Model checking large design spaces: Theory, tools, and experiments

by

Rohit Dureja

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Kristin Y. Rozier, Co-major Professor
Gianfranco Ciardo, Co-major Professor
Samik Basu
Robyn Lutz
Hridesh Rajan

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the early stages of design, there are frequently many different models of the system under development constituting a *design space*. The different models arise out of a need to weigh different design choices, to check core capabilities of system versions with varying features, or to analyze a future version against previous ones in the product line. Every unique combinations of choices yields competing system models that differ in terms of assumptions, implementations, and configurations. Formal verification techniques, like *model checking*, can aid system development by systematically comparing the different models in terms of *functional correctness*, however, applying model checking off-the-shelf may not scale due to the large size of the design spaces for today's complex systems. We present scalable algorithms for *design-space exploration* using model checking that enable exhaustive comparison of all competing models in large design spaces.

Model checking a design space entails checking multiple models and properties. Given a formal representation of the design space and properties expressing system specifications, we present algorithms that automatically prune the design space by finding inter-model relationships and property dependencies. Our design-space reduction technique is compatible with off-the-shelf model checkers, and only requires checking a small subset of models and properties to provide verification results for every model-property pair in the original design space. We evaluate our methodology on case-studies from NASA and Boeing; our techniques offer up to 9.4× speedup compared to traditional approaches.

We observe that sequential enumeration of the design space generates models with small incremental differences. Typical model-checking algorithms do not take advantage of this

information; they end up re-verifying "already-explored" state spaces across models. We present algorithms that learn and reuse information from solving related models against a property in sequential model-checking runs. We formalize heuristics to maximize reuse between runs by efficient "hashing" of models. Extensive experiments show that information reuse boosts runtime performance of sequential model-checking by up to 5.48×.

Model checking design spaces often mandates checking several properties on individual models. State-of-the-art tools do not optimally exploit subproblem sharing between properties, leaving an opportunity to save verification resource via concurrent verification of "nearly-identical" properties. We present a near-linear runtime algorithm for partitioning properties into provably high-affinity groups for individual model-checking tasks. The verification effort expended for one property in a group can be directly reused to accelerate the verification of the others. The high-affinity groups may be refined based on semantic feedback, to provide an optimal multi-property localization solution. Our techniques significantly improve multi-property model-checking performance, and often yield >4.0× speedup.

Building upon these ideas, we optimize parallel verification to maximize the benefits of our proposed techniques. Model checking tools utilize parallelism, either in *portfolio* mode where different algorithm strategies run concurrently, or in *partitioning* mode where disjoint property subsets are verified independently. However, both approaches often degrade into highly-redundant work across processes, or under-utilize available processes. We propose methods to minimize redundant computation, and dynamically optimize work distribution when checking multiple properties for individual models. Our techniques offer a median 2.4× speedup for complex parallel verification tasks with thousands of properties.

CHAPTER 1. INTRODUCTION

Safety-critical systems have become an integral part of our daily lives. They fly our planes, navigate autonomous vehicles, protect financial transactions, and even run our medical devices. We are increasingly dependent on these systems whose failure might endanger human life, lead to substantial economic loss, or cause extensive environmental damage. There are several well-known examples of safety-critical system failures that have occurred including the Ariane V launch failure [Dow97], the 2003 blackout of northeastern United States [ASH07], the 'failsafe mode' bug in 1.9 million Toyota Prius cars that caused a moving car's engine to stall [Koo14], and the more recent Boeing 737 Max's MCAS failure [JH19]. How can we make sure that safety-critical systems of the future are safe? The ever-increasing complexity of these life-critical hardware and software systems makes reasoning about safety extremely difficult. As we push more transistors in our integrated circuits and make our programming languages more expressive and high-level, guaranteeing safe operation of such systems becomes inherently challenging. There is a significant need to develop scalable techniques for specification, design and verification of critical systems.

We can use *formal methods* to guarantee safety of critical systems. Formal verification techniques utilize mathematical logic to provide correctness checks and very high levels of safety assurance. Their efficient usage guarantees that the designed system behaves according to the specification, and more importantly guarantee that the system does not do anything that is outside the specified behavior. It is important to note that the latter is considerably harder: the verification algorithm has to enumerate all possible behaviors of the system to check absence of any unspecified behaviors. Formal verification is exhaustive in nature, i.e., the system behavior is evaluated over all possible inputs. The system under design

is modeled in a high-level expressive language, like System-C [BDBK09], Verilog [ver06], or SMV [CCD+14], and specifications (design requirements) are expressed in mathematical logic, like System Verilog Assertions (SVA) [sva18], Property Specification Language (PSL) [psl10], or Linear Temporal Logic (LTL) [Pnu77]. The formal verification algorithm then exhaustively checks the system model against the specification to demonstrate the presence of bugs as well as the absence of bugs. On the other hand, time-honored techniques of *simulation and testing* are useful debugging tools in the early stages of system design. These methods evaluate the behavior of the system on a large, but rarely exhaustive, set of expected inputs. The restricted set of testing inputs limits the number of bugs that can be uncovered by these techniques: exhaustive evaluation of all possible inputs is often impractical for very large designs. Simulation and testing can be used to demonstrate the presence of bugs but never the absence of bugs. Moreover, the utility diminishes as the design is refined and remaining bugs become fewer and more subtle, and require more time to uncover. It has been proved that simulation and testing alone cannot guarantee high levels of reliability within any realistic time period [BF93]. For some systems this is an acceptable risk. However, for safety-critical systems, the absolute assurance that the system adheres to the specification via exhaustive analysis of intended, unexpected and unintended behaviors is key to guarantee reliability and safety. Therefore, formal verification complements simulation and testing to provide ultra-reliable safety-critical systems.

While there are a range of formal verification techniques [DKW08, KG99], *model-checking* has become one of the most widely-used formal method due to its automated nature and ease-of-use [CHV18]. Model checking is the process through which a desired *property* (system specification) is verified to hold for a given system *model* via an exhaustive enumeration of all reachable states and the behaviors that cause transitions between these states. A model checker will consider every possible combination of inputs and state, making the verification equivalent to exhaustive testing of the model. If the specification is found to not hold in all

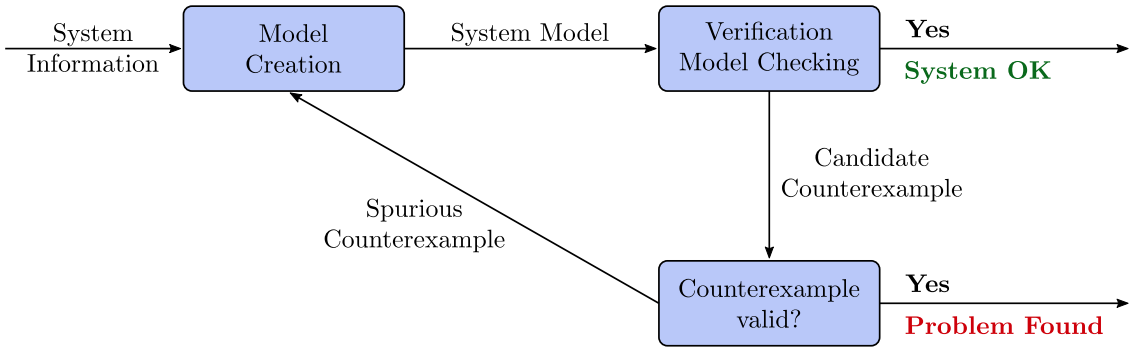


Figure 1.1: Model checking involves entering the system information (as a model) and requirements into a model-checking tool. If there is disagreement between the model's operation and its requirements, the model checker returns a *counterexample trace*. Otherwise, the system satisfies the specification

possible executions of the system, a *counterexample* is produced that shows an execution of the system from the start state to an error state that violates the specification. The ability to explain specification violation via counterexamples is a very helpful tool for debugging the system design. Figure 1.1 shows a typical model-checking workflow. Model checking has witnessed widespread industry adoption and has been used to verify systems such as air traffic controllers [GCM+16], autopilots [MAW+12], CPU designs [Fix08, KGN+09], encryption protocols [BCM18], financial transactions [PI17], medical equipment [JPM+12], network infrastructure [ME04], and many systems that ensure human safety and prevent financial loss.

1.1 Motivation

The technique of model-checking was developed independently by Clarke and Emerson in 1981 [CE81], and Quielle and Sifakis in 1982 [QS82]. Model checking provides a feasible and comprehensible infrastructure that permits bug detection as well as verification for correctness. Once the system model and properties have been determined, model-checking provides "push-button" and "automatic" verification. The counterexample returned in the case where

a bug is found provides the necessary diagnostic feedback. The shallow learning curve of model-checking has enabled its integration into industrial product life-cycles [GV08]; model checking requires minimal levels of user interaction and specialized expertise when compared to other methods of formal verification. However, certain systems that require finer-grained control of the verification process may benefit from the use of alternative techniques, like *theorem proving* that involves proving correctness using formal deduction that the system model implies the desired properties. Nevertheless, model checking has several qualities that make it the preferred formal method for verification of safety-critical systems:

1. The automatic high-quality counterexamples in model checking quickly provide a wealth of insight into the detected faulty system behavior. However, the quality of insight obtained from a negative result when using theorem proving is highly dependent on the skill set of the person providing the proof.

2. While a methodology of designing a system hand-in-hand with its proof using theorem proving has its merits, it cannot be readily automated. Model checking enables separation of system development from verification and debugging; the development and verification teams can work in parallel, and regular back-and-forth between teams can influence bug-fixes and future design decisions.

Since model checking requires the writing of formal properties, it has also helped fuel the industrial adoption of *property-based design*, wherein formal specifications are written early in the system-design process and communicated across all design phases [Roz16].

Model-checking technology has advanced considerably over the last three decades. Much progress has been made from the early days of *explicit-state model-checking* that involves exhaustive traversal of all reachable states of the system using graph-search algorithms, to *symbolic model-checking* [BCM+90] that reasons over logical formulas representing reachable states and properties. Several algorithmic advances, including partial-order reduc-

tion [Pel18], compositional verification [GNP18], bi-simulation equivalences [Mil71], bounded model checking [BCC+03], abstraction-refinement [DG18], and property-directed reachability [Bra12, EMB11], have increased the complexity and size of systems that can be verified using formal methods. Even though property-based design and model-checking for verification requires more time and has a higher up-front cost (specialized expertise, formal modeling and specification) compared to simulation or testing, the higher cost is outweighed by high-levels of assurance provided by formal verification for critical systems where human-life or safety is of utmost importance. However, the ever-increasing complexity and diversity of today's systems often evades the current capabilities of formal verification techniques. Formal verification practitioners face a tradeoff: spend considerable time manually guiding "automatic" model-checking using assume-guarantee reasoning, assumption tightening, constraint learning etc., to verify properties versus quickly maximize the number of partially-verified properties for a subset of model behaviors under relaxed assumptions or using bounded model checking. While the latter is acceptable for some systems but not safety-critical systems, the former is extremely important for safety-critical systems where maximizing reliability is the primary goal. There is an urgent need to extend research in model checking that enables verification of complex and challenging systems, and retains the push-button characteristic of model checking.

1.2 Design Space

Several design choices or *parameters* dictate system architecture and features in the very-early phases of system design. The system designer thoroughly evaluates different choices to decide core system capabilities with varying features, analyze system performance, or analyze a new system version against previous ones. Every unique combination of choices yields competing systems that differ in terms of assumptions, implementations, and configurations.

The designer narrows down on the final system design after a thorough qualitative and quantitative comparison of all competing systems. Each competing system must adhere to system specifications, plus per-design-choice specifications. The set of competing system designs under different parameter configurations constitute a *design space*. There are several advantages to design complex systems as design spaces, including but not limited to:

1. **Exhaustive enumeration**: Exhaustive enumeration and evaluation of all design alternatives. While domain-knowledge can help narrow the initial set of parameters, a thorough comparison is required to discard, prioritize, or amend design choices based on desired levels of system reliability.

2. **System behavior**: Provide better understanding of system behavior under different assumptions and architectures, and evaluate inter-component interactions with different parameter configurations.

3. **Explore tradeoffs**: Since every competing system must meet design specifications, analysis of the design space can help determine the parameter configurations that meet or violate specifications.

Figure 1.2 shows an air-traffic control system's design space for ensuring no loss of separation between four aircraft in the airspace. The ground controller and on-board controllers (if available) coordinate air traffic and trajectories to maintain safe flying distance. There can be multiple types of aircraft in the airspace. The *ground-separated aircraft* rely on the ground controller to maintain separation, while *self-separated aircraft* perform on-board separation-assurance reasoning with inputs from both the ground controller and on-board controller. Moreover, self-separated aircraft may communicate with other self-separated aircraft and the ground controller, whereas ground-separated aircraft only communicate with the ground controller. The exact trajectories and mitigation actions followed by the aircraft, in the case

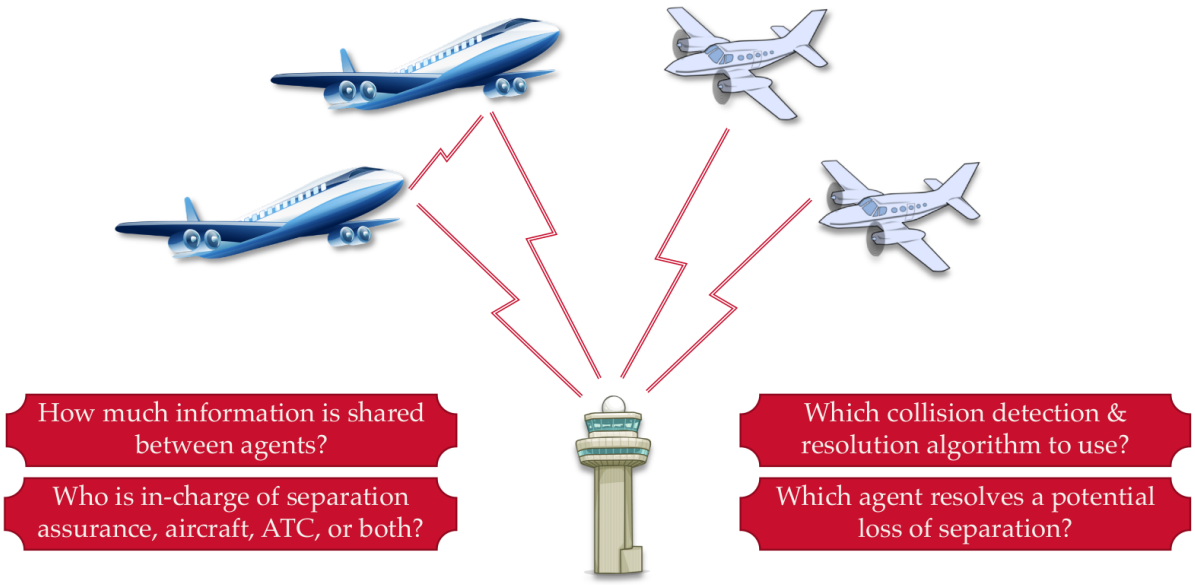


Figure 1.2: Design-space for an air-traffic control system.

of a potential loss of separation, depend on the choice of collision detection and recovery algorithms used by the ground and on-board controllers. Each design choice leads to a different airspace scenario based on: 1) a mix of aircraft types (ground-separated vs. self-separated); 2) different control algorithms (on ground vs. on-board), 3) communication configurations (communicate with ground vs. other aircraft), and; 4) different trajectory manipulation algorithms in case of a potential loss of separation. All these scenario variations constitute the design space of an air-traffic control system that maintains safe separation between different types of aircraft. The system designer thoroughly evaluates all design choices to determine scenarios that meet or violate design specifications. The specifications can range from low-level properties that specify system behavior for certain choices (e.g., ground-separated aircraft can always communicate with the ground controller, self-separated aircraft can always communicate with each other) to more encompassing properties that specify overall system behavior (e.g., there is no loss of separation between aircraft, a potential loss of

separation can always be detected by the ground or on-board controllers). The system designer can limit certain combinations of design choices, for e.g., choice of on-board control algorithm in a scenario with all ground-separated aircraft (these types of aircraft rely on the ground controller for separation assurance), however, such restrictions require very deep domain knowledge. Nevertheless, the different combinations of design choices often lead to a combinatorial explosion in the size of the design space.

1.3 Design-Space Exploration

The systematic analysis of discovering and evaluating design choices for a system under development is referred to *design-space exploration* (DSE). The exploration process is very complex since the same system functionality can be implemented in a variety of ways. The tradeoff between implementation choices, different parameter configurations, and evaluation metrics forms the basis of design-space exploration. Design-space exploration has many uses in the safety-critical systems engineering, including [KJS11]:

1. **Rapid prototyping**: The different parameter configurations generate a set of system prototypes. Analysis and profiling of these prototypes can impact design decisions while taking complex design dynamics into account.

2. **Objective optimization**: The different system prototypes can be compared in terms of power consumption, performance, cost, and safety. This helps eliminate inferior designs and collect a set of candidate prototypes that may be studied further.

3. **System integration**: The compatibility of multiple component behaviors and configurations of a system under varying parameter configurations can be analyzed to find a subset of configurations that satisfy design specifications.

Design-space exploration must be performed carefully due to the large number of design alternatives to be explored to determine which design configurations are 'optimal', i.e., meet design specifications. Design-space exploration may either be driven manually, with the system designer choosing parameter configurations based on intuition or domain-knowledge, or automatically, wherein a tool explores and evaluates all possible parameter configurations based on selected evaluation metrics. The manual approach to design-space exploration is tedious, error-prone, and does not scale for design spaces with millions of alternatives. An effective automatic design-space exploration framework consists of three ingredients [KJS11]:

1. **Design representation**: Automated design-space exploration requires a suitable formal representation of the design. The complex system may have a large number of design specification constraints that must be satisfied by every valid design alternative. The representation should be expressive enough to capture complex specification constraints: arithmetic operations, Boolean expressions, and datatype constraints.

2. **Exploration method**: The large number of alternatives make one-by-one ad-hoc enumeration of designs undesirable as some alternatives may be considered similar. The framework must provide a method for pruning the design space and quickly navigating to distinct and interesting designs, thereby, reducing the overall design-space exploration effort.

3. **Analysis techniques**: The framework must use machine-assisted techniques for discovering potential design candidates, and also check them against design specifications. These techniques must scale with the number and complexity of specifications while maintaining reasonable computational costs.

Design-space exploration has been effectively used in the engineering of several embedded systems [Pim17, YCY20], network architectures [LXX+09, ZBG20], communication protocols

[DR20a], processor protocols [HC13, SSZ11, KJCH19], and compiler optimizations [STF16]. Design-space exploration can be used to evaluate *functional* or *operational* correctness of different parameter configurations. The former pertains to design specifications that evaluate system alternatives with respect to safety and reliability (e.g., deadlock, starvation, etc.), and refers to the input-output behavior of the system, while the latter pertains to design specifications in terms of power consumption, performance, cost, etc. Both evaluation criteria explore a plethora of design choices ranging from choice of components, number of components, operating modes, choice of algorithms, and inter-component connections.

1.3.1 Taxonomy

The search for optimal design alternatives with respect to design criteria entails two distinct elements: 1) the evaluation of a single design alternative using defined evaluation metrics, and 2) the search strategy for covering all parameter configurations in the design space during the design-space exploration process. Methods for evaluating a single design in the design space broadly fall into three categories [Tho12]: measurements on a prototype implementation; simulation-based evaluation; and estimations based on some kind of analytical model. Each of these methods is characteristically distinct in terms of evaluation time and accuracy. The evaluation of prototype implementations provides the highest accuracy, but long development times prohibit evaluation of many design options. Estimations based on analytical models is fastest with limited accuracy since these models are typically unable to capture intricate system behavior. Simulation-based evaluation fills the gap between the other two methods: both highly accurate (but slower) and fast (but less accurate) simulation techniques are available. This tradeoff between accuracy and speed is very important for design-space exploration. The ability to evaluate a single design, and the ability to efficiently search the entire design space is critical for successful design-space exploration.

It is important to note that design-space exploration is a multi-objective optimization problem and is generally considered NP-hard [He10]. It finds design alternatives that are optimized in terms of design specifications: performance, cost, safety, etc. The search strategies to explore the design space can either be *open-loop* or *closed-loop*. In open-loop algorithms, the set of parameter configurations to evaluate is determined at the start of an exploration, and continues unchanged, regardless of the results obtained for individual designs. Therefore, open-loop search strategies are *exact* by their very nature. Exact methods, like those implemented using integer linear programming [NM, LGHT08] or branch and bound algorithms [PCC11], guarantee that all optimal parameter configurations that meet design specifications will be found. Closed-loop approaches attempt to find optimal parameter configurations without having to evaluate all possible configurations. These *heuristic* methods make a best-effort estimate by evaluating only a finite number of parameter configurations, however, they do not guarantee that all optimal configurations will be found. Examples of closed-loop methods are hill climbing, tableau search, simulated annealing, ant colony optimization, particle swarm optimization, and genetic algorithms. Successful design-space exploration requires a tradeoff between the methods for exploring parameter configurations and evaluation of single designs in terms of speed and accuracy. Exact methods are compute intensive and require clever design-space pruning to handle large design spaces but ensure exhaustive exploration of all parameter configurations. On the other hand, heuristic methods are often more scalable for large design spaces but may skip exploration of some parameter configurations. Regardless of the search strategy and single design evaluation method, the ultimate goal of design-space exploration is to provide 100% confidence that every design in the design space, i.e., all possible parameter configurations, is throughly evaluated, either by exact or heuristic methods, for functional and operational correctness.

1.3.2 Formal Methods

Exhaustive enumeration of all parameter configurations in a design space is very important for safety-critical systems. Heuristic methods are therefore not desirable for such systems, where 100% confidence is vital for ensuring all parameter configurations of a design space are explored. Moreover, simulation-based evaluation of single designs in the design space, although scalable, fails to guarantee high-levels of reliability. Therefore, formal methods play a major role in enabling design-space exploration of safety-critical systems. The three ingredients for automatic design-space exploration integrate nicely with the application of formal methods: 1) the formal representation of the design-space and specifications can be readily utilized; 2) formal methods can guarantee exhaustive exploration of all parameter configurations; and 3) formal method techniques, like model checking and theorem proving, can evaluate individual designs and check them against specifications. However, out-of-the-box application of formal methods may not scale to handle large design spaces. The combinatorial problem space limits the utility of formal verification, e.g., a design space with P Boolean parameters and N specifications may require $2^P \times N$ single design evaluations to ensure the design space is exhaustively explored.

Significant advances have enabled utilizing formal methods for design-space exploration: development of expressive formal languages for design representation [CCD+14], improvements to constraint satisfaction tools (e.g., Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solvers) [BT18] for design enumeration, and faster symbolic execution techniques [YFB+19] and faster model-checking algorithms [GR16] for design evaluation. However, most formal techniques are either application-specific and don't generalize across design spaces that arise in different domains, or fail to scale with the size of the design space. There is an urgent need to develop scalable formal tools and algorithms that generalize over several problem domains, and enable design-space exploration of safety-critical systems with 100%

confidence and safety-assurance. The use of formal methods for design-space exploration can be broadly categorized into *exploration* and *evaluation*.

1.3.2.1 Exploration

Given a formal representation of the design space and associated design constraints, formal methods can be used to prune the design space for potential candidates that may be individually evaluated. Each component (and their parameters) are associated with constraints that must be satisfied by every valid design solution. Moreover, each candidate design must satisfy global system constraints. Prior work utilizes *constraint satisfaction* solvers to find parameter configurations that meet operational constraints. Each design in the pruned design space is then evaluated using simulation techniques against functional design specifications. Other methods compile system specifications and component constraints into a satisfiability problem, which is tackled by a constraint solver to generate a single design solution that satisfies the specifications and constraints. The advances in constraint solver technology has significantly driven their use in exploration and design-space pruning. Another orthogonal technique is that of *parameter synthesis* [CGMT13] that generates parameter configurations that meet design specifications. The design representation specifies the datatype (Boolean, Integer, etc.) and range of parameters (minimum and maximum values). The synthesis algorithm then computes the values of all parameters for which the corresponding design meets specifications. Both techniques guarantee the enumeration of all valid design solutions that may be studied further.

1.3.2.2 Evaluation

Although formal exploration techniques provide high-levels of assurance for design enumeration, simulation and semi-formal techniques (like symbolic execution [YFB+19]) are the major workhorse for design evaluation. However, exhaustive evaluation of all behaviors of

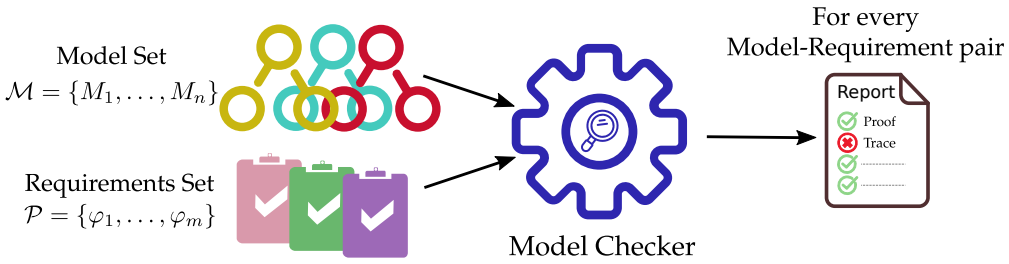


Figure 1.3: Model checking of multiple models and properties for design-space exploration of large design spaces. The model-checking engine outputs the verification result for every model-property pair.

a design using formal methods is vital for safety-critical systems. Several tools utilize theorem proving to check the design representation against specifications [KJS11], but require manual guidance for proof formulation. The design and specifications can be modeled as a constraint satisfaction problem at higher levels of abstraction [KJS11]. The constraint solver then automatically and exhaustively evaluates abstract system behaviors against specifications. However, such approaches are limited in terms of the types of specifications that can be checked; they only support specifications expressed in propositional or first-order logic.

