Incremental verification with mode variable invariants in state machines [⋆]

Temesghen Kahsai[1], Pierre-Loïc Garoche[1,2], Cesare Tinelli[1], and Mike Whalen[3]

[1] The University of Iowa
[2] Onera, the French Aerospace Lab
[3] University of Minnesota

Abstract. We describe two complementary techniques to aid the automatic verification of safety properties of synchronous systems by model checking. A first technique allows the automatic generation of certain inductive invariants for mode variables. Such invariants are crucial in the verification of safety properties in systems with complex modal behavior. A second technique allows the simultaneous verification of multiple properties incrementally. Specifically, the outcome of a property—valid or invalid—is communicated to the user as soon as it is known. Moreover, each property proven valid is used immediately as an invariant in the model checking procedure to aid the verification of the remaining properties. We have implemented these techniques as new options in the KIND model checker. Experimental evidence shows that these two techniques combine synergistically to increase KIND's precision as well as its speed.

1 Introduction

Embedded systems often contain complex modal behavior that describes how the system will interact with its environment. In these systems, the *modes* of the software drive the behavior of the device. In a flight guidance system, these modes cause a particular control algorithm to be chosen; an *approach* mode enables a control algorithm that attempts to land the airplane, while a *go-around* mode enables a controller that attempts to climb the aircraft to a suitable safe altitude. These modes are often designed as state machines or mode transition tables. In addition, embedded systems typically have several parallel mode machines that communicate with one another to define the control state of the system. For instance, in flight guidance, there are separate lateral and vertical modes that manage the lateral and vertical aspects of flight.

Understanding which variables in a system's model represent system modes, and discovering relationships between such variables often determine whether or not a property can be proven about a system. However, such variable, which from now on we will refer informally to as *mode variables*, may not be easily identifiable among all of the system's variables. In addition, once identified, determining correct invariants between different mode variables is non-trivial. As an example of these challenges consider the

[⋆] This work was partially supported by AFOSR grant #AF9550-09-1-0517, NSF grant CNS-1035715, DGA/MRIS/ERE support and FNRAE Cavale project.

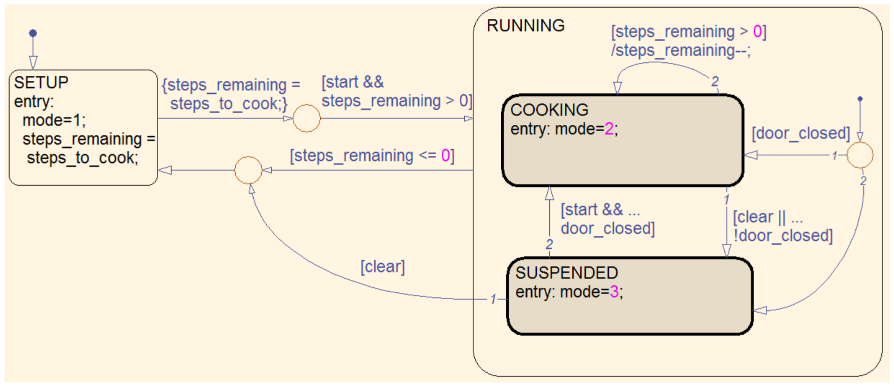


Fig. 1: State machine of a microwave model.

hierarchical state machines (HSMs) described in Figure 1 that illustrates the modal behavior of a microwave.[4] HSMs are used by model-based development notations such as Simulink and SCADE which are becoming widespread for software development in avionics and other industries. In the example, mode information is encoded both explicitly, in the states of the HSM, and implicitly, through the integer variable mode. This sort of hybrid encoding of mode information occurs regularly in industrial models that we have analyzed. An additional complication is that when HSMs are compiled to a lower level modeling language or to code, their state are usually encoded into integer variables that are not immediately distinguishable from other integer variables.

The focus of this paper is on leveraging mode information for *k-induction*-based model checking. In this approach, a prover attempts to prove a safety property of a system inductively by showing for some $k \geq 0$ that (i) the property holds in all states reachable in up to k steps, and (ii) for all sequences of $k + 1$ states along the system's transition relation, the last state satisfies the property whenever all the previous ones do. As with mathematical induction, sometimes safety properties are not strong enough to be provable by k-induction, for any k. In that case, it is helpful to strengthen the induction hypotheses with known invariants properties of the system. We believe that invariants involving mode variables are critical for the success of inductive methods when proving properties of control systems, and we provide initial experimental evidence to support this conjecture.

This paper describes two complementary techniques to aid, more generally, the automatic verification of safety properties of synchronous systems with modal behavior. The first technique, described in Section 3, allows the automatic identification of likely mode variables, and the discovery of invariant relationships among them by adapting an invariant generation method, described in Section 2, we developed in previous work. We heuristically consider as a mode variables any system variable that (i) ranges over a (small) finite set of values and (ii) whose next-state value is determined in part by its current value. We generalize this idea slightly to *mode variable sets* in which strongly-connected variables define a particular system mode. We develop a general invariant generation method to identify implicative relationships between values of mode vari-

[4] We thank Steve Miller and Lucas Wagner at Rockwell Collins for the example.

ables. The second technique, described in Section 4 and motivated by the industrial use of model checkers for the verification of large numbers of safety properties on the same system, allows the simultaneous and incremental verification of multiple properties. It is incremental in the sense that the status of a property—whether it holds in the system or not—is communicated to the user as soon as the checker determines it. Moreover, each property proven to hold is used immediately as an auxiliary invariant to aid the verification of the remaining properties. Our experimental results, described in Section 5 for selected benchmarks, indicate that our two techniques are quite effective in practice, especially in combination. As we show, using them together considerably increases the number of provable safety properties, as well as speeding up the verification process.

