p is an invariant. From Definition 1, M and M' entail the same set of reachable states. Therefore, the base case of $IND_M(k)(p)$ is true iff it is true in $IND_{M'}(k)(p)$. As $R' \subseteq R$, $path(s_0, ..., s_k)$ in M' implies $path(s_0, ..., s_k)$ in M, thus, $IND_M(k)(p)$ implies $IND_{M'}(k)(p)$. We have proven that if k-induction can verify p in M it can also prove it in M'. We now show that it is possible that $\{p|IND_M(k)(p)\} \subset \{p|IND_{M'}(k)(p)\}$. Assume that $\neg P(s_k)$, $path(s_0, ..., s_k)$ in M for some $s_0, ..., s_k$ but $\neg path(s_0, ..., s_k)$ in M' for any $s_0, ..., s_k$. This is possible if $R' \subset R$. Finally, we prove that general completeness cannot be guaranteed. Assume $\neg P(s_k)$ and $path(s_0, ..., s_k)$ in M'. In this case, $IND_{M'}(k)(p)$ is false even if p is an invariant. □

2.3 Discussion

Refutation. In general, better completeness with strengthened transitions comes at a price. Since the new transition relation is restricted with respect to the original one, it is possible that the shortest path between two reachable states s_0 and s increases from k (in M) to $k' > k$ (in M'). As a result, the base case of the induction might find counterexamples with greater depths. For example, assume that $P(s)$ does not hold and k' is the length of the shortest path to s in M'. In this case, a counterexample in M' can only be found with depth k', whereas, it suffices to use depth k in M. As finding counterexamples quickly is particularly important in early phases of the system development, we propose using M for refutation and M' for verification. However, making this differentiation is not always needed. In the next Section, we show an implementation of transition strengthening which never degrades the method's ability to disprove invariants.

Soundness. We remark that our method, while improving completeness of k-induction under favorable circumstances, preserves the general soundness property. This directly comes from (i) the restriction to state invariants and (ii) the condition that the new system must entail the same space of reachable states. Accordingly, if a state s is reachable in M it is also reachable in M', therefore, any state invariant is true in M iff it is true in M'. The last statement and the fact that k-induction is sound imply that the verification of M via M' is sound.

3. AN IMPLEMENTATION

Previously, we introduced a theoretical framework to enhance the completeness of induction-based BMC. The main premise was that it is possible to create a system with strengthened transitions. But how can we do it in practice? In this section, we give a solution as a possible implementation of the general framework. The heart of the solution is the assumption that a fixed control flow of the system exists.

3.1 Preliminaries

To ease further discussion, we use interpreted first order formulas to describe the system. The predicates and function symbols as well as the translation from (and to) the tuple-based representation of Section 2 are defined similarly to those in [6]. Let $V = \{v_1, ..., v_n\}$ be the set of *system variables*. Variables range over a finite *domain* D. A state $s : V \rightarrow D$ is interpreted as a valuation of V assigning values from D to a subset of variables. $\mathcal{I}$ denotes a first order formula which is only true for valuations representing *initial states*. $\mathcal{R}(V, V')$ is the formula which corresponds to the *transition relation*. Variables in V are thought of as current state variables and those in V' are the next state variables. The formula is true if the valuations of V and V' represent a current and next state of the system. Finally, *properties* are defined based on atomic propositions of the form $v = d$ where $v \in V$ and $d \in D$. A proposition is true in a state s if $s(v) = d$.

Using *guarded commands* is a common way to describe a system's transitions. The simple example of Figure 2 also used guards to rule the execution of transitions. The next values of variables can be updated whenever a guard condition based on current values is true. In general, we can assume that the transition formula implements the following template: $(guard_1 \wedge update_1) \vee (guard_2 \wedge update_2) \vee ...$, where $guard_i$ is restricted to be defined over V. We say that the system is defined with guarded commands if $\mathcal{R}(V, V')$ implements the previous template. In addition to the definition of all possible state transitions, a *scheduler* is attached to the system to decide which transitions are executed and in which order. For example, it is the scheduler's job to resolve conflicts when more than one guard is enabled. The SAL scheduler, for example, selects one of the enabled transitions non-deterministically or, if none of the guards is true, it executes the default transition if it is defined (using the `ELSE` keyword). The system deadlocks if no transition can be executed.