Model checking can be used to evaluate designs against specifications. The ability to automatically check specifications written in more expressive logics (like linear temporal logic [Pnu77]) make model checking very useful in a design-space exploration framework. Moreover, the counterexamples provided by model checking can help influence design decisions and catch catastrophic bugs in early stages of design. The ability to explain why certain parameter configurations violate a design specification using counterexamples is an added bonus. Given M models (one for each design in the pruned design space) and N specifications, design-space evaluation using model checking requires $M \times N$ individual model-specification runs. We refer to this problem as **model checking of multiple models and properties**, or simply **model checking of design spaces** as shown in Figure 1.3. The model-checking algorithms inputs multiple parameter-configured models and design properties, and outputs

the model-checking result (pass or counterexample) for every model-property pair. However, due to the inherent complexity of model checking, existing tools and algorithms fail to handle large design spaces with thousands, or even hundreds of valid parameter configurations. *The work presented in this dissertation advances state-of-the-art in model checking to evaluate multiple models and properties for design-space exploration of safety-critical systems.* We focus on the closed-loop and exact exploration of the design-space, and utilize model checking to analyze individual design candidates against specifications for functional correctness.

1.4 Application Domains

The problem of model-checking multiple models and properties is not just limited to design-space exploration. Several industrial verification tasks entail: (**T1**) checking a design model against multiple properties, (**T2**) checking multiple design models against a single property, and (**T3**) checking multiple models against multiple properties. Naive application of model checking to accomplish these tasks is inherently complex and prohibits usage on large designs [GCM+16] with thousands of properties. The algorithms and techniques presented in this dissertation are applicable to several practical verification tasks including *functional verification* (**T1**), *incremental verification* (**T1**, **T2**), *regression verification* (**T2**), *equivalence checking* (**T1**), and *product-line verification* (**T3**).

1.4.1 Functional Verification

Functional verification is the process of demonstrating the functional correctness of a design with respect to design specifications. It is inherently complex because of the sheer volume of possible test-cases that need to be checked for a design, and takes the majority of time and effort in most large safety-critical hardware and/or software projects. Formal verification techniques, like model checking, can attempt to mathematically prove that cer-

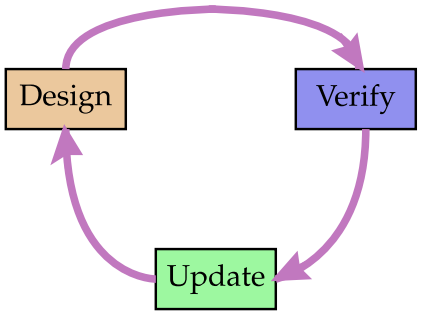


Figure 1.4: The verification workflow for systems where verification feedback constantly influences refinement of the design. A new design is re-verified against new or modified specifications after every refinement.

tain requirements are satisfied by the design, or that certain undesired behaviors (such as deadlock) cannot occur. Complex functional verification tasks often entail model-checking a large number of properties on the same design model. Despite the prevalence of such multi-property verification tasks, much research in model-checking has focused on the verification of individual properties [CCL+18]. How can we ensure that the verification effort expended to check a single property can be reused when checking thousands of properties on the same design? The work presented in this dissertation boosts the scalability of multi-property model checking of large and complex designs.

1.4.2 Incremental Verification

Modern approaches for the development of a hardware or software system require repeated design refinement based on verification feedback as shown in Figure 1.4. Despite the increasing effectiveness of model-checking tools, automatically re-verifying a design whenever a new revision is created is often not feasible using existing tools. When small changes are introduced into the design or the specification, for example due to a bug fix or an upgrade, the whole design needs to be re-verified, generally requiring the same amount of resources as for the initial verification [CIM+11]. Incremental verification aims at facilitat-

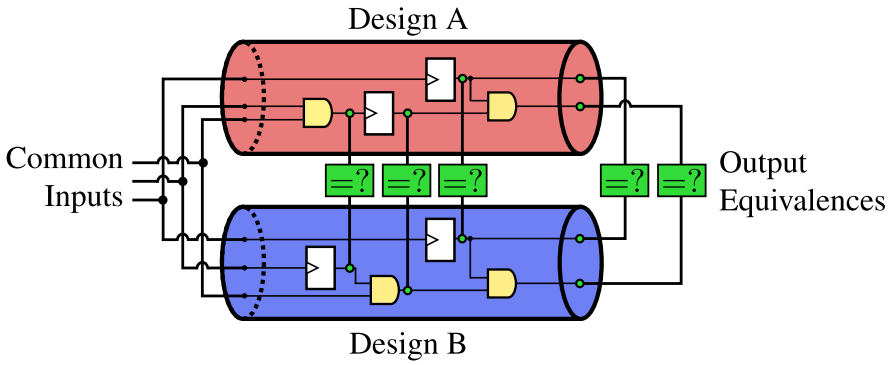


Figure 1.5: Equivalence checking by merging inputs and proving equivalence over outputs for two designs; proving internal equivalences boosts the scalability of equivalence checking. Each equivalence is a property to be model-checked against the composite model.

ing re-verification by reusing partial results from previous verification runs. The problem is especially acute when similar specifications are checked one-by-one on the same design (Section 1.4.1). Closed-loop design-space verification may entail verifying design models with very minor differences or related specifications; incremental verification greatly benefits verification by reusing previous results. The best option for scalable incremental verification is to reuse results from previous model-checking runs, and only verify the change. However, what prior results to reuse, and how to reuse remains an open question. We present new algorithms that enable efficient incremental verification in this dissertation.

1.4.3 Equivalence Checking

Equivalence checking is a process in electronic design automation (EDA) to formally prove that two representations of a design exhibit exactly the same behavior. The two design representations can have different implementations, but must be equivalent in terms of input/output behavior. Equivalence checking is used to prove design equivalence across different levels of abstraction (Verilog vs. And-Inverter graphs, C-language vs. assembly instructions, etc.), or between different versions of the same design. The equivalence checking algorithm merges the inputs of the two designs, and uses model checking to prove output

equivalences as shown in Figure 1.5. Equivalence checking also benefits from proving internal equivalences between the two designs, i.e., pairwise equivalence between intermediate points in the model. The output equivalences and internal equivalences form properties that are verified using model checking against the composite model with merged inputs. The two designs are equivalent when all output equivalences hold for the composite model. Equivalence checking is the most popular formal verification technique for hardware [Str09]. The problem is considered "easier" than functional verification as it circumvents the problem of specifying requirements, but requires the same or more verification effort; failure to prove even a single output equivalence within resource limits stalls equivalence checking. A related technique is that of *redundancy removal* [CBMK11] wherein redundancies added at design-time for boosting performance, error resilience, and debugging are identified and removed prior to functional verification; it is well known that redundancy removal can often make an intractable verification problem tractable [MBPK05] due to reduction in design size. Internal equivalences are identified to form properties that are evaluated using model checking. Both equivalence checking and redundancy removal mandate scalable verification of multiple properties. The work presented in this dissertation heavily impacts multi-property verification, and enables equivalence checking and redundancy removal of very large designs.

1.4.4 Product-line Verification

Product-line technology is increasingly used in mission-critical and safety-critical applications. A software product line is a family of software systems that differ in terms of features, i.e., a design space. Software product line verification entails evaluating every possible feature combinations with respect to design specifications. The product line is modeled formally with each feature represented by Boolean combination of parameters to the system. An example of a software product line is shown in Figure 1.6. The product line may comprise of a multitude of products; every unique feature parameter configuration is a prod-

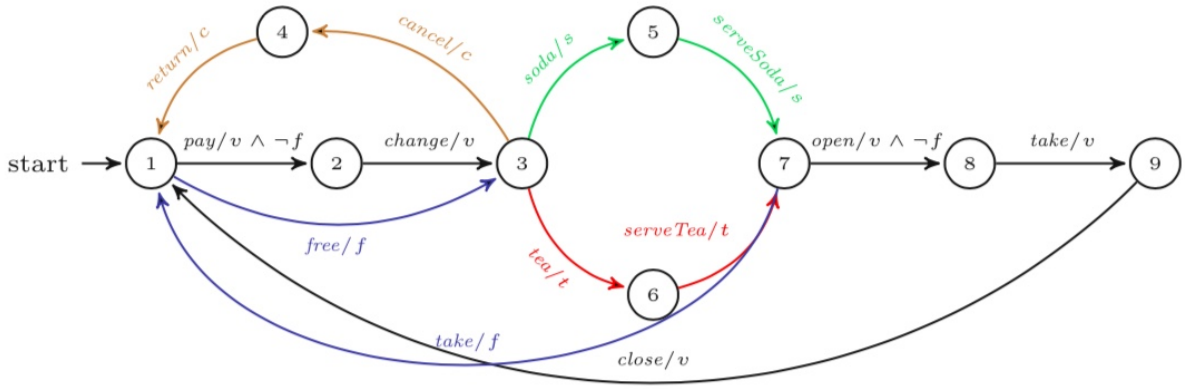


Figure 1.6: The formal representation of a software product line for a vending machine [CCH+12]. The individual features are represented by Boolean combinations of parameters to the system.

uct. Model-checking can be applied to analyze and verify software product lines [CCH+12] by reducing the problem to model checking of multiple models and properties. The formal model for each product (generated by parameter instantiation) can be checked against design specifications. Moreover, model-checking technology can be used to analyze selected products after pruning the product line using specialized techniques [DR18, AvW+13]. The techniques presented in this dissertation scale out-of-the-box application of model checking to product-line verification.

1.5 Contributions

We make several contributions that enable efficient design-space exploration using model checking. We focus on the more general problem of model checking multiple models and properties, and optimize design-space model checking, and other related application domains by making significant contributions across every step of the model-checking process.

1.5.1 Design-Space Reduction

Model checking a design space entails checking multiple models and properties. Given a formal representation of the design space and properties expressing system specifications, we present algorithms that automatically prune the design space by finding inter-model relationships and property dependencies [DR18]. Our design-space reduction technique is compatible with off-the-shelf model checkers, and only requires checking a small subset of models and properties to provide verification results for every model-property pair in the original design space. We make the following contributions (Chapter 2):

1. A fully automated, general, and scalable algorithm for checking design spaces; it can be applied to LTL model checking problems without major modifications to the system designers' verification workflow.

2. Modification to the general model-checking procedure of sequentially checking properties against a model to a dynamic procedure; the next property to check is chosen to maximize the number of yet-to-be-checked properties for which the result can be determined from inter-property dependencies.

3. Formal definition of two new formula-ordering heuristics with a comparative analysis of their individual and combined impact on performance.

1.5.2 Incremental Verification

We observe that sequential enumeration of the design space generates models with small incremental differences. Typical model-checking algorithms do not take advantage of this information; they end up re-verifying "already-explored" state spaces across models. We present algorithms that learn and reuse information from solving related models against a

property in sequential model-checking runs [DR17, DR20b]. We make the following contributions (Chapter 3):

1. Fully automated, general, and scalable incremental model-checking algorithm for checking design spaces that reuses model-checking information across runs.

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. Overview of locality-sensitive hashing [AI08] techniques to mine model specifications expressed as And-Inverter-Graph circuits.

4. 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 by FuseIC3 and improve overall performance.

1.5.3 Multi-Property Verification

Design space model-checking tasks often mandate checking several properties. State-of-the-art tools do not optimally exploit subproblem sharing between properties, leaving an opportunity to save verification resource via concurrent verification of "nearly-identical" properties. The verification effort expended for one property in a group can be directly reused to accelerate the verification of the others. We present a near-linear runtime algorithm for partitioning properties into provably high-affinity groups for individual model-checking tasks [DBI+19]. We make the following contributions (Chapter 4):

1. An online algorithm to partition properties based on structural information, readily available in low-level design representations, into provably high-affinity groups.

2. Efficient procedure to compute cones of influence of multiple properties, and data structures that allow CPU-speed comparison between properties.

3. A systematic methodology to learn semantic information, and refine high-structural-affinity groups in a localization abstraction framework.

4. An optimized multi-property localization abstraction solution that is resistant to performance slowdown that may occur when verifying very-large property groups.

1.5.4 Parallel Orchestration

We optimize parallel verification to maximize the benefits of our proposed techniques. Model checking tools utilize parallelism, either in portfolio mode where different algorithm strategies run concurrently, or in partitioning mode where disjoint property subsets are verified independently. However, both approaches often degrade into highly-redundant work across processes, or under-utilize available processes. We propose methods to minimize redundant computation, and dynamically optimize work distribution when checking multiple properties for individual models [DBK+20]. We make the following contributions (Chapter 5):

1. We present a scalable property partitioning algorithm, extending [DBI+19] to guarantee *complete* utilization of available processes with provable partition quality.

2. We propose parallel scheduling improvements, such as resource-constrained irredundant group iteration, incremental repetition, and group decomposition to dynamically cope with more-difficult groups or slower workers.

3. We address irredundant strategy exploration of a localization portfolio in a sequential redundancy removal framework, which we have found to be the most-scalable strategy to prove non-inductive redundancies.

4. We propose improvements to *semantic group partitioning* within localization. To our knowledge, this is the first published approach to mutually-optimize property partitioning and strategy exploration within a multi-property localization abstraction portfolio.

CHAPTER 2. DESIGN-SPACE REDUCTION

Modern system design often requires comparing several design alternatives over a large design space. The combinatorial size of the design space hinders out-of-the-box application of formal verification. Each design in the design space is modeled formally, and evaluated against a set of design specifications. The different models arise out of a need to weigh different design choices, to check core capabilities of versions with varying features, or to analyze a future version against previous ones. Model checking can compare the different models for functional correctness, however, applying model checking off-the-shelf may not scale due to the large size of the design space for today's complex systems because of several reasons. First, building and validating models for individual designs in the design space is tedious, and becomes intractable when the number of designs is large. Moreover, maintaining and updating models is extremely error-prone: a design update may require modifying several models. Second, the number of models to be verified individually using model checking can be very large. The number of models may be reduced by restricting certain combinations of parameter configurations or by identifying redundant designs, i.e., two models that exhibit same behavior under different parameter configurations, however, such restrictions and reductions either require very deep domain knowledge or expensive preprocessing to analyze design behaviors (e.g. using simulation). Third, the number of properties to check against individual models may be too large, or extremely hard for a model checker to verify in a reasonable amount of time. In this chapter, we present algorithms and techniques to scale the applicability of model checking for design-space exploration by answering the following questions: 1) *How to represent the design space and associated parameters, and design specifications formally that allows easier maintainability, design updates, and is amenable*

to model checking? 2) *How to identify redundant design-alternatives in the design space to reduce the number of models to check?* 3) *How to minimize the model-checking effort for evaluating a single model against several design specifications?*

The rest of the chapter is organized as follows: Section 2.1 gives a high-level overview of our contributions to efficiently model and model-check design-spaces, and contrasts with related work. Section 2.2 gives background information and introduces modeling formalisms for design spaces with parameters. Section 2.3 presents our algorithm to minimize the number of design-alternatives to check for a design space, and also minimize the number of properties to evaluate for each model, albeit providing the complete model-checking verdict for every individual model-property pair in the design space. We experimentally evaluate our modeling technique and algorithms on large-scale design spaces for NASA's NextGen air traffic control system, Boeing's AIR6110 wheel braking system in Section 2.4. Section 2.5 concludes the chapter by highlighting future optimizations and applicability of our algorithms and modeling techniques to other verification scenarios.

2.1 Introduction

In the early phases of design, there are frequently many different models of the system under development [BLBM07, GCM+16, MCG+15] constituting a *design space*. We may need to evaluate different design choices, to check core capabilities of system versions with varying feature-levels, or to analyze a future version against previous ones in the product line. The models may differ in their assumptions, implementations, and configurations. We can use model checking to aid system development via a thorough comparison of the set of system models against a set of properties representing requirements. Model checking, in combination with related techniques like fault-tree analysis, can provide an effective comparative analysis [MCG+15, GCM+16]. The classical approach checks each model one-by-one,

as a set of independent model-checking runs. For large and complex design spaces, performance can be inefficient or even fail to scale to handle the combinatorial size of the design space. Nevertheless, the classical approach remains the most widely used method in industry [BCFP+15, GCM+16, JMN+14, MCG+15, MNR+13]. Algorithms for family-based model checking [CHSL11, CCH+12] mitigate this problem but their efficiency and applicability still depends on the use of custom model checkers to deal with model families.

We assume that each model in the design space can be parameterized over a finite set of parametric inputs that enable/disable individual assumptions, implementations, or behaviors. It might be the case that for any pair of models the assumptions are dependent, their implementations contradict each other, or they have the same behavior. Since the different models of the same system are related, it is possible to exploit the known relationships between them, if they exist, to optimize the model checking search. These relationships can exist in two ways: relationships between the models, and relationships between the properties checked for each model.

We present an algorithm that automatically prunes and dynamically orders the model-checking search space by exploiting inter-model relationships. The algorithm, Discover Design-Space Dependencies (D^3), reduces both the number of models to check, and the number of LTL properties that need to be checked for each model. Rather than using a custom model checker, D^3 works with any off-the-shelf checker. This allows practitioners to use state-of-the-art, optimized model-checking algorithms, and to choose their preferred model checker, which enables adoption of our method by practitioners who already use model checking with minimum change in their verification workflow. We reason about a set of system models, corresponding to a design space, by introducing the notion of a *Combinatorial Transition System* (CTS). Each individual model, or *instance*, can be derived from the CTS by configuring it with a set of parameters. Each transition in the CTS is enabled/disabled by the parameters. We model check each instance of the CTS against sets of properties.

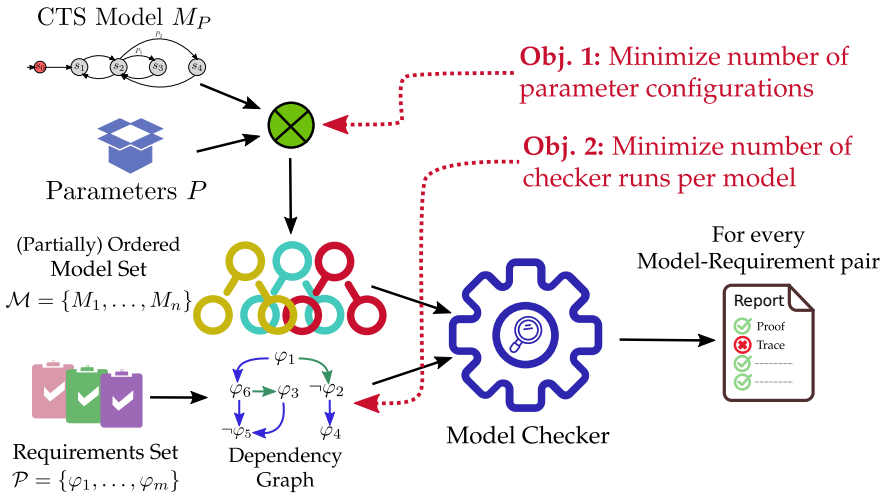


Figure 2.1: Typical verification workflow for design-space reduction using the D^3 algorithm thats minimize the number of parameter configured models and the number of properties checked per model, but provides results for every model-property pair.

We assume the properties are in Linear Temporal Logic (LTL) and are independent of the choice of parameters, though not all properties may apply to all instances. D^3 preprocesses the CTS to find relationships between parameters and minimizes the number of instances that need to be checked to produce results for the whole set. It uses LTL satisfiability checking [RV07] to determine the dependencies between pairs of LTL properties, then reduces the number of properties that are checked for each instance. D^3 returns results for every model-property pair in the design space, aiming to compose these results from a reduced series of model-checking runs compared to the classical approach of checking every model-property pair. Figure 2.1 shows the workflow for model checking design spaces using the D^3 algorithm. We demonstrate the industrial scalability of D^3 using a set of 1,620 real-life, publicly-available SMV-language benchmark models with LTL specifications; these model NASA's NextGen air traffic control system [CCD+14, GCM+16, MCG+15]. We also evaluate the property-dependence analysis separately on real-life models of Boeing AIR 6110 Wheel Braking System [BCFP+15] to evaluate D^3 in multi-property verification workflows.

2.1.1 Related Work

One striking contrast between D^3 and related work is that D^3 is a preprocessing algorithm, does not require custom modeling, and works with any off-the-shelf LTL model checker. Parameter synthesis [CGMT13] can generate the many models in a design space that can be analyzed by D^3; however existing parameter synthesis techniques require custom modeling of a system. We take the easier path of reasoning over an already-restricted set of models of interest to system designers. D^3 efficiently compares any set of models rather than finding all models that meet the requirements. Several parameter synthesis approaches designed for parametric Markov models [DJJ+15, DJJ+16, HHZ11, QDJ+16] use PRISM and compute the region of parameters for which the model satisfies a given probabilistic property (PCTL or PLTL); D^3 is an LTL-based algorithm. Parameter synthesis of a parametric Markov model with non-probabilistic transitions can generate the many models that D^3 can analyze. In multi-objective model checking [BDK+14, EKVY07, FKN+11, KNPQ13], given a Markov decision process and a set of LTL properties, the algorithms find a controller strategy such that the Markov process satisfies all properties with some set probability. Differently from multi-objective model checking, which generates "trade-off" Pareto curves, D^3 gives a boolean result. After making early-stage-design choices using D^3, multi-objective model checking can verify selected configurations. The parameterized model checking problem (PCMP) [EK00] deals with infinite families of homogeneous processes in a system; in our case, the models are finite and heterogeneous. Specialized model-set checking algorithms [DR17] can check the reduced set of D^3 processed models.

In multi-property model checking, multiple properties are checked on the same system. Existing approaches simplify the task by algorithm modifications [CCG+09, CGM+10], SAT-solver modifications [KNPH06, KN12], and property grouping [CN11a, CCL+17]. The inter-

property dependence analysis of D^3 can be used in multi-property checking. We compare D^3 against the *affinity*[CN11a] based approach to property grouping.

Product line verification techniques, e.g., with Software Product Lines (SPL), also verify parametric models describing large design spaces. We borrow the notion of an *instance*, from SPL literature [RS10, SS09]. An extension to NuSMV in [CHSL11] performs symbolic model checking of feature-oriented CTL. The symbolic analysis is extended to the explicit case and support for feature-oriented LTL in [CCH+12, CCS+13a]. The work most closely related to ours is [DASBW15] where product line verification is done without a family-based model checker. D^3 outputs model-checking results for every model-property pair in the design space (e.g. all parameter configurations) without dependence on any *feature* whereas in SPL verification using an off-the-shelf checker, if a property fails then it isn't possible to know which models *do* satisfy the property [CHS+10, DASBW15].

2.1.2 Contributions

The preprocessing algorithm presented is an important stepping stone to smarter algorithms for checking large design spaces. Our contributions are summarized as follows:

1. A fully automated, general, and scalable algorithm for checking design spaces; it can be applied to LTL model checking problems without major modifications to the system designers' verification workflow.

2. Modification to the general model-checking procedure of sequentially checking properties against a model to a dynamic procedure; the next property to check is chosen to maximize the number of yet-to-be-checked properties for which the result can be determined from inter-property dependencies.

3. Comparison of our novel inter-property dependence analysis to existing work in multi-property verification workflows [CN11a].

4. Extensive experimental analysis using real-life benchmarks; all reproducibility artifacts and source code are publicly available.[1]

5. Formal definition of two new formula-ordering heuristics with a comparative analysis of their individual and combined impact on performance.

2.2 Preliminaries

2.2.1 Temporal Logic Model Checking

Definition 2.2.1. A *labeled transition system* (LTS) is a system model of the form $M = (\Sigma, S, s_0, L, \delta)$ where,

1. Σ is a finite alphabet, or set of atomic propositions,

2. S is a finite set of states,

3. $s_0 \in S$ is an initial state,

4. $L : S \to 2^{\Sigma}$ is a labeling function that maps each state to the set of atomic propositions that hold in it, and

5. $\delta : S \to S$ is the transition function.

A computation trace, or *run* of LTS M is a sequence of states $\pi = s_0 \to s_1 \to \ldots \to s_n$ over the word $w = L(s_0), L(s_1), \ldots, L(s_n)$ such that $s_i \in S$ for $0 \leq i \leq n$, and $(s_i, s_{i+1}) \in \delta$ for $0 \leq i < n$.

Linear temporal logic (LTL) reasons over linear computation traces. LTL formulas are composed of a finite set Σ of atomic propositions, the Boolean connectives $\neg, \wedge, \vee$, and $\to$, and the temporal connectives $\mathcal{U}$ (until), $\mathcal{R}$ (release), $\mathcal{X}$ (also called $\bigcirc$ for "next time"),

[1]Raw experimental results available at http://temporallogic.org/research/TACAS18/

$\Box$ (also called $\mathcal{G}$ for "globally") and $\Diamond$ (also called $\mathcal{F}$ for "in the future"). We define LTL formulas inductively.