Related Work. Automatic invariant generation has been intensively investigated since the 1970s, producing a large body of literature. Manna and Pnueli [13] provide an early compendium of this research and an extensive set of references. In this paper, we focus on discovering invariants related to system modes. The idea of automatically discovering mode machines for hardware is (very briefly) referenced in [7]. The idea of generation of mode-specific invariants for the SCR notation was introduced in [8] and improved in [9] then generalized to LTSs by Damas [2, 1]. This work supports discovery of invariants between a (known) state machine and variables used in guard expressions for transitions of the machine, using syntactic fixpoint algorithm that operates over the state machine graph. Our approach, based on our own previous work invariant discovery [10], is more general; it automatically identifies mode (state machine) variables and uses symbolic analysis to discover a superset of the implications in [9, 2] to include variables not explicitly referenced in the definition of the state machine. On the other hand, the other approaches can quickly determine “local” mode invariants through simple graph traversal algorithms. It may be possible to combine both approaches to improve the scalability of invariant generation. In [4], query checking is used to discover mode invariants. That work uses symbolic methods and is in principle more general (a single query can discover *all* state invariants), but has serious scaling problems. The idea of simultaneous verification of multiple properties is not new [12, e.g.]. Our approach contrasts with previous work by using a parallel architecture that allows the incorporation of invariant generators to enhance the basic verification process.

1.1 Formal Preliminaries

Our work is built on logic-based model checking techniques that phrase reachability problems as entailment problems in a suitable logic for which efficient solvers exist. Relevant examples of such logics are propositional logic or any of the many logics used in SMT. For generality, we consider any of these logic $\mathcal{L}$ (with classical semantics) extending propositional logic, and rely on $\mathcal{L}$'s notion of variable, term, formula, free variable, model, satisfiability in a model, and entailment (which we denote as $\models_{\mathcal{L}}$). If F is a formula of $\mathcal{L}$ and $(x_1, \ldots, x_k)$ a tuple of distinct variables, we write $F[x_1, \ldots, x_k]$ to express that the free variables of F are in $(x_1, \ldots, x_k)$. If $t_1, \ldots, t_k$ are any terms, we write $F[t_1, \ldots, t_k]$ to denote the formula obtained from $F[x_1, \ldots, x_k]$ by simultaneously replacing each occurrence of x_i in F by t_i, for $i = 1, \ldots, k$. We denote finite tuples of elements by letters in bold font, and use comma (,) for tuple concatenation.

Let Q be a set of *states*, a *state space*. We assume some encoding of the state space Q in terms of n-tuples of ground terms in $\mathcal{L}$, for some fixed n.[5] Then, we say that (the encoding of) a state $\boldsymbol{q}$ *satisfies* a formula $F[\boldsymbol{x}]$, where $\boldsymbol{x}$ is an n-tuple of distinct variables, if $F[\boldsymbol{x}]$ is satisfied by every model of $\mathcal{L}$ interpreting $\boldsymbol{x}$ as $\boldsymbol{q}$. This terminology extends to formulas over several n-tuples of free variables in the obvious way.

A *transition system* $\mathcal{S}$ *over* Q is a pair $(\mathcal{S}_\mathrm{I}, S_\mathrm{T})$ where $\mathcal{S}_\mathrm{I} \subseteq Q$ is the set of $\mathcal{S}$'s *initial states*, and $S_\mathrm{T} \subseteq Q \times Q$ is $\mathcal{S}$'s *transition relation*. A state $q \in Q$ is 0-*reachable* if $q \in \mathcal{S}_\mathrm{I}$; it is k-*reachable* with $k > 0$ if it is $(k-1)$-reachable or $(s, q) \in S_\mathrm{T}$ for some $(k-1)$-reachable state s. A state is *($\mathcal{S}$-)reachable* if it is k-reachable for some $k \geq 0$. A *(state) property* is any formula $P[\boldsymbol{x}]$ for some n-tuple $\boldsymbol{x}$ of variables. It is *invariant (for $\mathcal{S}$)* if it is satisfied by all $\mathcal{S}$-reachable states. For automated verification purposes one does not work directly with a transition system $\mathcal{S}$ itself, but with an *encoding* of it in some logic $\mathcal{L}$, namely, a pair $(I[\boldsymbol{x}], T[\boldsymbol{x}, \boldsymbol{x}'])$ of formulas of $\mathcal{L}$, with $\boldsymbol{x}$ and $\boldsymbol{x}'$ both of size n, where

- $I[\boldsymbol{x}]$ is a formula satisfied exactly by the initial states of $\mathcal{S}$;
- $T[\boldsymbol{x}, \boldsymbol{x}']$ is a formula satisfied by two reachable states $\boldsymbol{q}, \boldsymbol{q}'$ iff $(\boldsymbol{q}, \boldsymbol{q}') \in \mathcal{S}_\mathrm{T}$.

k-induction Given an $\mathcal{L}$-encoding $(I[\boldsymbol{x}], T[\boldsymbol{x}, \boldsymbol{y}])$ of some transition system $\mathcal{S}$, one can prove that a property P is invariant for $\mathcal{S}$ by showing that P is k-inductive.

Definition 1. *A state property $P[\boldsymbol{x}]$ is k-inductive (wrt T) for some $k \geq 0$ if*

$$I[\boldsymbol{x}_0] \wedge T[\boldsymbol{x}_0, \boldsymbol{x}_1] \wedge \cdots \wedge T[\boldsymbol{x}_{k-1}, \boldsymbol{x}_k] \models_\mathcal{L} P[\boldsymbol{x}_0] \wedge \cdots \wedge P[\boldsymbol{x}_k] \tag{1}$$

$$T[\boldsymbol{x}_0, \boldsymbol{x}_1] \wedge \cdots \wedge T[\boldsymbol{x}_k, \boldsymbol{x}_{k+1}] \wedge P[\boldsymbol{x}_0] \wedge \cdots \wedge P[\boldsymbol{x}_k] \models_\mathcal{L} P[\boldsymbol{x}_{k+1}] \tag{2}$$

When entailment in $\mathcal{L}$ is decidable and an $\mathcal{L}$*-solver* is available for that, the k-inductiveness of a property P can be established by asking the $\mathcal{L}$-solver to prove both entailments in the definition above for some initial choice of k, retrying with an increasingly larger k until either the base case (1) is shown not to hold or both the base and the induction step (2) are shown to hold. In the second situation, P has been shown to hold for all reachable states, which means it is invariant. In the first situation, P is not invariant and a counterexample path can be generated from a counter-model of (1) above if the $\mathcal{L}$-solver is able to return models.