3.2 Execution Plan: Fixed Control Flow

Our implementation of systems with strengthened transitions is based on the assumption that the control flow of the system is a-priori known. For example, in the Bakery protocol each process executes the same transitions in the same order. More generally, we assume that an *execution plan* is available which contains information about the order of executed transitions. Let $trans_i$ denote the formula $(guard_i \wedge update_i)$. In the simplest case, an execution plan is a sequence of positive integers $i_1, i_2, ...$ meaning that $trans_{i_j}$ is the j^{th} transition executed by the system. We call it *total* execution plan. However, even if a system adheres to a control flow, partial non-determinism is possible. Therefore, we define *partial* execution plans to be a similar sequence of numbers with the guarantee that $trans_{i_k}$ is executed after $trans_{i_j}$ iff $j < k$. We allow multiple partial execution plans for the same system, however, with the restriction that they contain distinct transitions. This makes sense otherwise there was an uncertainty about what transitions are preceded by the one appearing in multiple plans. For example, the Bakery protocol with two processes can be associated with two partial execution plans each of them containing the three transitions between lines 12 and 20 (Figure 2). Note that it is not possible to associate a total execution plan with the Bakery protocol as the participating processes execute transitions asynchronously without an a-priori global order of transitions.

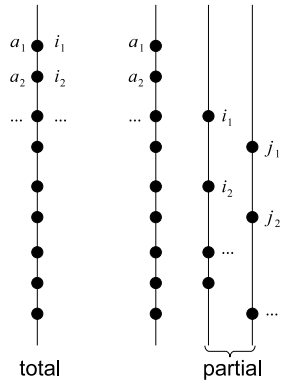

Figure 3: Execution plans and actual executions

Denote the sequence $a_1, a_2, ...$ an *actual execution* where $trans_{a_i}$ is the i^{th} transition executed by the system. The actual execution models an arbitrary run of the real system. The relation between execution plans and actual execution is depicted in Figure 3.

3.3 Strengthened Guards

The main idea of strengthened guards is to prune some of the impossible behavior by taking into account the scheduling of the transitions in the real execution of the system. We only make a natural assumption to require that a transition can only be executed if its guard is true. To eliminate impossible runs we use the updates of previously executed transitions to strengthen the guard. If the system adheres to the execution plan the new guard will be true exactly when the old one is true because no further updates have been done to the variables. In k-induction, the values of state variables in spurious counterexamples are usually not in accordance with the execution plan. Consequently, such runs can be ruled out by using strengthened transitions.

The first solution is to replace the original guard $guard_i$ with $guard_i \wedge (\bigvee update_{i_{j-1}})$ for all $i_j = i$ in the execution plan. It is possible that a transition is preceded by different transitions in the execution plan. Therefore, we add all of them such that only one needs to be true. If a transition is executed as first, the possible initial assignments can also be added to the strengthened guard. Since execution plans are disjunct or total, at most one execution plan is used for the replacement of a guard. Before formally defining the strengthened system, consider the following two issues.

- Formally, $update_i$ is defined over V and V'. Therefore, its syntactical rewriting is needed when used in a guard because guards can only use current state variables. For example, the update `y1'=y2+1` can be used as `y1=y2+1` in the strengthened guard. However, this is not correct if current state variables that are used in the assignment are modified by the same update. In the previous example, changing the value of `y2` would

mean that `y1=y2+1` will not be true. In general, if variable v is used as both current and next state variable by the same update, its old value stored in an (auxiliary) variable *v_old* can be used in the strengthened guards (e.g., `y1=y2_old+1` in the previous case).

- We have to be careful with partial execution plans because transitions that are not part of the plan can modify the assignments with respect to the ones in $update_{i_{j-1}}$. For example, the update `y1=y2+1` could not be directly used to strengthen the guard in line 15 (Figure 2) because the other process might change `y2` in the meanwhile. Therefore, for the general case, we assume that there is a function fun which extracts the possibly strongest condition which can be safely used to strengthen the guard; fun takes an update as input and returns a formula over V. For example, in the strengthened version of the Bakery protocol, $fun(y1' = y2 + 1)$ returns $y1 > 0$ to use it for strengthening the guard at line 15. Techniques for implementing fun in an effective and automated way are part of our future work (see Section 3.6 for more).

The system with strengthened guards M' can now be defined based on the original specification M of the system. Note that the transformation from M to M' can be automated if we suppose that the function fun and the execution plans are given.