Definition 2.2.2. For every $p \in \Sigma$, atomic proposition p is an LTL formula. If φ and ψ are LTL formulas, then so are:

- $\neg\varphi$
- $\varphi \rightarrow \psi$
- $\mathcal{X}\varphi$
- $\varphi \wedge \psi$
- $\varphi \,\mathcal{U}\, \psi$
- $\Box\varphi$
- $\varphi \vee \psi$
- $\varphi \,\mathcal{R}\, \psi$
- $\Diamond\varphi$

Definition 2.2.3. We interpret LTL formulas over computations of the form $\pi : \omega \rightarrow 2^{\Sigma}$, where ω is used in the standard way to denote the set of non-negative integers. We define $\pi, i \models \varphi$ (computation π at time instant $i \in \omega$ satisfies LTL formula φ) as follows:

- $\pi, i \models p$ for $p \in \Sigma$ *iff* $p \in \pi(i)$.
- $\pi, i \models \neg\varphi$ *iff* $\pi, i \not\models \varphi$.
- $\pi, i \models \varphi \wedge \psi$ *iff* $\pi, i \models \varphi$ and $\pi, i \models \psi$.
- $\pi, i \models \varphi \vee \psi$ *iff* $\pi, i \models \varphi$ or $\pi, i \models \psi$.
- $\pi, i \models \mathcal{X}\varphi$ *iff* $\pi, i = 1 \models \varphi$.
- $\pi, i \models \varphi\mathcal{U}\psi$ *iff* $\exists j \geq i$, such that $\pi, j \models \psi$ and $\forall k, i \leq k < j$, we have $\pi, k \models \varphi$.
- $\pi, i \models \varphi\mathcal{R}\psi$ *iff* $\forall j \geq i$, *iff* $\pi, j \not\models \psi$, then $\exists k, i \leq k < j$, such that $\pi, k \models \varphi$.
- $\pi, i \models \Box\varphi$ *iff* $\forall j \geq i$, we have $\pi, j \models \varphi$.
- $\pi, i \models \Diamond\varphi$ *iff* $\exists j \geq i$, such that $\pi, j \models \varphi$.

We take $\models(\varphi)$ to be the set of computations that satisfy φ at time 0, i.e., $\{\pi : \pi, 0 \models \varphi\}$. We define the prefix of an infinite computation π to be the finite sequence starting from the zeroth time step, $\pi_0, \pi_1, \ldots, \pi_i$ for some $i \geq 0$. Let Π denote the set of all computations of an LTS M starting from the zeroth time step. Given an LTL property φ and a LTS M, M *models* or *satisfies* φ, denoted $M \models \varphi$, iff $\forall \pi \in \Pi$, we have $\pi, 0 \models \varphi$, i.e., φ holds in all possible computation paths of M.

2.2.2 Design-Space Model Checking

Definition 2.2.4. A *parameter* P_i is a variable with the following properties.

1. The *domain* of P_i, denoted $[\![P_i]\!]$, is a finite set of possible assignments to P_i.
2. Parameter P_i is *set* by assigning a single value from $[\![P_i]\!]$, i.e. $P_i = d_{P_i} \in [\![P_i]\!]$. A non-assigned parameter is considered *unset*.
3. Parameter setting is static, i.e., it does not change during a run of the system.

Let P be a finite set of parameters. $|P|$ denotes the number of parameters. For each $P_i \in P$, $|P_i|$ denotes the size of the domain of P_i. Let $Form(P)$ denote the set of all Boolean formulas over P generated using the BNF grammar $\varphi ::= \top \mid P_i == \mathrm{D}$ and $\mathrm{D} ::= P_{i_1} \mid P_{i_2} \mid \ldots \mid P_{i_n}$; for each $P_i \in P$, $n = |P_i|$, and $[\![P_i]\!]=\{P_{i_1}, P_{i_2}, \ldots, P_{i_n}\}$. Therefore, $Form(P)$ contains $\top$ and equality constraints over parameters in P.

Definition 2.2.5. A *combinatorial transition system* (CTS) is a combinatorial system model $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$, such that $(\Sigma, S, s_0, L, \delta)$ is a LTS and

1. P is a finite set of parameters to the system, and
2. $L_P : \delta \to Form(P)$ is function labeling transitions with a guard condition.

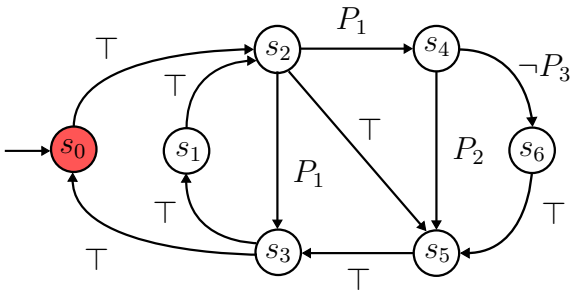


Figure 2.2: A combinatorial transition system $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$ with Boolean parameters $P = \{P_1, P_2, P_3\}$ that enable (or disable) state transitions in δ

We limit the guard condition over a transition to $\top$ or an equality constraint over a single parameter for simpler expressiveness and formalization. However, there can be multiple transitions between any two states with different guards. A transition is *enabled* if its guard condition evaluates to true, otherwise, it is *disabled*. A label of $\top$ implies the transition is always enabled. A *possible run* of a CTS is a sequence of states $\pi_P = s_0 \xrightarrow{\nu_1} s_1 \xrightarrow{\nu_2} \ldots \xrightarrow{\nu_n} s_n$ over the word $w = L(s_0), L(s_1), \ldots, L(s_n)$ such that $s_i \in S$ for $0 \leq i \leq n$, $\nu_i \in Form(P)$ for $0 < i \leq n$, and $(s_i, s_{i+1}) \in \delta$ and $(s_i, s_{i+1}, \nu_{i+1}) \in L_P$ for $0 \leq i < n$, i.e., there is transition from s_i to s_{i+1} with guard condition ν_{i+1}. A *prefix* α of a possible run $\pi_P = \alpha \xrightarrow{\nu_i} \ldots \xrightarrow{\nu_n} s_n$ is also a possible run.

Example 2.2.1. A Boolean parameter has domain $\{true, false\}$. Figure 2.2 shows a CTS with Boolean parameters $P = \{P_1, P_2, P_3\}$. For brevity, guard condition P_i==$true$ is written as P_i, while P_i==$false$ is written as $\neg P_i$. A transition with label P_1 is enabled if P_1 is set to *true*. Similarly, a label of $\neg P_3$ implies the transition is enabled if P_3 is set to *false*.

Definition 2.2.6. A *parameter configuration* c for a set of parameters P is a k-tuple $(d_{P_1}, d_{P_2}, \ldots, d_{P_k})$, for $k = |P|$, that sets each parameter in P, i.e., for every $1 \leq i \leq k$, $P_i = d_{P_i}$ and $d_{P_i} \in [\![P_i]\!]$ is a setting. The set of all possible configurations $\mathbb{C}$ over P is equal to $P_1 \times P_2 \times \ldots \times P_k$ where $\times$ denotes the cross product. The setting for P_i in configuration c is denoted by $c(P_i)$.

A *configured run* of a CTS M_P over a configuration c, or c-run, is a sequence of states $\pi_{P(c)} = s_0 \xrightarrow{\nu_1} s_1 \xrightarrow{\nu_2} \ldots \xrightarrow{\nu_n} s_n$ such that $\pi_{P(c)}$ is a possible run, and $c \vdash \nu_i$ for $0 < i \leq n$, where $\vdash$ denotes propositional logic satisfaction of the guard condition ν_i under parameter configuration c. Given a CTS M_P and a parameter configuration c, a state t is *reachable* iff there exists a c-run such that $s_n = t$, denoted $s_0 \xrightarrow[c]{*} t$, i.e., t can be reached in zero or more transitions. A transition with guard ν is *reachable* iff $(s_j, s_{j+1}, \nu) \in L_P$, $(s_j, s_{j+1}) \in \delta$, and $s_0 \xrightarrow[c]{*} s_j$.

Definition 2.2.7. An *instance* of a CTS $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$ for parameter configuration c is a LTS $M_{P(c)} = (\Sigma, S, s_0, L, \delta')$ where $\delta' = \{t \in \delta \mid c \vdash L_P(t)\}$.

Given a LTL property φ and a CTS $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$, the *model checking problem* for M_P is to find all parameter configurations $c \in \mathbb{C}$ over P such that φ holds in all c-runs of M_P, or all computation paths of LTS $M_{P(c)}$.

Definition 2.2.8. Given a CTS M_P with parameters P_i, P_j, and a parameter configuration c, P_j is *dependent on* P_i, denoted $P_j \rightsquigarrow_c P_i$, iff

- In all possible runs with a transition guard over P_j, a transition with guard over P_i appears before a transition with guard over P_j, and
- In all configured runs, the setting for P_i in c makes transitions with guard conditions over P_j unreachable.

Example 2.2.2. In Figure 2.2, if P_1 is set to false, execution never reaches the transition labeled $\neg P_3$. Therefore, if configuration $c = (false, true, true)$ then $P_3 \rightsquigarrow_c P_1$.

Definition 2.2.9. A *universal model* U is a LTS that generates all possible computations paths over its atomic propositions.

```
MODULE system
VAR
        p: boolean;
        q: boolean;
        ...
FROZENVAR
        parameter: boolean;
        ...
INIT
        parameter = 0;
        ...
TRANS
        p & !q & parameter ->
                p & q;
        p & !q & !parameter ->
                !p & q;
        ...
```

Model 1

```
MODULE system
VAR
        p: boolean;
        q: boolean;
        ...
FROZENVAR
        parameter: boolean;
        ...
INIT
        parameter = 1;
        ...
TRANS
        p & !q & parameter ->
                p & q;
        p & !q & !parameter ->
                !p & q;
        ...
```

Model 2

```
MODULE system
VAR
        p: boolean;
        q: boolean;
        ...
FROZENVAR
        parameter: boolean;
        ...
INIT
        parameter = PARAMETER_CONF;
        ...
TRANS
        p & !q & parameter ->
                p & q;
        p & !q & !parameter ->
                !p & q;
        ...
```

CTS Model

Figure 2.3: Model 1 and Model 2 written in the SMV language can be combined to form a CTS model with the use of `PARAMETER_CONF` preprocessor directive.

2.2.3 Temporal Logic Satisfiability

Theorem 2.2.1 (LTL Satisfiability)**.** [RV07] Given a LTL property φ and a universal model U, φ is satisfiable if and only if $U \not\models \neg\varphi$.

This theorem reduces LTL satisfiability checking to LTL model checking. Therefore, φ is satisfiable when the model checker finds a counterexample.[2]

2.2.4 Modeling a Design Space

Efficient modeling of a design space using a combinatorial transition system requires language constructs to deal with parameters. Since our goal is to use an existing model checker, language extensions are outside the scope of this work. An alternative way to add parameters to any system description is by utilizing the C preprocessor (`cpp`). Given a set of parameters P, and a combinatorial model M_P, each run of the preprocessor with a configuration $c \in \mathbb{C}$ generates an instance $M_{P(c)}$. Figure 2.3 demonstrates generating a CTS from two related

[2]This is why we do not consider CTL; CTL satisfiability is EXPTIME-complete and cannot be accomplished via linear time CTL model checking.

SMV models. Model 1 and Model 2 differ in the initial configuration of the parameter. The corresponding CTS replaces the parameter initiation with the `PARAMETER_CONF` preprocessor directive. The `cpp` is run on the CTS model with `#define PARAMETER_CONF 0`, and `#define PARAMETER_CONF 1` to generate the two models.

2.3 Discovering Design-Space Dependencies

In this section we describe the D^3 algorithm. Our approach speeds up model checking of combinatorial transitions systems by preprocessing of the input instances; it therefore increases efficiency of both BDD-based and SAT-based model checkers. The problem reduction is along two dimensions: number of instances, and number of properties.

2.3.1 Individual Model Redundancies

Given a set of parameters P, a combinatorial transition system M_P, and a property φ, M_P is model checked by sending, for all parameter configuration $c \in \mathbb{C}$, instance $M_{P(c)}$ to the LTS model checker, along with the property φ. The output is aggregated for $|\mathbb{C}|$ runs of the model checker, and all parameter configurations c, such that $M_{P(c)} \models \varphi$ are returned. In principle, parameters can be encoded as state variables, and the parametric model can be posed as one big model-checking obligation, however there are caveats.

1. State space explosion before any useful results are obtained.

2. The counterexample generated from one run of the model checker gives a single undesirable configuration.

Our goal is to make the classical approach of individual-model checking more scalable as the design space grows by intelligently integrating possible dependencies between parameter configurations. A combinatorial transition system is a directed unweighted graph with Boolean

constraints on its edges. The states and transitions in the CTS represent the vertices and edges of the graph, respectively. A possible run of a CTS is a path through the graph. The number of instances that need to be checked can be reduced by exploiting the order in which guarded transitions appear along a possible run.

Lemma 2.3.1. *Given a CTS* $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$ *with parameters* $A, B \in P$*, if* $B \rightsquigarrow_c A$ *for some parameter configuration* c*, then there does not exist any possible run of* M_P *with prefix* $\alpha = s_0 \xrightarrow{*} s_i \xrightarrow{\nu_B} s_j \xrightarrow{*} s_k \xrightarrow{\nu_A} s_l$*, where* ν_A *and* ν_B *are guards over parameters* A *and* B*, resp., and* $s_i, s_j, s_k, s_l \in S$*, i.e., a transition with guard over parameter* B *does not appear before a transition with guard over parameter* A*.*

Proof. Follows from Definition 2.2.8. Let Π_P be the set of all possible runs of CTS M_P. Let $\pi_p \in \Pi_P$ be a possible run with prefix $s_0 \xrightarrow{*} s_i \xrightarrow{\nu_B} s_j \xrightarrow{*} s_k \xrightarrow{\nu_A} s_l$. In configured run $\pi_{P(c)}$, transition with guard over B is reachable irrespective of the setting for A, violating our premise of $B \rightsquigarrow_c A$. Therefore, when $B \rightsquigarrow_c A$, a transition with guard over B never appears before a transition with guard over A in all possible runs. □

As a corollary to Lemma 2.3.1, there also do not exist possible runs with transition guards only over B (and no other $P_i \in P$). Therefore, given a CTS M_P with states $s_i, s_j, s_k, s_l \in S$ and parameters $A, B \in P$, if $B \rightsquigarrow_c A$ for some parameter configuration c, then all possible runs of M_P have one of the following prefixes:

1. $s_0 \xrightarrow{*} s_i \xrightarrow{\nu_A} s_j \xrightarrow{*} s_k \xrightarrow{\nu_B} s_l$ (guard over A before guard over B)
2. $s_0 \xrightarrow{*} s_i \xrightarrow{\nu_A} s_j \xrightarrow{*} s_k \xrightarrow{\nu_A} s_l$ (guards only over A)
3. $s_0 \xrightarrow{*} s_i \xrightarrow{*} s_j \xrightarrow{*} s_k \xrightarrow{*} s_l$ (guards neither over A nor B)

Similarly, if $A \rightsquigarrow_c B$ for some parameter configuration c, then all possible runs of M_P have one of the following prefixes:

1. $s_0 \xrightarrow{*} s_i \xrightarrow{\nu_B} s_j \xrightarrow{*} s_k \xrightarrow{\nu_A} s_l$ (guard over B before guard over A)

2. $s_0 \xrightarrow{*} s_i \xrightarrow{\nu_B} s_j \xrightarrow{*} s_k \xrightarrow{\nu_B} s_l$ (guards only over B)

3. $s_0 \xrightarrow{*} s_i \xrightarrow{*} s_j \xrightarrow{*} s_k \xrightarrow{*} s_l$ (guards neither over A nor B)

Therefore, when A and B are not dependent, there is no possible run with transition guards over both A and B. Note that for a CTS M_P with $A, B \in P$, if A and B are dependent, then either $A \rightsquigarrow_c B$ or $B \rightsquigarrow_c A$ but not both for any configuration c. We only show formalization for $B \rightsquigarrow_c A$; $A \rightsquigarrow_c B$ follows directly.

Theorem 2.3.2 (Redundant Instance)**.** *Given a CTS $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$ with parameters $A, B \in P$ such that $B \rightsquigarrow_c A$ for some configuration c, and a LTL property φ, there exist configurations $c_1, c_2, \ldots c_k \in \mathbb{C}$ for $k = |B|$ such that*

- *$c_i(A) = c(A)$ for $0 < i \leq k$, and*
- *$c_i(B) = d_{B_i} \in [\![B]\!]$ for $0 < i \leq k$ and $[\![B]\!] = \{d_{B_1}, d_{B_2}, \ldots, d_{B_k}\}$*

For such configurations $M_{P(c_1)} \models \varphi \equiv M_{P(c_2)} \models \varphi \equiv \ldots \equiv M_{P(c_k)} \models \varphi$.

Proof. From Lemma 2.3.1 we know that if $B \rightsquigarrow_c A$, then a transition with guard over B never appears before a transition with guard over A in all possible runs of M_P. Also, from Definition 2.2.8 we know that if $B \rightsquigarrow_c A$ then there exists a parameter setting for A that makes transitions with guard over B unreachable in configured runs of M_P. Let this setting be $c(A) = d_A$ for $d_A \in [\![A]\!]$. For parameter configurations $c_1, c_2, \ldots c_k \in \mathbb{C}$ such that $c_i(A) = c(A)$ for $0 < i \leq k$, execution never reaches the transition with guard over B in all c-runs $\pi_{P(c_i)}$ (if it did, $B \not\rightsquigarrow_c A$). Irrespective of the setting to B, the same set of states are reachable, and c-runs $\pi_{P(c_i)}$, for $0 < i \leq k$, are identical. Therefore, $M_{P(c_1)} \models \varphi \equiv M_{P(c_2)} \models \varphi \equiv \ldots \equiv M_{P(c_k)} \models \varphi$. □

function FINDUP (M_P, $\hat{c}$)

Input: M_P = CTS $(\Sigma, S, s_o, L, \delta, P, L_P)$, $\hat{c}$ = partial **configuration**

Output: P_u = unset parameter **queue**

1: **if** all parameters are set in $\hat{c}$: **return** $\emptyset$ *# do not proceed*

2: P_u = empty *# initially P_u is empty.*

3: **traverse** M_P using depth-first traversal

4: **if** $t \in \hat{\delta}$ is reachable **and** $L_P(t)$ is **undefined** :

 # $L_P(t)$ is undefined when its parameter is NOT set in partial configuration $\hat{c}$.

5: enqueue ($p : L_P(t)$ is guard over p) in P_u

6: **return** P_u

Figure 2.4: FINDUP algorithm to find unset parameters in a partially configured CTS.

Theorem 2.3.2 allows us to reduce the number of model checker runs by exploiting redundancy between instances. The question that needs to be answered is how to find dependent parameters? One way would be to use domain knowledge to decide which parameter configurations effect one another; we instead calculate this automatically. A *partial parameter configuration*, $\hat{c}$, is a parameter configuration in which not all parameters have been set. Given a CTS $M_P = (\Sigma, S, s_0, L, \delta, P, L_P)$, for a transition $t \in \delta$, such that $L_P(t) = \nu$, the guard ν is

- *defined*, if its corresponding parameter is set in $\hat{c}$, and
- *undefined*, otherwise.

A defined guard evaluates to true when $\hat{c} \vdash L_P(t)$, or false when $\hat{c} \nvdash L_P(t)$. Algorithm FINDUP (**Find** Unset **P**arameters) in Figure 2.4 solves the dual problem of finding independent parameters. It takes as input a CTS M_P and a partial parameter configuration $\hat{c}$,

and returns unset parameters for which guard conditions are undefined and their corresponding transitions are reachable. It traverses (depth-first) the CTS starting from a node for the initial state s_0. During traversal, an edge (transition) $t = (s_i, s_j)$ connects two nodes (states) $s_i, s_j \in S$ if $t \in \delta$ and $\hat{c} \vdash L_P(t)$. The edge is disconnected if $t \notin \delta$ or $\hat{c} \nvdash L_P(t)$. Since M_P is defined relationally in the annotated SMV language with preprocessor directives (§ 2.2.4), in the worst case, FindUP takes polynomial time in the number of symbolic states and transitions. From an implementation point of view, FindUP invokes the `cpp` for parameter settings in $\hat{c}$ on the input model, and parses the output for unset parameters.

Lemma 2.3.3. FindUP *returns unset parameters $P_i \in P$ for all reachable transitions $t \in \delta$ such that guard $L_P(t)$ is a guard over P_i, and is undefined.*

Proof. Depth-first traversal (DFT) of the CTS visits nodes (states) in the order they appear in possible runs of M_P, while branching and backtracking for nodes that have more than one adjacent node. Consider a possible run $\pi_P = s_0 \xrightarrow{\nu_1} s_1 \xrightarrow{\nu_2} s_2 \ldots \xrightarrow{\nu_n} s_n$ such that $\hat{c} \vdash \nu_1$ and ν_2 is undefined. From Definition 2.2.6, transition $t = (s_1, s_2)$ is reachable, while all transitions after s_2 are not reachable. Hence, the unset parameter for guard ν_2 is added to the return set of FindUP. Depth-first traversal (DFT) allows scanning all possible runs of a CTS without enumerating all of them. The traversal backtracks whenever a transition with an undefined guard is visited. Therefore unset parameters for all edges $t \in \delta$ that can be reached starting from an initial node in DFT, and for which $L_P(t)$ is undefined, are returned by FindUP. □

Algorithm GenPC (**Gen**erate **P**arameter **C**onfigurations) in Figure 2.5 uses FindUP as a subroutine to recursively find parameter configurations that need to be checked. It takes as input a CTS M_P, queue of unset parameters P_u, and a partial parameter configuration $\hat{c}$. Initially, $\hat{c}$ contains no set parameters and P_u =FindUP(M_P, $\hat{c}$). Upon termination of GenPC, $\hat{\mathbb{C}}$ contains the set of partial parameter configurations that need to be checked. On every iteration, GenPC picks a parameter p from P_u, assigns it a value from its domain

```
1: configuration set Ĉ  # initially empty
function GenPC (M_P, P_u, ĉ)
Input: M_P = CTS (Σ, S, s_o, L, δ, P, L_P), P_u = unset parameter queue
       ĉ = partial config  # initially empty
2: while P_u not empty :
3:   p = dequeue element from P_u
4:   for each p_d in [[p]] :  # iterate on possible assignments to p
5:     set parameter p to p_d in ĉ  # make an assignment to parameter p
6:     P_u = FindUP(M_P, ĉ)  # get unset parameters
7:     if P_u is empty :  # all parameters set
8:       add ĉ to Ĉ and return
9:     else:  # set unset parameters
10:      GenPC(M_P, P_u, ĉ)  # call function recursively to assign unset parameters
```

Figure 2.5: GenPC algorithm to generate parameter configurations to be checked.

$[\![p]\!]$ in $\hat{c}$, and uses FindUP to find unset parameters in CTS M_P. If the returned unset parameter queue is empty, $\hat{c}$ added to $\hat{\mathbb{C}}$. Otherwise, GenPC is called again with the new unset parameter queue.

Theorem 2.3.4 (GenPC is sound)**.** *Given a CTS M_P with parameters $A, B \in P$, if there exists a partial configuration $\hat{c} \in \hat{\mathbb{C}}$ with $\hat{c}(A) = d_{A_n} \in [\![A]\!]$ and B unset, then there exist configurations $c_1, c_2, \ldots c_k \in \mathbb{C}$ for $k = |B|$ such that*

- *$c_i(A) = \hat{c}(A)$ for $0 < i \leq k$, and*
- *$c_i(B) = d_{B_i} \in [\![B]\!]$ for $0 < i \leq k$ and $[\![B]\!] = \{d_{B_1}, d_{B_2}, \ldots, d_{B_k}\}$*

for which $B \rightsquigarrow_{c_i} A$.

Proof. We prove the contrapositive of the statement. When parameters A and B are not dependent, there is no possible run of M_P that contains transitions with guards over both A and B (follows from Lemma 2.3.1). Therefore, every possible run of M_P is of the form $\pi_P = s_0 \xrightarrow{\nu_1} s_1 \xrightarrow{\nu_2} \dots \xrightarrow{\nu_n} s_n$ where all ν_i, for $0 < i \leq n$, are either *true* or guards over $P \setminus \{A, B\}$, or guards over either parameter A or B. A call to FindUP with a setting for A, returns unset parameter B (follows from Lemma 2.3.3) that is then set to every value in $[\![B]\!]$ domain in GenPC. Therefore, if A is set to $d_{A_n} \in [\![A]\!]$ in the call to FindUP, then $\hat{\mathbb{C}}$ contains $k = |B|$ partial configurations $\hat{c}_i$ such that

- $\hat{c}_i = d_{A_n}$ for $0 < i \leq k$
- $\hat{c}_i(B) = d_{B_i} \in [\![B]\!]$ for $0 < i \leq k$ and $[\![B]\!] = \{d_{B_1}, d_{B_2}, \dots, d_{B_k}\}$

Therefore, when A and B are not dependent, for every setting of A, $\hat{\mathbb{C}}$ contains $|B|$ partial parameter configurations; one for every different setting of B. □

Theorem 2.3.5 (GenPC is complete)**.** *Given a CTS M_P with parameters $A, B \in P$, if there exist configurations $c_1, c_2, \dots c_k \in \mathbb{C}$ for $k = |B|$ such that*

- $c_i(A) = d_{A_n}$ *for* $0 < i \leq k$ *and* $d_{A_n} \in [\![A]\!]$*, and*
- $c_i(B) = d_{B_i} \in [\![B]\!]$ *for* $0 < i \leq k$ *and* $[\![B]\!] = \{d_{B_1}, d_{B_2}, \dots, d_{B_k}\}$

for which $B \rightsquigarrow_{c_i} A$, then $\exists \hat{c} \in \hat{\mathbb{C}}$ with $\hat{c}(A) = d_{A_n}$ and B unset.