Since k-inductiveness is a sufficient condition for invariance, the k-induction procedure above is a sound verifier for invariance. The procedure, however, is not complete since there exist systems with invariant properties that are not k-inductive for any k. For those properties, the procedure will keep increasing k indefinitely. A number of improvements are possible to increase the procedure's *precision*, the set of invariant properties it can prove [15, 3, 5]. In particular, if Y is another state property already known to be invariant, one can strengthen the antecedent of the entailment in the induction step (2) by adding (conjunctively) the formula $Y[\boldsymbol{x}_0] \wedge \cdots \wedge Y[\boldsymbol{x}_{k+1}]$ to it. The strengthening is beneficial for eliminating *spurious counter-examples* to the induction step, i.e., counter-models involving unreachable states.

[5] Depending on $\mathcal{L}$, states may be encoded for instance as n-tuples of Boolean constants or as n-tuples of integer constants, and so on.

2 Template-based invariant generation

In previous work [10] we described a general invariant discovery scheme that produces k-inductive invariants for a given transition system $\mathcal{S}$ from a *template* $R[x, y]$, a formula of $\mathcal{L}$ representing a decidable binary relation over one of the system's data types. The discovered invariants are instances $R[s, t]$ of the template generated with terms s, t from a set U of terms over the same n-tuple $\boldsymbol{x}$ of variables. The set U can be constructed heuristically in any number of ways from $\mathcal{S}$ and a given set of properties to be proven invariant for $\mathcal{S}$. In the experiments reported in [10], U included terms occurring in a given $\mathcal{L}$-encoding of $\mathcal{S}$, as well as a few distinguished constants.

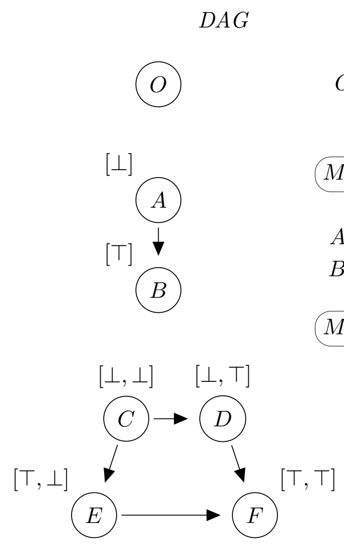
The general scheme relies on the availability of efficient reasoning engines, such as SAT and SMT solvers, for the given logic $\mathcal{L}$, and capitalizes on their ability to quickly generate counter-models. It consists of a simple two-phase procedure, with an optional third phase not discussed here. Given the template R and the term set U, the first phase starts with the (very crude) conjecture that the state property $C[\boldsymbol{x}] = \bigwedge_{s,t \in U} R[s, t]$ is invariant. Then, it uses the $\mathcal{L}$-solver to weaken that conjecture by eliminating from it as many conjuncts $R[s, t]$ as possible that have a counterexample—specifically, all conjuncts falsified by a k-reachable state, for some heuristically determined k. The resulting formula C is passed to the second phase, which attempts to prove C k-inductive by checking that it satisfies the inductive step of k-induction. Any counter-examples there are used, conservatively, to weaken C further by eliminating additional conjuncts until no counter-examples exists. The final formula—the empty conjunction in the worst case—is by construction k-inductive, and so invariant.

The scheme above is impractical in its full generality because the number of instances of R over U can be very large. So we devised two specializations to relations R that are partial orders, one for general posets and one specific to binary posets. These specializations rely on the properties of partial orders to represent the conjunctive conjecture C compactly, and weaken it efficiently. In the following, we briefly illustrate the case of binary posets (see [10] for a more formal treatment and for the general case).

Invariant generation for binary posets. For concreteness, and because it is relevant to our goal of learning invariants on mode variables, let us consider the poset $(\{\bot, \top\}, \rightarrow)$ of the Booleans, with logical implication $\rightarrow$ as the partial order, and with linear integer arithmetic as $\mathcal{L}$. In this case, the instances of R have the form $F \rightarrow G$, where F and G are any arithmetic predicates, i.e., quantifier-free arithmetic formulas.

The invariant generation procedure maintains a directed acyclic graph where each node contains a set of arithmetic predicates and stands for the conjecture that those predicates all imply each other (i.e., are all equivalent) in every reachable state of $\mathcal{S}$. An edge from a node A to a node B in the graph represents the weaker conjecture that the predicates in A imply the predicates in B, again in all reachable states.

The graph starts with a single node containing all the predicates in the candidate set U; it is then updated incrementally using a sequence $(M_1, M_2, \ldots)$ of models of $\mathcal{L}$, each containing a reachable state $\boldsymbol{q}$ that falsifies one of the conjectures in the current graph. Let G_0 be the initial graph and G_i the version of the graph updated after observing model M_i. The graph G_i is updated to G_{i+1} using model M_{i+1} as follows. If M_{i+1} falsifies (the conjecture expressed by) an edge of G_i, the edge is removed; if it falsifies



Conjecture

$$O : \{x = 0 \leftrightarrow x = 1 \leftrightarrow x \neq 0 \leftrightarrow x \neq 1 \leftrightarrow y = 3 \leftrightarrow y = 4 \\ \leftrightarrow y \neq 3 \leftrightarrow y \neq 4 \leftrightarrow x + y \geq 3 \leftrightarrow (2 * x) - y \geq y\}$$