Definition 2. Supposed that executions plan(s) $i_1, i_2, ...$ of a system with guarded commands, described with $\mathcal{I}$ and $\mathcal{R}(V, V')$, are available; the system with *strengthened guards* is described with $\mathcal{I}$ and $\mathcal{R}'(V, V')$ such that every $guard_i$ in $\mathcal{R}(V, V')$ is replaced by $guard_i \wedge (\mathcal{I} \vee \bigvee fun(update_{i_{j-1}}))$ for all $i_j = i, j > 1$. The strengthened guard only includes $\mathcal{I}$ if $i_1 = i$.

We claim that Definition 2 is a special case of Definition 1. Therefore, the result of Theorem 1 and the discussion in Section 2.3 is valid.

COROLLARY 1. *Assume that a system A is defined with guarded commands. The system A with strengthened guards (call it A') is a system with strengthened transitions with respect to A if A' is scheduled as A.*

PROOF SKETCH. Assume that A and A' are represented by tuples $M = (S, I, R, L)$ and $M' = (S, I, R', L)$ respectively. This assumption is valid because Definition 2 only modifies $\mathcal{R}(V, V')$. By intuition, $R' \subseteq R$ because guards are never weakened and A and A' share the same schedule (see Section 3.4 for more details). We have to prove that $Reach_M = Reach_{M'}$. Assume that the execution plan $i_1, i_2, ...$ is total. In this case, $fun(update_{i_j}) = update_{i_j}$. For every reachable state s in M, there is $s_0, s_1, ...$ such that $I(s_0)$ and $path(s_0, ..., s)$. We use induction by the position of states in the path. $I(s_0)$ is true in both M and M'. s_j is computed by executing $guard_{i_j} \wedge update_{i_j}$. The same s_j can be computed by $trans_{i_j}$ in M' if $guard_{i_j} \wedge update_{i_{j-1}}$ (the strengthened guard) is true. The condition is indeed true because $update_{i_{j-1}}$ represents the assignments in s_{j-1} which is the current state. If the current state is an initial state, $guard_{i_1} \wedge \mathcal{I}$ is true. If the execution plan is not total, the proof is dependent of the implementation of fun. Assuming that $fun(update_{i_j}) = update_{i_j}$, the proof is similar to the one presented above. □

Note that our technique does not eliminate transitions, it only strengthens the guards. Since every new guard is trivially true on paths starting from initial states (this is guaranteed by construction), valid *counterexamples* can be found with the same depth as for the original system. Therefore, unlike in the general case (see Section 2.3), there is no need to use the original system for seeking counterexamples. In the next Section, we explain why Corollary 1 makes the assumption on the schedule of transitions.

3.4 Preserving Execution Semantics

We say that a system with strengthened transitions A' is scheduled as the original system A if they execute the same sequence of transitions if possible. This ensures that if a strengthened guard is not true then no other transition is taken and the execution stops, i.e., no path of depth k exists.

Note that the scheduler which triggers the execution of transitions might prevent the previous condition from being true. It is the core of our technique that the execution stops if a transition which is supposed to be executed is not enabled. However, the scheduler might select another transition which can be alternatively executed. In SAL, for example, having the default transition might cause the strengthened system to explore states that are not entailed by the original system. There are two options to circumvent the undesired interplay of the scheduler. In theory, it is possible to directly translate the specification into the tuple-based representation without using the built-in scheduler of the execution environment. This solution is cumbersome and can only be viable if the process of translation is fully automated. Another option is to use the scheduler at hand and enforce it to implement the desired schedule. Usually it is not a hard thing to do. For example, ELSE-branches can easily be eliminated from the original SAL specification by replacing them with regular guarded commands. We show an example of how to do it in Section 4.

3.5 Optimizations

Usually, an update only changes the assignment of a real subset of all state variables instead of changing each of them. Therefore, guards can be further strengthened with updates that are not defined by direct predecessors of a transition plan but whose assignments (or some of them) are not changed by the subsequent transitions.

Another refinement of our basic technique is to strengthened guards based on variables (say *v_old*) that store old (e.g. the previous) values of another variable (v). In this case, the assignments of v can be used in guards even if value of v has already been re-assigned. It is also possible to store more than one old value (i.e., from different states) of the same variable. In this case, a good tradeoff has to be found between the overhead of the verification and the benefit of using strengthened guards. Note that *v_old* is not necessarily

an auxiliary variable of the system that is only introduced to implement strengthened guards. It is a common technique to use such variables in order to express liveness properties as invariants. Variables storing earlier values of other variables can also be a functional part of the system's model if some correlation between the current and previous values needs to be established. Our case study shows an example of both optimization techniques.