Proof. Let $A, B \in P$ be dependent parameters such that $B \rightsquigarrow_c A$ for some configuration c and $c(A) = d_{A_n} \in [\![A]\!]$. When $B \rightsquigarrow_c A$, there is no possible run of M_P in which a transition with guard over B appears before a transition with guards over A (follows from Lemma 2.3.1). A call to FindUP with a partial configuration $\hat{c}$ such that $\hat{c}(A) = d_{A_n}$ does not return B as an unset parameter (follows from Lemma 2.3.3). Therefore, GenPC generates a partial configuration $\hat{c} \in \hat{\mathbb{C}}$ with $\hat{c}(A) = d_{A_n}$ and B unset. □

GenPC returns partial configurations $\hat{c} \in \hat{\mathbb{C}}$ over parameters. A partial configuration $\hat{c}$ is converted to a parameter configuration c by setting the unset parameters in $\hat{c}$ to an arbitrary value from their domain. Note that this operation is safe since the arbitrarily set parameters are not reachable in the instance $M_{P(c)}$. As a result of this operation, $\hat{\mathbb{C}}$ contains configurations c that have all parameters set to a value from their domain.

Theorem 2.3.6 (Minimality)**.** *The minimal set of parameter configurations is* $\hat{\mathbb{C}}$*.*

Proof. Suppose towards contradiction that $\hat{\mathbb{C}}$ is not minimal. Then there is a minimal set of configurations $\hat{\mathbb{C}}^*$ with $\hat{\mathbb{C}}^* \subset \hat{\mathbb{C}}$. Take $c \in \hat{\mathbb{C}} \setminus \hat{\mathbb{C}}^*$. Now $c \notin \hat{\mathbb{C}}^*$ implies that there exists a $c_i \in \hat{\mathbb{C}} \cap \hat{\mathbb{C}}^*$ for which $B \rightsquigarrow_{c_i} A$ with $c_i(A) = c(A)$ and $c_i(B) \neq c(B)$, i.e., the setting of A in c_i makes transitions with guards over B unreachable and hence the setting of B does not effect configured runs. Since $\hat{\mathbb{C}}$ contains both c and c_i, then from the correctness of GenPC, $B \not\rightsquigarrow_{c_i} A$ (follows from Theorem 2.3.4 and Theorem 2.3.5). This contradicts our premise, and thus $\hat{\mathbb{C}}$ must be minimal. □

2.3.2 Identifying Property Dependencies

In model checking, properties describe the intended behavior of the system. Usually, properties are iteratively refined to express the designer's intentions. For small systems, it can be manually determined if two properties are dependent on one another. However, practically determining property dependence for large and complex systems requires automation. Given a set of properties $\mathcal{P}$, and LTS M, an off-the-shelf model checker is called $N = |\mathcal{P}|$ times.

In order to check all properties in $\mathcal{P}$, a straightforward possibility is to generate a grouped property φ_g given by the conjunction of all properties $\varphi_i \in \mathcal{P}$, i.e., $\varphi_g = \bigwedge_i \varphi_i$. However, the straightforward approach may not scale [CN11a] due to

1. State-space explosion due to orthogonal cone-of-influences of properties.

2. Need for additional analysis of individual properties one-by-one in order to discriminate failed ones and generate individual counterexamples.

3. Computational cost of verifying grouped properties in one run can be significantly higher than verifying individual properties in a series of runs.

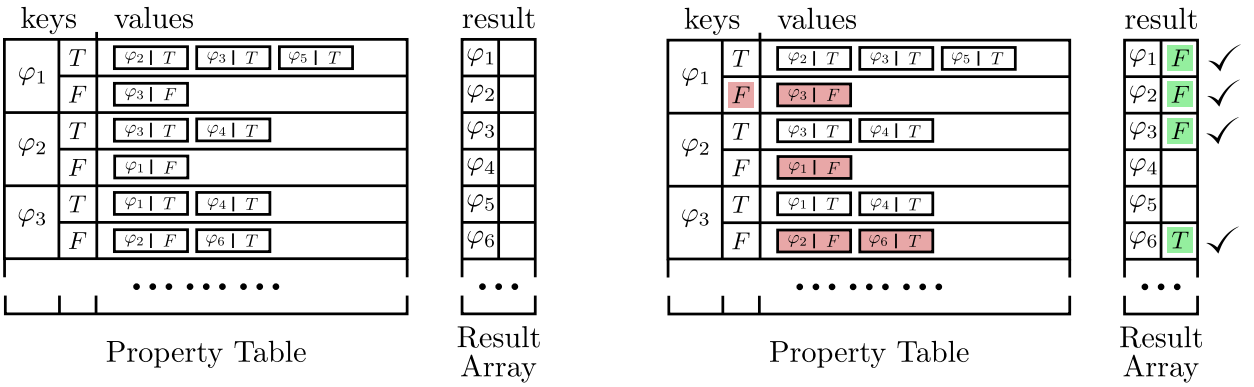
Our goal is to minimize the number of properties checked by intelligently using implicit dependencies between LTL properties. For two LTL properties φ_1 and φ_2 *dependence* can be characterized in four ways: $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \rightarrow \neg\varphi_2)$, $(\neg\varphi_1 \rightarrow \varphi_2)$, and $(\neg\varphi_1 \rightarrow \neg\varphi_2)$. However, knowing which implication holds for a pair of properties is a difficult task, simply because they may have been introduced by different verification engineers at different times. Theorem 2.3.7 allows us to find dependencies automatically.

Theorem 2.3.7 (Property Dependence)**.** *For two LTL properties φ_1 and φ_2 dependence can be established by model checking with universal model U.*

Proof. There are a total of four cases to consider: $(\varphi_1 \rightarrow \varphi_2)$, $(\varphi_1 \rightarrow \neg\varphi_2)$, $(\neg\varphi_1 \rightarrow \varphi_2)$, and $(\neg\varphi_1 \rightarrow \neg\varphi_2)$. We show proof for $(\varphi_1 \rightarrow \varphi_2)$, since other dependencies follow a similar proof. From Theorem 2.2.1 we know that a LTL formula φ is satisfiable iff $U \not\models \neg\varphi$. Therefore, formula φ is unsatisfiable iff $U \models \neg\varphi$. Let $\varphi = \neg(\varphi_1 \rightarrow \varphi_2)$. Therefore, if $U \models (\varphi_1 \rightarrow \varphi_2)$ then $\neg(\varphi_1 \rightarrow \varphi_2)$ is unsatisfiable or $(\varphi_1 \rightarrow \varphi_2)$ is valid, and vice-versa. □

The dependencies learned as a result of Theorem 2.3.7 have implications on the verification workflow. For instance, if $\varphi_1 \rightarrow \varphi_2$ is valid, then for a model M, if $M \models \varphi_1$ then $M \models \varphi_2$. Of particular interest are $(\varphi_1 \rightarrow \varphi_2)$, $(\neg\varphi_1 \rightarrow \varphi_2)$, and $(\neg\varphi_1 \rightarrow \neg\varphi_2)$ because they allow use of previous counterexamples (for $(\varphi_1 \rightarrow \neg\varphi_2)$, even if property φ_1 is *true*, there is no counterexample to prove that property φ_2 is *false* for model M).

The pairwise property dependencies are stored in a property table as shown in Figure 2.6a. Each row in the table is a $(key, value)$ pair. For LTL properties φ_1, φ_2, and φ_3 in $\mathcal{P}$, if



(a) Initial layout of the property table

(b) Results that can be determined based on knowing φ_1 does not hold in model M

Figure 2.6: Property table to store dependence between every LTL property pair in property set $\mathcal{P}$. Each row entry in the table is a $(key, value)$ pair. Multiple entries with the same *key* have been merged in a single row. E.g., if $\varphi_1 \rightarrow \varphi_2$, the table contains a row $(\varphi_1 : T, \varphi_2 : T)$ implying that if property φ_1 holds for model M then property φ_2 also holds.

$(\varphi_1 \rightarrow \varphi_2)$ is valid, then the table contains a row $(\varphi_1 : T, \varphi_2 : T)$ implying that if φ_1 holds for a model M then φ_2 also holds. Similarly, for $(\neg\varphi_3 \rightarrow \neg\varphi_2)$ the table entry $(\varphi_3 : F, \varphi_2 : F)$ implies that if φ_3 doesn't hold for M then φ_2 doesn't hold. Algorithm CheckRP (**Check** **R**educed **P**roperties) in Figure 2.7 takes as input a LTS M, a set of LTL properties $\mathcal{P}$, and a property table T over $\mathcal{P}$. CheckRP selects an unchecked LTL property φ, checks whether φ holds in M, and stores the outcome. Based on the outcome, it uses the property table to determine checking results for all dependent properties and stores them. For example, in Figure 2.6b, if $M \not\models \varphi_1$, then $M \not\models \varphi_3$, $M \not\models \varphi_2$, and $M \models \varphi_6$. The LTL property to check is selected using two heuristics **H1** and **H2**.

2.3.2.1 Maximum Dependence Heuristic (H1)

The tabular layout of property dependencies is used to calculate the number of dependencies for each property. The unchecked LTL property with the most right-hand side entries is selected for verification against the model. If $\mathcal{U} \subseteq \mathcal{P}$ are unchecked properties in table D,

1: **array** results *# initially empty*

function CHECKRP (M, $\mathcal{P}$, T)

Input: M = LTS $(\Sigma, S, s_0, L, \delta)$, $\mathcal{P}$ = **set** of LTL properties, D = property **table**

2: **while** unchecked properties remain :

3: φ = get unchecked property

4: outcome = MODELCHECK(M, φ) *# outcome = T if $M \models \varphi$, else F*

5: **set** $S = \{(\varphi : \text{outcome})\}$

6: **while** S is **not** empty :

7: $(p : \text{result})$ = pop element from S

8: results[p] = result *# update result*

9: $S = S \cup$ unchecked properties dependent on $(p :$ result) in D

Figure 2.7: CHECKRP algorithm to check LTL properties against a model.

the next LTL property to check is then

$$\varphi \in \mathcal{U} : count(\varphi) = max(\{count(\psi) \mid \forall \psi \in \mathcal{U}\})$$

where $count(x) = |D[x : T] \cup D[x : F]|$ returns the number of dependencies for a LTL property in table D, and $max(S)$ returns the largest element from S.

2.3.2.2 Property Grouping Heuristic (H2)

Most model-checking techniques are computationally sensitive to the cone-of-influence (COI) size. Grouping properties based on overlap between their COI can speed up checking. Property *affinity* [CN11a, CCL+17] based on *Jaccard Index* can compare the similarity between COI. For two LTL properties φ_i and φ_j, let V_i and V_j, respectively, denote the

function D^3 (M_P, $\mathcal{P}$)

Input: M_P = CTS $(\Sigma, S, s_0, L, \delta, P, L_P)$, $\mathcal{P}$ = **set** of LTL properties

1: configuration **set** $\hat{\mathbb{C}}$ *# initially empty*

2: parameter **queue** P_u = FINDUP(M_P, _)

3: $\hat{\mathbb{C}}$ = GENPC(M_P, P_u, _) *# See § 2.3.1*

 # generate property table, see § 2.3.2

4: property **table** D *# initially empty*

5: **for** every property pair (φ_1, φ_2) in $\mathcal{P}$: *# iterate over all property pairs*

6: check if φ_1 and φ_2 are dependent and add to property table D

7: **for each** c in $\hat{\mathbb{C}}$: *# check configured instances*

8: generate **instance** $M_{P(c)}$ *# See § 2.2.4*

9: **array** results *# initially empty*

10: CHECKRP($M_{P(c)}$, $\mathcal{P}$, D) *# See § 2.3.2*

11: **return** results

Figure 2.8: Discovering Design-Space Dependencies (D^3) algorithm.

variables in their COI with respect to a model M. The affinity α_{ij} for φ_i and φ_j is given by

$$\alpha_{ij} = \frac{|V_i \cap V_j|}{|V_i| + |V_j| - |V_i \cap V_j|}$$

If α_{ij} is larger than a given threshold, then properties φ_i and φ_j are grouped together. The model M is then checked against $\varphi_i \wedge \varphi_j$. If verification fails, then LTL properties φ_i and φ_j are checked individually against model M.

2.4 Experimental Analysis

Our revised model checking procedure D^3 is shown in Figure 2.8. D^3 takes as input a CTS M_P and a set of LTL properties $\mathcal{P}$. It uses GENPC to find the parameter configurations

that need to be checked. It then generates a property table to store dependencies between LTL properties. Lastly, CheckRP checks each instance against properties in $\mathcal{P}$. Results are collated for every model-property pair.

2.4.1 Benchmarks

We evaluate D^3 on two benchmarks derived from real-world case studies.

2.4.1.1 Air Traffic Controller (ATC) Models

These are a set of 1,620 real-world models representing different possible designs for NASA's NextGen air traffic control (ATC) system. In previous work, this set of models were generated from a contract-based, parameterized nuXmv model; individual-model checking enabled their comparative analysis with respect to a set of requirements for the system [GCM+16]. In the formulation of [GCM+16], the checking problem for each model is split in to five phases.[3]. In each phase, all 1,620 models are checked. For our analysis and to gain better understanding of the experimental results, we categories the phases based on the property verification results (unsat if property holds for the model, and sat if it does not). Each of the 1,620 models can be seen as instances of a CTS with seven parameters. Each of the 1620 instances is checked against a total of 191 LTL properties. The original nuXmv code additionally uses OCRA [CDT13] for compositional modeling, though we do not rely on its features when using the generated model-set.

2.4.1.2 Boeing Wheel Braking System (WBS) Models

These are a set of seven real-world nuXmv models representing possible designs for the Boeing AIR 6110 wheel braking system [BCFP+15]. Each model in the set is checked against ∼200 LTL properties. However, the seven models are not generated from a CTS. We

[3]For a detailed explanation we refer the reader to [GCM+16]

Table 2.1: Timing results of 1,620 models for each phase using individual-model checking, and D^3 algorithm. For individual-model checking, **Time** indicates model checking time, whereas, for D^3, **Time** indicates preprocessing time + model checking time.

Phase	**Property Mix**	**Properties**	**Model Checking Time** (in hours)		**Speedup**	**Overall Speedup**
		Total (median)	Individual	D^3		
I	UNSAT	25 (24)	6.02	4.02	1.5×	4.5×
II	UNSAT	29 (19)	12.76	5.17	2.5×	
III	UNSAT	29 (1)	139.79	14.80	9.4×	
IV	SAT+UNSAT	54 (43)	24.81	14.25	1.7×	1.8×
V	SAT+UNSAT	54 (44)	31.15	16.03	1.9×	
TOTAL		191	214.53	54.27	4.0×	-

evaluate D^3 against this benchmark to evaluate performance on multi-property verification workflows, and compare with existing work on property grouping [CN11a].

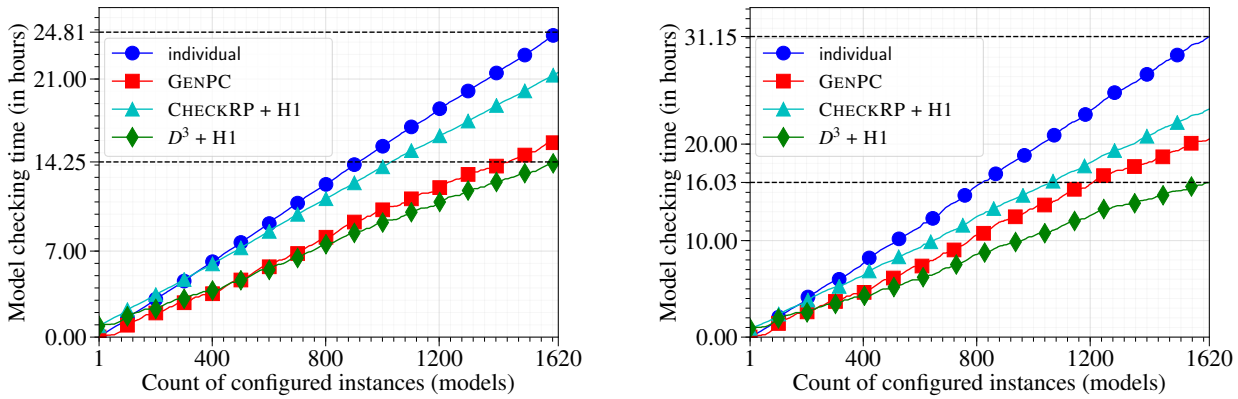
2.4.2 Experiment Setup

D^3 is implemented as a preprocessing script in ∼2,000 lines of Python code. We model check using NUXMV 1.1.1 with the IC3-based back-end. 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 has dedicated access to a node, which guarantees that there are no resource conflicts with other jobs running on that node.

2.4.3 Experimental Results

2.4.3.1 Air Traffic Controller (ATC) Models

All possible models are generated by running the C preprocessor (`cpp`) on the annotated composite SMV model representing the CTS. Table 2.1 summarizes the results for complete verification of the ATC design space: 191 LTL properties for each of 1,620 models.



(a) Phase IV **(b)** Phase V

Figure 2.9: Cumulative time for checking each model for all properties one-by-one (individual), checking reduced instances for all properties (GenPC), checking all models for reduced properties (CheckRP + H1), and checking reduced instances for reduced properties (D^3 + H1) for phases IV (Figure 2.9a) and V (Figure 2.9b).

Compared to individual model checking, wherein every model-property pair is checked one-by-one, verification of the ATC design space using D^3 is 4.0× faster. It reduces the the 1,620 models in the design space to 1,028 models. D^3 takes roughly three hours to find dependencies between LTL properties for all phases. Dependencies established are local to each model-checking phase and are computed only once per phase. The number of reduced LTL properties checked for each model in a phase vary; we use CheckRP with the Maximum Dependence heuristic (H1). Although the logical dependencies are global for each phase, the property verification results vary for different models. In phases containing UNSAT properties, speedup achieved by D^3 varies between 1.5× to 9.4×; since all properties are true for the model, only $(\varphi_1 : T \rightarrow \varphi_2 : T)$ dependencies in the property table are used. A median of one property is checked per model in phase III. For phases IV and V, D^3's performance is consistent as shown in Figure 2.9a and Figure 2.9b, respectively.

Interesting Observation. D^3 requires a minimum number of models to be faster than individual-model checking. When the design space is small, individually checking the models is faster than verifying using D^3. This is due to the fact that D^3 requires an initial set-up time. The number of models after which D^3 is faster is called the "*crossover point*". For the benchmark, the crossover happens after ∼120 models. As the number of models, and the relationships between them increase, the time speedup due to the D^3 algorithm also increases. Moreover, the number properties checked by D^3 for every model vary. This is due to the fact that the next property to check is chosen to maximize the number of yet-to-be-checked properties for which the result can be determined from inter-property dependencies. Figure 2.10a and Figure 2.10b shows the number of LTL properties checked per model in phases IV and V, respectively. Note that all 54 LTL properties are never checked for a model in either of the phases. D^3 dynamically reduces the number of properties checked for individual models, but provides results for all properties nevertheless.

Overall. From the initial problem of checking 1,620 models against 191 LTL properties, D^3 checks 1,028 models with a median of 129 properties per model (45% reduction of design space). Once D^3 terminates, the model-checking results for each model are compared using the data analysis technique of [GCM+16].

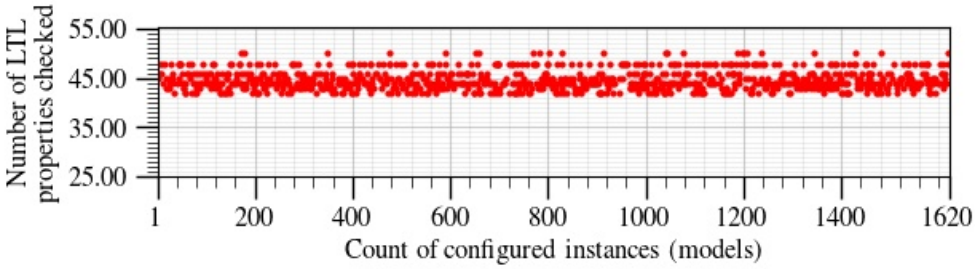
2.4.3.2 Boeing Wheel Braking System (WBS) Models

LTL Properties for each of the seven models are checked using four algorithms:

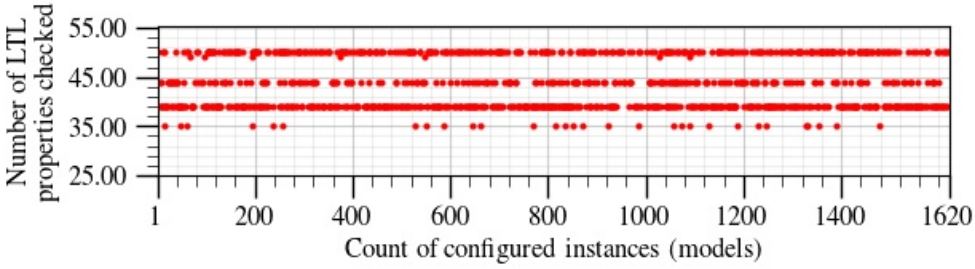
1) Single: properties are checked one-by-one against the model,

2) CheckRP: properties are checked using inter-property dependencies,

3) CheckRP + Maximum Dependence (H1): unchecked property with the maximum dependent properties as per inter-property dependencies is checked,



(a) Phase IV



(b) Phase V

Figure 2.10: Number of properties dynamically checked for individual models for NASA's NextGen air-traffic control system's design space using the D^3 algorithm for phases IV (Figure 2.10a) and V (Figure 2.10b)

4) CheckRP + Property Grouping (H2): properties are pairwise grouped and the unchecked pair with the maximum dependent properties is checked.

Figure 2.11 summarizes the results of verifying properties for every model. On every call to the model checker, a single or grouped LTL property is checked. CheckRP is successful in reducing the number of checker runs by using inter-property dependencies. The Maximal Dependences (H1) and Property Grouping (H2) heuristics improve the performance of CheckRP, the former more than the latter. The timing results for each algorithm is shown in Table 2.2.

Analysis. For H2, we limit our experiments to pairwise groupings, however, larger groupings may be possible (trade-off required between property inter-dependencies and

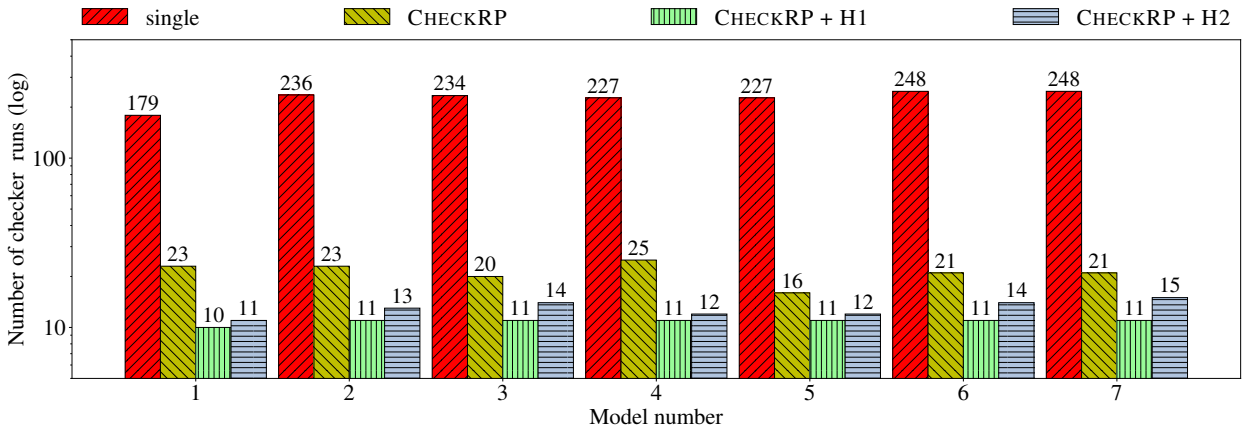


Figure 2.11: Number of calls made to the model checker to verify all properties in the set for a model. Every call to the checker verifies one property: single or grouped. For CheckRP, multiple property results are determined (based on inter-property dependencies) on every checker run. Heuristics H1 and H2 improve performance of CheckRP.

groupings). It takes ∼50 minutes to establish dependence between properties for a model, which is much higher than checking them one-by-one without using CheckRP. This brings us back to the question of estimating a *crossover point*. However, as the number of models increase for the same set of properties, CheckRP starts reaping benefits. Nevertheless, the CheckRP algorithm is suited for multi-property verification in large design spaces and provides significant end-to-end speedup.

2.5 Summary and Discussion

Our formalisms to model design spaces using Combinatorial transitions systems (CTS) is extremely versatile, allow for easier maintainability, and is amenable to model checking. We present an algorithm, Discovering Design-space Dependencies (D^3), to increase the efficiency of LTL model checking for large design spaces. It is successful in reducing the number of models that need to be verified, and also the properties verified for each model. In contrast to software product line model checking techniques using an off-the-shelf checker, D^3 returns

Table 2.2: Timing results (in seconds) for performance of D^3's inter-property dependence analysis. A property: single or grouped, is verified on each checker run. Overall time indicates the total time to verify all properties for a model.

Model	Single		CheckRP		CheckRP+H1		CheckRP+H2	
	Overall Time	Checker Runs	Overall Time	Checker Runs	Overall Time	Checker Runs	Overall Time	Checker Runs
1	17.81	179	2.92	23	1.28	10	2.05	11
2	64.37	236	9.35	23	3.94	11	5.67	13
3	54.22	234	7.11	20	3.40	11	4.97	14
4	53.18	227	9.71	25	3.41	11	5.89	12
5	61.02	227	6.86	16	4.01	11	5.58	12
6	68.24	248	8.34	21	3.93	11	5.34	14
7	58.40	248	7.74	21	3.39	11	5.98	15

the model-checking results for all models, and for all properties. D^3 is general and extensible; it can be combined with optimized checking algorithms implemented in off-the-shelf model checkers. We demonstrate the practical scalability of D^3 on a real-life benchmark models. We calculate a crossover point as a crucial measure of when D^3 can be used to speed up checking. D^3 is fully automated and requires no special input-language modifications; it can easily be introduced in a verification work-flow with minimal effort. Heuristics for predicting the cross-over point for other model sets are a promising topic for future work.