$\boxed{M_1 : x \mapsto 0,\ y \mapsto 4}$

$$A : \{x = 1 \leftrightarrow x \neq 0 \leftrightarrow y = 3 \leftrightarrow y \neq 4 \leftrightarrow (2 * x) - y \geq y\}$$
$$B : \{x = 0 \leftrightarrow x \neq 1 \leftrightarrow y \neq 3 \leftrightarrow y = 4 \leftrightarrow x + y \geq 3\}$$

$\boxed{M_2 : x \mapsto 0,\ y \mapsto 3}$

$$C : \{x = 1 \leftrightarrow x \neq 0 \leftrightarrow (2 * x) - y \geq y\}$$
$$D : \{y = 3 \leftrightarrow y \neq 4\}$$
$$F : \{y \neq 3 \leftrightarrow y = 4\}$$
$$E : \{x = 0 \leftrightarrow x \neq 1 \leftrightarrow x + y \geq 3\}$$

Fig. 2: In each graph, a node stands for a set of predicates that have evaluated to the same Boolean value ($\perp$ or $\top$) in each model considered until them. The predicates in a node are shown, as a double implication chain, in the *Conjecture* column. The list of observed values for the predicates in each node is shown on top of the node.

a node N, then (i) the node is split in two new nodes $N_\perp$ and $N_\top$ connected with an edge from $N_\perp$ and $N_\top$, (ii) N's predicates are assigned to $N_\perp$ and $N_\top$ depending on whether they are respectively falsified or satisfied by M_{i+1}, and (iii) all edges involving N are updated so that the set of conjectures represented by G_{i+1} is consistent with all the models observed so far and weakens the previous set only as little as needed to accommodate M_{i+1}. The procedure is perhaps best illustrated with an example.

Example 1. Consider a system whose $\mathcal{L}$-encoding contains exactly the predicates $x + y \geq 3$ and $(2*x) - y \geq y$, with $x \in [0..1]$ and $y \in [3..4]$, say. In the invariant generation procedure in [10], the set U would be just $\{x + y \geq 3, (2 * x) - y \geq y\}$. In the version we discuss here, if x and y are identified as mode variables of interest (see later) U is augmented with the predicates from the following set

$$V := \{x = 0, x = 1, y = 3, y = 4\} \cup \{x \neq 0, x \neq 1, y \neq 3, y \neq 4\} .$$

Figure 2 shows how the graph evolves with a sample sequence of two models. The procedure starts with the implication graph consisting of the node O, conjecturing that all predicates in U are equivalent. Nodes A and B are the result of splitting O. Nodes C and D are the result of splitting A, and node E and F of splitting B. ■

The addition to U of predicates like those in the set V in Example 1 allows our procedure to discover, among others, invariants of the form $x = a \rightarrow y = b$ where x and y are mode variables and a and b are specific values in their range. Together with range constraints, negative predicates of the form $y \neq b$, allow the procedure to discover, in effect, also invariants of the form $x = a \rightarrow \bigvee_{i \in I} y = b_i$ where $[b_1..b_n]$ is y's range and $I \subseteq 1, \ldots, n$.[6] We will call these two kinds of invariants *mode invariants*.

3 Identifying mode variables

In this section we propose a technique to identify a relatively tight number of system variables as mode variables and a set of predicates on them to be used to produce mode invariants as described in the previous section. The overall goal is to capture with these invariants enough mode information about a software system under analysis—or, more accurately, about its encoding as a transition system in some logic $\mathcal{L}$.

The logic $\mathcal{L}$ used here will be the two-sorted logic consisting of the quantifier-free fragment of (mixed integer and real) linear arithmetic.

3.1 State machines in synchronous models

Embedded systems, controllers for instances, are usually modeled as a set of synchronous dataflow computations governed by an overall mode logic. In aircraft control command, the mode logic could be a finite state machine iterating through the phases: taxi, take-off, flying, landing. For a car cruise controller, it could be a state machine describing how the controller engages and disengages depending on a number of parameters and actions. As mentioned in the introduction, when encoding these models as transition systems for verification purposes, the state machine expressing the original system's mode logic is often encoded with the introduction of *mode variables* to model the mode logic's finite state machine. These are variables over an enumeration type or, more often, Boolean variables or variables over a finite integer range.

While this approach is rather general, it has the disadvantage that the structure of state machine gets lost in the translation. This has important consequences for verification methods based on inductive arguments, such as k-induction, because the logical encoding ends up creating a state space with states that do not correspond to any state of the original state machine, and so are unreachable by the resulting transition system. These states are problematic because they typically lead to spurious counter-examples for the inductive step of the verification process.

To illustrate the problem with an example, consider again the microwave model of Figure 1, but without the variable mode. Consider then a layered encoding of the model into a transition system where a mode variable $top \in [1..2]$ represents the top states SETUP and RUNNING, with $top = 1$ for the first and $top = 2$ for the second, and a mode variable $running \in [0..2]$ represents the running state, with 0 meaning not running, 1 meaning SUSPENDED and 2 COOKING. The state space of this transition system contains the unreachable states $\{top \mapsto 1, run \mapsto 1\}$ and $\{top \mapsto 1, run \mapsto 2\}$ which may cause problems during induction. Those states can be ruled out during the verification process if $(top = 1) \leftrightarrow (running = 0)$ is discovered to be an invariant for the system.

[6] The reason is that such an invariant is equivalent to $\bigwedge_{j \in [b_1..b_n] \setminus I} (x = a \rightarrow y \neq b_j)$.

$$\begin{array}{rcl} T_1 & := & x' = z \\ & \wedge & y' = \mathsf{if}\ c_1'\ \mathsf{then}\ 2\ \mathsf{else} \\ & & \qquad \mathsf{if}\ c_2'\ \mathsf{then}\ 1\ \mathsf{else}\ x' \\ & \wedge & z' = \mathsf{if}\ c_3'\ \mathsf{then}\ 0\ \mathsf{else}\ y' \\ & \wedge & x, y, z \in [0..2] \end{array} \qquad \begin{array}{rcl} T_2 & := & a' = z \ \wedge\ x' = b \\ & \wedge & b' = \mathsf{if}\ c_4' \wedge a' = 2\ \mathsf{then}\ 1\ \mathsf{else}\ \mathsf{if}\ c_5'\ \mathsf{then}\ y'\ \mathsf{else}\ 2 \\ & \wedge & y' = \mathsf{if}\ c_1'\ \mathsf{then}\ 2\ \mathsf{else}\ \mathsf{if}\ c_2'\ \mathsf{then}\ 1\ \mathsf{else}\ x' \\ & \wedge & z' = \mathsf{if}\ c_3'\ \mathsf{then}\ 0\ \mathsf{else}\ y' \\ & \wedge & a, b, x, y, z \in [0..2] \end{array}$$

Fig. 3: Transition relations over integer and Boolean variables. The latter are unconstrained just for simplicity.