3.6 Discussion

Transition-validated assertions. The technique of transition validated assertions (TVAs) is a bottom-up approach for strengthened invariants [13]. By definition, a TVA is an invariant q such that $update_i \rightarrow q$ for every $trans_i$ interfering with q. Accordingly, every TVA is 1-inductive and can be used to strengthen the original property p. Our technique of strengthening the guards based on a fixed control flow can be expressed via TVAs as well. Assume that the transitions of the system are uniquely labeled. For example, $trans_i$ can be labeled by i. In the Bakery example, the values of `pc.1` and `pc.2` provide an appropriate labeling. Labels serve, besides providing the means of identifying statements, as locations of control. Note that a location can designate more than one label at the same time. For example, if the control of the Bakery protocol is at `pc.1=sleeping`, it might be also at `pc.2=trying`. Let the auxiliary predicate $l(i)$ be true iff the control is at $trans_i$. For simplicity, denote the strengthened guard of $trans_i$ by $guard_i \wedge (\bigvee rewr\_update)$. Accordingly, the formula $l(i) \rightarrow (\bigvee rewr\_update)$ is a TVA because every update that might lead to location described by $l(i)$ is included in $(\bigvee rewr\_update)$. This is guaranteed by the execution plan. It can be seen the formula is still 1-inductive if our optimization of using earlier updates to strengthen a guard is applied (see Section 3.5).

Our work differs from TVAs in that it is presented in the context of strengthened transitions. Even though strengthened guards and TVAs are logically equivalent, their implementation can entail different overheads as shown in our case study. Furthermore, we use execution plans to derive strengthened guards which can be automated or easily implemented by the user.

Control flow-determinism. One might argue that an execution plan about the system does not exist or is unknown. In fact, strengthened guards are useless in the former case. However, we believe that a static sequence of actions in the system's execution can be observed for a variety of applications. For example, safety-critical embedded systems utilize deterministic protocols like diagnosis, membership [16] or startup [19] for predictability. We also think that execution plans can be, in many practical cases, easily determined based on the system's specification. For example, the execution plans of the Bakery protocol can be obtained from its original description [11] without the need of understanding how the protocol works: the specification of each process is given as a sequence of actions which corresponds to partial execution plans. In other cases, the implementation restricts the unconstrained schedule of the high-level formal description. For example, the compiler injects control flow information of the hardware design with respect to the abstract functional specification on which the formal analysis is performed [18]. Therefore, the guards in the specification can be strengthened if we know what the compiler does in synthesizing the schedule. The automation of determining the control flow is part of our future work.

Update rewriting. Although the implementation of fun is kept open as future work, we speculate about some aspects of its possible automation. Given an update upd, we look for the result of $fun(upd)$ to strengthen the guards in accordance with the execution plan(s). Call a variable in V' *stable* with respect to upd if its value remains unchanged until the execution of the strengthened transition. Otherwise, the variable is called *unstable*. Now, $fun(upd)$ is the identity function if all variables of V' appearing in upd are stable. This means that upd can be used in the strengthened guard after the trivial syntactical rewriting described in Section 3.3. Otherwise, the empty constraint ($true$) can be used which corresponds to discarding upd. Decision procedures can be used to obtain more sophisticated solutions. For example, unstable variables in upd can be replaced by symbolic constants and automatic static analysis can be used to derive a provably valid but non-empty constraint. In the example of the Bakery protocol, considering the domain of the variables, $fun(y1' = y2 + 1) = (y1 > 0)$ can be computed without knowing the value of $y2$.

4. EXECUTION PLANS IN SAL: VERIFYING A REAL PROTOCOL

As a proof of concept, we use execution plans to verify a diagnosis protocol with the SAL model checker. The techniques of strengthened guards, transition-validated assertions and lemmas are compared. Our experiments show that all techniques are able to prove the properties that are non-inductive in the original model. However, additional overhead is induced due to the manipulation of the model. Using strengthened guards, the pure execution time of the k-induction rule is the shortest among all applied techniques.

In the remainder of this Section, we first briefly describe the subjected protocol and its implementation in the SAL language (Section 4.1), then we present how different techniques can be implemented to reduce the depth of k-induction based on the a-priori known control flow of the system (Section 4.2). Finally, we compare the performance of these techniques using the BMC model checker of the SAL environment (Section 4.3).