Design-space pruning is extremely essential for exact design-space exploration methods. Our techniques quickly prune the design space for scalable model checking. The front-end techniques presented in this chapter can benefit from advanced model-checking back-ends that can reuse verification artifacts across different model-checking runs. Traditionally, these incremental model-checking algorithms have been limited in applicability: they either save too much information across runs, or are not practical for large designs. The different models generated by D^3 can be sequentially checked by incremental algorithms to boost model-checking performance. Instead of restarting verification for the next model, the algorithms may "salvage" verification artifacts from prior runs and achieve significant end-to-end

speedups. In the next chapter, we look under the covers of a model checker to optimize model-checking search for design spaces.

CHAPTER 3. INCREMENTAL VERIFICATION

Combinatorial transitions systems (Section 2.2) provide an extremely efficient methodology to model design spaces with parameters that enable or disable transitions between reachable states. Each unique combination of parameter configurations generate a model that represents a valid individual design in the design space. Some of the designs maybe redundant and may be pruned by the D^3 algorithm presented in Chapter 2. The remaining models are then checked sequentially using off-the-shelf model checkers against properties expressing system specifications. Although model checking the pruned design space is considerably faster compared to checking every model in the original design space, specialized model-checking algorithms can greatly benefit the overall design-space exploration process.

We observe that sequential enumeration of the design space, by parameter-configuring the associated combinatorial transition system, generates models with small incremental differences. Typical model-checking algorithms do not utilize this information. They reset every time a new model is checked, thereby, losing all model-checking artifacts learned for the previous model. Since the models in the design-space are *related*, the model checker can benefit from reusing these artifacts across model-checking runs. Therefore, learning and reusing information from solving related models becomes very important for future checking efforts. Figure 3.1 shows the reachable state-spaces for four related models M_1, M_2, M_3, and M_4 in a design space that are checked sequentially against a safety property φ. The model checker first verifies model M_1 by exploring its reachable state-space, and concludes that $M_1 \models \varphi$. A typical model-checking algorithm then resets, and starts verifying model M_2. This is extremely wasteful. Significant verification resources can be saved by reusing

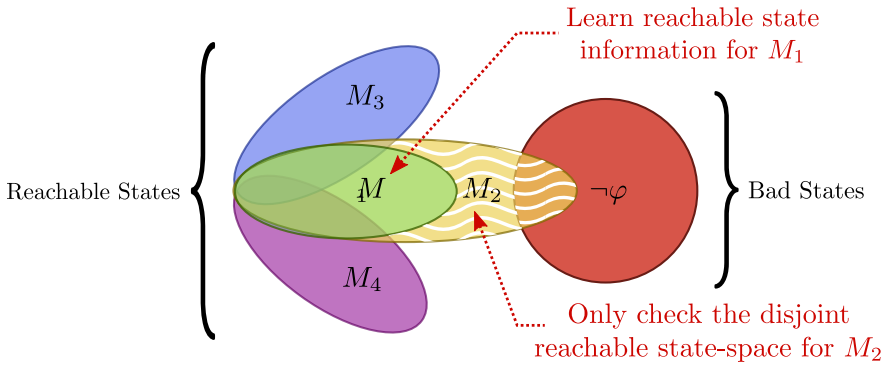


Figure 3.1: Venn diagram of reachable state-spaces for four related models M_1, M_2, M_3, and M_4. The model checker learns state-space information for M_1, and only verifies the disjoint state-space for M_2

the reachable state-space information learned by the model checker for M_1, and then only verifying the disjoint state-space for M_2 (marked by wavy lines).

There is no doubt that incremental verification can tremendously benefit model-checking of large design spaces. In this chapter, we present a state-of-the-art incremental model-checking algorithm that saves minimal model-checking information across runs, and efficiently reuses the saved information across different models in the design-space, albeit, after careful repair. We specifically answer the following questions: 1) *What type of model-checking algorithms can benefit from information reuse?* 2) *What type of information is learned, and can be saved during model-checking runs?* 3) *How to efficiently maximize the reuse of saved information across different models?*

The rest of the chapter is organized as follows: Section 3.1 gives an overview of incremental verification and its applicability to model checking design spaces, highlights our contributions for identifying, customizing and efficiently reusing model-checking information, and contrasts with related work. Section 3.2 gives background information, and formally introduces incremental verification for design spaces. Section 3.3 details our state-of-the-art incremental model-checking algorithm and provides proofs of correctness. A large-scale experimental evaluation on design spaces from NASA and Boeing, and several hardware veri-

fication problems forms Section 3.5. Lastly, Section 3.6 concludes the chapter by highlighting future work and possible extensions to our algorithms.

3.1 Introduction

In the early phases of design, there are several models of the system under development constituting a *design space* [BLBM07, GCM+16, MCG+15]. 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 [BCFP+15, GCM+16, JMN+14, MCG+15, MNR+13].

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 grow, 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 [Bra11], also known as *property directed reachability* (PDR) [EMB11]. 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 [GCM+16, MCG+15], selected benchmarks from HWMCC 2015 [Bie15], and a set of seven models for the Boeing AIR6110 wheel braking system [BCFP+15]. 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.

3.1.1 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 [YBO+98, BBDEL96]. Similarly, intermediate SAT solver clauses and interpolants are reused in bounded model checking [MS07a, SKB+16]. Reusing learned invariants in IC3 speeds up convergence of the algorithm [CIM+11]. These techniques enable efficient incremental model checking and are useful in *regression verification* [YDR09] and *coverage computation* [CKV06]. The FuseIC3 algorithm is an incremental model-checking algorithm and is applicable for these scenarios.

Product line verification techniques, e.g., with Software Product Lines (SPL), also verify models describing large design spaces [CHS+10, CHSL11, CCH+12, BDSAB15]. The several *instances* of feature transition systems (FTS) [CCS+13b] 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 [DR18]. *Parameter synthesis* [CGMT13] can generate the many models in a design space that can be checked using FuseIC3. The parameterized model-checking problem [EK00] deals with infinite homogeneous models. In our case, the models in a set for the design-space are heterogeneous and finite.

The work most closely related to ours is a state-of-the-art algorithm for incremental verification of hardware [CIM+11]. 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.

3.1.2 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. Overview of locality-sensitive hashing [AI08] techniques to mine model specifications expressed as And-Inverter-Graph circuits.

4. 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 by FuseIC3 and improve overall performance.

5. Extensive experimental analysis using real-life benchmarks and comparison with existing state-of-the-art incremental algorithm for IC3.

6. We make all reproducibility artifacts and source code publicly available.[1] We provide detailed explanations, and theorem proofs of correctness for the several sub-algorithms.

3.2 Preliminaries

Definition 3.2.1. A Boolean transition system, or system model M is represented using the tuple $M = (\Sigma, Q, Q_0, \delta)$ where

1. Σ is a finite set of atomic propositions or state variables,

[1]Raw experimental results available at http://temporallogic.org/research/FMCAD17/

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 $\pi = s_0 \to s_1 \to \ldots \to s_n$ 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 3.2.2. 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 3.2.3. 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 (disjunction) of clauses (cubes) is said to be in *Conjunctive Normal Form* (CNF) (*Disjunctive Normal Form* (DNF)).

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 $c \Rightarrow \varphi$ 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 3.2.4. Two finite sets ψ_1 and ψ_2 *overlap* iff $\psi_1 \cap \psi_2 \neq \emptyset$.

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

Definition 3.2.5. 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 [DR18], or software-product lines [CHS+10]. Moreover, updates to a sequential circuit design in regression verification, either due to a bug fix or feature addition, generate related transitions systems [YDR09] that may have overlapping reachable states.

3.2.1 Safety Verification

The safety verification problem is to decide whether a transition system 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. $Q_0 \Rightarrow \mathcal{I}$, i.e., the initial states satisfy the invariant,
2. $\mathcal{I} \wedge \delta \Rightarrow \mathcal{I}'$, i.e., the invariant is inductive, and
3. $\mathcal{I} \Rightarrow \varphi$, i.e., the invariants satisfies safety property φ.

In SAT-based model checking algorithms [Bra11, BCCZ99, McM03, VG09], 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.

3.2.2 Property-Directed Reachability

IC3/PDR [Bra11, Bra12, EMB11, GR16, SB11] 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. $S_{i+1} \subseteq S_i$, i.e., the frame sequence is monotone,
3. $S_i \wedge \delta \Rightarrow S'_{i+1}$, i.e., states in S_{i+1} are reachable from S_i, and
4. for $i < k$, $S_i \Rightarrow \varphi$, i.e., each frame satisfies safety property φ.

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 3.2.1.

3.2.3 Problem Formulation

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.

3.2.3.1 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 3.2.2. 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 3.2.2 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 $S_i \wedge \delta_N \Rightarrow \hat{R}_{i+1}$.

3.2.3.2 SAT with Assumptions

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

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

3.3 Algorithm for Incremental Verification

In this section, we present our main contribution, 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.

3.3.1 Information Learning

Recall that the frame sequences 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 3.2.2). 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 Figure 3.2. The parallelogram and ellipse represent clauses c_1 and c_2 learned by IC3 during a model-checking run, 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

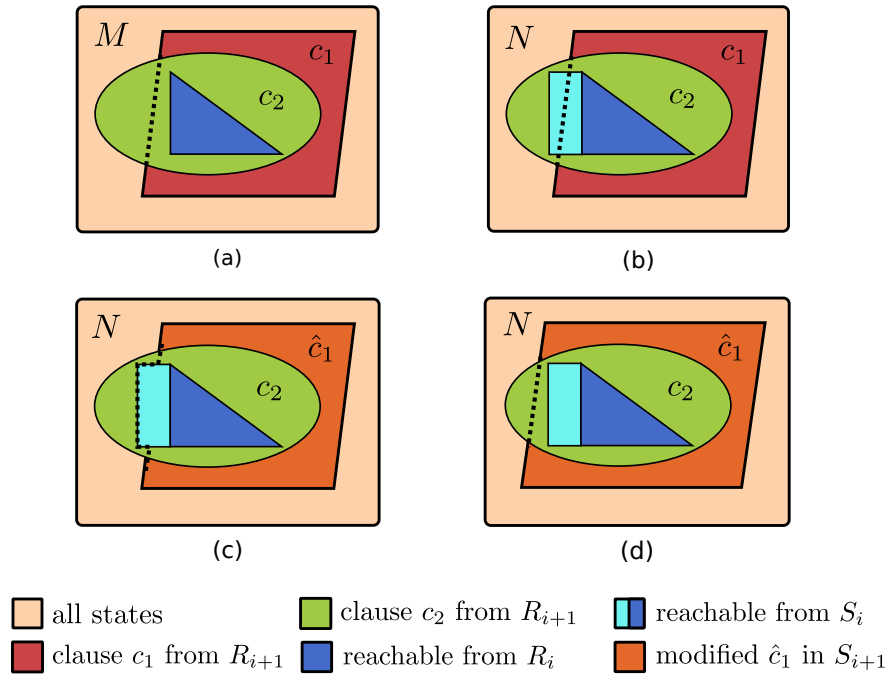


Figure 3.2: Intuition behind repairing frames computed for one model by IC3, and reusing them for checking another related model in the design space.

one step, i.e., $R_i \wedge \delta_M$. Therefore, $R_i \wedge \delta_M \Rightarrow R'_{i+1}$. Now consider a scenario in which we recycle the clauses in R_{i+1} when verifying N. The triangle and the rectangle in Figure 3.2b 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. Figure 3.2c and 3.2d 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. Therefore, reusing clauses from model M, saves a lot of effort in rediscovering these clauses for model N, and thus helps IC3 converge faster in finding an invariant or counterexample.

3.3.2 Information Repair and Reuse

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 Figure 3.3.

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 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.

The FrameRepair algorithm of Figure 3.4 takes as input an integer i. It checks if frame sequence R_{i+1} from model M 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.

bool FuseIC3 (Q_0, δ, φ)

Input: Q_0 = initial states, δ = transition relation, φ = safety property

1: **if** CheckInvar(Q_0, δ, last_invariant, φ) : **return** *true*

2: **if** SimulateCex(Q_0, δ, last_cex, φ) : **return** *false*

3: $k \leftarrow 0$, $S_k \leftarrow Q_0$ *# first frame is initial state*

4: **while** *true* : *# main algorithm 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*

Figure 3.3: High-level description of the FuseIC3 algorithm. Parts of the algorithm for typical IC3 are based on the description in [EMB11, GR16].

frame FrameRepair (**int** i)

Input: i = current frame number in the sequence

1: **if not** $\mathsf{sat}(S_i \wedge \delta \wedge \neg R'_{i+1})$: **return** R_{i+1}

2: $\mathcal{G} \leftarrow$ FindClauses(S_i, δ, 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, δ, c) *# weaken clause c*

6: $\hat{c} \leftarrow$ ShrinkClause(S_i, δ, c, $\hat{c}$) *# strengthen clause 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}

Figure 3.4: The FrameRepair algorithm to reuse and reachable state sequences across model-checking runs by efficiently repairing violating clauses. The violating clauses are either expanded (weakened) or shrunk (strengthen) to enable their reuse for the current run.

3.3.2.1 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. $Q_{0_M} \Rightarrow \mathcal{I}_M$, i.e., initial states of M satisfy invariant $\mathcal{I}_M$,

2. $\mathcal{I}_M \wedge \delta_M \Rightarrow \mathcal{I}'_M$, i.e., invariant $\mathcal{I}_M$ is inductive with respect to δ_M, and

3. $\mathcal{I}_M \Rightarrow \varphi$, i.e., invariant satisfies safety property φ.

If the 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 following conditions hold for N:

1. $Q_{0_N} \Rightarrow \mathcal{I}_M$, i.e., initial states of N satisfy invariant $\mathcal{I}_M$,

2. $\mathcal{I}_M \wedge \delta_N \Rightarrow \mathcal{I}'_M$, i.e., invariant $\mathcal{I}_M$ is inductive with respect to δ_N and, and

3. $\mathcal{I}_M \Rightarrow \varphi$, i.e., invariant satisfies safety property φ.

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

1. $s_0 \in Q_{0_M}$, i.e., state s_0 is an initial state,

2. $(s_i, s_{i+1}) \in \delta_M$ for $i < k$, i.e., state S_{i+1} is reachable from state S_i in model M, and

3. $s_k \in \neg\varphi$, i.e., state s_k is a bad-state and violates property φ.

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 Figure 3.5.

bool CHECKINVARIANT (Q_0, δ, $\mathcal{I}$, φ)

Input: Q_0 = initial states, δ = transition relation, $\mathcal{I}$ = saved invariant, φ = safety property

1: **if not** $\mathsf{sat}(Q_0 \wedge \neg\mathcal{I})$ **and not** $\mathsf{sat}(\mathcal{I} \wedge \delta \wedge \neg\mathcal{I})$ **and not** $\mathsf{sat}(\mathcal{I} \wedge \neg\varphi)$: **return** *true*

2: **else return** *false*

bool SIMULATECEX (Q_0, δ, s, φ)

Input: Q_0 = initial states, δ = transition relation, s = saved counterexample trace

φ = safety property

1: **if not** $\mathsf{sat}(s_0 \wedge Q_0)$: **return** *false*

2: **if not** $\mathsf{sat}(s_k \wedge \neg\varphi)$: **return** *false*

3: **for** $i \leftarrow 0$ to len(s) : *# simulate counterexample trace*

4: **if not** $\mathsf{sat}(s_i \wedge \delta \wedge s'_{i+1})$: **return** *false*

5: **return** *true* *# valid counterexample*

Figure 3.5: 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*.

3.3.2.2 Frame Repair

We want to find all clauses in frame R_{i+1} that are responsible for the violation of $S_i \wedge \delta_N \Rightarrow R'_{i+1}$. 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 $S_i \wedge \delta_N \Rightarrow \hat{R}'_{i+1}$ 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.

Finding Violating Clauses: Let's assume that frame R_{i+1} is composed of a set of clauses $C = \{c_1, c_2, \ldots c_n\}$. Then there are clauses $\mathcal{G} \subseteq C$ such that the assertion $S_i \wedge \delta_N \Rightarrow c'$ 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 for complex verification problems as IC3 frames can often have thousands of clauses. Algorithm FindClauses, which is inspired by the *Invariant Finder* algorithm in [CIM+11], efficiently finds all such violating clauses.Figure 3.6 shows a query-efficient algorithm to find all violating clauses.

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 3.3.1. *Given the current frame sequence S, transition relation δ, and frame sequence to reuse R, the* FindClauses *algorithm (Figure 3.6) 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 $\neg l_1' \vee y_i$, $\neg l_2' \vee y_i$, $\ldots$, $\neg l_k' \vee y_i$ to the solver. Therefore, the overall assertion for clause c_i

FindClauses (S, δ, R)

Input: S = current frame for model N, δ = transition relation for model N,

R = frame to reuse from model M

Output: $\mathcal{G}$ = violating clauses in 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 $\neg l \vee y_i$ to solver

5: $\mathcal{G} \leftarrow \emptyset$ *# set is initially empty*

6: **while** sat$(S \wedge \delta, (\neg y_1 \vee \neg y_2 \vee \ldots \vee \neg y_k))$:

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*

Figure 3.6: FindClauses algorithm to find all violating clauses $c_i \in R$ such that $S \wedge \delta \not\Rightarrow c'$. Upon termination, the set $\mathcal{G}$ contains all violating clauses.

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

$$
\begin{aligned}
& (\neg l_1' \vee y_i) \wedge (\neg l_2' \vee y_i) \wedge \ldots \wedge (\neg l_k' \vee y_i) \\
\Leftrightarrow \quad & (\neg l_1' \wedge \neg l_2' \wedge \ldots \wedge \neg l_k') \vee y_i \\
\Leftrightarrow \quad & \neg (l_1' \vee l_2' \vee \ldots \vee l_k') \vee y_i \\
\Leftrightarrow \quad & \neg c_i' \vee y_i \\
\Leftrightarrow \quad & c_i' \Rightarrow y_i
\end{aligned}
$$

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 $\neg l'_1 \vee y_1$, $\neg l'_2 \vee y_1$, $\ldots$, $\neg l'_k \vee y_1$. 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 based on whether 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, $\neg l'_1$, $\neg l'_2$, $\ldots$, $\neg l'_k$ 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 $S \wedge \delta \not\Rightarrow c'_i$. □

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.

Expanding 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. [2] Instead, we use an efficient randomized algorithm, ExpandClause, to add literals to clause c. The pseudo-code for the algorithm is given in Figure 3.7.

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.

[2] The resulting query is of the form $\exists\forall$. and in 2QBF.

ExpandClause (S, δ, c)

Input: S = current frame for model N, δ = transition relation for model N,

c = violating clause

Output: $\hat{c}$ = weakened clause

1: $v \leftarrow$ all *primed* variables in δ

2: $l \leftarrow$ all variables in clause c'

3: $B \leftarrow v \setminus l$ *# variables not in clause c*

4: $\hat{c} \leftarrow c$ *# initially* $\hat{c} = c$

5: **while** $|B| > 0$ and $\mathsf{sat}(S \wedge \delta \wedge \neg\hat{c}')$: *# iterative until no more literals are left*

6: $\alpha \leftarrow$ get-sat-model() *# get variable assignments*

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 *# no longer under consideration*

11: **if** $\mathsf{sat}(S \wedge \delta \wedge \neg\hat{c}')$: **return** $\emptyset$

12: **return** $\hat{c}$ *# expanded clause;* $S \wedge \delta \Rightarrow \hat{c}'$

Figure 3.7: 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.

Theorem 3.3.2. *Given the current frame sequence S, transition relation δ, and violating clause c, the* ExpandClause *algorithm (Figure 3.7) weakens violating clause c to generate clause $\hat{c}$ such that $S_i \wedge \delta \Rightarrow \hat{c}'$.*

Proof. In line 3, set 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 $a \in S$, $(a, b') \in \delta$, 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 $\hat{c} = \mathit{true}$ (all states are reachable from S) and returned. □

Shrinking Expanded Clauses: Due to the randomized nature of the ExpandClause algorithm, 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 $\tilde{c}$. In principle, we can use the minimal UNSAT assumptions for the query to provide the smallest clause $\tilde{c}$ such that $S_i \wedge \delta \Rightarrow \tilde{c}'$. However, finding minimal UNSAT assumptions is often very expensive. Moreover, the need for minimality can often be traded for faster implementation by utilizing first-available UNSAT assumptions. 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 Figure 3.8.

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

ShrinkClause (S, δ, c, $\hat{c}$)

Input: S = current frame for model N, δ = transition relation for model N,
c = violating clause, $\hat{c}$ = expanded clause

Output: $\tilde{c}$ = strengthened clause

assert(**not** sat($S \wedge \delta \wedge \neg\hat{c}'$))

1: $v \leftarrow$ {literals in $\hat{c}$} $\setminus$ {literals in c} *# find excess literals*

2: $\tilde{c} \leftarrow c$ *# initially $\tilde{c} \leftarrow c$*

3: **for each** $l \in v$: *# try dropping literals one-by-one*

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 $j \mid j' \in$ get-unsat-assumptions()} *# required literals*

7: **return** $\tilde{c} \leftarrow \tilde{c} \vee \bigvee${literals in v} *# shrunk clause; $S \wedge \delta \Rightarrow \tilde{c}'$*

Figure 3.8: ShrinkClause algorithm to remove excess literals from clause c while maintaining $S \wedge \delta \Rightarrow c'$.

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 3.3.3. *Given the current frame sequence S, transition relation δ, and violating clause c, the* ShrinkClause *algorithm (Figure 3.8) 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, set v contains excess literals added to expand 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, set v contains the minimum number of literals, which when added to clause c to generate clause $\tilde{c}$, 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 3.3.4. *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 (Figure 3.4) 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 directly from Theorems 3.3.1, 3.3.2, and 3.3.3. All violating clauses in R_{i+1} are found by the FindClauses algorithm. (Theorem 3.3.1). The ExpandClause algorithm (Theorem 3.3.2 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 3.3.3). 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 sequence$\hat{R}_{i+1}$ is added to the set of frame sequences 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 checked for intersection with the negated safety property φ representing bad states, followed by normal execution of blocking and propagation phases of the IC3 algorithm.

3.4 Organizing the Design Space

If models 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 [AI08] 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.

3.4.1 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, $H : U \to V$, 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 $X = Y$, i.e., identical objects map identical hash values, and
2. $H(X) \neq H(Y)$ for $X \neq Y$, i.e., different objects map to different hash values.

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 [CZ17] 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) [AI08] 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 [BCFM00] 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 3.9 shows locality-sensitive hashing on a set of five documents D_1, D_2, D_3, D_4, and D_5. 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.

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 $(\frac{1}{b})^{\frac{1}{r}}$ [RU11]. To estimate the values of b and r for $k = 400$ and a Jaccard

		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

Figure 3.9: Locality-sensitive hashing to find similar documents in a set. Documents D_1 and D_3, and documents 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.

similarity threshold of $t = 0.9$, we have

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

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 [RU11]. 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 [App] with different seed values.

3.4.2 Partial Model Ordering

Let model-set $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ consist of related models of a design space, e.g., generated by parameter instantiating a combinatorial transition system [DR18]. 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 representing the transition relation for the models is expressed in DIMACS CNF format[3], 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} for model $M_i \in \mathcal{M}$ 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 partial model ordering (MO) 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 containing similar models are checked in random order, or in parallel as discussed in Chapter 5. We use a Jaccard similarity coefficient of 0.9 for the partial model ordering heuristic to group similar models.

3.4.3 Property Grouping

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* [CCL+17, CN11b] 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 with respect to a model M. The affinity α_{ij} is then calculated as

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

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

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

from the conjunction, and the remaining property is checked. The property grouping (PG) 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).
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 can be checked sequentially in random order, or in parallel for maximum throughput. We use a Jaccard similarity coefficient of 0.9 for finding similar properties, and a property affinity threshold of 0.95 for grouping properties.

3.5 Experimental Analysis

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

3.5.1 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) [Bie15] 2015.

3.5.1.1 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 [GCM+16]. 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.

3.5.1.2 Selected Benchmarks from HWMCC 2015

We consider a total of 548 benchmark models from the single safety property track [Bie15]. 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, 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.

3.5.1.3 Wheel Braking System (WBS) Models

The benchmark consists of seven real-world models representing possible designs for the Boeing AIR6110 wheel braking system [BCFP+15]. 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 in the set of seven models is checked using a timeout of 120 minutes.

3.5.2 Experiment Setup

FuseIC3 is implemented in C++ and uses MathSAT5 [CGSS13] as the underlying SMT solver. It takes SMV models or AIGER files as input. The IC3 part of FuseIC3 is based

Table 3.1: Results for 34 sets of 1,620 models each for NASA Air Traffic Control System.

Algorithm	Cumulative Time (min)	Median 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)

on the description in [EMB11] and `ic3ia`.[4] 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 [BBDEL96]. We implemented inc based on the description in [CIM+11] 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.

