Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Incremental design-space model checking via reusable reachable state approximations

Rohit Dureja · Kristin Yvonne Rozier

Received: date / Accepted: date

Abstract The design of safety-critical systems often requires *design space exploration*: comparing several system models that differ in terms of design choices, capabilities, and implementations. Model checking can compare different models in such a set, however, it is continuously challenged by the state space explosion problem. Therefore, learning and reusing information from solving related models becomes very important for future checking efforts. For example, reusing variable ordering in BDD-based model checking leads to substantial performance improvement. In this paper, we present a SAT-based algorithm for checking a set of models. Our algorithm, FuseIC3, extends IC3 to minimize time spent in exploring the common state space between related models. Specifically, FuseIC3 accumulates artifacts from the sequence of over-approximated reachable states, called *frames*, from earlier runs when checking new models, albeit, after careful repair. It uses bidirectional reachability; forward reachability to repair frames, and IC3-type backward reachability to block predecessors to bad states. We extensively evaluate FuseIC3 over a large collection of challenging benchmarks. FuseIC3 is on-average up to 5.48× (median 1.75×) faster than checking each model individually, and up to 3.67× (median 1.72×) faster than the state-of-the-art incremental IC3 algorithm. Moreover, we evaluate the performance improvement of FuseIC3 by smarter ordering of models and property grouping using a linear-time hashing approach.

Keywords Model Checking · Design Space Exploration · Reachability Analysis · Incremental Verification · Air Traffic Control

A preliminary version of this work was reported in "Dureja, R., Rozier, K.Y.: *FuseIC3: An Algorithm for Checking Large Design Spaces.* In: Proceedings of the 17th International Conference on Formal Methods in Computer-Aided Design (FMCAD '17), IEEE/ACM, Vienna, Austria, October 2–6, 2017, pp. 164–171."

Rohit Dureja
Iowa State University, Ames, IA, USA
E-mail: dureja@iastate.edu

Kristin Yvonne Rozier
Iowa State University, Ames, IA, USA
E-mail: kyrozier@iastate.edu

1 Introduction

In the early phases of design, there are several models of the system under development constituting a *design space* [3, 29, 34]. Each model in such a set is a valid design of the system, and the different models differ in terms of core capabilities, assumptions, component implementations, or configurations. We may need to evaluate the different design choices, or to analyze a future version against previous ones in the product line. Model checking can be used to aid system development via a thorough comparison of the set of models. Each model in the set is checked one-by-one against a set of properties representing requirements. However, for large and complex design spaces, such an approach can be inefficient or even fail to scale to handle the combinatorial size of the design space. Nevertheless, model checking remains the most widely used method in industry when dealing with such systems [8, 29, 32, 34, 36].

We assume that different models in the design space have overlapping reachable states, and the models are checked sequentially. In a typical scenario, a model-checking algorithm doesn't take advantage of this information and ends up re-verifying "already explored" state spaces across models. For large models this can be extremely wasteful as every model-checking run re-explores already known reachable states. The problem becomes acute when model differences are small, or when changes in the models are outside the cone-of-influence of the property being checked, i.e., although the reachable states in the models vary, none of them are bad. Therefore, as the number of models grows, learning and reusing information from solving related models becomes very important for future checking efforts.

We present an algorithm that automatically reuses information from earlier model-checking runs to minimize the time spent in exploring the symbolic state space in common between related models. The algorithm, FuseIC3, is an extension to one of the fastest bit-level verification methods, IC3 [9], also known as *property directed reachability* (PDR) [27]. Given a set of models and a safety property, FuseIC3 sequentially checks each model by reusing information: reachable state approximations, counterexamples (cex), and invariants, learned in earlier runs to reduce the set's total checking time. When the difference between two subsequent models is small or beyond the cone-of-influence of the property, the invariant or counterexample from the earlier model may be directly used to verify the current model. Otherwise, FuseIC3 uses reachable state approximations as inputs to IC3 to only explore undiscovered reachable states in the current model. In the former, verification completes almost instantly, while in the latter, significant time is saved. When the stored information cannot be used directly, FuseIC3 repairs and patches it using an efficient SAT-based algorithm. The repair algorithm is the main strength of FuseIC3, and uses features present in modern SAT solvers. It adds "just enough" extra information to the saved reachable states to enable reuse. We demonstrate the industrial scalability of FuseIC3 on a large set of 1,620 real-life models for the NASA NextGen air traffic control system[29, 34], selected benchmarks from HWMCC 2015[6], and a set of seven models for the Boeing AIR6110 wheel braking system[8]. Our experiments evaluate FuseIC3 along two dimensions; checking all models with the same property, and checking each model with several properties. Lastly, we evaluate the impact of smarter model ordering and property grouping on the performance of FuseIC3.

1.1 Related Work

The idea of reusing model-checking information, like variable orderings, between runs has been extensively used in BDD-based model checking leading to substantial performance improvement[41, 4]. Similarly, intermediate SAT solver clauses and interpolants are reused in bounded model checking[33, 38]. Reusing learned invariants in IC3 speeds up convergence of the algorithm[15]. These techniques enable efficient incremental model checking and are useful in *regression verification*[42] and *coverage computation*[16]. FuseIC3 is an incremental algorithm and is applicable in these scenarios.

Product line verification techniques, e.g., with Software Product Lines (SPL), also verify models describing large design spaces[23, 22, 20, 5]. The several *instances* of feature transition systems (FTS)[21] describe a set of models. FuseIC3 relaxes this requirement and can be used to check models that cannot be combined into a FTS. It outputs model-checking results for every model-property pair in the design space without dependence on any *feature*. Nevertheless, SPL instances can be checked using FuseIC3. Large design spaces can also be generated by models that are parametric over a set of inputs [26]. *Parameter synthesis*[17] can generate the many models in a design space that can be checked using FuseIC3. The parameterized model-checking problem[28] deals with infinite homogeneous models. In our case, the models in a set are heterogeneous and finite. The paradigm of "just-assume" (JA) verification [30] provides a semantic approach to derive a debugging set of properties to fix before verifying others, implying a property; our incremental algorithm can speed up JA-verification by reusing information across different property checking runs.

The work most closely related to ours is a state-of-the-art algorithm for incremental verification of hardware[15]. It extends IC3 to reuse the generated proof, or counterexample, in future checker runs. It extracts minimal inductive subclauses from an earlier invariant with respect to the current model. In our analysis, we compare FuseIC3 with this algorithm, and show that with the same amount of information storage, FuseIC3 is faster when checking large design spaces.

1.2 Contributions

We present a query-efficient SAT-based algorithm for checking large design spaces, and incremental verification. Our contributions are summarized as follows:

1. Fully automated, general, and scalable algorithm for checking design spaces.
2. Systematic methodology to reuse reachable state approximations to guide bad-state search in IC3. Our novel procedure to repair state approximations requires little computation effort and is of individual interest.
3. Extensive experimental analysis using real-life benchmarks and comparison with existing state-of-the-art incremental algorithm for IC3.
4. We make all reproducibility artifacts and source code publicly available [1].

This article is an expanded version of a peer-reviewed conference paper presented at FMCAD 2017 [25] and extends it along the following new contributions:

5. More detailed explanations, and theorems and proofs supporting the correctness of the several sub-algorithms (Section 3.3).

[1] `http://temporallogic.org/research/FMCAD17/`

6. Overview of locality-sensitive hashing [1] techniques to mine model specifications expressed as And-Inverter-Graph circuits (Section 4.1).
7. Heuristics to organize the design space, i.e., partially order models in a set and group properties based on similarity, to enable higher reuse of reachable state approximations and improve performance (Sections 4.2 and 4.3).
8. Detailed experimental results incorporating model ordering and property grouping heuristics with an in-depth analysis (Section 5).

1.3 Structure

Section 2 details background information, overviews the typical IC3 algorithm, and defines the notation used throughout the paper. Section 3 presents the FuseIC3 algorithm. Locality-sensitive hashing and its usage as a heuristic to measure similarity is detailed in Section 4. A large-scale experimental evaluation forms Section 5, and Section 6 concludes by highlighting future work and possible extensions.

2 Preliminaries

2.1 Definitions

Definition 1 A Boolean transition system, or model M is represented using the tuple (Σ, Q, Q_0, δ) where

1. Σ is a finite set of atomic propositions or state variables,
2. Q is a finite set of states,
3. $Q_0 \subseteq Q$ is the set of initial states,
4. $\delta : Q \times Q$ is the transition relation.

A sequence of states $\pi = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ is a *path* in M if s_0 is an initial state, each $s_i \in Q$ for $0 \leq i \leq n$, and for $0 < i < n$, $(s_i, s_{i+1}) \in \delta$, i.e., there is a valid transition from state s_i to state s_{i+1}. A state t in a model is *reachable* iff there exists a path such that $s_n = t$.

Definition 2 A *safety property* is a Boolean formula φ over Σ.

A transition system M is SAFE, represented as $M \models \varphi$, iff φ holds in all reachable states of M. Similarly, M is UNSAFE, represented as $M \not\models \varphi$, iff φ does not hold in atleast one reachable state of M.