4.1 Diagnosis with Hybrid Faults

Diagnosis is a service that is able to locate faults in a system. In our case study, we use a distributed diagnostic protocol applicable in *synchronous* environments[3]. The execution model is that the nodes proceed through a parallel sequence of rounds such that each round is split into communication and computation phase. The algorithm is based on round-based consensus with malicious (aka. Byzantine) faults [12]. The plain model of worst-case faults is augmented with other less severe faults such as *benign* faults or *symmetric* value

[3]Communication based on message sending is assumed where correct nodes are able to send and deliver a message on time.

faults. A benign node executes the protocol but it is unable to send a message or sends wrongly formatted data. A symmetric node is malicious faulty with the restriction that it sends the same message to everyone. The benefit of the hybrid fault model is that it entails enhanced fault tolerance with the same number of replicas. The protocol used in this case study, called hybrid diagnosis (HD), executes consensus on the local syndromes about the health status of the system [20]. It guarantees that, under the fault hypothesis, (i) all non-faulty nodes agree on the diagnosis of the system (*consistency*), (ii) all benign faults are eventually detected (*completeness*) and (iii) non-faulty nodes are never diagnosed faulty (*correctness*). The fault hypothesis defines that the overall number of nodes N is greater than $(2a + 2s + b + 1)$ where a, s and b denote the number of malicious, symmetric and benign faults, respectively.

The basic assumption of the protocol is, besides synchrony, that distributed nodes can run computation and send/receive messages in parallel with each other. This might not be the case in systems where resources are shared to reduce cost. Consequently, the protocol (e.g., [16]) and its formal model must be modified to accommodate the conditions. We remark that the latter is not always needed. For example, the same model of the presented protocol can be used even if the system uses a shared communication bus and if different applications run on the same computer [4]. This is achieved by using an abstraction and showing a bi-simulation between the abstracted and the original models.

The protocol. The SAL language allows the user to define *modules* which can be thought of as building-blocks of the overall model. The model of the protocol contains two modules, one describing the fault model, the second defining the protocol's operations executed by the distributed nodes. As every node is supposed to execute the same code, the module defining the protocol is parameterized with the ID of the corresponding node. We use SAL's guarded commands to identify the different stages of the protocol and to execute the part of the code which is associated with that stage. Additional stages are defined when auxiliary operations (e.g., fault generation) are being computed which cannot overlap with the protocol's operation. In accordance with the synchrony assumption, the modules are composed together using SAL's *synchronous composition*. This ensures that transitions are executed in a lock-step manner, i.e., the execution stops unless an enabled transition can be selected in every module and the selected transitions can be executed following parallel execution semantics (non-conflicting updates, etc.) [7]. Recall that the Bakery example of Section 1 used asynchronous composition, where only one transition is selected and executed at each step of the model[4].

The protocol HD is a sequential and periodic execution of the following two rounds:

[4]We remark that SAL's different composition semantics can be considered as instructions for compiler about how the transition relation is calculated.

Variable	Stage	Dependency	Valid at stage
`ls_symm'`	0	`fvec`	1-3 (+0 w/ `fvec_old`)
`ls'[i]`	0	`fvec`	1-3 (+0 w/ `fvec_old`)
`sm'[i]`	1	`fvec` `ls[1..N]` `ls_prev[1..N]`	2-3 (+0 w/ `fvec_old`)
`chv'[i]`	2	`sm[i]`	3,0,1
`fvec'`	3	`fvec`	0,1-3 (all w/ `fvec_old`)

Table 1: Variables and their dependencies in the SAL model of the HD protocol: N is the number of nodes, `var[i]` denotes `var` at node i and the primed version of a variable means its next value.

Round 1

1. *Sending workload (communication):* every node i broadcasts a message.
2. *Local detection (computation):* every node i diagnoses the other nodes based on the messages received from them and forms a *local syndrome* (`ls[i]`) indicating the health status of each node. The j^{th} value of this vector is 0 if node j is diagnosed faulty and 1 if it is correct.

Round 2

2. *Global dispersion (communication):* every node i broadcasts its local syndrome.
3. *Global assimilation...:* the local syndromes received from the other nodes, together with the syndrome obtained in Round 1, are compiled into a *syndrome matrix* (`sm[i]`). The j^{th} row of the matrix is the local syndrome received from node j.
4. *...and analysis (computation):* Every node i computes hybrid majority on the values of each column in the syndrome matrix and derives a *consistent health vector* (`chv[i]`).

The protocol uses a special form of majority function, called *hybrid majority*, where wrongly formatted (i.e., semantically incorrect) local syndromes and the node's opinion about itself are omitted in the voting; the default outcome is "correct" if no value is in majority.