3.2 Selecting mode variables

To generate mode invariants for a transition system $\mathcal{S}$ it is necessary to identify its mode variables in the first place. In the absence of explicit user-provided information, a possibility is to perform interval analysis on $\mathcal{S}$ to uncover variables that have a finite domain in all reachable states, and treat all such variables as mode variables. In general, examples of finite domain variables would be Boolean variables, enumeration type variables, and integer variables over a finite range. Then, one can strengthen $\mathcal{S}$'s transition relation as needed with the discovered finite domain constraints on those variables, and apply the invariant generation technique presented in Section 2 based on a set of predicates that contains all equations of the form $x = v$ and their negation, for each finite domain variable x and value v in its domain.

One problem with this approach is that it does not to scale well with respect to the number of finite domain variables or the size of their domains. Furthermore, many finite domain variables are uninteresting from a mode invariant generation perspective because they simply store intermediate values in the system's computation. For example, consider the two transition relations T_1 and T_2 in Figure 3, already strengthened with finite domain constraints for some of their variables. While artificial and somewhat contrived, they illustrate a common situation in which several of the finite domain variables can be ignored for depending functionally on other variables.

It is easy to see that in T_1 the values of x', y' and z'—i.e., the *next-state values* of x, y and z—are all determined by the value of the tuple (z, c_1', c_2', c_3'). A closer look reveals that they are also all determined by the value of (x, c_1', c_2', c_3'). As we will argue later, this suggests that it is enough to consider just z or just x as a mode variable for invariant generation purposes. In contrast, it would not be advantageous to consider just y because the next-state value of x is not determined by $(y, c_1', \ldots, c_3')$. In T_2, no tuple consisting of the Boolean variables and just one of the integer variables determines the next-state value of all the other variables. However, a tuple made of b and z and the Boolean variables will do. We formalize this intuition in the following and discuss a mode variable selection heuristics based on it.

Definition 2. *Let $F[\boldsymbol{z}]$ be a formula in $\mathcal{L}$ and let $F^{\mathcal{L}}$ be the relation denoted by F in $\mathcal{L}$. A variable y in $\boldsymbol{z}$* depends *(in F) on a tuple $\boldsymbol{x}$ of variables from $\boldsymbol{z}$, if the projection $\pi_{\boldsymbol{x},y}(F^{\mathcal{L}})$ of $F^{\mathcal{L}}$ over $\boldsymbol{x}, y$, in the sense of relational algebra, is functional; that is, if $\pi_{\boldsymbol{x},y}(F^{\mathcal{L}})$ contains no two distinct tuples of the form $(\boldsymbol{v}, u_1), (\boldsymbol{v}, u_2)$; the variable y* strictly *depends on $\boldsymbol{x}$ if, additionally, it depends on no proper subtuple of $\boldsymbol{x}$.*

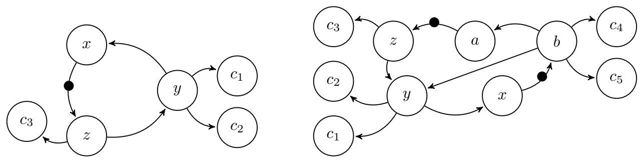


Fig. 4: Dependency graphs for the formulas T_1 and T_2, respectively, from Figure 3.

Now, let's consider a formula $T[\boldsymbol{x}, \boldsymbol{x}']$ of $\mathcal{L}$ encoding the transition relation of some system $\mathcal{S}$. Suppose we are given a mapping dep from each variable y' of $\boldsymbol{x}'$ to a tuple of variables (of F) that y' strictly depends on. This mapping induces a directed labelled multigraph (V, E), a *dependency graph for* T, where

$$V = \boldsymbol{x} \qquad\qquad E = \{y \longrightarrow z \mid z' \in \mathrm{dep}(y')\} \cup \{y \mathrel{-\!\bullet\!\!\rightarrow} z \mid z \in \mathrm{dep}(y')\}\ .$$

Intuitively, there is an edge $\longrightarrow$ in the graph between y and z iff the next-state value of y depends on the next-state value of z, and there is an edge $\mathrel{-\!\bullet\!\!\rightarrow}$ between them iff the next-state value of y depends on the current-state value of z. For the transition relation T_1 in Figure 3, a suitable mapping dep would be $\{x' \mapsto (z), y' \mapsto (c_1', c_2', x'), z' \mapsto (c_3', y')\}$. That mapping and its analogous for T_2 induce the multigraphs depicted in Figure 4.

We assume here that, for a given transition relation formula T, it is possible to compute from it a mapping dep, and hence its induced dependency graph. The ease, or in fact the possibility, of doing this automatically depends in general on T's format. In our experiments, where transition relation formulas are generated from system models written in Lustre [6], the process is straightforward because there each variable is given an explicit equational definition, as in the formulas of Figure 3.