Definition 3 A state variable $a \in \Sigma$ is called an *atom*, and *literal* l is an atom a or its negated form $\neg a$. A conjunction of literals, i.e., $l_1 \wedge l_2 \wedge \ldots \wedge l_k$, for $k \geq 1$, is called a *cube*. A disjunction of a set of literals, i.e., $l_1 \vee l_2 \vee \ldots \vee l_k$, for $k \geq 1$, is called a *clause*. A Boolean formula containing a conjunction of clauses is said to be in *Conjunctive Normal Form* (CNF).

A primed variable a', such that $a \in \Sigma$, represents a in the next time step. If ψ is a Boolean formula over Σ, ψ' is obtained by replacing each variable in ψ with the corresponding primed variable. We assume that a cube (or clause) c can be treated as a Boolean formula, set of literals, or set of states depending on the context it is used. For example, in the formula $c \Rightarrow \phi$ we treat c as a Boolean formula, in the statement $c_1 \subseteq c_2$ we treat c_1 and c_2 as sets of literals, and if we say a state t is in c, i.e., $c(t) = 1$, then we treat c as a set of states. Similarly, a Boolean formula ψ can be treated as a set of clauses or cubes, or a set of states depending on the context it is used. A clause c can be *weakened* (or *strengthened*) to clause $\hat{c}$ by adding (or removing) literals such that $\hat{c} \supseteq c$ (or $\hat{c} \subseteq c$).

Definition 4 Two finite sets ψ_1 and ψ_2 *overlap* iff $\psi_1 \cap \psi_2 \neq \emptyset$.

For transition systems $M = (\Sigma, Q_M, Q_{0_M}, \delta_M)$ and $N = (\Sigma, Q_N, Q_{0_N}, \delta_N)$ the set of reachable states are $R_M = \{s \in Q_M \mid s \text{ is reachable in M}\}$ and $R_N = \{s \in Q_N \mid s \text{ is reachable in N}\}$, respectively.

Definition 5 Given two transition system models $M = (\Sigma, Q_M, Q_{0_M}, \delta_M)$ and $N = (\Sigma, Q_N, Q_{0_N}, \delta_N)$, we say that M and N are *related* iff there exists a transformation function τ such that $\delta_N = \tau(\delta_M)$.

The transformation function may be defined by a set of rules that map transitions in model M to transitions in model N. We assume the existence of such a transformation function. Note that $R_M \cap R_N \neq \emptyset$ for related models M and N. A *set of models* is a collection of related models. Parameter instantiation generates a set of models from meta-models representing design-spaces [26], or software-product lines [23]. Moreover, updates to a sequential circuit design in regression verification, either due to a bug fix or feature addition, generate related transitions systems [42].

2.2 Safety Verification

The safety verification problem is to decide whether model $M = (\Sigma, Q, Q_0, \delta)$ is UNSAFE or SAFE with respect to a safety property φ, i.e., whether there exists an initial state in Q_0 that can reach a bad state in $\neg\varphi$, or generate an inductive invariant $\mathcal{I}$ that satisfies three conditions:

1. $Q_0 \Rightarrow \mathcal{I}$
2. $\mathcal{I} \wedge \delta \Rightarrow \mathcal{I}'$
3. $\mathcal{I} \Rightarrow \varphi$

In SAT-based model checking algorithms [9, 7, 35, 40], the verification problem is solved by computing over-approximations of reachable states in M, and using them to either construct an inductive invariant, or find a counterexample.

2.3 Overview of IC3

IC3/PDR [9, 10, 27, 31, 39] is a novel SAT-based verification method based on property directed invariant generation. Given a model $M = (\Sigma, Q, Q_0, \delta)$, and a safety property φ, IC3 incrementally generates an inductive strengthening of φ to prove whether $M \models \varphi$. It maintains a sequence of frames $S_0 = Q_0, S_1, \ldots S_k$ such that each S_i, for $0 < i < k$, satisfies φ and is an over-approximation of states reachable in i-steps or less. If two adjacent frames become equivalent, IC3 has found an inductive invariant and the property holds for the model. If a state violating the property is reachable, a counterexample trace is returned. Throughout IC3's execution, it maintains the following invariants on the sequence of frames:

1. for $i > 0$, S_i is a CNF formula, i.e., conjunction of clauses,
2. $S_{i+1} \subseteq S_i$,
3. $S_i \wedge \delta \Rightarrow S'_{i+1}$, and
4. for $i < k$, $S_i \Rightarrow \varphi$.

Each clause added to the frames is an intermediate lemma constructed by IC3 to prove whether $M \models \varphi$. The algorithm proceeds in two phases: a *blocking* phase, and a *propagation* phase. In the blocking phase, S_k is checked for intersection with $\neg\varphi$. If an intersection is found, S_k violates φ. IC3 continues by recursively blocking the intersecting state at S_{k-1}, and so on. If at any point, IC3 finds an intersection with S_0, $M \not\models \varphi$ and a counterexample can be extracted. The propagation phase moves forward the clauses from preceding S_i to S_{i+1}, for $0 < i \leq k$. During propagation, if two consecutive frames become equal, a fix-point has been found and IC3 terminates. The fix-point $\mathcal{I}$ represents the strengthening of φ and is an inductive invariant that satisfies the three conditions of Section 2.2.

2.4 SAT with Assumptions

In our formulation, we consider SAT queries of the form $\mathsf{sat}(\phi, \gamma)$, where ϕ is a CNF formula, and γ is a set of assumption clauses. A query with no assumptions is simply written as $\mathsf{sat}(\phi)$. Essentially, the query $\mathsf{sat}(\phi, \gamma)$ is equivalent to $\mathsf{sat}(\phi \wedge \gamma)$ but the implementation of the former is typically more efficient. If $\phi \wedge \gamma$ is:

1. SAT, get-sat-model() returns a satisfying assignment.
2. UNSAT, get-unsat-assumptions() returns a unsatisfiable core β of the assumption clauses γ, such that $\beta \subseteq \gamma$, and $\phi \wedge \beta$ is UNSAT.

We abstract the implementation details of the underlying SAT solver, and assume interaction using the above three functions.

2.5 Notation

We reduce the task of verifying a set of models by restricting the description of our algorithm to two related models $M = (\Sigma, Q_M, Q_{0_M}, \delta_M)$ and $N = (\Sigma, Q_N, Q_{0_N}, \delta_N)$ in the set. Each model has to be checked against a safety property φ. Assume that model M is checked first. The algorithm computes frame sequence R and S for M and N, respectively. $|R|$ denotes number of frames in the sequence R.

2.6 Problem Definition

Given two related models $M = (\Sigma, Q_M, Q_{0_M}, \delta_M)$ and $N = (\Sigma, Q_N, Q_{0_N}, \delta_N)$, and a safety property φ, let $R = R_0, R_1, R_2, \ldots, R_m$ be the sequence of frames computed by IC3 that satisfies the invariants of Section 2.3. We want to reuse the reachable state approximations of M to model-check property φ against model N, i.e., compute frame sequence $S = S_0, S_1, S_2, \ldots, S_n$ for model N that satisfies invariants of Section 2.3 by reusing frame sequence R such that $S_{i+1} = \hat{R}_{i+1}$, where $\hat{R}_{i+1} = R_{i+1}$ if $S_i \wedge \delta_N \Rightarrow R'_{i+1}$, otherwise $\hat{R}_{i+1}$ is obtained by strengthening or weakening clauses in R_{i+1} such that $\forall c \in R_{i+1}$, we have $S_i \wedge \delta_N \Rightarrow \hat{c}'$ and $S_i \wedge \delta_N \Rightarrow \hat{R}_{i+1}$.

3 Algorithm

In this section, we present the main contribution of our paper, FuseIC3. We start with the core idea behind the algorithm by giving the intuition behind recycling IC3-generated intermediate lemmas. We then provide a general overview of different sub-algorithms that help FuseIC3 achieve its performance. We next describe the two main components: *basic check* and *frame repair* of FuseIC3.

3.1 Intuition

Recall that frames computed by IC3 represent over-approximated states. When M is checked with IC3, frames $R_0, R_1, \ldots, R_j$, are computed such that $R_i \wedge \delta_M \Rightarrow R'_{i+1}$ for $i < j$ (invariant 3, Section 2.3). In the classical case, checking N after M requires resetting and restarting IC3, which then computes frames $S_0, S_1, \ldots, S_k$ for N. Due to the reset, all intermediate lemmas are lost and verification for N has to start from the beginning. However, since M and N are related, the frames for M and N overlap, and therefore, frames for M can be recycled and potentially reused in the verification for N. The idea is illustrated using Venn diagrams in Fig. 1.

In Fig. 1a, the parallelogram and ellipse represent clauses c_1 and c_2, respectively, in frame R_{i+1} such that $R_{i+1} = c_1 \wedge c_2$, and the triangle represents states reachable from R_i in one step, i.e., $R_i \wedge \delta_M$. So, $R_i \wedge \delta_M \Rightarrow R'_{i+1}$. Now consider a scenario in which we recycle the clauses in R_{i+1} when verifying N. The triangle and the rectangle in Fig. 1b represent the states reachable from S_i in one step. If we were to make $S_{i+1} = R_{i+1}$, we end up with $S_i \wedge \delta_N \not\Rightarrow S'_{i+1}$ since c_1 doesn't contain some states reachable from S_i. Therefore, we have to modify c_1 such that the invariant holds. Fig. 1c and 1d show the two possible modifications of c_1. In the former case, we add states $(S_i \wedge \delta_N) \setminus c_1$ to c_1 such that $\hat{c_1} = c_1 \cup (S_i \wedge \delta_N) \setminus c_1$. In the latter, we over-approximate c_1 to $\hat{c}_1$ such that $S_i \wedge \delta_N \Rightarrow \hat{c}_1$ (a trivial over-approximation is to make c_1 equal to the set of all states). Irrespective of the approach used, we end up with $S_i \wedge \delta_N \Rightarrow \hat{R}'_{i+1} = S'_{i+1}$, where $\hat{R}_{i+1} = \hat{c}_1 \wedge c_2$. Then we check the $(i+1)$-th step over-approximation for intersection with $\neg\varphi$ and IC3 continues. In this way, reusing clauses from model M, saves a lot of effort in rediscovering these clauses for model N.