Table 1 depicts the variables that are used to encode HD in the SAL language[5]. The different stages indicate the protocol's steps and auxiliary operations. The current stage is stored in a control variable called `pc` (program counter); this variable is re-set to 0 when it reaches 3 to enable periodic execution. In addition, the following variables are defined: the fault vector (`fvec`), its old value (`fvec_old`)

[5]The source of models in this Section are available at http://www.deeds.informatik.tu-darmstadt.de/peter/sal/sources. The model of the HD protocol can be found under diagnosis.sal.

and a variable to store semantically incorrect but symmetrically disseminated local syndromes (`ls_symm`). The fault vector is updated at the end of each round such that the fault hypothesis is satisfied. It is assumed that, in each instance of the protocol, a node can be faulty according to at most one fault class. For example, it cannot be that a node is benign faulty at round 1 and Byzantine during round 2. The variable `ls_symm` contains the local syndrome that a symmetric faulty node sends to the other nodes. In our experiments, we only verify systems with $N < 6$; therefore, it suffices to have just one such a vector since s (the number of symmetric faults) is at most one. Note that overlapping instances of the protocol can be executed. This means that in every communication round k instance A executes Round 2 and another instance B launches the protocol by executing Round 1. Consequently, in our SAL model, the syndrome matrix of node i (`sm[i]`) is not updated based on the local syndromes of this round but on those from the last round (`ls_prev[1],...,ls_prev[N]`).

Properties. We define all three properties of the HD protocol: consistency, correctness and completeness. Consistency can be naturally defined as a state invariant because it requires that the consistent health vector is consistent among the nodes at the termination of each instance, i.e., at stage 3. Since the protocol terminates after Round 2, liveness properties have to be verified with one round delay. Accordingly, correctness requires that no node that was correct during round $(k-1)$ is diagnosed as faulty in round k; completeness means that every benign fault occurring in round $(k-1)$ must be detected at round k. It is possible to define correctness and completeness as invariants as well, if we store the fault vector of the previous round. For example, diagnosis can be defined as follows (`chm[i]` corresponds to `chv[i]`):

```
diagnosis_completeness: THEOREM system |-
    G(FORALL(l,m:nodes):pc=3 => (
        fvec_old[l]=benign => chm[m][l]=0));
```

4.2 Inductive Invariants with Execution Plan

If the control flow of the system is a-priori known, the system's model can be modified such that the properties of the system (expressed as invariants) are more likely to inductive. The presented model of the HD protocol gives rise to a total execution plan. This is because the protocol is fully deterministic, the modules of the SAL model are connected via synchronous composition and the auxiliary operations (the updates of `fvec` and `ls_symm`) do not cause non-determinism in the control flow. Given that one is implemented via four stages, the j^{th} transition in the execution plan is always the transition guarded by stage j modulo 4. Note that, although the existence of total execution plans requires that the control flow is deterministic, the data flow can contain non-determinism. For example, our model non-deterministically generates local syndromes of Byzantine nodes, the activation of faults, etc.

Strengthened guards. We have implemented a model of the HD protocol with strengthened guards. The complete model is available in the source file diagnosisSG.sal. Now, we discuss the outline of our implementation. The technique of strengthened transitions is based on restricting the global transition relation. In case of synchronously composed modules, the global transition relation can be derived from the corresponding transitions of the modules. Since a module is only allowed to update local variables (which might depend on input variables), it is possible to strengthen the guards "locally" in each module. For example, this is how we strengthen the guard `pc=3` with the assignment of `chv` from the last transition (note that the update part of the transition is empty because stage 3 is an auxiliary transition to update the fault vector in the fault module):

```
pc=3
%---- Guard strengthening STARTS --
AND chv=[[n:nodes] h_maj(sm)[n]]
%---- Guard strengthening ENDS ----
-->
```

We do not use SAL's ELSE branch in the model to preserve execution semantics. The program is forced to execute the transition (at stage 0,1,2 or 3) determined by the execution plan. Otherwise, the program's execution is blocked. Once again, this does not happen in regular runs, i.e., when the the program is started from a proper initial state. We applied the optimization when updates that are not in direct predecessors of a transition are also used to strengthen the guard of that transition. Table 1 helps determining the strengthened guards that can be used at different stages of the protocol. The third column depicts the dependencies of each state variable. An update of a variable can be used to strengthen a guard until none of its dependencies is re-updated. For example, `chv` is dependent of `sm` which is updated at stage 1. Consequently, the update of `chv` at stage 2 can be used from stage 3 to stage 1 in the next round. Furthermore, the updates of variables depending on `fvec` can be used in guards even after `fvec` is updated because its old version (`fvec_old`) is also available. Finally, we note that this example of guard strengthening assumes the trivial implementation of fun where $fun(upd) = upd$. This is because the execution plan is total and the optimization takes the dependencies into account.