Definition 3. *Let $G = (V, E)$ be a dependency graph and let C be a strongly connected component (SCC) of G.*[7] *The* base *of C is the set*

$$\{y \in C \mid x \mathrel{-\!\bullet\!\!\rightarrow} y \in E \textit{ for some } x, y \in C\} \tag{3}$$

if this set is non-empty; otherwise it is C itself. A variable is a base variable *of G if it is in the base of one of G's SCCs; it is a* stateful *base variable if it is in a base like (3).*

The SCCs of the left-hand graph of Figure 4 are $\{c_1\}, \{c_2\}, \{c_3\}$, and $\{x, y, z\}$; their respective bases are $\{c_1\}, \{c_2\}, \{c_3\}$, and $\{z\}$. The SCCs of the other graph are $\{c_1\}, \ldots, \{c_5\}, \{a, b, c, x, y, z\}$; their respective bases are are $\{c_1\}, \ldots, \{c_5\}$, $\{b, z\}$.

It is not difficult to show that the following holds.

Proposition 1. *Let $G = (V, E)$ be a dependency graph for a transition relation T, and let $N = \{x_1, \ldots, x_k\}$ be the union of all the base variables of G. Then, every variable in $V \setminus N$ depends on $(x_1, \ldots, x_k)$ in T.*

The proposition above suggests that for invariant generation purposes it is enough to restrict attention to base variables only since they determine the values of all the other

[7] With respect to paths built with any of the two edge types of G.

variables. Therefore, it is enough to constraint their values only. In fact, one can go even further and ignore any invariants containing only non-stateful base variables. For instance, invariants over just the base variables $c_1, \ldots, c_3$ and $c_1, \ldots, c_5$ in the graphs of Figure 4. The reason is that the current state values of such variables constraints only current state values of other variables, but no next state values. This means that invariants containing only such variables will be entailed by the transition relation. Such invariants are useless for induction because they do not not strengthen the transition relation.[8] In this work we take a more draconian approach and simply discard *all* non-stateful base variables, to reduce as much as possible the number of predicates $x = v$ fed to our invariant discovery procedure. Of course, we also discard all stateful base variables that do not (or that we cannot determine to have) a finite domain.

The rationale behind this selection heuristics is that each independently defined state machines in the original system model—in particular, submachines of a hierarchical state machine—typically end up generating separate SCCs over mode variables in the dependency graph. Our conjecture is that enough useful invariants about these submachines and their relationships, are captured by considering just the finite-domain stateful base variables of each SCC.

A variable selection procedure. To summarize, to limit the number of variables used for mode invariant generation for a transition relation formula T we use a procedure that (i) computes a dependency graph $G = (V, E)$ for T, (ii) identifies G's strongly connected components, and (iii) collects and returns all and only the finite-domain stateful base variables of these components.

4 Multi-property incremental verification

In this section we present a technique to verify simultaneously and incrementally multiple safety properties. Its relevance in this work is that it combines synergistically with the invariant generation techniques described in the previous sections.

Given a transition system encoding (I, T) and a list of properties $P^1, \ldots, P^n$ all to be checked for invariance, there are two possible ways of doing that with k-induction. One is to check each property individually. This is however time consuming and not very effective because a conjunction of formulas is usually easier to prove by induction that its individual conjuncts. Another way then is to check the property $P = P^1 \wedge \cdots \wedge P^n$. But this has its drawbacks as well. To start, if even just one of the individual properties fails to be invariant so does the whole P. However, even when P is indeed invariant, it is often the case that its individual constituents are k-inductive for different values of k. So the k-induction procedure has to wait until the largest of these values is reached before succeeding. In the worst case, one of the individual properties may not be k-inductive for any k, forcing the basic k-induction procedure to diverge.

Our solution to the problems above is to work with all properties at the same time but also keep track in each iteration of the k induction loop of the current status of each property P^i. In the base case, all properties that are falsified for a particular k are

[8] Note, however, that they might nevertheless be useful to *speed up* queries to an $\mathcal{L}$-solver, the way auxiliary (deductive) lemmas generally do.

```
proc base_proc ≡
  $k := 0$; $props := \{P^1, \ldots, P^n\}$;  for $P \in props$  $P.\mathsf{level} := \infty$
  while $(props \neq \emptyset)$
      $model := \mathsf{SAT}(T_0 \wedge \cdots \wedge T_k \wedge \neg \bigwedge_{P \in props}(P_0 \wedge \cdots \wedge P_k))$
      if $(model = \mathsf{Unsat})$ then  $k := k + 1$
      else
        $invalid := \mathsf{filter\_sat}(model, props)$
        $\mathsf{send}(\mathsf{INVALID}(invalid), \mathsf{ind\_proc})$
        $props := props \setminus invalid$
        print_out  $invalid$
      if $\mathsf{receive}(\mathsf{VALID}(possibly\_valid, k'),\ \mathsf{ind\_proc})$ then
        for $P \in possibly\_valid$  $P.\mathsf{level} := k'$
        $valid := \{P \in props \mid P.\mathsf{level} \leq k\}$
        $props := props \setminus valid$
        if $props \neq \emptyset$ then  $\mathsf{send}(\mathsf{INVAR}(valid), \mathsf{ind\_proc})$
        print_out  $valid$
  $\mathsf{send}(\mathsf{STOP}, [\mathsf{ind\_proc}, \mathsf{inv\_gen\_proc}])$
```

Fig. 5: **Base step process**. For each i, T_i abbreviates $I[\boldsymbol{x}_i]$ if $i = 0$ and $T[\boldsymbol{x}_{i-1}, \boldsymbol{x}_i]$ otherwise. P_i abbreviates $P[\boldsymbol{x}_i]$.

removed from consideration before increasing the value of k. In the induction step, all properties that are validated for a particular k (see later for details on how we check this) are also removed from the list of properties to be checked but immediately added back as invariants, to aid the verification of the remaining ones.