3.2 Overview

FuseIC3 is a bidirectional reachability algorithm. It uses forward reachability to reuse frames from a previously-checked related model, and IC3-type backward reachability to recursively block predecessors to bad states. The algorithm description appears in Fig. 2.

FuseIC3 takes as input the initial states Q_0 and the transition relation δ for the current model, and a safety property φ. The internal state maintained by the algorithm is last_invariant, last_cex, and the frames R computed for the last model verified. Initially, the state is empty. Lines 1–2 perform basic checks in an attempt to reuse proofs from an earlier run to verify the current model. Lines 4–15 loop until an invariant or a counterexample is found. FuseIC3 maintains a sequence of frames $S_0, S_1, \ldots, S_k$ for the current model being checked. Whenever a new frame

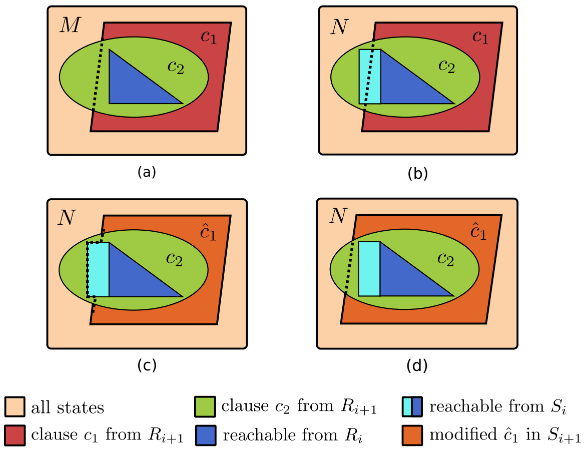

M
N
c_1
c_2
$\hat{c}_1$
(a)
(b)
(c)
(d)
all states
clause c_1 from R_{i+1}
clause c_2 from R_{i+1}
reachable from R_i
reachable from S_i
modified $\hat{c}_1$ in S_{i+1}

Fig. 1 Intuition behind repairing frames computed for one model by IC3, and reusing them for checking another related model in the design space.

```
bool FuseIC3 ($Q_0$, $\delta$, $\varphi$)
 1: if CheckInvar($Q_0$, $\delta$, last_invariant, $\varphi$) : return true
 2: if SimulateCex($Q_0$, $\delta$, last_cex, $\varphi$) : return false
 3: $k \leftarrow 0$, $S_k \leftarrow Q_0$   # first frame is initial state
 4: while true :   # main FuseIC3 loop
 5:     while sat($S_k \wedge \neg\varphi$) :   # blocking phase
 6:         $s \leftarrow$ get-sat-model()
 7:         if not recursive_block($s, k$) :
 8:             last_cex $\leftarrow$ extract_cex(), return false
 9:     $k \leftarrow k + 1$
10:     $S_k \leftarrow$ FrameRepair($k - 1$)
11:     for $i \leftarrow 1$ to $k - 1$ :   # propagation phase
12:         for each new clause $c \in S_i$ :
13:             if not sat($S_i \wedge c \wedge \delta \wedge \neg c'$) : add $c$ to $S_{i+1}$
14:         if $S_i \equiv S_{i+1}$ :   # found fix-point invariant
15:             last_invariant $\leftarrow S_i$, return true
frame FrameRepair (int i)
 1: if not sat($S_i \wedge \delta \wedge \neg R'_{i+1}$) : return $R_{i+1}$
 2: $\mathcal{G} \leftarrow$ FindClauses($S_i$, $\delta$, $R_{i+1}$)
 3: $\hat{R}_{i+1} \leftarrow R_{i+1} \setminus \mathcal{G}$
 4: for each clause $c \in \mathcal{G}$ :
 5:     $\hat{c} \leftarrow$ ExpandClause($S_i$, $\delta$, $c$)
 6:     $\hat{c} \leftarrow$ ShrinkClause($S_i$, $\delta$, $c$, $\hat{c}$)
 7:     $\hat{R}_{i+1} \leftarrow \hat{R}_{i+1} \wedge \hat{c}$
 8: return $\hat{R}_{i+1}$   # repaired frame $R_{i+1}$
```

Fig. 2 High-level description of FuseIC3. Parts of the algorithm for typical IC3 are based on the description in [27, 31].

S_k is introduced in line 10, the algorithm reuses a frame from R after repairing it with FrameRepair. The repaired frame is added to S_k, which after propagation in lines 11–15, is checked for intersection with a bad state. A typical execution of IC3 follows until a new frame is introduced. Upon termination, R is replaced with the current set of frames S, and last_invariant and last_cex are updated accordingly.

FrameRepair takes as input an integer i. It checks if R_{i+1} can be used as is in line 1. If yes, R_{i+1} is returned. Otherwise, the frame is repaired in lines 2–7. FindClauses finds violating clauses in R_{i+1}. Each of these clauses is repaired in lines 4–7 using ExpandClause and ShrinkClause. After repair, the updated frame $\hat{R}_{i+1}$ is returned.

The models in a set are checked sequentially. When FuseIC3 is run on the first model in the set, it reduces to running typical IC3. During propagation and when $k < |R|$, only repaired clauses (from FrameRepair) and discovered clauses for the current model are propagated. When $k \geq |R|$, FrameRepair returns an empty frame and all clauses from earlier frames take part in propagation.

3.3 Basic Checks

It is possible that the changes in design between two models are very small, and are outside the cone-of-influence of the verification procedure. Therefore, although the models are different, they might have the same over-approximated inductive invariant with respect to the property being checked. A similar argument applies for two models that fail a property. In this case, a counterexample for the first model might be a valid counterexample for the second model. Both these checks can be carried out in very little time as explained below. For the case when M and N have different state variables, cone-of-influence with respect to variables in N is applied on the invariant/counterexample before performing the checks.

Inductive Invariant. If $\mathcal{I}_M$ is an inductive invariant for M with respect to a safety property φ, it satisfies the following three conditions:

1. $Q_{0_M} \Rightarrow \mathcal{I}_M$,
2. $\mathcal{I}_M \wedge \delta_M \Rightarrow \mathcal{I}'_M$, and
3. $\mathcal{I}_M \Rightarrow \varphi$.

If model differences between M and N are small, or changes in N are outside the cone-of-influence of $\mathcal{I}_M$, then $N \models \varphi$ iff the above conditions hold for N, i.e.,

1. $Q_{0_N} \Rightarrow \mathcal{I}_M$,
2. $\mathcal{I}_M \wedge \delta_N \Rightarrow \mathcal{I}'_M$, and
3. $\mathcal{I}_M \Rightarrow \varphi$.

Counterexample Trace. If $M \not\models \varphi$, then IC3 generates a counterexample trace $s_0, s_1, \ldots s_k$ to prove satisfaction of $\neg\varphi$ such that

1. $s_0 \in Q_{0_M}$,
2. $(s_i, s_{i+1}) \in \delta_M$ for $i < k$, and
3. $s_k \in \neg\varphi$.

We simulate the counterexample trace for M on N and check if it satisfies the above three conditions (using $k + 1$ SAT calls). If the conditions are satisfied, the counterexample trace is a valid trace in N, and we conclude that $N \not\models \varphi$.

To summarize, if changes in two subsequent models are outside the cone-of-influence of the proofs generated by IC3, verification completes almost instantly. The pseudo-code for these two basic checks is given in Fig. 3.

```
bool CheckInvariant ($Q_0$, $\delta$, invariant $\mathcal{I}$, $\varphi$)
1: if not sat($Q_0 \land \neg\mathcal{I}$) and not sat($\mathcal{I} \land \delta \land \neg\mathcal{I}$)
   and not sat($\mathcal{I} \land \neg\varphi$) : return true
2: else return false

bool SimulateCex ($Q_0$, $\delta$, trace $s$, $\varphi$)
1: if not sat($s_0 \land Q_0$) : return false
2: if not sat($s_k \land \neg\varphi$) : return false
3: for $i \leftarrow 0$ to len($s$) :
4:     if not sat($s_i \land \delta \land s'_{i+1}$) : return false
5: return true  # valid counterexample
```

Fig. 3 CheckInvariant evaluates the last known invariant against the current model, and returns *true* if invariant holds, otherwise *false.* SimulateCex simulates the last known counterexample on the current model, and returns *true* if successful, otherwise, *false.*

```
FindClauses (frame $S$, $\delta$, frame $R$)
 1: for each clause $c_i \in R$ :  # configure solver assertions
 2:     introduce auxiliary variable $y_i$
 3:     for each literal $l \in c'_i$ :
 4:         add assertion $\neg l \lor y_i$ to solver
 5: $\mathcal{G} \leftarrow \emptyset$  # set is initially empty
 6: while sat($S \land \delta, (\neg y_1 \lor \neg y_2 \lor \ldots \lor \neg y_k)$) :
 7:     $\alpha \leftarrow$ get-sat-model()
 8:     for each $y_1, y_2, \ldots y_k$ :
 9:         if $\alpha(y_i) == \bot$ :
10:             add $c_i$ to $\mathcal{G}$ and remove $y_i$ from sat query
11: return $\mathcal{G}$  # set of violating clauses
```

Fig. 4 FindClauses algorithm to find all violating clauses $c_i \in R$ such that $S \land \delta \not\Rightarrow c'$. Upon termination, the set $\mathcal{G}$ contains all violating clauses.