Transition-validated assertions. We have strengthened the properties of the HD protocol with transition-validated assertions that can be obtained through the total execution plan. The model and its properties are available in the source file diagnosisTVA.sal. For a fair comparison, we used the same assertions that appear in the strengthened guards of the previous model. For example, the following assertion corresponds to the guards strengthened at stages 3, 0 and 1 with the update of `chv`.

```
pc/=2 => FORALL(m:nodes):
    chm[m]=[[n:nodes] h_maj(sm_vec[m])[n]]
```

Note that auxiliary variables are used to refer to local variables of parameterized modules (e.g., `sm_vec[i]` corresponds to `sm[i]`). The same model can be used for invariance checking with lemmas. We defined a lemma called `TVAs` which is comprised of the TVAs used to strengthen the properties.

Technique	Depth	EFA (s)	UQ (s)	BPCT (s)	BFMC (s)	BIF (s)	SAT time (s) (*)	Overall time (s)
-	7	0,11	0,07	0,01	0,23	0,48	0,38	1,46
SG	3	0,24	0,22	0,04	0,42	0,65	0,26	2,52
TVA	3	0,14	0,2	0,3	0,2	1,77	26,22	29,18
Lemma	3	0,15	0,21	0,3	0,2	0,83	9,44 (†)	11,17 + 1,59 (‡)

Table 2: Results of proving consistency of the HD protocol in a 4-node system by using sal-bmc (with Yices SAT solver). Abbreviations: Expanding function application (EFA), unfolding quantifiers (UQ), Boolean property conversion time (BPCT), Boolean flat module conversion (BFMC), building induction formula (BIF) as the time to build the base formula plus the time to build the induction formula. (*) It is the sum of Yices execution time for the base case and the inductive step. (†) The inductive step took 97% of SAT time. (‡) The lemma TVAs was proven in 11,17 seconds at depth 1; the consistency property was proven in 1,59 seconds at depth 3. All data in this row correspond to the proof of the lemma.

4.3 Experiments

Reduced induction depth. We verified the properties of the HD protocol in systems where $N < 6$ by using the models presented in the previous Section. For example, consistency can be proven at depth 3 in the model with strengthened guards. The original model fails to prove the property with the same depth and returns a spurious counterexample. In our setting, the counterexample starts in a state at stage 0 where the fault hypothesis is violated by fault vector. In addition, the local syndromes do not correspond to the fault status of the system. Consequently, since the syndrome matrix and the consistent health vector are computed based on data that cannot appear in the assumed system, consistency is violated. In the model with strengthened guards, the transition at stage 0 is strengthened with the assignments of the fault vector and the local syndromes. This entails that runs similar to the previous one are ruled out in the inductive step of 3-induction and the property can be proven. We remark that strengthened guards might be useful even if the property was not inductive. In fact, as a result of guard strengthening the counterexample resembles more a valid run and appear less "chaotic" than in case of the original model and it helps the user to better understand what goes wrong in the run. We expect that the other techniques are able to prove consistency at the same depth. Indeed, consistency strengthened with the established TVAs as well as the original consistency property using our lemma could be verified by 3-induction. In the following, we discuss how the use of these techniques affect the performance of SAL's BMC model checker.

Performance issues. The experiments of verifying consistency, completeness and correctness with different system size showed similar trends. Table 2 depicts the results of proving consistency for $N = 4$. The same induction depths could be measured for completeness and correctness as well. The experiments were run on a single processor of a double-core Intel Xeon 5130 at 2 GHz with 4 GBytes memory. In this case study, the properties of the protocol could be proven even in the original model by increasing the induction depth to at least 7. Table 2 compares the time of verification with the different approaches. As we can see, the original model is still the fastest (in terms of overall time) in spite of the increased depth. However, in general, increasing the depth might not be feasible or might not result in an inductive invariant. We can also see that the SAL implementation of strengthened guards (SG) outperforms that with TVAs. We speculate that this is because of the ability of strengthening the guards "locally", i.e., within a module based. It is very fast to prove consistency by using the lemma; however, the lemma must be proven separately which takes approximately six times more than the proof of the property[6]. The modified models entail more time to expand functions (EFA) and to unfold quantifying operators (UQ) because the strengthened guards and TVAs contain both function application and quantifiers. The TVA and lemma techniques spend more time converting the formulas into a Boolean representation (BPCT) because they use a modified form of the original property. On the other hand, the transformation of the modules into Boolean formulas (BFMC) takes the most time in the SG model because the strengthened guards augment the transition system of the original model. Our approach needs the shortest time among the depth-reduction techniques to build the k-induction formulas (BIF) and to run the SAT solver (SAT time).