3.5.3 Experimental Results

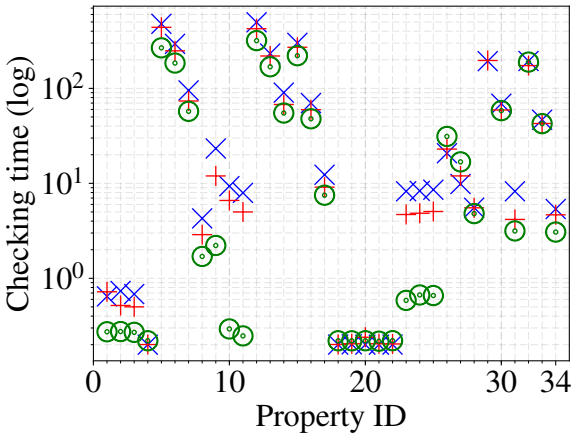
3.5.3.1 Air Traffic Controller (ATC) Models

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 3.1 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

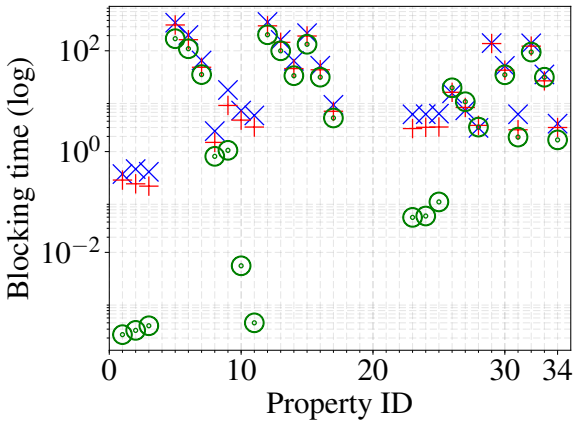
[4] https://es-static.fbk.eu/people/griggio/ic3ia/

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.

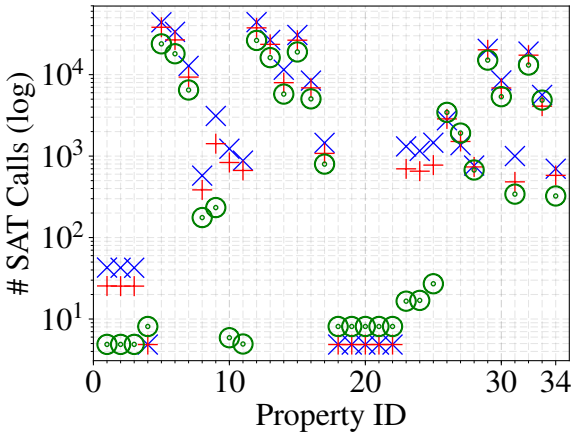
Figure 3.10a shows time taken by the algorithms on each model-set. FuseIC3 is almost always faster than typical IC3, and incremental IC3. However, for model-sets (corresponding to property IDs 4 and 18–22) containing models that trivially satisfy/falsify a property, typical IC3 is faster; both incremental 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). Figure 3.10b shows time taken by each algorithm in blocking discovered bad states. FuseIC3 spends considerably less time in the blocking 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. Figure 3.10c 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 significantly improves the overall performance of FuseIC3 as shown in Figure 3.10d. Checking partially ordered models is faster than random checking for all model-sets, as it enables the FuseIC3 algorithm to reuse more information between similar models in a group.



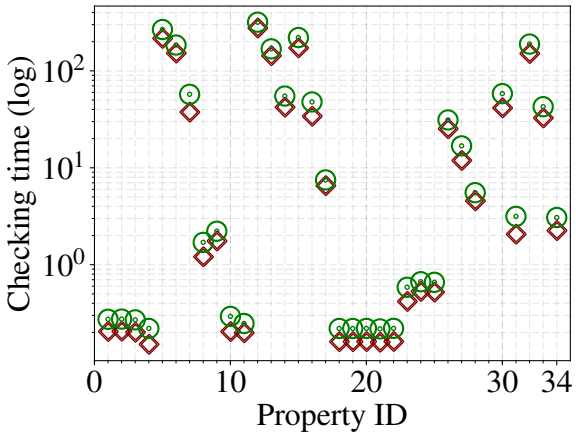
(a) Checking time per set (minutes)



(b) Blocking time per set (minutes)



(c) Number of SAT calls per set



(d) Time with model ordering (minutes)

Figure 3.10: 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.

Table 3.2: Results for 91 of 110 sets of 200 models each for selected HWMCC 2015 benchmarks.

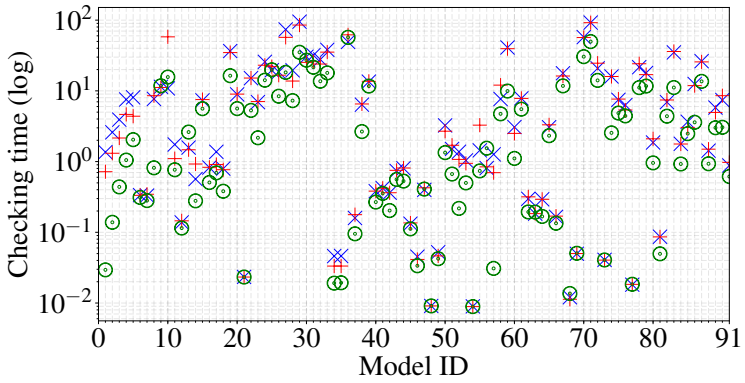
Algorithm	Cumulative Time (min)	Median Speedup	
		v/s typ (avg)	v/s inc (avg)
Typical IC3 (typ)	1024.60	-	-
Incremental IC3 (inc)	1026.30	1.04 (1.07)	-
FuseIC3	545.31	1.75 (3.18)	1.72 (2.56)
FuseIC3 + MO	396.65	2.32 (3.96)	2.05 (3.12)

3.5.3.2 Benchmarks from HWMCC 2015

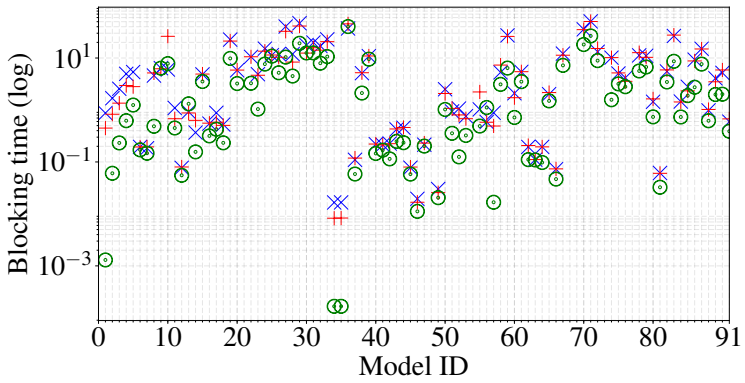
Each of the 110 model-sets are checked using a timeout of 120 minutes per algorithm. The models in a set are checked in random order, and then using model ordering (MO) heuristic. 91 of 110 model-sets were solved by all algorithms within the timeout. Incremental IC3 solved two more model-sets compared to typical IC3, while FuseIC3 solved five more compared to typical IC3. Table 3.2 gives a summary of results.

Figure 3.11a 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 when checking model-sets containing large models with FuseIC3. Performance for model-sets containing small models is similar for all algorithms. Figure 3.11b shows time spent by each algorithm in blocking predecessors to bad states. Lastly, Figure 3.11c shows the number of SAT queries made by the different algorithms.

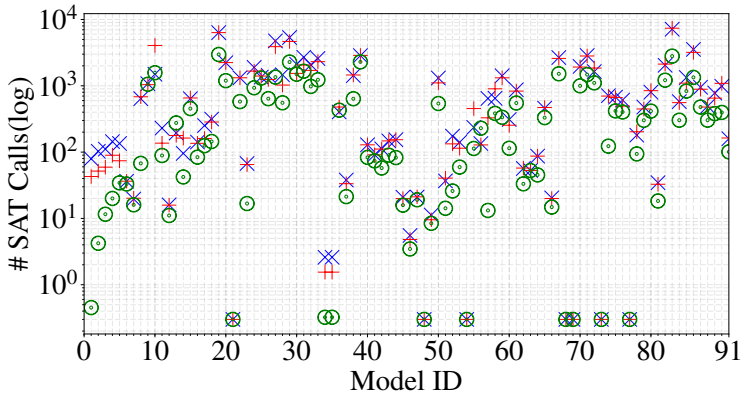
To estimate performance of FuseIC3 on model-sets with varying degree of overlap among models, we picked the `bobtuint18neg` benchmark from HWMCC 2015. 40 model-sets with varying degrees of mutation, between 0.5% to 20%, of the original model were generated. Each model-set consists of 100 models each. Each set was checked using a timeout of 300 minutes with typical IC3, and FuseIC3 with model ordering (MO). Model-sets corresponding



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



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



(c) Number of SAT calls per per design space

Figure 3.11: 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.

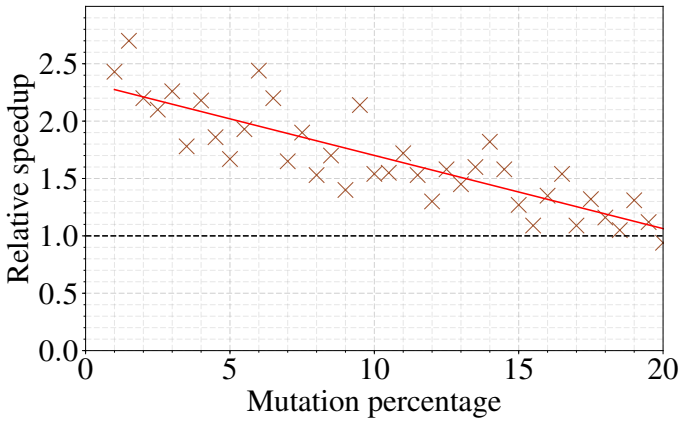


Figure 3.12: Relative speedup between checking using FuseIC3 with model ordering versus typical IC3. 100 models are generated for every mutation percentage between 0.5% to 20% in steps of 0.5%, and are checked against the same property. The 'red' line represents a linear fitting of all the points on the plot.

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 [CIM+11]) and are not reported. Figure 3.12 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.

3.5.3.3 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 3.3 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

Table 3.3: Comparison between typical IC3, Incremental IC3, and FuseIC3 for AIR6110 Wheel Braking System (reported 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)

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 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.

3.6 Summary and Discussion

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 (Figure 3.10b and Figure 3.11b) 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 [CIM+11]. 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.

Specialized incremental verification algorithms, like FuseIC3, immensely benefit design-space model checking. The models for the pruned design space generated by the D^3 often have small incremental differences, which allows FuseIC3 to reuse much of the learned information across model-checking runs. SAT-based model-checking algorithms that learn reachable-state information as clauses [LDP+18, DLP+19], can be easily extended for incremental verification using the techniques presented in this chapter. The situation becomes trickier when verifying multiple properties on individual models. For the wheel-braking system models presented in this chapter, fewer clauses are salvageable due to the random ordering of the checked properties; even though FuseIC3 with the proposed heuristics improves overall performance, it still spends significant time in learning new clauses (measured

by the number of calls to the SAT solver). The grouping approach of properties is useful, but requires expensive offline computation to find high-affinity properties. For models with thousands or even millions of properties, as is the case in equivalence checking, the property grouping procedure based on locality sensitive hashing is computationally prohibitive. Moreover, it is often the case that the proof or counterexample for a property only depends on a small subset of its cone-of-influence. It might be the case that for two properties with identical cone-of-influence, FuseIC3 learns completely different clauses; these two properties shouldn't be in the same group. Property grouping based on the "required" cone-of-influence of properties therefore may help maximize reuse of information between grouped properties. However, it is impossible to know the required cone-of-influence subset without consuming any verification resources. In the next chapter, we continue looking under the covers of a model checker, and focus our attention to optimizing verification of multiple properties that enables more efficient reuse of information across properties.

CHAPTER 4. MULTI-PROPERTY VERIFICATION

Individual model-checking tasks in design-space exploration using model checking entail checking a large number of properties on the model. The number of properties to check may be reduced by finding implicit dependencies (Section 2.3.2). Incremental verification algorithms can further speedup verification of multiple properties by learning and reusing model-checking artifacts across runs (Section 3.4.2). Several other verification tasks ranging from functional property verification to equivalence checking evaluate multiple properties on a model. Given the prevalence of multi-property verification problems, much research has mostly focused on optimizing model-checking for single property verification. State-of-the-art tools typically solve all properties concurrently, or one-at-a-time. They do not optimally exploit subproblem sharing between properties, leaving an opportunity to save considerable verification resource via incremental or concurrent verification of properties with nearly identical cone of influence (COI). The FuseIC3 algorithm presented in Chapter 3 benefits from verifying *similar* properties in sequentially. Existing techniques for multiple properties utilize heuristics to partition properties into high-affinity groups based on a "similarity-measure" as shown in Figure 4.1. The property groups are then checked independently, and different groups can be checked in parallel. However, existing techniques to group properties are either computationally prohibitive, or require expensive offline analysis. The locality sensitive hashing technique of Section 3.4 can speed up grouping significantly, but does not scale to problems with thousands, or even millions of properties as is the case in equivalence checking, due to the required large number of hashing operations.

A typical verification task should spend majority time in checking properties rather than finding optimal grouping. Therefore, there is a significant need to develop partitioning algo-

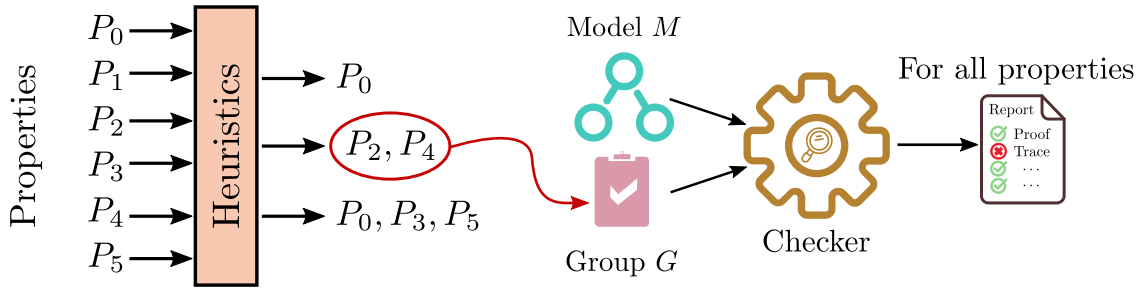


Figure 4.1: Typical methodology for verification of multiple properties. Several heuristics partition properties into high-affinity groups; the individual groups are verified independently using a model checker, and different groups in parallel to provide results for all properties.

rithms that scale with the number of properties, and take a few milliseconds on problems with millions of properties. We observe that trading accuracy vs. speed for property grouping impacts the overall verification workflow. The choice of the similarity metric impacts verification performance, and influences the design of property-grouping algorithms. Moreover, a very large property group, although high-affinity, can sometimes be harder to solve than checking smaller subgroups due to increased problem complexity. In this chapter, we present a super-fast, near-linear runtime algorithm that trades accuracy vs. speed for partitioning properties into high-affinity groups. We specifically answer the following questions: 1) *How to efficiently compute a similarity measure for properties based on model representations?* 2) *What readily available design information can be utilized to measure property similarity?* 3) *How to efficiently compare properties based the chosen similarity measure and avoiding pairwise comparisons?* 4) *How to repartition very large, and hard-to-prove property groups into smaller subgroups based on semantic information learned during verification?*

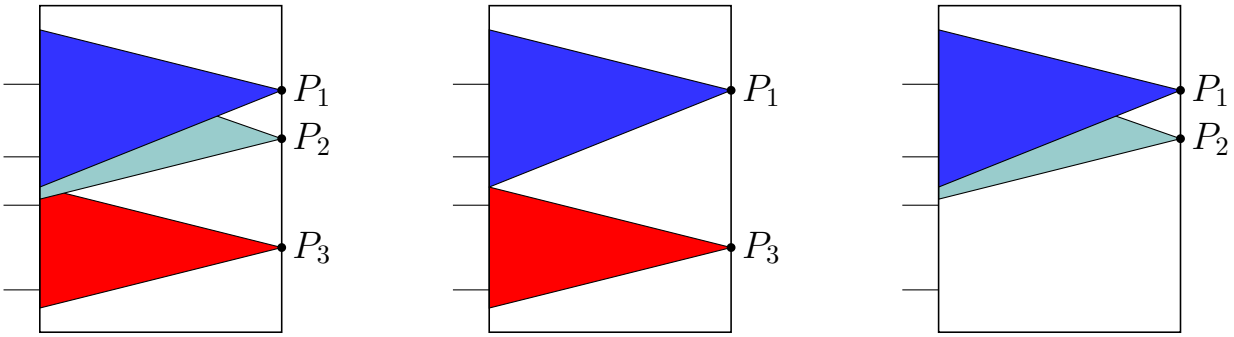
The rest of this paper is organized as follows: Section 4.1 overview our contributions to efficiently partition properties into high-affinity groups and utilize semantic information to repartition very large high-affinity groups, and contrasts with related work. Section 4.2 gives background information, introduces formalisms, and details a linear-time algorithm to compute cones of influences for multiple properties. We present a linear-time algorithm

to partition properties into provably high-affinity groups in Section 4.3. The large and hard-to-prove property groups may be repartitioned using semantic information learned by localization abstraction using the algorithms in Section 4.4. A large experimental evaluation on multiple property benchmarks forms Section 4.5. Section 4.6 concludes the chapter by addressing avenues for future optimizations and customizations of our algorithms.

4.1 Introduction

From equivalence checking to functional verification to design-space exploration, industrial verification tasks entail checking a large number of properties on the same design. For example, equivalence checking compares pairwise equality of each design output across two designs, and entails a distinct property per output. Functional verification checks designs against a large number of properties ranging from low-level assertions to high-level encompassing properties. Design-space exploration via model checking [GCM+16] verifies multiple properties against competing system designs differing in core capabilities or assumptions.

Each property has a distinct minimal *cone of influence (COI)*, or fan-in logic of the signals referenced in that property (Figure 4.2a). Verification of a set of properties often entails exponential complexity with respect to the size of its collective COI. *Concurrent* verification of multiple properties may thus be significantly slower than solving these properties one-at-a-time, in that each property of the group may add unique fan-in logic to the collective COI (Figure 4.2b). Conversely, sometimes two or more properties share nearly-identical COIs (Figure 4.2c). Concurrent verification of high-affinity properties may save considerable verification resource, as the effort expended for one can be directly reused for the others without significantly slowing the verification of any property within that group (e.g., reusing clauses within the SAT solves or reachable state-approximations [Bra11, LDP+18], and models abstractions learned using localization [AM04] across properties in a group).



(a) Mix **(b)** Low **(c)** High

Figure 4.2: Cone-of-influence of high- and low- affinity properties.

Despite the prevalence of multi-property testbenches, little research has addressed the problem of optimal grouping or *clustering* of properties into high-affinity groups. Selective past work [CCL+18, CN11b] has experimentally demonstrated that ideal grouping may save substantial verification resource. However, no scalable online property grouping procedure has been provided; this potential was illustrated as a proof-of-concept using computationally-prohibitive offline grouping algorithms with undisclosed runtime. A significant need thus remains for an effective solution of determining which high-affinity properties should be concurrently solved. To ensure overall scalability, it is essential that such a property-partitioning solution be as close to linear runtime as possible with respect to the number of properties, otherwise the grouping effort itself may severely degrade overall verification resource comprising grouping plus subsequent verification of the identified groups.

We present a near-linear runtime, fully-automated algorithm to partition properties into provably high-affinity groups based on structural COI similarity. COI support information is maintained as bitvectors [CCQ16], and grouping is performed in three configurable levels based on: identical COI, strongly-connected components (SCC) in the COI, and Hamming distance. The properties in each high-affinity group are verified concurrently; each group may be independently verified in parallel, using arbitrary solver algorithms. We also present an

algorithm to semantically refine high-structural-affinity groups in a localization abstraction framework, offering the first optimized multi-property localization solution, to our knowledge. Our partitioning requires negligible resources even on the largest problems, while offering substantial verification speedups as demonstrated by extensive experiments.

4.1.1 Related Work

Much prior work has addressed methods to incrementally reuse information across muliple properties to accelerate specific algorithms. E.g., incremental SAT across proofs of different properties [KNPH06, KN12], and reusing verification by-products like invariants [DR17] and interpolants [MS07b], can accelerate the verification of high-affinity properties.

Methods to group properties based on high-level design descriptions (for e.g., module hierarchies) extract similarity criteria from high-level information unavailable in low-level designs and benchmark formats such as AIGs [CM10]. The framework of *local* and *global* proofs [GGKM18] has been used to derive a "debugging set" of properties to fix before verifying others, implying a property ordering but not a partitioning for minimal collective resource. LTL satisfiability checking has been used to establish logical dependencies between properties [DR18] to dynamically reduce verification resource; however, this work requires a quadratic number of resource-intensive comparisons.

The work most similar to ours is a property-clustering procedure based on COI similarity [CCL+18, CN11b]. While a similar goal, their solution requires a quadratic number of comparisons between properties, rendering it prohibitively expensive on large testbenches. Their experiments do not disclose grouping resource, only subsequent verification speedup. Moreover, this generic clustering approach requires the number of desired groups as an algorithmic parameter. This metric is impossible to predict in practice; it is far superior to allow affinity analysis to automatically determine the optimal number of groups.

4.1.2 Contributions

We present a near-linear runtime algorithm to partition properties into high-affinity groups based on structural COI similarity. Our contributions are summarized as follows:

1. An online algorithm to partition properties based on structural information, readily available in low-level design representations, into provably high-affinity groups.

2. Efficient procedure to compute cones of influence of multiple properties, and data structures that allow CPU-speed comparison between properties.

3. A systematic methodology to learn semantic information, and refine high-structural-affinity groups in a localization abstraction framework.

4. An optimized multi-property localization abstraction solution that is resistant to performance slowdown that may occur when verifying very-large property groups.

5. Extensive experimental evaluation on large benchmarks derived from varied hardware verification problems that span functional verification and equivalence checking.[1]

4.2 Preliminaries

Definition 4.2.1. The logic design under verification is represented as a *netlist* N, which is a tuple $(\langle V, E\rangle, F)$ where $\langle V, E\rangle$ is a directed graph such that

1. V is a set of vertices representing *gates*,

2. $E \subseteq V \times V$ are edges representing interconnections between gates, and

3. $F : V \to \mathit{types}$ is a function that assigns vertices to gate *types*: constants, primary inputs, combination logic such as *AND* gates, and sequential logic such as *registers*.

[1]Raw experimental results available at http://temporallogic.org/research/FMCAD19

A *state* is a valuation to the registers. Each register has two associated gates that represent its next-state function, and its initial-value function. Semantically, the value of the register at time "0" equals the value of the initial-value function gate at time "0", and the value of the register at time "i+1" equals that of the next-state function gate at time "i". Certain gates in the netlist are labeled as *properties* that are formed through standard synthesis of the relevant property specification language.

Definition 4.2.2. Given a netlist $N = (\langle V, E\rangle, F)$, a gate $v_i \in V$ is in the *fan-in* of gate $v_j \in V$ if and only if $(v_i, v_j) \in E$ or there exist gates $\{v_1, v_2, \ldots, v_k\} \in V$, for $k \geq 1$, such that $\{(v_i, v_1), (v_1, v_2), \ldots, (v_k, v_j)\} \in E$.

The *fan-in cone* of a property gate p refers to the set of all gates in the netlist which may be reached by traversing the netlist edges backward from the property gate, and is denoted *fanin(p)*. This fan-in cone of the property gate is called the *cone-of-influence* (COI) of the property. The registers and inputs in the COI of the property are called *support variables*. The number of support variables in the property's COI is the COI *size*.

Definition 4.2.3. A *strongly connected component* in a netlist $N = (\langle V, E\rangle, F)$ is a set of gates $\mathcal{C} \subseteq V$ such that for every pair of gates $v_i, v_j \in \mathcal{C}$, $v_i \in fanin(v_j)$ and $v_j \in fanin(v_i)$.

Note that a primary input does not belong to any SCC, and in a *well-formed* SCC every directed cycle has at least one register gate because a netlist must be free of combinational cycles. The number of register gates in a SCC is the *weight* of the SCC.

4.2.1 Cone-of-Influence Computation

Support variable information may be represented as an indexed array of Boolean values, or bitvector, per property. Figure 4.3 gives a high-level procedure to compute a support bitvector for a property p. Every support variable in the netlist N is indexed to an unique

```
function support_bitvector (p, N)
Input: p = property gate, N = netlist
1: Bitvector bv  # initially set to all zeros
2: for each support variable v ∈ N :
3:   unsigned i = index(v)  # index of variable's bit in bv
4:   if v ∈ fanin(p) : bv[i] = 1 else bv[i] = 0
5: return bv
```

Figure 4.3: High-level procedure to compute support variable information for a property. Every variable is uniquely indexed into the bitvector.

position in the bitvector, and index(v) returns that index for variable v. The function fanin(p) recursively computes the fan-in structure, or COI, of the gate corresponding to property p. If a support variable v is in the fan-in of property p, then the index(v)'th bit is set to "1" in the bitvector; otherwise, the bit is set to "0". The length of such a bitvector, denoted *length(bv)* is equal to the total number of support variables in the netlist, and all bitvectors have the same length. The COI size of the property is the number of bits set to "1", and can be computed using fast population counting algorithms [War12].

The bitvectors can be *packed* by representing every SCC as a single weighted support variable; SCC bits have weight equal to the SCC weight, while others have unit-weight. The COI size of the property equals the weighted sum of the bits set to "1". Note that the choice of whether or not to represent SCCs as a single bit does not affect the resulting support size. Unless stated otherwise, a "support bitvector" is assumed packed.

Practically, it is far too computationally expensive to walk the fan-in cone of every property independently. Instead, the netlist may be traversed once in a topological manner, computing intermediate support bitvectors for internal gates [CCQ16]. E.g., for an AND gate a_1 with incoming edges i_1 and i_2, the intermediate bitvector for a_1 is simply the disjunction

over the bitvectors for i_1 and i_2. For more details on support bitvector computation and optimizations, we refer the reader to [CCQ16].