3.4 Frame Repair

We want to find all clauses in frame R_{i+1} that are responsible for the violation of $S_i \land \delta_N \Rightarrow R'_{i+1}$. The satisfiability model is a pair of states (a, b) such that $a \in S_i$, $b \notin R_{i+1}$, and $(a, b) \in \delta_M$. In other words, b is missing from some, or all clauses in R_{i+1}. If all such missing states are added to clauses in R_{i+1}, resulting in $\hat{R}_{i+1}$, the condition $S_i \land \delta_N \Rightarrow \hat{R}'_{i+1}$ becomes valid and $\hat{R}_{i+1}$ can be reused in checking N. Adding these states one-by-one requires several calls to the underlying SAT solver and is infeasible in practice (reduces to all-SAT). Instead, we approximate the violating clauses in R_{i+1}. The over-approximation ends up adding several states to R_{i+1} that are in the post-image of multiple states in S_i. As the first step in repairing the frame, we find all such violating clauses.

Find Violating Clauses: Let's assume frame R_{i+1} is composed of a set of clauses $C = \{c_1, c_2, \ldots c_n\}$. There are clauses $\mathcal{G} \subseteq C$ such that the assertion $S_i \land \delta_N \Rightarrow c'$ is violated for all $c \in \mathcal{G}$. Set $\mathcal{G}$ can be found by brute-forcing the assertion check for all clauses in C. However, such an approach doesn't scale since IC3 frames can have thousands of clauses. Algorithm FindClauses, which is inspired by the *Invariant Finder* algorithm in [15], efficiently finds all such violating clauses. The pseudo-code for the algorithm is given in Fig. 4.

FindClauses takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, and frame $R = R_{i+1}$. Upon termination, it returns all violating clauses. An auxiliary variable y_i is introduced for each clause c_i in R in line 2. Lines 3–4 are equivalent to adding the assertion $c_i \Rightarrow y_i$ to the solver. Lines 6–10 loop until the query in line 6 is SAT. On every iteration of the loop, there is at least one y_i that is assigned *false*. Clauses c_i corresponding to all such y_i are added to $\mathcal{G}$ and y_i is removed from the query. When the query becomes UNSAT, $\mathcal{G}$ contains all violating clauses in R, and is returned. In practice, multiple y_i are assigned *false* which helps terminate the loop faster.

Theorem 1 *Given the current frame sequence S, transition relation δ, and frame sequence to reuse R, the* FindClauses *algorithm (Fig. 4) returns all violating clauses $c_i \in R$ such that $S \wedge \delta \not\Rightarrow c'_i$.*

Proof For each clause $c_i \in R$, we introduce an auxiliary variable y_i. For each literal $l \in c'_i$, we add the assertion $\neg l \wedge y_i$ to the solver. Let's assume $c_i = l_1 \vee l_2 \vee \ldots \vee l_k$. We add asertions $\neg l'_1 \vee y_i$, $\neg l'_2 \vee y_i$, ..., $\neg l'_k \vee y_i$ to the solver. Therefore, the overall assertion for clause c_i added is $(\neg l'_1 \vee y_i) \wedge (\neg l'_2 \vee y_i) \wedge \ldots \wedge (\neg l'_k \vee y_i)$. Now

$$\begin{aligned} & (\neg l'_1 \vee y_i) \wedge (\neg l'_2 \vee y_i) \wedge \ldots \wedge (\neg l'_k \vee y_i) \\ \Leftrightarrow\ & (\neg l'_1 \wedge \neg l'_2 \wedge \ldots \wedge \neg l'_k) \vee y_i \\ \Leftrightarrow\ & \neg(l'_1 \vee l'_2 \vee \ldots \vee l'_k) \vee y_i \\ \Leftrightarrow\ & \neg c'_i \vee y_i \\ \Leftrightarrow\ & c'_i \Rightarrow y_i \end{aligned}$$

Therefore, the operation performed in lines 1–4 of FindClauses is equivalent to adding the assertion $c'_i \Rightarrow y_i$ for each clause $c_i \in R$. Initially, the set of violating clauses $\mathcal{G}$ is empty. For the sake of argument, let's assume R contains only one clause c_1. If $c_1 = l_1 \vee l_2 \vee \ldots \vee l_k$, then the assertions added to the solver are $\neg l'_1 \vee y_1$, $\neg l'_2 \vee y_1$, ..., $\neg l'_k \vee y_1$. Moreover, the SAT query of line 6 adds the assertions $S \wedge \delta$, and assumes $\neg y_1$. Combined, these assertions are equivalent to $(S \wedge \delta \wedge \neg y_1 \wedge \neg c'_1)$ or $(S \wedge \delta \wedge \neg y_1 \wedge \neg R')$. There are two cases to consider. If the assertion is

1. UNSAT: The post-image of all states in S is in R, and c_1 is not a violating clause. Therefore, FindClauses terminates and returns $\mathcal{G} = \emptyset$.
2. SAT: We know that the SAT model for $S \wedge \delta \wedge \neg y_1 \wedge \neg R'$ is a pair of states (a, b') such that $a \in S$, $(a, b') \in \delta$, but $b' \notin R'$, and an assignment to y_1. Since R contains only one clause, $b' \notin R'$ if and only if $b' \notin c'_1$. In other words, none of the literals in c'_1 match the literal assignments in state b'. Therefore, $\neg l'_1$, $\neg l'_2$, ..., $\neg l'_k$ are true, which makes $\neg c'_1$ true. The only possible assignment to y_1 is false. Therefore, since c_1 is a violating clause, the corresponding auxiliary variable is assigned false. Clause c_1 is added to $\mathcal{G}$ in lines 9–10, and the sat query is updated.

Therefore, upon termination $\mathcal{G} = \emptyset$ or $\mathcal{G} = \{c_1\}$ if $S \wedge \delta \wedge \neg R'$ is UNSAT and SAT, respectively. The argument for R containing only one violating clause can be extended to multiple clauses. If a state b' in the SAT model is missing from multiple clauses in R, their corresponding auxiliairy variables get assigned to false, and all such clauses are added to $\mathcal{G}$ and the query updated. On every iteration of the loop in lines 6–10, a new state pair is found until all violating clauses have been removed from R and added to $\mathcal{G}$. Therefore, upon termination, set $\mathcal{G}$ contains all violating clauses $c_i \in R$ such that $S \wedge \delta \not\Rightarrow c'_i$. □

```
ExpandClause (frame $S$, $\delta$, clause $c$)
1:  $v \leftarrow$ all primed variables in $\delta$
2:  $l \leftarrow$ all variables in clause $c'$
3:  $B \leftarrow v \backslash l$  # variables not in clause c
4:  $\hat{c} \leftarrow c$  # initially $\hat{c} = c$
5:  while $|B| > 0$ and sat$(S \wedge \delta \wedge \neg\hat{c}')$ :
6:      $\alpha \leftarrow$ get-sat-model()
7:      randomly pick any $b' \in B$
8:      if $\alpha(b') == \top$ : add $b$ to clause $\hat{c}$
9:      else if $\alpha(b') == \bot$ : add $\neg b$ to clause $\hat{c}$
10:     remove $b'$ from $B$
11: if sat$(S \wedge \delta \wedge \neg\hat{c}')$ : return $\emptyset$
12: return $\hat{c}$  # expanded clause; $S \wedge \delta \Rightarrow \hat{c}'$
```

Fig. 5 ExpandClause algorithm to add literals to violating clause c such that $S \wedge \delta \Rightarrow \hat{c}'$. Upon termination, an empty clause is returned if expansion fails.

After discovering all violating clauses, FuseIC3 attempts to expand them, by adding literals, before reusing R_{i+1} to check model N. In the trivial case, each violating clause can be removed from R_{i+1}. However, doing this is quite wasteful. For example, consider a frame in which all clauses are violating. Reusing this frame entails restarting IC3 from an empty frame, a scenario we want to avoid. Instead, we rely on efficient use of the SAT solver to over-approximate the violating clauses.

Expand Violating Clauses: A clause c is violating if none of its literals match the literals in state b (recall the model (a, b) to the SAT query $S_i \wedge \delta_N \Rightarrow R'_{i+1}$). If any literal from b is added to c, resulting in $\hat{c}$, then $b \in \hat{c}$. Fundamentally, we want to add literals to clause c without actually enumerating all such b such that the assertion $S_i \wedge \delta_N \Rightarrow \hat{c}'$ holds. A literal can be added as is, or in its negated form. Adding both makes the assertion trivially valid. For example, consider a system with variables x, y, z, and a violating clause $c = (x \vee y)$. Our aim is to add states to c. Either z or $\neg z$ can be added to c, but not both. However, deciding what to add to make the assertion valid is beyond the scope of a SAT solver. Instead, we use an efficient randomized algorithm, ExpandClause, to add literals to clause c. The pseudo-code for the algorithm is given in Fig. 5.