5. CONCLUSION

We have presented an alternate technique for making invariants inductive. The proposed framework is general allowing the invention of customized solutions. We have implemented a prototype solution which strengthens the guards of the transitions based on the the assumption the sequence of the executed transitions is known. We have reported the benefit and the overhead of using this method to model check a system that we develop in a parallel project. We remark that the general technique is not restricted to standard induction-based invariant checking [17] but can possibly be used to improve the proof-quality of any method using k-induction. For example, algorithms using fix-point iteration to prove equivalence of circuits have been augmented with k-induction to provide stronger completeness [3]. The resulting algorithm can be further improved by using strengthened transitions. However, as our technique manipulates the transitions, it is limited to induction schemes that work on the unfolding of the state transition relation.

We believe that the usability of strengthened guards heavily depends on the actual system. Therefore, we plan to apply it for the verification of other systems as well. In future work, we would like to elaborate the full automation of strengthened guards. The main issues are to automatically obtain execution plans and to derive strong strengthened guards

[6]The lemma can be proven by 1-induction; in general, TVAs are guaranteed to be provable via simple induction [13].

without user intervention.

6. ACKNOWLEDGMENTS

The authors would like to thank Marco Serafini and Áron Sisak for the valuable comments.

7. REFERENCES

[1] J. Baumgartner, A. Kuehlmann, and J. A. Abraham. Property Checking via Structural Analysis. In *Proceedings of Conference on Computer Aided Verification (CAV)*, pages 151–165, 2002. Springer-Verlag.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT Procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC)*, pages 317–320, 1999.

[3] P. Bjesse and K. Claessen. SAT-Based Verification without State Space Traversal. In *Proceedings of Formal Methods in Computer-Aided Design* (FMCAD), pages 372–389, 2000.

[4] P. Bokor, M. Serafini, A. Sisak, A. Pataricza, and N. Suri. Sustaining Property Verification of Synchronous Dependable Protocols over Implementation. In *Proceedings of High Assurance Systems Engineering Symposium* (HASE), pages 169–178, 2007.

[5] G. M. Brown and L. Pike. Easy Parameterized Verification of Biphase Mark and 8N1 Protocols. In *Proceedings of International Conference on Tools and the Construction of Algorithms (TACAS)*, pages 58–72., 2006.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] L. de Moura, S. Owre, and N.Shankar. *The SAL Language Manual*. Technical Report, SRI International, 2003.

[8] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Proceedings of Computer-Aided Verification (CAV)*, pages 496–500, 2004.

[9] L. de Moura, H. Rueß, and M. Sorea. Bounded Model Checking and Induction: From Refutation to Verification. In *Proceedings of Computer-Aided Verification conference (CAV)*, pages 14–26, 2003.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of Symposium on Logic in Computer Science*, pages 1–33, 1990.

[11] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communication of ACM*, 17(8):453–455, 1974.

[12] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[13] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[14] K. L. McMillan. Interpolation and SAT-based Model Checking. In *Proceedings of Computer-Aided Verification (CAV)*, pages 1–13, 2003.

[15] J. Rushby. Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification. In *Proceedings of Computer-Aided Verification (CAV)*, pages 508–520, 2000.

[16] M. Serafini, N. Suri, J. Vinter, A. Ademaj, W. Brandstaetter, F. Tagliabó, and J. Koch. A Tunable Add-On Diagnostic Protocol for Time Triggered Systems. In *Proceedings of Dependable Systems and Networks (DSN)*, pages 164–174, 2007.

[17] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties using Induction and a SAT-solver. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, pages 108–125, 2000.

[18] G. Singh and S. K. Shukla. Verifying Compiler based Refinement of Bluespec Specifications using the SPIN Model Checker (To Appear). In *Proceedings of Int. SPIN Workshop on Model Checking*, 2008.

[19] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *Proceedings of Conference on Dependable Systems and Networks (DSN)*, pages 189–198, 2004.

[20] C. J. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Trans. Software Eng.*, 23(11):684–721, 1997.