4.2.2 Property Affinity

The *unpacked* bitvectors for every property can be analyzed to determine *affinity* among the properties. We use "Hamming distance" as an affinity measure; high-affinity properties have nearly-identical bitvectors. The affinity between two properties p_1 and p_2 with *unpacked* bitvectors bv_1 and bv_2 is:

$$0 \leq \text{affinity}(p_1, p_2) = 1 - \frac{\text{hamming}(bv_1, bv_2)}{\text{length}(bv_1)} \leq 1.0$$

where $\text{hamming}(bv_1, bv_2)$ is the Hamming distance between the *unpacked* bitvectors, and $\text{length}(bv_1)$ is the number of support variables in the netlist (identical for every bitvector). Let V_1 and V_2 be the set of support variables in the COI of p_1 and p_2, respectively. Note that $\text{hamming}(bv_1, bv_2)$ equals $(|V_1 \cup V_2| - |V_1 \cap V_2|)$ and $\text{length}(bv_1) \geq |V_1 \cup V_2|$.

4.2.3 Group Center and Grouping Quality

A property p is selected in a group g that represents the group's *center*, or *representative* property, and is denoted as g^*. The *quality* of a group g, denoted $Q(g)$, is the minimum affinity between any property in g with respect to the center property g^*, i.e.,

$$Q(g) = \min(\{\text{affinity}(p_i, g^*) \mid \forall p_i \in g\})$$

A quality of t implies that *unpacked* bitvectors, of length l, for properties in a group have a maximum Hamming distance of $(1 - t) * l$. Our grouping algorithms guarantee that the quality of every group will be greater than a specified threshold.

4.2.4 Localization Abstraction

The proof or counterexample for a property often only depends on a small subset of its COI logic. *Localization abstraction* [MEB+13, MA03, AM04, CCK+02] is a powerful method aimed at reducing netlist size by removing irrelevant logic, transforming irrelevant gates to unconstrained primary input variables via *cutpoint* insertion. Since cutpoints can simulate the behavior of the original gates and more, the localized netlist over-approximates the behavior of the original netlist. Abstraction *refinement* is used to eliminate cutpoints which are deemed responsible for any spurious counterexamples, effectively re-introducing previously-eliminated logic. Ultimately, the abstract netlist is passed to a proof engine. It is desirable that the abstract netlist be as small as possible to enable more-efficient proofs, while being immune to spurious counterexamples.

4.3 Structural Grouping of Properties

Practical industrial verification tasks often entail hundreds of thousands of support variables, and tens of thousands of properties. The need for scalability obviates straight-forward approaches, such as pairwise-comparing each property to check for affinity. We use support bitvectors for a set of properties, and partition them into high-affinity property groups. Our affinity-based algorithm performs grouping in three configurable levels based on: identical bitvectors (level-1), weights of large SCCs in support (level-2), and Hamming distance between bitvectors (level-3). The underlying intuition is that properties with similar bitvectors, measured in terms of a distance metric like Hamming distance, have high structural affinity and can be most efficiently verified as one concurrent multi-property verification task. To ensure overall scalability, each level runs in as close to linear runtime as possible with respect to the number of properties, otherwise the grouping effort itself may severely degrade overall verification resource comprising grouping plus verification of the identified groups.

function structural_grouping (**Properties** P, **Netlist** N, **Level** l, **Affinity** t)

Input: P = properties to group, N = netlist, l = desired grouping level,
t = affinity threshold

Output: G = high-affinity property groups

1: **Groups** $G = \emptyset$ *# initially empty*

2: **for each** Property $p \in P$:

3: **Group** $g = \emptyset$, g.insert(p), G.insert(g) *# initially groups contains only one property*

4: **if** $l \geq 1$: *# idential COI*

5: grouping_level_1 (G, N) *# see Figure 4.5*

6: **if** $l \geq 2$: *# heavy-weight SCCs in COI*

7: grouping_level_2 (G, N, t) *# see Figure 4.6*

8: **if** $l \geq 3$: *# Hamming distance*

9: grouping_level_3 (G, N, t) *# see Figure 4.10*

10: **return** G

Figure 4.4: Algorithm to group properties based on structural affinity.

Figure 4.4 shows our leveled structural grouping algorithm for partitioning properties into high-affinity groups. The algorithm takes as input the properties P, netlist N, and desired grouping level l. Additionally, an affinity threshold t controls the quality of groups formed. Each property is initially assigned its own distinct group, i.e., each group contains only one property. Upon termination, properties in a group are checked concurrently using a verification algorithm portfolio, and different groups are verified independently.

```
function grouping_level_1 (G, N)
Input: G = property groups, N = netlist
1: Hash_function hfun, Hash_table ht
2: for each Group g ∈ G :
3:   Property p = g*   # center property in group
4:   Bitvector bv = support_bitvector (p, N)
5:   unsigned val = hfun (bv)  # hash the bitvector for fast comparison
       # check if another group has identical bitvector
6:   if Group h = hash_lookup (ht, ⟨val, bv⟩) :
7:     group_merge (g, h)  # merge properties in g with h
8:   else # store in hash table for later comparison
9:     hash_insert (ht, ⟨⟨val, bv⟩, g⟩)
```

Figure 4.5: Algorithm to group properties based on identical COI. Properties for which bitvectors hash to the same value are grouped together.

4.3.1 Identical Cones of Influence

The procedure to perform property grouping based on identical support bitvectors is demonstrated in Figure 4.5. The procedure takes an initial property grouping as input, and then merges groups that have identical support bitvectors. g^* denotes the representative property in a group, i.e., g^* is the center. The choice of g^* is trivial at level-1 because every group contains only one property. Next, the support bitvector for the center property in the group is hashed to an integer value. The choice of the hash function is implementation-dependent. We use Murmur3 [App] to hash bitvectors as being very fast and accurate with minimal collisions, however, other functions can also be used. Groups for which the bitvector hashes to the same integer value, and further which have identical bitvectors, are

then merged. Any property in the merged group can be chosen as the new center property without affecting subsequent results.

Theorem 4.3.1. *The level-1 grouping procedure (Figure 4.5) generates high-affinity property groups G such that $\forall g \in G : Q(g) = 1.0$.*

Proof. Initially, every group contains one property, and therefore $Q(g) = 1.0$. The procedure then merges properties with identical bitvectors, i.e., properties with affinity = 1.0 are grouped, and therefore the generated groups have $Q(g) = 1.0$, irrespective of the center. □

While scalable (near-linear runtime) and able to group properties with 100% affinity, in practice it is desirable to perform additional grouping of properties which have a small tolerable Hamming distance yet are still high-affinity. Again, we stress that a simple procedure of pairwise comparison between properties to check whether properties are within a small tolerance is prohibitively slow in practice, rendering prior techniques as [CCL+18, CN11b] unusable in practice. The following algorithms solve this goal of high-affinity group merging, with high scalability and provide guarantees on the grouping quality of generated groups.

4.3.2 Strongly Connected Components

Many practical netlists contain at least one very large SCC, comprising the majority of its registers. For such netlists, all properties that contain the same heavy-weight SCCs in their COI can often be grouped together as having high affinity. Figure 4.6 demonstrates the procedure to perform property grouping based on heavy-weight SCCs. The procedure takes as input an affinity threshold t. For every group g, we find all SCCs in the COI of the center property $p = g^*$, with weight at least w. We use Tarjan's algorithm to find SCCs in the COI of property p in linear runtime. Practically, it is very expensive to find SCCs in the COI of every property independently. Instead, all SCCs are computed once for netlist N along with the linear traversal to compute support bitvectors for properties [LPP+13].

```
function grouping_level_2 (G, N, t)
Input: G = property groups, N = netlist, t = affinity threshold
1:  Trie trie  # initially empty
2:  Weight w  # set heuristically
3:  for each Group g ∈ G :
4:    Property p = g*  # center property in group
5:    Bitvector bv = support_bitvector (p, N)
6:    Set S = find_sccs (p, N, w)  # find SCCs with weight ≥ w in COI of property p
7:    unsigned scc_weight = cumulative_weight(S)
8:    if scc_weight/length(bv) < t :  # check if SCCs contain t% of support variables
9:      continue  # SCCs can't decide affinity for group g
10:   if Group h = trie_lookup (trie, S) :  # check if another group has exact same SCCs
11:     group_merge (g, h)  # merge properties in g with h
12:   else trie_insert (trie, ⟨S, g⟩)  # store in trie for later comparison
```

Figure 4.6: Algorithm to group properties based on heavy-weight SCCs in the COI. Properties that share the same heavy-weight SCCs are grouped together.

If the cumulative SCC weight is at least t times the number of support variables in netlist N, this set of SCCs is inserted into a prefix tree or trie (for fast ∼linear time lookup and prefix matching). A hash table may be used, at the expense of possibly-increased memory footprint. If the trie already contains this set of SCCs, albeit for another group h, the two groups are merged. Any property in the merged group can be chosen as the new center property without affecting subsequent results.

Theorem 4.3.2. *Given affinity threshold t, the level-2 grouping procedure (Figure. 4.6) generates property groups G such that $\forall g \in G : Q(g) \geq t$.*

Proof. Initially, $Q(g) = 1.0$ for all groups $g \in G$. Let n be the number of support variables, therefore for the center property $p = g^* \in g$, we have length$(bv) = n$, where bv represents the support bitvector for property p (lines 4–5). Let $\mathcal{C}$ be the set of SCCs in the cone of influence of property p. The procedure finds set $S \subseteq \mathcal{G}$ such that $\forall s \in S$ we have weight$(s) \geq w$ (line 6). Let $cw = \Sigma_{\forall s \in S}$weight$(s)$, i.e., the cumulative number of support variables in S (line 7). The algorithm then compares cw and $n * t$. We have two cases:

1. $cw < n * t$: SCCs in set S contain fewer than t% of the total number of support variables in the netlist. The SCC's alone cannot be used for deciding the affinity-merge of group g and the procedure proceeds to the next group (lines 8–9).

2. $cw \geq n * t$: SCCs in set S contain greater than t% of the total number of support variables in the netlist. The procedure then finds group $h \in G$ with S in the COI of h^* (trie lookup), merges group g with h, and proceeds to the next group (lines 10–11). If no such group exists, set S is added to the trie for later comparisons (line 12).

The procedure performs a successful merge of group g and h if their respective center properties g^* and h^*, respectively, have identical heavy-weight SCCs (set S) in their COI that contain at least t% of the variables ($cw \geq n * t$). Both g^* and h^* have the same $\geq n * t$ bits set to "1" in their *unpacked* support bitvectors, implying a maximum Hamming distance of $(1 - t) * n$ or minimum affinity of t. Therefore, level-2 grouping generates groups $g \in G$ such that either $Q(g) = 1.0$ (unsuccessful merges), or $Q(g) \geq t$ (successful merges). □

Properties sharing a small number of common large SCCs may thus be adequately high-affinity to group based solely upon analysis of these SCCs, without needing to consider a potentially very large number of non-SCC support variables or smaller SCCs. In contrast, storing every full bitvector in a trie may become computationally expensive and serve little benefit. Since the subsequent level-3 grouping does take non-SCC support variables into

account, minimum SCC weight w is typically set to at least 1% of the total number of support variables in the netlist, and possibly substantially larger like 10%, for fastest runtime without impacting the overall grouping results.

4.3.3 Hamming Distance

Classical clustering techniques, like k-medoids [PJ09] ($\mathcal{O}(n^2)$ time complexity) and heirarchical clustering based on a distance metric like Hamming distance[RM05] ($\mathcal{O}(n^2 \log n)$ time complexity), are slow and do not scale well with the number of clustered items [AV06]. They require expensive computation of a distance matrix that maintains the distance between every pair of items (guaranteed to require at least quadratic resources), and the number of clusters to generate as an input parameter. In a verification context, it is prohibitively slow to perform a quadratic number of bitvector comparisons [CCL+18, CN11b] on netlists with millions of support variables. Plus, it is impossible to a-priori know how many high-affinity groups are a natural fit for the given multi-property netlist, until the affinity analysis and grouping are completed. Classical clustering algorithms are thus unsuitable for our goal.

A third component of our grouping procedure is an approximate clustering algorithm to scalably cluster bitvectors based on Hamming distance. Figure 4.7 demonstrates the clustering algorithm. The algorithm takes as input a set of *unpacked* bitvectors BV, word size n, and an affinity threshold t. As an initialization step, the algorithm first uses an off-the-shelf clustering algorithm [RM05, Gon85] to cluster all n-bit numbers into k clusters such that quality of every cluster is at least $\hat{t} = 1 - \lfloor (1-t) * n \rceil \div n$ ($\lfloor x \rceil$ is the nearest integer function); a map m is maintained that maps every n-bit number $(0, 1, \ldots, 2^n - 1)$ to the allotted cluster center $(1, \ldots, k)$. For a fixed value of n, the number of clusters k can be increased one-by-one until quality of each is at least $\hat{t}$, i.e. the maximum Hamming distance allowed per n-bit segment in a cluster is $(1 - \hat{t}) * n$. E.g. for $n = 32$ and $t = 0.95$, the

function bitvector_cluster (BV, t, n)

Input: BV = set of bitvectors to cluster, t = affinity threshold

n = word size for clustering

 # initialization step

Map m, **unsigned** k

m = generate_map (t, n) *# see Figure 4.8*

 # clustering step

Hash_function hfun, **Hash_table** ht

Clusters $C = \emptyset$ *# initially empty*

for each Bitvector bv in BV :

  unsigned num = ceil(length(bv)/n) *# number of words*

  unsigned $mbv[num]$ *# mapped bitvector*

  for i in $0, \ldots, num - 1$: *# generate mapped bitvector*

    $mbv[i] = m[bv[i]]$

 # hash and insert into ht. If $\langle val \rangle$ exists as key, add new $\langle bv \rangle$ value to this key

  unsigned val = hfun(mbv), hash_insert_multi($ht, \langle val, bv \rangle$)

for each entry $\langle \langle val \rangle, bv[] \rangle$ in ht :

  Cluster $c = bv[]$, C.append(c) *# bitvectors with key $\langle val \rangle$*

return C

Figure 4.7: Algorithm to cluster bitvectors based on Hamming distance. The initialization step may be computed offline and reused across runs.

function generate_map (t, n)

Input: t = affinity threshold, n = word size of numbers to cluster

1: **Set** $S = \{0, 1, \ldots, 2^n - 1\}$ *# all n-bit numbers*

2: **unsigned** k *# number of clusters for items in S*

3: **Map** m *# m stores map of n-bit number* $\rightarrow 1, \ldots, k$

 # generate clusters s.t. each has quality $\hat{t} = 1 - \lfloor (1-t)*n \rceil \div n$

4: m = cluster (S, t) *# increase k to match* $\hat{t}$

5: **return** m

Figure 4.8: Algorithm to cluster n-bit numbers based on Hamming distance. The procedure return a function $m : n \rightarrow c$ that assigns each n-bit number to a cluster index c. The n-bit numbers with identical cluster indexes are within a Hamming distance of $(1 - \hat{t}) * n$.

maximum hamming distance is $(1 - 0.95) * 32 = 1.6 \approx 2$ for which $\hat{t} = 0.9375$. Note that the initialization step involving clustering does not hinder scalability because:

1. The value of n is typically less than the maximum CPU word size that allows fast single-cycle Hamming distance computation between two numbers (XOR); clustering with $t = 0.9$ for n=16 and 32 takes <1s and <1min, resp.

2. The map can be computed once offline, and reused in all future runs of the algorithm (e.g. embedded into a verification tool) for various ranges of threshold t.

3. For online computation, an approximate linear-time algorithm, like Gonzalez [Gon85], can be used on S that may only contain n-bit numbers appearing in bitvectors BV.

In the clustering step, every *unpacked* bitvector bv is read in n-bit segments to generate a piecewise-mapped bitvector mbv using map m. Figure 4.9 gives an example to generate mapped bitvectors from *unpacked* bitvectors for $n = 16$. Bitvectors for which the corresponding mapped bitvectors hash to the same value are put in the same cluster.

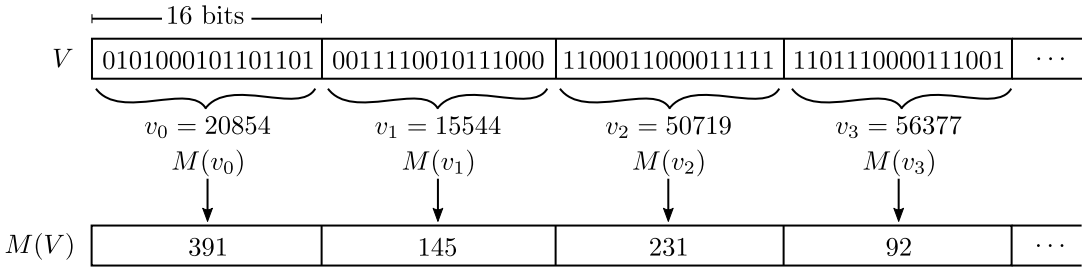


Figure 4.9: Generate mapped bitvector $M(V)$ by mapping 16-bit segments of bitvector V using a map that assigns all numbers $0, 1, 2, \ldots, 2^{16} - 1$ to an index between $1, 2, \ldots, k$. Bitvectors with identical mapped bitvectors are grouped.

Theorem 4.3.3. *Given a set of unpacked bitvectors BV, affinity t, and word size n, the* bitvector_cluster *procedure (Figure 4.7) returns cluster set C of bitvectors such that:*

1. $\forall c \in C : c \subseteq BV$ *and* $|c| > 0$, *i.e., each cluster is a subset of bitvector BV,*
2. *for each bitvector $bv \in BV$ and clusters $c_i, c_j \in C$, if $bv \in c_i$ and $bv \in c_j$, then $i = j$, i.e., bitvector bv is present in at most one cluster,*
3. $\bigcup_{c \in C} c = BV$, *i.e., every bitvector is assigned a cluster, and*
4. $\forall c \in C : Q(c) \geq \hat{t}$, *where* $\hat{t} = 1 - \lfloor (1-t) * n \rceil \div n$.

Proof. We assume that the procedure generate_map (Figure 4.8) returns a map that assigns numbers $0, 1, 2, \ldots, 2^n - 1$ to indexes $1, 2, \ldots, k$, and numbers with identical indexes have their corresponding n-size bitvectors within a Hamming distance of $(1 - \hat{t}) * n$, where $\hat{t} = 1 - \lfloor (1-t) * n \rceil \div n$, implying an affinity of $\hat{t}$ (line 2). The set of clusters C is initially empty. The procedure iterates over all *unpacked* bitvectors $bv \in BV$ (lines 5–10). Let num be the number of full n-bit words (segments) in bitvector bv (line 6), i.e., length$(bv) \leq n * num$ and length$(bv) > n * (num - 1)$. The procedure generates a mapped bitvector mbv by mapping every n-bit segment to an index between $1, 2, \ldots, k$ (lines 8–9). The mapped bitvector is then inserted into a hash table for later comparisons (line 10). We assume that the hash function

used is collision-free.[2] The procedure then iterates over all entries in the hash table (lines 11-12). For the mapped bitvectors that hash to the same value, the corresponding unpacked bitvectors are collected to form clusters $c \in C$. All other *unpacked* bitvectors form singleton clusters. Therefore, every $bv \in BV$ is assigned to a cluster $c \in C$. If two mapped bitvectors hash to the same value, then every i^{th} n-bit segment in the two original *unpacked* bitvectors is at a maximum distance of $(1-\hat{t}) * n$. Therefore, the maximum distance between the two *unpacked* bitvectors is $(1-\hat{t}) * n * num$ or $(1-\hat{t}) * \text{length}(bv)$, implying a minimum affinity of $\hat{t}$. Every cluster $c \in C$ is either singleton (implying $Q(c) = 1.0$) or contains unpacked bitvectors within a Hamming distance of $(1-\hat{t}) * \text{length}(bv)$ (implying $Q(g) \geq t$, irrespective of the chosen cluster center). □

Figure 4.10 demonstrates the procedure to perform property grouping based on Hamming distance using the bitvector clustering algorithm of Figure 4.7. The algorithm generates a map m of n-bit numbers to cluster centers $1, \ldots, k$ as an initialization step. The center property *unpacked* bitvector for every group is read per n-bit segment, to generate a mapped bitvector using map m. The mapped bitvector is hashed to an integer value. The groups for which the center property mapped bitvectors hash to the same value, and further which have identical mapped bitvectors, are immediately merged.

Theorem 4.3.4. *Given affinity t and word size n, the level-3 grouping procedure (Figure 4.10) generates property groups G such that $\forall g \in G : Q(g) \geq 2 * t + \hat{t} - 2$, where $\hat{t} = 1 - \lfloor (1-t) * n \rceil \div n$.*

Proof. We assume that the procedure generate_map (Figure 4.8) returns a map that assigns numbers $0, 1, 2, \ldots, 2^n - 1$ to indexes $1, 2, \ldots, k$ (line 2), and the hash function is collision-free. Initially, G contains singleton groups, level-1 groups, or level-2 groups. The procedure iterates over every group $g \in G$ (lines 4–15). Let bv be the *unpacked* support bitvector

[2] A hash function H is collision free if $H(x) = H(y)$ if and only if $x = y$

```
function grouping_level_3 (G, N, t, n)
Input: G = property groups, N = netlist, t = affinity threshold,
       n = word size for clustering
   # initialization step (can be computed online/offline)
1: Map m, unsigned k
2: m = generate_map (t, n)  # see Figure 4.8
   # clustering step
3: Hash_function hfun, Hash_table ht
4: for each Group g ∈ G :
5:   Property p = g*  # center property in group
6:   Bitvector bv = support_bitvector (p, N)
7:   unsigned num = ⌈length(bv)/n⌉  # number of words
8:   unsigned mbv[num]  # mapped bitvector
9:   for i in 0, ..., num − 1 :  # generate mapped bitvector
10:    mbv[i] = m[bv[i]]
11:  unsigned val = hfun (mbv)  # hash the mapped bitvector for fast comparison
       # check if another group has identical mapped bitvector
12:  if Group h = hash_lookup (ht, ⟨val, mbv⟩) :
13:    group_merge (g, h)  # merge properties in g with h
14:  else # store in hash table for later comparison
15:    hash_insert (ht, ⟨⟨val, mbv⟩, g⟩)
```

Figure 4.10: Algorithm to group properties based on Hamming distance. Properties for which mapped bitvectors hash to the same value are grouped together.

for property g^*. The procedure generates a mapped bitvector mbv by mapping every n-bit segment of bv to an index between $1, 2, \ldots, k$ (7–10). It then checks if there exists another group h with identical mapped bitvector mbv for $h^* \in h$ (line 12). If no such group exists, the mapped bitvector for property g^* is then inserted into a hash table for later comparisons. If group h exists, then the *unpacked* bitvectors for g^* and h^* are within a Hamming distance of $(1 - \hat{t}) * \text{length}(bv)$ (follows from Theorem 4.3.3). The remainder of the proof follows from triangle inequality of Hamming distance. Properties within groups g and h are at a maximum distance of $(1 - t) * \text{length}(bv)$ (follows from Theorem 4.3.1 and Theorem 4.3.2) from their respective center properties. Therefore, maximum distance between a property in g and another property in h is $2*(1-t)*\text{length}(bv)+(1-\hat{t})*\text{length}(bv)$, or $(1 - (2 * t + \hat{t} - 2)) * \text{length}(bv)$, implying a minimum affinity of $2 * t + \hat{t} - 2$. Therefore, level-3 grouping generates groups $g \in G$ such that either $Q(g) = 1.0$ (for singleton and level-1 groups), $Q(g) \geq t$ (for level-2 groups), or $Q(g) \geq 2 * t + \hat{t} - 2$ (for groups generated by merging singleton, level-1, or level-2 groups). □

When $\hat{t} = t$, level-3 returns groups with $Q(g) \geq 3 * t - 2$. Despite its provable threshold, there is some asymmetry in this approach, in that two fairly-high-affinity bitvectors which differ too much in a single segment will not be merged, whereas if the difference was small per-segment with multiple segments differentiated, they may be merged, albeit respecting the quality bound. The highly scalable analysis can be repeated if higher precision and symmetry is desired. This can be done either as-is on the entire netlist under different permutations or segment-partitioning of bitvector indices (i.e., by varying the starting index of the first n-bit segment in the bitvector), or on individual (sets of) groups obtained from the prior run. Since re-running on a subset of properties implies a smaller cone-of-influence, bitvectors can be compacted for faster runtime to only include support variables in the COI of any considered property, and this indexing will differ from the prior run over a larger

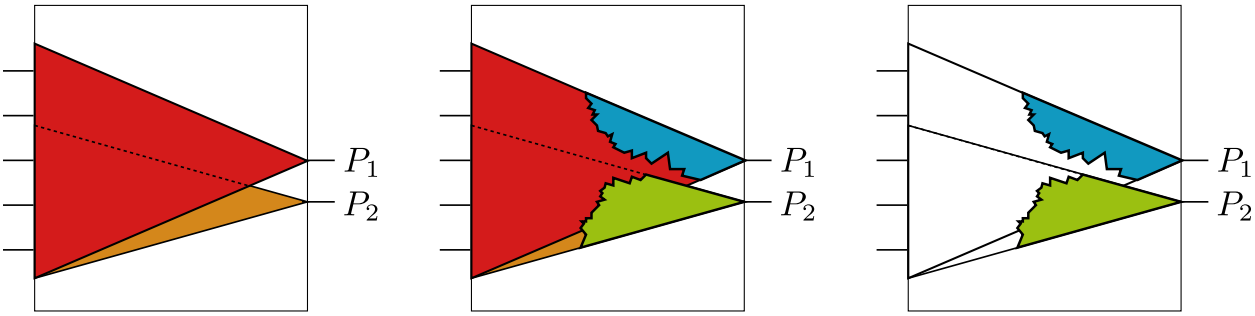
set of properties. Moreover, support variables present in the COI of every property can be completely projected out of the bitvectors to offer further compaction and speedup.