ExpandClause takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, and the violating clause $c \in R_{i+1}$. Initially, $\hat{c} = c$. Lines 1–3 find all variables that are missing from c and store them in set B. The loop in lines 4–9 is repeated until set B becomes empty, or the query $S \wedge \delta \Rightarrow \hat{c}'$ becomes valid. In the latter case, enough literals have been added to expand c and the algorithm can terminate. From the SAT model α, randomly pick an assignment to a variable in B. If the assignment is *true*, add the variable as is to $\hat{c}$, otherwise, negate variable and add to $\hat{c}$. The added variable is removed from B and the loop continues. When all possible variables have been added to $\hat{c}$ and the assertion is still SAT, return $\hat{c}$ to be the empty clause ($c =$ *true*, or set of all states) in line 10.

Theorem 2 *Given the current frame sequence S, transition relation δ, and violating clause c, the* ExpandClause *algorithm (Fig. 5) weakens violating clause c to generate clause $\hat{c}$ such that $S_i \wedge \delta \Rightarrow \hat{c}'$.*

Proof In line 3, B contains all primed variables not in clause c'. Initially, $\hat{c}' = c'$. The SAT model of the query $S \wedge \delta \wedge \neg\hat{c}'$ is pair of states (a, b') such that $a \in S$, $(a, b') \in \delta$,

```
ShrinkClause (frame $S$, $\delta$, clause $c$, clause $\hat{c}$)
   assert(not sat($S \wedge \delta \wedge \neg\hat{c}'$))
1: $v \leftarrow$ {literals in $\hat{c}$} \ {literals in $c$}
2: $\tilde{c} \leftarrow c$
3: for each $l \in v$ :
4:     $g \leftarrow v \setminus l$   # drop literal $l$
5:     if not sat($S \wedge \delta \wedge \neg c', \neg g'$) :
6:         $v \leftarrow$ {literal $j$ | $j' \in$ get-unsat-assumptions()}
7: return $\tilde{c} \leftarrow \tilde{c} \vee \bigvee$ {literals in $v$}
```

Fig. 6 ShrinkClause algorithm to remove excess literals from clause c while maintaining $S \wedge \delta \Rightarrow c'$.

but $b' \notin \hat{c}'$. We know that $b \notin \hat{c}$ if none of the literals in $\hat{c}$ match a literal in state b. If we pick a literal in b and add it to $\hat{c}$, then $b \in \hat{c}$. The variable corresponding to the added literal is removed from B and the loop repeats. On every iteration of the loop in lines 5–10, multiple states are added to $\hat{c}$. The loop terminates when $S \wedge \delta \wedge \neg\hat{c}'$ is UNSAT, or B is empty. In the former case, ExpandClause returns $\hat{c}$, while in the latter, c is weakened to $\hat{c} = true$ (all states are reachable from S) and returned. □

Shrink Expanded Clauses: Due to the randomized nature of ExpandClause, we may end up adding more states than required to the expanded clauses. As a last step in repairing the frame, we remove the excess states added from all such clauses, albeit, maintaining the over-approximation. FuseIC3 uses UNSAT assumptions generated in the proof for $S_i \wedge \delta \Rightarrow \hat{c}'$ to shrink clause $\hat{c}$ to $\tilde{c}$. The ShrinkClause algorithm strengthens $\hat{c}$ by dropping a subset of the newly added literals from $\hat{c}$. The pseudo-code for the algorithm is given in Fig. 6.

ShrinkClause takes as input frame $S = S_i$, transition relation $\delta = \delta_N$, violating clause c, and the expanded clause $\hat{c}$. Set v contains all literals that were added to clause c by ExpandClause to generate clause $\hat{c}$. Lines 2–5 loop until enough literals have been dropped from $\hat{c}$ such that the $S_i \wedge \delta_N \wedge \neg c' \wedge \neg v'$ is valid. On each iteration of the loop, a literal l to drop from v is chosen. If the assertion is UNSAT, we can successfully drop l from v, and replace v with the UNSAT assumption literals in the query. However, if the assertion is SAT, l is a required literal in v and needs to be retained, so we try dropping another literal.

Theorem 3 *Given the current frame sequence S, transition relation δ, and violating clause c, the* ShrinkClause *algorithm (Fig. 6) strengthens clause $\hat{c}$ to generate clause $\tilde{c}$ such that $S \wedge \delta \Rightarrow \tilde{c}'$ and $|\tilde{c}| \leq |\hat{c}|$.*

Proof In line 1, v contains literals added to weaken c to $\hat{c}$, i.e., all literals that are added to c such that $S \wedge \delta \wedge \neg\hat{c}'$ is UNSAT. Initially, $\tilde{c} = c$. On every iteration of the loop in lines 3–6, we pick a literal l to drop from $\hat{c}$. If $S \wedge \delta \neg c' \wedge \neg g'$ is SAT, where $g = v \setminus l$, then l is a required literal and we try dropping another literal. If $S \wedge \delta \neg c' \wedge \neg g'$ is UNSAT, we extract the unsat core of the assumption literals. The unsat core is not necessarily minimal. v is made equal to the unsat assumption literals and the loop repeats. Upon termination, v contains the minimum literals that when added to c to give $\tilde{c}$ are enough to ensure that $S \wedge \delta \Rightarrow \tilde{c}'$. □

The violating clause may appear in future frames in R (due to the propagation phase when checking M). The modification is reflected in all occurrences of the clause. All such violating clauses in R_{i+1} are repaired.

Theorem 4 *Given the current frame sequence S_i and transition relation δ for model N, and frame sequence R_{i+1} for model M, the* FrameRepair *algorithm (Fig. 2) repairs frame R_{i+1} to $\hat{R}_{i+1}$ such that $S_i \wedge \delta \Rightarrow \hat{R}'_{i+1}$.*

Proof The proof follows from Theorems 1, 2, and 3. All violating clauses in R_{i+1} are found by the FindClauses algorithm. (Theorem 1). The ExpandClause algorithm (Theorem 2 weakens every violating clause $c \in R_{i+1}$ to generate clause $\hat{c}$. The expanded clause $\hat{c}$ is then strengthened to clause $\tilde{c}$ by the ShrinkClause algorithm (Theorem 3). The repaired clause is added $\hat{R}_{i+1}$. Therefore, upon termination, the FrameRepair algorithm returns repaired frame $\hat{R}_{i+1}$ such that $S_i \wedge \delta \Rightarrow \hat{R}'_{i+1}$. □

The repaired frame $\hat{R}_{i+1}$ is added to the set of frames for N at step $i+1$. Therefore, $S_{i+1} = \hat{R}_{i+1}$. Clauses are propagated from frames S_j, for $j \leq i$, to S_{i+1}, which is then checked for intersection with bad states, and the normal execution of blocking and propagation phases of IC3 follows.

4 Organizing the Design Space

If M and N have similar reachable states, FuseIC3 can reuse most of the reachability clauses learned for M when verifying N. However, determining models that have similar states is hard. The situation worsens when we are dealing with design spaces containing hundreds of models. We use two preprocessing heuristics to organize the design space: partially order the models, and group similar properties, that improve the performance of FuseIC3. We use locality-sensitive hashing [1] to order models in the design space, and group properties. We assume that the transition relation δ, for a model M, is a CNF formula over current- and next-state variables.

4.1 Hashing Techniques and Similarity Measure

Traditional hashing techniques map data from one domain to another. An ideal *hash function h* is an injective function that maps arbitrary sized data to data of fixed size. For example, a mapping from a string of characters to a 32-bit integer. Formally, $H : U \rightarrow V$, where U and V are the domain of input objects, and fixed size hash value, respectively. Ideally, for two objects $X, Y \in U$,

1. $H(X) = H(Y)$ for $X = Y$, and
2. $H(X) \neq H(Y)$ for $X \neq Y$.

A good hash function produces a large change in output for small changes in input. Hashing techniques find widespread use in databases, cryptography, and DNA sequencing [14] to find duplicates. Two objects X and Y are *same*, or equivalent, if $H(X) = H(Y)$. However, traditional hashing techniques do not allow to find objects that are *similar*, e.g., the words “color” and “colors” are similar, but not same; a hash function will produce vastly different outputs for these two inputs.