Our incremental approach builds on the parallel k-induction-based model checking architecture we developed in recent work [11]. That architecture is designed to minimize synchronization delays and facilitate the incorporation of concurrent invariant generators, and has the following basic structure:

$$\mathsf{base\_proc} \;||\; \mathsf{ind\_proc} \;||\; \mathsf{inv\_gen\_proc}_i$$

The base and the inductive step of k-induction execute concurrently respectively in the base_proc and the ind_proc process, as do one or more independent processes $\mathsf{inv\_gen\_proc}_i$ that incrementally generate auxiliary invariants for the system being verified. These invariants are fed to the k-induction loop as soon as they are produced and used to strengthen the induction hypothesis. The processes communicate with one another by asynchronous messages passing, with non-blocking send and receive operations relying on message queues. The operation $\mathsf{receive}(pat, source)$ matches the pattern pat with a message from process $source$, if any; it returns true if there is a message and the matching succeeds, and returns false otherwise. Some more details on each process are described below, assuming for simplicity just one invariant generator.

Base case process. Figure 5 shows the pseudo-code for base_proc. Its main task is to partition incrementally the initial set of properties, the initial value of $props$, into *valid* (i.e., invariant) properties and *invalid* (i.e., non-invariant) ones.

The process checks the entailment in Case (1) of Definition 1 for increasing values of k starting from 0. The function SAT implements the $\mathcal{L}$-solver. It takes a formula F over n states and returns either unsat or a model $model$ of F, i.e., a sequence of n

```
proc ind_proc ≡
  $k := 0;\ props := \{P^1, \ldots, P^n\};\ invs := \emptyset$
  while ($props \neq \emptyset$)
      assert($T_{k+1} \wedge \bigwedge_{Y \in invs} Y_{k+1} \wedge \bigwedge_{P \in props} P_k$)
      if entailed($\bigwedge_{P \in props} P_{k+1}$) then send(VALID($props, k$), base_proc); exit
      else
        $possibly\_valid :=$ recheck_validity($props, k$)
        send(VALID($possibly\_valid, k$), base_proc)
        $props := props \setminus possibly\_valid$
      if receive($msg$, _) then
        match $msg$ with
            STOP → exit
          | INVALID($invalid$) → $props := props \setminus invalid$
          | INVAR($new\_invs$)→ for $i = 0$ to $k+1$ assert($\bigwedge_{Y \in new\_invs} Y_i$)
                                $invs := invs \cup new\_invs$
      $k := k + 1$
```

Fig. 6: **Inductive step process.** For each i, Y_i abbreviates $Y[\boldsymbol{x}_i]$.

states that satisfies F. The function filter_sat returns the set of properties in $props$ that are falsified by one of the states in $model$. Those properties are definitely invalid. They are both printed for the user and sent the inductive step process, and then removed from the current set $props$ of properties. Note that the counter k left unchanged as long as the solver keeps finding counter-models for some of the current properties.

Before repeating the main loop the process checks its message queue; a message from ind_proc stating that it has successfully proven the inductive step (2) of Definition 1 for a some k' and a subset $possibly\_valid$ of the input properties. The value k' need not be the same as k since the two processes increase their own induction level independently. As a consequence, the base_proc first annotates each property in $possibly\_valid$ with k', storing it in the level field of the property. Then it collects in $valid$ all properties from $props$ whose level is at that point smaller or equal to the current k. Each property P in $valid$ has been cooperatively shown by the two processes to be (P.level)-inductive. So it is removed from the list of properties to be proven and sent back to ind_proc to be used as an invariant, provided there are still properties to be proven. The process terminates when $props$ becomes empty, sending a termination signal to the other processes as well.

Inductive step process. Pseudo-code for this process is provided in Figure 6. There we assume a stateful $\mathcal{L}$-solver that allows one to assert formulas (with the assert procedure) and then check (with the entailed Boolean function) whether the current set of asserted formulas entails a given one.

The process checks the inductive step entailment for increasing values of k. However, it strengthens the induction hypothesis with any invariants at its disposal (in $invs$). If the entailment holds for the current k and set $props$ of properties, they are both sent to the base case process, and the inductive process terminates. As discussed earlier, base_proc will confirm their individual invariance, or not, by checking that they have no counter-examples of length up to k. If the entailment fails, the process passes the properties to the auxiliary function recheck_validity which (using a separate copy of the

$\mathcal{L}$-solver) computes the largest subset of $props$ for which the entailment test succeeds. This set is sent to base_proc as in the previous case, and removed from $props$.

The remaining properties are rechecked for an increased value of k. Before proceeding, however, the process checks its message queue. If it sees a message (from base_proc) with a set of properties found to be invalid, it removes them from $props$. If it sees a message from an invariant generation process, providing a set of auxiliary invariants, or from base_proc, providing a set of properties confirmed to be valid and so usable as invariants, it asserts all those invariants for all steps from 0 to $k + 1$ and then adds them to the current invariant set $invs$. The process terminates if it sees a termination message from base_proc.

Incremental invariant generator. This process can be any incremental invariant generator for the given transition system. It is supposed to keep sending any newly discovered invariants to the induction step process until it can generate no more, or it receives a termination message from the base case process. In our current implementation, we have one such process that essentially implements the general template-based invariant discovery procedure seen in Section 2. The process is composed of three main modules: the *Candidate generator*, which constructs the initial set C of candidate invariants from predefined templates, the *Int invariant generator*, which produces from C invariants of the form $s \leq t$ where s and t are integer terms, and the *Bool invariant generator*, which produces invariants of the forms $F \rightarrow G$ as discussed in Section 3.

5 Experimental results

To evaluate experimentally the techniques presented in the previous sections, we have implemented it as new options in our k-induction-based model checker KIND.[9] KIND can simultaneously check multiple invariant properties of programs written in an idealized version of the specification/programming language Lustre [6].[10] The underlying logic of KIND is a quantifier-free logic that includes both propositional logic and mixed real-integer linear arithmetic. Lustre programs can be readily encoded as transition systems in this logic [5]. KIND uses the SMT solvers CVC3 and Yices, in alternative, as satisfiability solvers for this logic. The version discussed here is based on the incremental parallel architecture discussed in the previous section.

The experiments discussed below were run, using Yices version 1.0.9 as the background solver, on a 12-core 2.10 GHz AMD Opteron machine under Ubuntu 11.10. The experiments used benchmark derived from the following problem.