Locality-sensitive hashing (LSH) [1] is a technique that finds similar objects. LSH hashes inputs such that similar items map to the same *bucket*. In contrast to traditional hashing, LSH aims to maximize the probability of a collision for similar items. An LSH scheme for a universe of objects U, and similarity function $S : U \times U \rightarrow [0,1]$ is a probability distribution over a set $\mathcal{H}$ of hash functions such that

$$Pr_{H \in \mathcal{H}}[H(X) = H(Y)] = S(X,Y) \qquad \text{for any } X, Y \in U$$

Hash collisions capture the similarity between two objects. Possible measures for the similarity function include Euclidean distance, Jaccard similarity, Hamming distance, edit distance, etc. For our heuristic to partially order models in the design space, we use LSH with Jaccard distance as the similarity function. The Jaccard similarity coefficient for two sets X and Y is given by

$$S(X,Y) = J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

The goal of LSH is to find all similar objects in U based on their Jaccard similarity. The *MinHash* algorithm [11] is used to estimate the Jaccard similarity coefficient. Assuming that objects correspond to text documents, for every document D_i, we compute k minhash signatures using random hash functions. A minhash signature for a document D using a random hash function h is given by

$$h_{min}(D) = min(\{h(x) \mid x \in D\})$$

The signatures for each of the n documents are then divided into b bands of r rows each such that $b * r = k$. Two documents are similar if they share the exact same minhash signature on all rows of atleast one band. Figure 7 shows locality-sensitive hashing on a set of five documents. D_1 and D_3 are similar because they have the exact same minhash signatures for all rows in band 1. Documents D_2 and D_4 are also similar as they have signatures in all rows of band 5.

		D_1	D_2	D_3	D_4	D_5
Band 1	MinHash 1	172	29	172	193	41
	MinHash 2	33	125	33	59	98
	MinHash 3	45	156	45	44	56
	MinHash 4	13	34	13	71	153
	MinHash 5	101	51	101	143	67

• • •

		D_1	D_2	D_3	D_4	D_5
Band 5	MinHash 21	41	34	45	34	112
	MinHash 22	73	163	111	163	146
	MinHash 23	189	49	77	49	67
	MinHash 24	106	113	46	113	91
	MinHash 25	77	69	93	69	119

Fig. 7 Locality-sensitive hashing to find similar documents. D_1 and D_3, and D_2 and D_4 are similar from bands 1 and 5, respectively because they have the exact same minhash signatures on all rows of at least one band.

The probability that two documents A and B share the same signatures on all rows of atleast one band is given by $1 - (1 - J(A,B)^r)^b$ and can be estimated using the step function approximation $(\frac{1}{b})^{\frac{1}{r}}$ [37]. To estimate the values of b and r for $k = 400$ and a Jaccard similarity threshold of $t = 0.9$, we have

$$\left(\frac{1}{b}\right)^{\frac{1}{r}} = t \quad \Rightarrow \quad \left(\frac{1}{b = 20}\right)^{\frac{1}{r = 20}} = 0.86 \approx 0.9$$

Locality-sensitive hashing with minhash signatures will map documents that have their Jaccard coefficient higher than t to the same bands with high probability. For more details on locality-sensitive hashing with minhash we refer the reader to [37]. An important point to note is that LSH gives an $O(n)$ approximate algorithm to find similarities, compared to the quadratic algorithm for pairwise similarity. For our heuristics, the k hash functions for minhash signatures are generated by MurmurHash3 [2] with different seed values.

4.2 Partial Model Ordering (MO)

Let model-set $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ consist of related models of a design space. Locality-sensitive hashing is a favorable technique to find similar models in the design space; there is a high probability that models contain the same transition relation clauses. If the CNF formula is expressed in DIMACS CNF format[2] then a clause can be interpreted as a string of integers separated by whitespace and terminated with 0, and the CNF formula is a set of strings. Therefore, the transition relation δ_{M_i} can be viewed as a text document D_i containing strings representing clauses. Our LSH routine takes as input a set of documents corresponding to every model in the model-set. The heuristic works as follows:

1. Find groups of similar models using locality-sensitive hashing.
2. Consecutively check models in a group using FuseIC3 with a property φ.

The different groups are checked in random order. We use a Jaccard similarity coefficient of 0.9 for partial model ordering.

4.3 Property Grouping (PG)

Model checking techniques are computationally sensitive to the cone-of-influence (COI) size. Therefore, grouping properties based on overlap between support variables, or clauses containing support variables, in the COI of the property can speed up checking. Property *affinity* [12, 13] based on Jaccard similarity can compare the degree of overlap between COI. We generalize affinity to measure overlap between clauses. For two properties, φ_i and φ_j, let C_i and C_j, respectively, denote the clauses containing support variables in the cones of influences of the properties with respect to a model M. The affinity α_{ij} is then

$$\alpha_{ij} = \frac{|C_i \cap C_j|}{|C_i| + |C_j| - |C_i \cap C_j|}$$

If α_{ij} is larger than a given threshold, then properties φ_i and φ_j are conjoined together. The model M is then checked against $\varphi_i \wedge \varphi_j$. If verification fails, the violated property is removed from the conjunction, and the remaining property is checked. The heuristic works as follows:

1. Find groups of similar properties using locality-sensitive hashing (approximate).
2. Conjoin similar properties that have affinity larger than a threshold (exact).

[2] `http://www.satcompetition.org/2009/format-benchmarks2009.html`

3. Consecutively check conjoined properties using FuseIC3 with a model M.

The document to hash consists of clauses containing support variables, and the safety property clauses. The groups are checked in random order. We use a Jaccard similarity coefficient of 0.9 for finding similar properties, and a property affinity threshold of 0.95 for grouping properties.

5 Experimental Analysis

In this section, we report on our extensive experimental analysis with FuseIC3. We summarize the setup used for the experiments, briefly detail our benchmarks, and end with experimental results.

5.1 Setup

FuseIC3 is implemented in C++ and uses MathSAT5[18] as the underlying SMT solver. It takes SMV models or AIGER files as input. The IC3 part of FuseIC3 is based on the description in [27] and `ic3ia`.[3] We compare the performance of FuseIC3 with typical IC3 (`typ`), and incremental IC3 (`inc`). The algorithm for incremental IC3 is part of IBM's RuleBase model checker[4]. We implemented `inc` based on the description in [15] to the best of our understanding. We study the impact of partial model ordering (`MO`) and property grouping (`PG`) heuristic on the performance of FuseIC3. Locality-sensitive hashing using minhash signatures is implemented as a preprocessing Python script. All experiments were performed on Iowa State University's Condo Cluster comprising of nodes having two 2.6GHz 8-core Intel E5-2640 processors, 128 GB memory, and running Enterprise Linux 7.3. Each model-checking run had exclusive access to a node, which guarantees that no resource conflict with other jobs will occur.

5.2 Benchmarks

We evaluate FuseIC3 over a large collection of challenging benchmarks. The benchmarks are derived from real-world case studies and modified benchmarks from the Hardware Model Checking Competition (HWMCC) [6] 2015.

5.2.1 Air Traffic Controller (ATC) Models

The benchmark consists of a large set of 1,620 real-world models representing different possible designs for NASA's NextGen air traffic control (ATC) system[29]. The set of models are generated from a contract-based, parameterized nuXmv model. Each model is checked against 34 safety properties. The entire evaluation consists of 34 model-sets (one for each property) containing 1,620 models.

[3] `https://es-static.fbk.eu/people/griggio/ic3ia/`

5.2.2 Selected Benchmarks from HWMCC 2015

We consider a total of 548 benchmark models from the single safety property track[6]. Of the 548, 110 models are solved using our implementation of IC3 within a timeout of 5 minutes. To create a model-set, we generate 200 mutations of each of the 110 benchmarks. The original model is mutated to only modify the transition system of the cone-of-influence reduced model, and not the safety property implicit in the AIGER file; 1% of the assignments are randomly modified. An assignment of the form $g = g_1 \wedge g_2$ is selected with probability 0.01 and changed to $g = 0$, $g = 1$, $g = \neg g_1 \wedge g_2$, $g = g_1 \wedge \neg g_2$, $g = \neg g_1 \wedge \neg g_2$, $g = g_1 \wedge g_2$, $g = g_1$, $g = \neg g_1$, $g = g_2$, or $g = \neg g_2$, with equal probability. Therefore, the full evaluation consists of 110 model-sets, each consisting of one property and 200 models.

5.2.3 Wheel Braking System (WBS) Models

The benchmark consists of seven real-world models representing possible designs for the Boeing AIR6110 wheel braking system[8]. Each model is checked against ∼250 safety properties. However, the properties checked for each model are not the same. We evaluate FuseIC3 using this benchmark to measure performance when a model is checked against several related or similar properties. Each model is checked using a timeout of 120 minutes.

5.3 Results

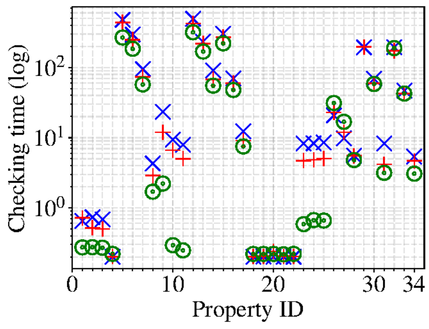
5.3.1 Air Traffic Controller (ATC) Models

Each of the 34 model-sets are checked using a timeout of 720 minutes per algorithm. The models in a set are checked in random order, and then using the model ordering (MO) heuristic. We experiment with ten different random orderings and report averaged results. Table 1 gives a summary of the results. FuseIC3 is median 1.75× (average 5.48×) faster compared to typical IC3, and median 1.34× (average 3.67×) faster compared to incremental IC3. On the other hand, incremental IC3 is median 1.29× (average 1.3×) faster than typical IC3. The model ordering heuristic improves the performance of FuseIC3 making it median 2.23× (average 6.89×) and 1.87× (4.47×) faster than typical and incremental IC3, respectively. We use a value of $k = 20,000$ with $b = 500$, and $r = 40$ for the heuristic. It takes ∼30 minutes to find a partial order among 1,620 models. The impact of model ordering is clearly evident: two similar models share the reachable state space, and FuseIC3 is able to reuse several reachable state clauses.

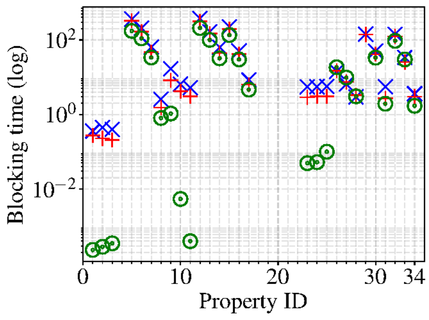
Fig. 8a shows time taken by the algorithms on each model-set. FuseIC3 is almost always faster than typical IC3, and incremental IC3. However, for model-sets (corresponding to property IDs 4 and 18–22) containing models that trivially satisfy/falsify a property, typical IC3 is faster; both incremental IC3 and FuseIC3 require a certain overhead in extracting information from the last checker run. FuseIC3 tries minimizing the time spent in exploring the common state space between models. In terms of the IC3 algorithm, this relates to time spent in finding bad states and blocking them at earlier steps (blocking phase). Fig. 8b shows time taken by each algorithm in blocking discovered bad states. FuseIC3 spends considerably less time in the blocking

Table 1 Results for 34 sets of 1,620 models each for NASA Air Traffic Control System.

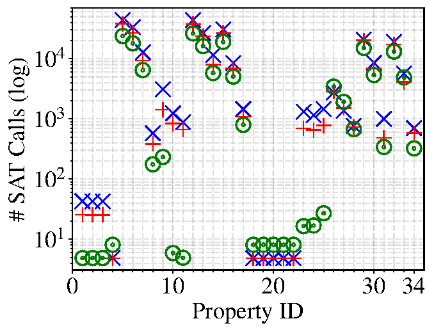
Algorithm	Cumulative Time (minutes)	Median (Average) Speedup	
		v/s typ (avg)	v/s inc (avg)
Typical IC3 (typ)	2502.70	-	-
Incremental IC3 (inc)	2180.57	1.29 (1.3)	-
FuseIC3	1683.53	1.75 (5.48)	1.34 (3.67)
FuseIC3 + MO	1352.53	2.23 (6.89)	1.87 (4.47)



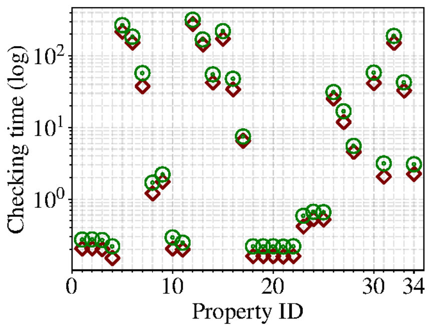
(a) Checking time per set (minutes)



(b) Blocking time per set (minutes)



(c) Number of SAT calls per set



(d) Time with model ordering (minutes)

Fig. 8 Comparison between IC3 (×), incremental IC3 (+), FuseIC3 (⊙) and FuseIC3 with model ordering (◇) on NASA Air Traffic Control System models. There are a total of 34 properties. 1,620 models are checked per property. Every property ID corresponds to a model-set. A point represents cumulative time taken to check all models for a property by an algorithm.

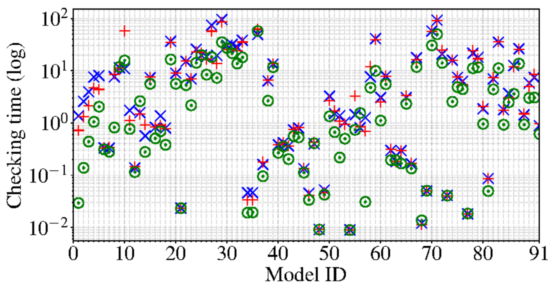
phase compared to typical IC3 and incremental IC3. Therefore, FuseIC3 is successful in reusing a major part of the already-discovered state space between different checker runs, a major requirement when checking large design spaces. Fig. 8c shows the total number of calls made to the underlying SAT solver by each algorithm. FuseIC3 makes fewer SAT calls and takes less time to check each model-set. The model ordering heuristic improves the performance of FuseIC3 as shown in Fig. 8d. Checking partially ordered models is faster than random checking for all model-sets.

Table 2 Results for 91 of 110 sets of 200 models each for selected HWMCC 2015 benchmarks.

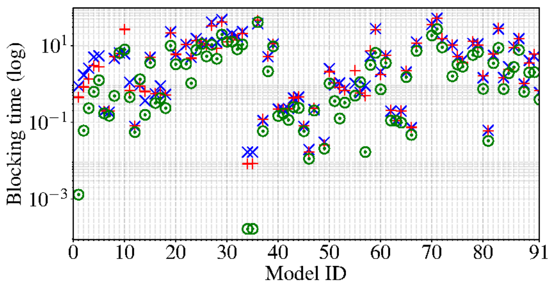
Algorithm	Cumulative Time (minutes)	Median (Avergae) Speedup	
		v/s typ (avg)	v/s inc (avg)
Typical IC3 (typ)	1024.60	-	-
Incremental IC3 (inc)	1026.30	1.04 (1.07)	-
FuseIC3	545.31	1.75 (3.18)	1.72 (2.56)
FuseIC3 + MO	396.65	2.32 (3.96)	2.05 (3.12)

5.3.2 Benchmarks from HWMCC 2015

Each of the 110 model-sets are checked using a timeout of 120 minutes per algorithm. The models in a set are checked in random order, and then using model ordering (MO) heuristic. 91 of 110 model-sets were solved by all algorithms within the timeout. Incremental IC3 solved two more model-sets compared to typical IC3, while FuseIC3 solved five more compared to typical IC3. Table 2 gives a summary of results.



(a) Cumulative checking time per design space (minutes)



(b) Cumulative blocking time per design space (minutes)

Fig. 9 Comparison between IC3 (×), incremental IC3 (+), and FuseIC3 (⊙) on 91 benchmarks from HWMCC 2015. Each model is converted to a model-set containing 200 models, generated by 1% mutation of the original. Every model ID corresponds to a model-set. A point represents cumulative time for checking all mutated versions of a model.

Fig. 9a shows time taken by the algorithms in checking each benchmark model-set. FuseIC3 is median 1.75× (average 3.18×) faster than typical IC3, and median 1.72× (average 2.56×) faster than incremental IC3. Significant speedup is achieved

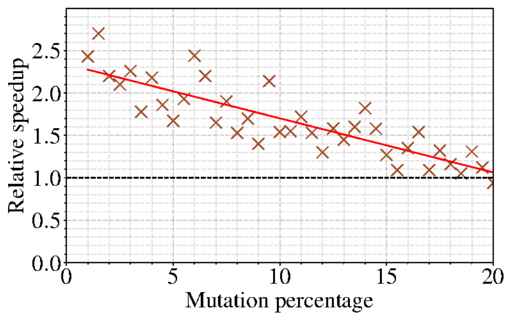


Fig. 10 Relative speedup between checking using FuseIC3 with model ordering versus typical IC3. 100 models are generated for every mutation percentage between 0.5% to 20% in steps of 0.5%, and are checked against the same property. The line is a linear fitting of the points.

when checking model-sets containing large models with FuseIC3. Performance for model-sets containing small models is similar for all algorithms. Fig. 9b shows time spent by each algorithm in blocking predecessors to bad states.

To estimate performance of FuseIC3 on model-sets with varying degree of overlap among models, we picked the `bobtuint18neg` benchmark from HWMCC 2015. 40 model-sets with varying degrees of mutation, between 0.5% to 20%, of the original model were generated. Each model-set consists of 100 models each. Each set was checked using a timeout of 300 minutes with typical IC3, and FuseIC3 with model ordering (MO). Model-sets corresponding to higher mutation values (greater than 20%) time out (SAT solvers are tuned for practical designs and random mutations create SAT instances that don't always correspond to real designs [15]) and are not reported. Fig. 10 gives a summary of the speedup between checking using FuseIC3 with MO versus typical IC3. Even at higher mutation percentages, checking a model-set using FuseIC3 is significantly faster than typical IC3.

5.3.3 Wheel Braking System Models

A model in the design space was checked against several properties, differently from the other benchmarks that checked all models in a set with the same property. Each model was checked using a timeout of 120 minutes. The properties for each model were checked in random order, and then using the property grouping (PG) heuristic. Table 3 gives a summary of the results.

Compared to other benchmarks, FuseIC3 achieves a smaller speedup when checking the WBS models. Although some properties being checked for the models are similar, i.e., the bad states representing the negation of the property overlap, the order in which they are checked greatly influences the performance of FuseIC3. In the random ordering used for the experiment, FuseIC3 is able to reuse frames without any repair (the same model is being checked), however, it spends a lot of time in blocking predecessors to bad states. Nevertheless, it is faster than checking all properties on a model using typical IC3. On the other hand, incremental IC3 is slower compared to typical IC3. It is able to extract the minimal inductive invariant (invariant finder) instantly, however, suffers from the same problem as FuseIC3. Incremental IC3, and FuseIC3 will benefit if similar properties are checked in order. Our property grouping (PG) heuristic conjoins properties that have overlapping cone-of-influence. The 247

Table 3 Comparison between typical IC3, Incremental IC3, and FuseIC3 for AIR6110 Wheel Braking System (time is in minutes).