NASA Docking Approach Example. This is a complex hierarchical problem that describes the approach behavior of the Space Shuttle when docking with the International Space Station [14]. As the shuttle approaches the ISS it goes through several operational modes related to how the shuttle is to orient itself for capture, dock with the ISS, and capture the ISS docking latch, among several other operational modes. The model describing this behavior is quite intricate and consists of a hierarchical and parallel state machine with three levels of hierarchy and multiple parallel state machines, including a

[9] Tools and experimental data can be found at http://clc.cs.uiowa.edu/Kind.

[10] The idealization consists in treating Lustre's numerical types as infinite precision types.

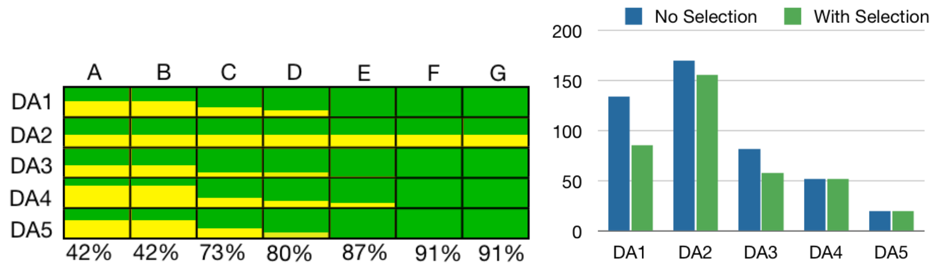


Fig. 7: The left graph illustrates the distribution of solved and unsolved properties for the different benchmarks, DA1 ... DA5, using configurations A though G for KIND. Darker areas indicate the portion of solved properties. The right graph indicates the number of variables considered for mode invariant generation before and after applying the selection procedure from Section 3 to the 5 benchmarks.

total of 64 states. For the purposes of this experiment, we created five reduced versions of the docking approach model in which we replaced one of the complex hierarchical states with a simple state that approximates its behavior. This allows us to examine the behavior of the invariant generation over a range of state machine models with different characteristics (the hierarchical states vary substantially in size). Note that it also causes some of the original properties to be violated.

We ran KIND on the five problems above in different configurations: (A) *single-prop*, *no invars*; (B) *multi-prop*, *no invars*; (C) *single-prop*, *no mode invars*; (D) *multi-prop*, *no mode invars*; (E) *single-prop*, *mode invars*; (F) *multi-prop*, *mode invars*; (G) *multi-prop*, *selected mode invars*. In the *single-prop* configurations, each property was checked individually; in *multi-prop* configurations the properties were checked together incrementally as discussed in Section 4. In *no invars*, no invariants were generated at all. In *no mode invars*, invariants were generated, but no mode invariants. In *mode invars*, invariants included mode invariants generated for *all* finite domain variables. In the *selected mode invars* configuration, invariants included mode invariants generated only for those variables selected by the procedure discussed in Section 3.

Precision results. The first graph of Figure 7 summarizes the precision achieved by KIND under the configurations above. In cases A and B, KIND is able to solve 42% of all the properties without relying on auxiliary invariants. The percentage of solved properties goes up to 73%, 80% and 87% in cases C, D and E, respectively, illustrating the effectiveness of invariant generation and of incremental multi-property verification. In particular, general invariants (case C) increase precision by 31 percentage points over configurations A and B. The further addition of mode invariants in the single property case increases precision by 14 more points (from C to E). Going from single to incremental multi-property verification but without mode invariants (from C to D) increases precision by 7 points. Finally, the combination of multi-property verification and mode invariants does noticeably better than each of them alone (91% vs 80% and 87%).

Runtime results. As we conjectured, reducing the number of variables to generate mode invariants using our variable selection procedure reduces runtimes in general without impacting precision. In particular, in case F, KIND can to solve all the valid properties in a total time of 15s; in case G, such value goes down to 14.3s. As shown by the right-hand side graph of Figure 7, our selection procedure reduces the number of mode variables to consider in problems DA1, DA2 and DA3—although not in DA4 and DA5, perhaps because of their small number there. As a result, the total time for the first three benchmarks goes respectively from 4089ms, 57ms and 6025ms before the selection of mode variables (case F) to 3728ms, 33ms and 5752ms after (case G).

6 Conclusion

We have presented two complementary techniques for the verification of safety properties in synchronous systems with complex modal behavior. A first technique allows the automatic generation of certain inductive invariants for system variables identified heuristically as containing mode information. A second technique allows the simultaneous verification of multiple properties in an incremental fashion. The synergy between these two techniques allowed us to verify safety properties of complex systems like the NASA docking benchmarks.

References

1. C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *SE, IEEE Transactions on*, 31(12):1056–1073, 2005.
2. C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde. Analyzing critical process models through behavior model synthesis. In *ICSE '09*, pages 441–451. IEEE, 2009.
3. L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *CAV 2003*, volume 2725 of *LNCS*. Springer, 2003.
4. A. Gurfinkel, M. Chechik, and B. Devereux. Temporal logic query checking: a tool for model exploration. *SE, IEEE Transactions on*, 29(10):898 – 914, oct. 2003.
5. G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD '08*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.
6. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
7. S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *DAC '05*, pages 775 – 778, june 2005.
8. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 23:56–69, November 1998.
9. R. Jeffords and C. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *RE 2011*, pages 182 –191, 2001.
10. T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *NFM 2011*, volume 6617 of *LNCS*, pages 192–207. Springer, 2011.
11. T. Kahsai and C. Tinelli. PKIND: a parallel k-induction based model checker. In *PDMC 2011*, EPTCS 72, pages 55–62, 2011.
12. Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna. Simultaneous SAT-based model checking of safety properties. In *Haifa Verification Conference'05*, pages 56–75, 2005.
13. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
14. M. Sampson and V. Derevenko. Interface definition document (IDD) for international space station (ISS) visiting vehicles (VVs). Technical report, NASA, 2000.
15. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00*, pages 108–125. Springer, 2000.