Model	Typical IC3	Incremental IC3		FuseIC3			FuseIC3 + PG		
	Time	Time	v/s typ	Time	v/s typ	v/s inc	Time	v/s typ	v/s inc
M_1	4.36	5.02	0.87	**3.72**	1.17	1.35	**2.03**	2.14	2.47
M_2	15.78	16.65	0.95	**14.80**	1.07	1.13	**5.64**	2.79	2.95
M_3	12.43	13.48	0.92	**11.24**	1.11	1.20	**4.34**	2.86	3.10
M_4	12.45	13.66	0.91	**11.09**	1.12	1.23	**4.67**	2.66	2.92
M_5	15.92	17.04	0.93	**14.71**	1.08	1.16	**6.03**	2.64	2.82
M_6	**16.85**	17.79	0.95	17.04	0.99	1.04	**6.57**	2.56	2.70
M_7	12.95	13.67	0.95	**12.12**	1.07	1.13	**4.59**	2.82	2.97
	90.73 (total)	97.31 (total)	0.95 (median)	84.72 (total)	1.11 (median)	1.20 (median)	34.57 (total)	2.66 (median)	2.92 (median)

safety properties were distributed in 73 groups, and each group was checking against a model. The PG heuristic improves model checking performance making FuseIC3 upto 2.86× faster than typical IC3, and upto 3.10× faster than incremental IC3. The boost in performance is primarily due to the reduced number of model checking runs for groups compared to checking each property individually.

6 Conclusions and Future Work

FuseIC3, a SAT-query efficient algorithm, significantly speeds up model checking of large design spaces. It extends IC3 to minimize time spent in exploring the state space in common between related models. FuseIC3 spends less time during the blocking phase (Fig. 8b and Fig. 9b) due to success in reusing several clauses, has to learn fewer new clauses, and makes fewer SAT queries. The smallest salvageable unit in FuseIC3 is a clause; due to this granularity, FuseIC3 is able to selectively reuse stored information and is faster than the state-of-the-art algorithms that rely on reusing a coarser CNF invariant [15]. FuseIC3 is industrially applicable and scalable as witnessed by its superior performance on a real-life set of 1,620 NASA air traffic control system models (achieving an average 5.48× speedup), and benchmarks from HWMCC 2015 (achieving an average 3.18× speedup). Despite spending significant time in learning new clauses for the Boeing wheel braking system models, FuseIC3 is still faster than the previous best algorithm, typical IC3, when checking properties in random order; FuseIC3's performance improves by ordering models in a set, and checking similar properties together.

Ordering of models and properties in the design space improves the performance of FuseIC3, much like variable ordering in BDDs. Heuristics for optimizing model ordering are a promising topic for future work. Faster hashing and cone-of-influence computation techniques will greatly benefit faster ordering of models and property grouping. Preprocessing the models and properties, based on knowledge about the design space, before checking them with FuseIC3 may remove redundancies in the design space. Online property grouping algorithms [24] can by extended to dynamically reorder properties based on information reuse and semantic information learned during a model checking. Exploring synergies between offline and online property grouping algorithms is a promising research direction. We plan to extend FuseIC3 to checking liveness properties by using it as a safety checker[19]. The variation in the COI-overlap across properties for a model impacts the performance of FuseIC3 understanding the relationship between COI variations and clause reuse by FuseIC3 is future work. We also to plan to investigate extending FuseIC3 to reuse intermediate results of SAT queries, generalized clauses, and IC3 proof obligations across models. Finally, since checking large design spaces is becoming commonplace, we plan to develop more model-set benchmarks and make them publicly available.

Acknowledgements Thanks to NSF CAREER Award CNS-1664356 for supporting this work. We thank Armin Biere for discussion and feedback on the use of hashing to find the exact same models in a design space.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Commun. ACM **51**(1), 117–122 (2008). doi: 10.1145/1327452.1327494. URL `http://doi.acm.org/10.1145/1327452.1327494`
2. Appleby, A.: SMHasher and MurmurHash3. `https://github.com/aappleby/smhasher`
3. Bauer, C., Lagadec, K., Bès, C., Mongeau, M.: Flight control system architecture optimization for fly-by-wire airliners. J. Guidance, Control, and Dynamics **30**(4) (2007)
4. Beer, I., Ben-David, S., Eisner, C., Landver, A.: RuleBase: An industry-oriented formal verification tool. In: DAC (1996)
5. Ben-David, S., Sterin, B., Atlee, J.M., Beidu, S.: Symbolic model checking of product-line requirements using SAT-based methods. In: ICSE, vol. 1, pp. 189–199 (2015)
6. Biere, A.: HWMCC. `http://fmv.jku.at/hwmcc15/` (2015)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bdds. In: W.R. Cleaveland (ed.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
8. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal design and safety analysis of AIR6110 wheel brake system. In: CAV (2015)
9. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI, pp. 70–87 (2011)
10. Bradley, A.R.: Understanding IC3. In: SAT, pp. 1–14 (2012)
11. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. Journal of Computer and System Sciences **60**(3), 630 – 659 (2000). doi: https://doi.org/10.1006/jcss.1999.1690. URL `http://www.sciencedirect.com/science/article/pii/S0022000099916902`
12. Cabodi, G., Camurati, P.E., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S.: To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking. International Journal on Software Tools for Technology Transfer (2017). doi: 10.1007/s10009-017-0451-8. URL `https://doi.org/10.1007/s10009-017-0451-8`
13. Cabodi, G., Nocco, S.: Optimized model checking of multiple properties. In: 2011 Design, Automation Test in Europe, pp. 1–4 (2011). doi: 10.1109/DATE.2011.5763279

14. Chi, L., Zhu, X.: Hashing techniques: A survey and taxonomy. ACM Comput. Surv. **50**(1), 11:1–11:36 (2017). doi: 10.1145/3047307. URL `http://doi.acm.org/10.1145/3047307`
15. Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental Formal Verification of Hardware. In: FMCAD, pp. 135–143 (2011)
16. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for temporal logic model checking. FMSD **28**(3), 189–212 (2006)
17. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD, pp. 165–168 (2013)
18. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS, pp. 93–107 (2013)
19. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: FMCAD, pp. 52–59 (2012)
20. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with snip. (STTT) pp. 1–24 (2012)
21. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. IEEE Trans. Softw. Eng. **39**(8), 1069–1089 (2013)
22. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE, pp. 321–330 (2011)
23. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE, pp. 335–344 (2010)
24. Dureja, R., Baumgartner, J., Ivrii, A., Kanzelman, R., Rozier, K.Y.: Boosting verification scalability via structural grouping and semantic partitioning of properties. In: Formal Methods in Computer Aided Design (FMCAD), pp. 1–9 (2019). doi: 10.23919/FMCAD.2019.8894265
25. Dureja, R., Rozier, K.Y.: FuseIC3: An Algorithm for Checking Large Design Spaces. In: Proceedings of the 17th International Conference on Formal Methods in Computer-Aided Design, FMCAD '17, pp. 164–171. FMCAD Inc, Austin, TX (2017)
26. Dureja, R., Rozier, K.Y.: More Scalable LTL Model Checking via Discovering Design-Space Dependencies (D^3). In: D. Beyer, M. Huisman (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 309–327. Springer International Publishing, Cham (2018)
27. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: FMCAD, pp. 125–134 (2011)
28. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE, pp. 236–254 (2000)
29. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: CAV (2016)
30. Goldberg, E., GÃijdemann, M., Kroening, D., Mukherjee, R.: Efficient verification of multi-property designs (The benefit of wrong assumptions). In: Design, Automation Test in Europe (DATE), pp. 43–48 (2018). doi: 10.23919/DATE.2018.8341977
31. Griggio, A., Roveri, M.: Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **35**(6), 1026–1039 (2016)

32. James, P., Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: On modelling and verifying railway interlockings: Tracking train lengths. Science of Computer Programming **96**(3) (2014)
33. Marques-Silva, J.: Interpolant learning and reuse in sat-based model checking. Theoretical Computer Science **174**(3), 31 – 43 (2007)
34. Mattarei, C., Cimatti, A., Gario, M., Tonetta, S., Rozier, K.Y.: Comparing different functional allocations in automated air traffic control design. In: FMCAD (2015)
35. McMillan, K.L.: Interpolation and sat-based model checking. In: W.A. Hunt, F. Somenzi (eds.) Computer Aided Verification, pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
36. Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Defining and model checking abstractions of complex railway models using CSP—B. In: HVC (2013)
37. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press, New York, NY, USA (2011)
38. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software. Formal Aspects of Computing (2016)
39. Somenzi, F., Bradley, A.R.: IC3: Where Monolithic and Incremental Meet. In: FMCAD, pp. 3–8 (2011)
40. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Formal Methods in Computer-Aided Design, pp. 1–8 (2009). doi: 10.1109/FMCAD.2009.5351148
41. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of bdd-based model checking. In: FMCAD, pp. 255–289 (1998)
42. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: ICSM, pp. 115–124 (2009)