4.4 Semantic Refinement of Property Groups

It is desirable that the netlist generated by localization abstraction be as small as possible to enable efficient proofs. Localization cutpoints are property-specific, hence concurrent localization of properties with disjoint COIs - or even similar COIs - may yield significantly larger netlists which are less scalable to verify. Our structural property grouping procedure ensures that only high-affinity properties in a group will be localized concurrently, which helps ensure smaller multi-property abstractions. Figure 4.11a shows two high-affinity properties P_1 and P_2. However, it might be the case that a cutpoint is refined for one property in a high-affinity group, whereas that refinement may be unnecessary for another property in the group. As a result, properties in a high-affinity group without localization cutpoints may have vastly different COI in the localized netlist. Figure 4.11b shows the abstract netlist for properties P_1 and P_2 learned using localization abstraction. Therefore, partitioning the group obtained from Figure 4.4 into high-affinity localized subgroups based upon localization decisions can improve overall verification scalability. Figure 4.11c shows the abstract netlist for properties P_1 and P_2 that may be utilized to repartition properties P_1 and P_2 using structural grouping with respect to the abstract netlist.

4.4.1 Abstract Cone-of-Influence Computation

Various techniques have been proposed [MA03, CCK+02, AM04] to guide the abstraction-refinement process of localization. Most state-of-the-art localization implementations use SAT-based bounded model checking (BMC) [BCCZ99] to select the localized netlist upon which an unbounded proof is attempted. In our implementation we run BMC iteratively



(a) High-affinity **(b)** Learned abstraction **(c)** Abstract affinity

Figure 4.11: Two high-affinity properties with respect to the original netlist can have vastly different cones of influence with respect to the localized abstract netlist.

until there is no change in the localized netlist. Figure 4.12 shows our localization abstraction framework that supports high-affinity group partitioning. We start with a localized netlist only containing property gates. For a given BMC depth k, we iterate over properties in group g to eliminate all spurious counterexamples of length k. Cutpoints deemed necessary to refine for a property p are collected (line 12). If a cutpoint is also a support variable, it is then added to the support bitvector maintained for property p (line 13). The abstraction is then refined using the collected cutpoints, and BMC is run again at depth k. When all properties hold for the abstract model at depth k, BMC is run again with depth $k+1$. The repeated BMC runs add new cutpoints to the support bitvector for every property, which in turn can be used to partition group g into high-affinity subgroups with respect to the localized netlist. Various strategies may be used to decide when to terminate BMC: an upper-bound on BMC depth or runtime can be used. In our framework, we prefer increasing BMC depth until there is no change in the localized netlist for n consecutive steps (lines 16–17). The value of n can be varied to increase confidence in the abstracted model such that it is immune to spurious counterexamples.

```
function localization (g, N, n, t)
Input: g = group to partition, N = netlist, n = word size for clustering,
       t = affinity threshold
1:  Netlist L = initial_abstraction(g)  # localized netlist, add gates for every property
2:  unsigned k = 0  # bmc depth
3:  bool stop = 0  # some properties fail at depth k
4:  while not stop :  # loop until all properties pass at depth k
5:    stop = 1
6:    Gates c = {}  # cutpoints to refine in L, initially empty
7:    for each Property p ∈ g :
8:      Result r = run_bmc(L, p, k)  # run bmc with depth k
9:      if r == unsat : continue  # property passes
10:     if cex not spurious : report_solved(p, cex), continue  # check counterexample
11:     stop = 0  # property fails
12:     Gates d = cutpoints_to_refine(), c = c ∪ d
13:     collect_support_info(p, d)  # add to support bitvector
      # at least one property passes at depth k
14:   if not stop : refine_abstraction(L, c), unchanged = 0
15:   else unchanged += 1  # no change in abstraction
    # check if netlist unchanged for last n bmc steps
16: if unchanged < n : k = k + 1, goto line 3  # increment depth
17: else Groups Ĝ = structural_grouping(g, L, 3, t)
    # run proof engine for each group in Ĝ with netlist L
```

Figure 4.12: Localization to partition a group g of high-affinity properties. BMC is run for increasing depth until there is no change in the localized netlist, after which partitioning is attempted to split g into subgroups $\hat{G}$.

4.4.2 Semantic Partitioning

Once BMC converges, group g is then partitioned into subgroups $\hat{G}$ based on support bitvector information. Note that the problem is analogous to grouping of properties in g with respect to the localized netlist. Therefore, we use the property grouping procedure of Figure 4.4 to generate high-affinity property groups for overall scalability.[3] The properties in each subgroup are then passed to a proof engine for verification with respect to each COI-reduced localized subgroup's netlist.

4.5 Experimental Analysis

In this section, we report on our extensive experimental analysis with our high-affinity property grouping algorithm, and semantic group partitioning based on localization on end-to-end verification scalability. We briefly detail our benchmarks, summarize the setup used for the experiments, and end with experimental results and a discussion of results.

4.5.1 Benchmarks

4.5.1.1 Benchmarks from HWMCC

We evaluate 48 benchmarks from HWMCC that contain more than 100 safety properties (Figure 4.13a). These are obtained by simplifying all the benchmarks by standard logic synthesis (similar to `&dc2` in ABC [BM10]) to solve easy properties, and disjunctive decomposition to fragment each OR-gate property into a sub-property of its literals. Each property, or property group, is solved using a portfolio comprising BMC [BCCZ99], IC3 [Bra11, EMB11],

[3] Off-the-shelf clustering is more applicable here than on the original netlist if desired, because: (1) the localized netlist and support bitvectors are often immensely smaller than the original netlist; (2) the number of properties per structural group being localized is often smaller than the number of overall netlist properties. However, there is no guarantee of either of these points.

and localization (LOC) without semantic partitioning. Each can process multiple properties: IC3 and BMC in a time-sharing manner, and LOC concurrently abstracting a set of properties which are solved using IC3.

4.5.1.2 Proprietary Benchmarks

Post-silicon observability solutions often leverage monitoring logic instrumented throughout a hardware design. This *debug bus* logic monitors a configurable set of internal signals in real-time, non-intrusively while the chip is functionally running. Debug bus verification entails a large number of properties (often one per monitor point), within very large design components - sometimes entire chips [GBS+06]. We evaluate the impact of high-affinity property grouping on 9 debug bus benchmarks from IBM. The number of properties in each evaluated benchmark ranges from a few tens to thousands.

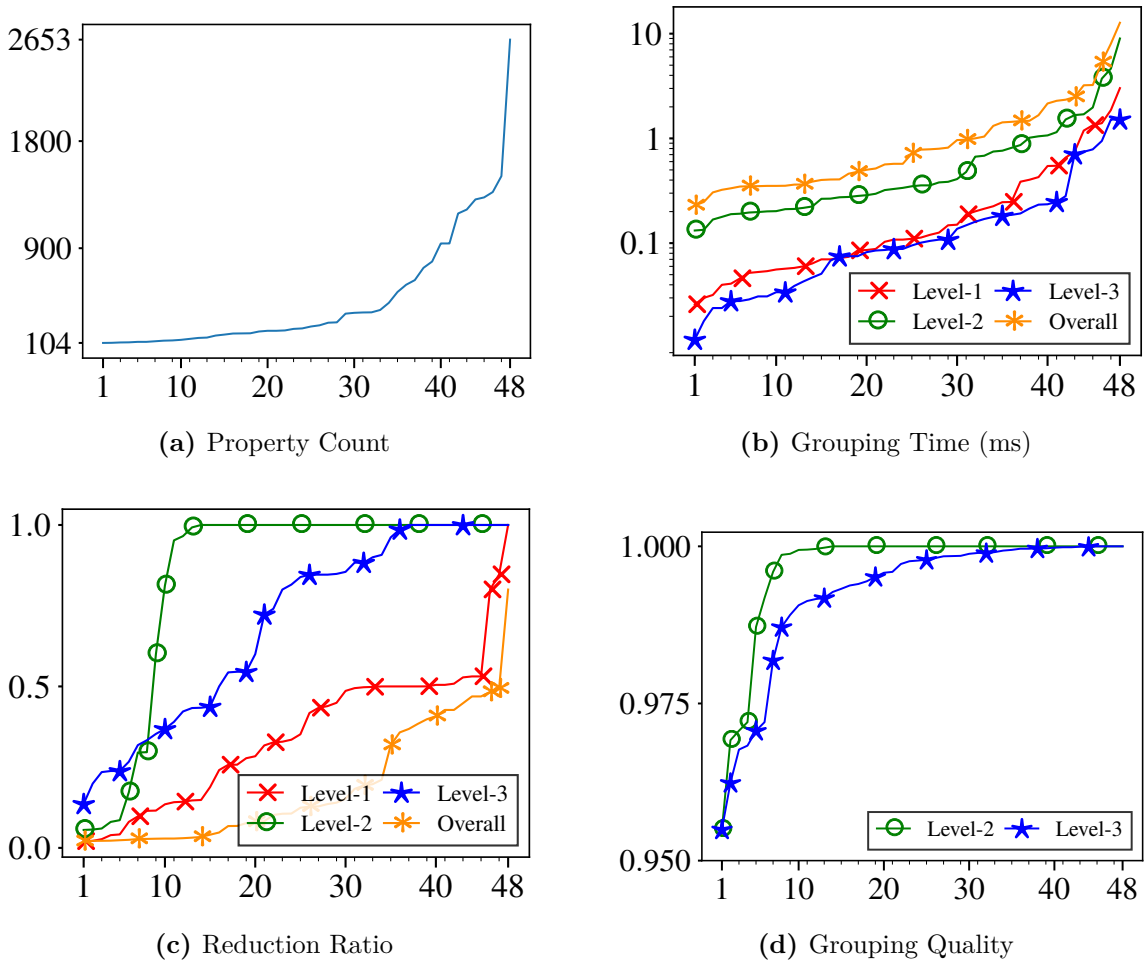
4.5.2 Experiment Setup

Our grouping procedure is implemented within *Rulebase: Sixthsense Edition*[MBP+04]. All experiments were run on Linux machines, with 32GB memory. Time reported is 'cpu' time. We refer to different grouping levels as L_1, L_2, and L_3.

4.5.3 Experimental Results

4.5.3.1 Benchmarks from HWMCC

Property Grouping Support bitvector computation is fast, and takes less than five seconds on the largest benchmark. The ideal threshold is benchmark- and solver-specific. Given the exponential penalty of grouping lower-affinity properties vs. linear penalty of splitting higher-affinity properties (offset by parallel solving), we find it best to err to the latter using a higher affinity t=0.9. L_3 is done using 16-bit words and $\hat{t} = 0.875$, i.e.,



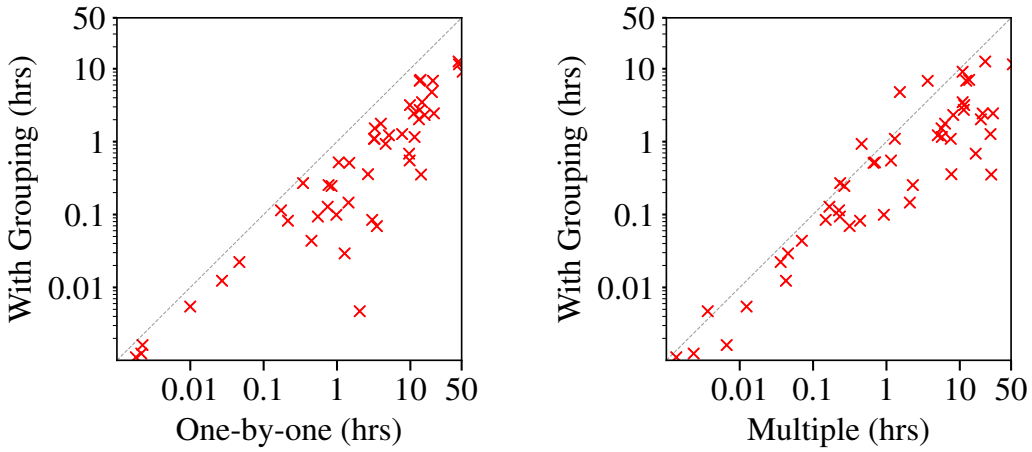
(a) Property Count

(b) Grouping Time (ms)

(c) Reduction Ratio

(d) Grouping Quality

Figure 4.13: Grouping on 48 HWMCC benchmarks with more than 100 properties, and maximum 50 properties/group. Level-1 grouping quality is 1.0.



(a) Single property **(b)** All properties

Figure 4.14: End-to-end verification with grouping vs. portfolio which (a) checks properties one-at-a-time, and (b) check all properties together. Points below diagonal are in favor of verification with grouped properties.

maximum distance of 2 between words (Figure 4.7). Initially each property is assigned to its own distinct group. The grouping takes less than 10ms for all benchmarks (Figure 4.13b). The group count reduction ratio for each level with respect to the preceding level, and overall reduction ratio, i.e., number of groups relative to preceding level, is shown in Figure 4.13c. L_2 merges properties for 13 benchmarks: <0.5 ratio for 8 benchmarks, and is critical to the performance of L_3. Without L_1 and L_2, not all properties merged by L_2 are merged by L_3 due to inherent asymmetry, and L_3 can merge the same properties as L_1, albeit, with small runtime penalty. Therefore, the leveling order is crucial and gives tighter control on group affinity. Figure 4.13d shows the minimum quality of all non-singleton groups in a benchmark.

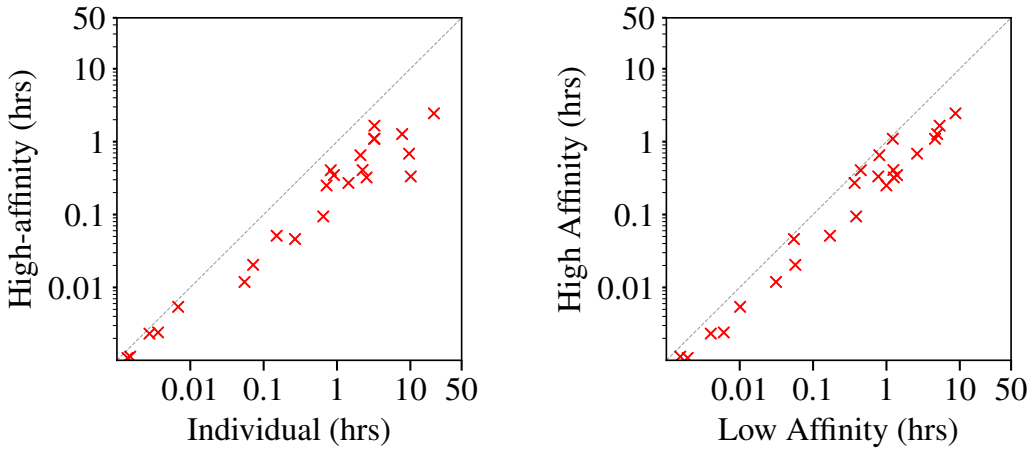
End-to-end Verification We compare the runtime of checking each property one-by-one vs. checking property groups in Figure 4.14a; verification with structural grouping is up to 400× (median 4.3×) faster. A fairer comparison of the runtime of checking all properties together vs. checking property groups is shown in Figure 4.14b; grouped verification is up

Table 4.1: Performance evaluation with one-by-one, multiple, and grouped properties

Name	#Prop	One-by-one	Multiple	#G	Grouped
6s281	105	0.32h	0.22h	84	0.26h
bobsmvhd3	138	4.36h	0.43h	53	0.92h
6s380	149	1.92h	12.54s	12	16.95s
bob12s08	206	13.04h	3.45h	88	6.80h
6s381	1506	18.60h	1.45h	192	4.76h

to 72× (median 3.5×) faster. Table 4.1 shows benchmarks for which checking all properties together is faster. LOC solves very few properties for these benchmarks, whereas, BMC/IC3 quickly verify all properties together: 145 properties in 6s380 are falsified by BMC in a few unrollings and remaining proved by LOC, while all properties are proved by LOC or IC3 for other benchmarks. The benchmarks in Table 4.1 have properties where a large majority are either all falsified, or proved. The advantage of checking high-affinity groups is outweighed by the added cost of repeating BMC/IC3 across groups for these benchmarks, which could be adjusted for using a lower affinity threshold. However, grouping advantage is apparent for benchmarks in which no single algorithm solves all properties, and properties have different verification outcomes, i.e., proved or failed with respect to the model.

Localization Abstraction We select 24 benchmarks having at least 50 properties solved by LOC. Properties not solved by LOC are not considered. Figure 4.15 shows the impact of high and low affinity grouping on the performance of LOC. If high-affinity structural grouping returns N groups, low-affinity grouping is done by sorting properties by COI size, and partitioning into equally-sized N groups. Figure 4.15a compares verification of high-affinity grouped properties and one-by-one checking of each property with LOC; verification is up to 30× (median 2.9×) faster. On the other hand, low-affinity groups often degrade LOC performance compared to one-by-one checking. Figure 4.15b compares high and low



(a) Structural Grouping **(b)** Random Grouping

Figure 4.15: Verification performance of (a) high and (b) low affinity grouping on LOC with respect to checking all properties one-by-one.

affinity group verification with LOC. Five benchmarks have comparable performance due to grouping of large number of properties into very few groups. Nevertheless, high-affinity verification is always faster: up to 3.7× (median 2.5×).

Semantic Partitioning LOC generates a localized netlist using BMC for every property group which is then checked by a proof engine. If the localization is sufficient, the proof engine may prove all properties in a single run. Otherwise, it generates a possibly-spurious counterexample. Table 4.2 shows benchmarks in which some non-singleton groups are proved by LOC in a single proof-engine run. We perform semantic partitioning on these groups. 'Total' shows the #**G**roups generated by structural grouping for #**P**rops, whereas, 'Single Run' shows the #**G**roups and #**P**rops solved by one proof engine run after generating a sufficient localized netlist. All groups are solved by LOC one-by-one. As is evident, semantic partitioning boosts the performance of LOC for hard problems (in bold). However, there is a marginal slowdown for easy problems due to the overhead of restarting the proof engine on semantically partitioned subgroups.

Table 4.2: Verification performance with semantic partitioning of high-affinity groups

Name	Total		Single Run		Verification Time		
	#G	#P	#G	#P	Disabled	Enabled	Speedup
6s384	2	51	1	27	22.65s	36.76s	0.61×
6s344	12	247	3	65	2.04h	0.65h	3.13×
6s405	13	593	3	134	0.28h	0.21h	1.34×
6s410	15	735	4	121	0.18h	0.12h	1.50×
6s110	15	186	5	73	82.13s	81.43s	1.00×
6s391	30	144	9	32	25.61s	43.12s	0.60×
6s332	77	163	16	45	1.21h	0.75h	1.62×

Table 4.3: Performance comparison between high-affinity property grouping and property grouping based on hierarchical clustering

Name	#P	Our Procedure			Hierarchical			#Loss
		#G	G.Time	V.Time	#G	G.Time	V.Time	
6s405	593	13	2.16ms	1.04h	13	12.43s	1.04h	0
6s381	1506	192	5.93ms	4.76h	76	36.62s	3.01h	96
6s361	2653	84	12.71ms	355.76s	62	107.14s	293.14s	11
6s117*	8063	173	25.53ms	8.13h	165	1.07h	7.45h	4
6s114*	30628	1612	0.42s	0.76h	873	2.65h	0.52h	412

* Not simplified by logic synthesis

Lossy Grouping Lastly, we compare the grouping loss using our procedure with hierarchical clustering (HC) [RM05]. We measure loss as #properties assigned a group by HC but not our procedure (maximum 50 properties/group). Table 4.3 summarizes results for five representative benchmarks. HC always takes more grouping time. There is no loss in benchmarks for which both methods return very few groups (e.g., 6s405). Verification with fewer groups from HC is faster (e.g., 6s381) when our procedure has higher loss. This loss may be due to **1)** properties having an almost identical set of SCCs but differing in a few small SCCs: these are not grouped due to trie prefix mismatch, and **2)** asymmetry in L3, which can be mitigated by using techniques in Sec. 4.3.3. In most cases, HC gives fewer

Table 4.4: End-to-end verification speedup on debug bus designs with high-affinity property grouping

ID	#P	#G	G.Time (ms)	Verification Time		
				One-by-one	Grouped	Speedup
1	36	9	0.48	32.69s	19.27s	1.70×
2	45	3	0.49	26.24s	12.63s	2.08×
3	56	5	0.94	11.9s	6.34s	1.88×
4	76	36	3.87	0.21h	0.14h	1.40×
5	148	4	0.68	95.83s	22.68s	4.23×
6	224	6	0.74	65.52s	19.65s	3.34×
7	1506	53	9.16	0.93h	0.21h	4.32×
8	9371	1027	137.72	52.65h	11.89h	4.43×
9	11035	1238	146.32	7.94h	2.81h	2.82×

groups which may result in less verification time, but HC grouping resource results in an end-to-end runtime degradation vs. our approach. It is clear that HC gives tighter groups but overall verification resource is dominated by the time it takes to perform grouping.

4.5.3.2 Proprietary Benchmarks

Post-silicon observability solutions often leverage monitoring logic instrumented throughout a hardware design. This *debug bus* logic monitors a configurable set of internal signals in real-time, non-intrusively while the chip is functionally running. Debug bus verification entails a large number of properties (often one per monitor point), within very large design components - sometimes entire chips [GBS+06].

Localization is the dominant method to verify debug bus designs as they often contain >10M gates [GBS+06]. Note that concurrent verification of all properties is completely intractable. Table 4.4 summarizes our results. 'One-by-one' shows verification time by localizing one property at a time, whereas, 'Grouped' represents concurrent localization of

properties in a high-affinity group. All designs benefit from high-affinity group verification, and the speedup is clearly evident for large designs (in bold) with thousands of properties.

4.6 Summary and Discussion

Scalable property grouping is a hard problem. Existing approaches are either syntax-based [CM10], or resource intensive [CCL+18]. The need for scalability cannot be over-stated; traditional grouping algorithms require at least quadratic runtime vs. number of properties, and are prohibitively slow–adding to and easily outweighing the benefit they bring to the verification process. We present a 2-step grouping strategy: strucural grouping followed by semantic partitioning, that offers massive end-to-end verification speedup. Experiments demonstrate the usefulness of our method on several verification tasks: structural grouping is trivially fast regardless of subsequent verification engines, and semantic partitioning accelerates difficult localization problems. We advance state-of-the-art in localization by providing an optimal multi-property solution.

An efficient multi-property verification algorithm is key to minimize the effort required for verifying individual designs in a design space. The high-affinity groups of properties can be checked by incremental algorithms, like FuseIC3 (Chapter 3), to maximize the amount of information reused across runs. The different groups can be checked in parallel to maximize throughput. The locality sensitive hashing (LSH)technique (Section 3.4.1) to group properties helps improve end-to-end verification performance, but the offline grouping procedure itself can be computationally prohibitive on designs with thousands of properties. The approximate three-level grouping procedure trades accuracy vs. speed, and obviates the need for pairwise comparisons, or multiple hashes on large documents (Section 3.4.3). Nevertheless, the LSH-based grouping technique is extremely useful for tasks where exact affinity computation may be required. Property grouping also impacts the performance

of BDD-based model-checking algorithms; concurrent verification of high-affinity properties leads to smaller BDDs and faster verification closure. While grouped-property verification helps in reducing CPU-time by concurrently verifying properties, thereby reducing power consumption, competition for available machines and IT costs, optimizing overall wall-time for design-space exploration is a comparably-important goal. Parallel verification of groups helps in this regard, however, existing approaches to verify groups in parallel suffer from serious limitations. They degrade into highly-redundant work across processes, and fail to optimally utilize available processes for work distribution. The problem is acute for verification tasks with thousands, or even millions of properties, as is often the case in equivalence checking. In the next chapter, we optimize the property grouping algorithm (Figure 4.4) for parallel verification. We discuss heuristics that help improve the throughput of parallel verification tasks, and help organize parallel verification tasks to minimize redundant work and optimize work distribution across workers in a parallel verification environment.

CHAPTER 5. PARALLEL ORCHESTRATION

Multi-property verification combined with efficient incremental verification algorithms is key for scalable design-space exploration using model checking. Multiple properties can be partitioned into high-affinity groups using the algorithm presented in Chapter 4. The high-affinity groups of properties can be checked against the reduced set of models, corresponding to the pruned design space, by incremental algorithms, like FuseIC3 presented in Chapter 3, to maximize the amount of information reused across runs. The different high-affinity property groups and individual models can be checked in parallel to maximize throughput. However, utilizing a parallel verification environment to improve overall verification performance requires careful considerations.

Verification tools often utilize parallelism in their solving orchestration to improve scalability, either in *portfolio* mode where different solver strategies run concurrently, or in *partitioning* mode where disjoint property subsets are verified independently. While most tools focus solely upon reducing end-to-end wall-time, reducing overall CPU-time is a comparably-important goal influencing power consumption, competition for available machines, and IT costs. Portfolio approaches often degrade into highly-redundant work across processes, where similar strategies address properties in nearly-identical order. Partitioning should take *property affinity* into account, atomically verifying high-affinity properties to minimize redundant work of applying identical strategies on individual properties with nearly-identical logic cones. Existing algorithms for property partitioning are either computationally-prohibitive, or do not optimally utilize available parallel processes. They may generate fewer groups than processes, or lose affinity guarantees when requiring number of groups as an algorithmic parameter. Therefore, utilizing parallelism to boost verification performance is far from

trivial. In this chapter, we extend the property grouping algorithm of Chapter 4 to optimally utilize the number of available parallel workers, and propose heuristics that improve the performance of parallel verification tasks. We specifically answer the following questions: 1) *What are the common problems with parallel verification that limit verification throughput?* 2) *How to optimize property grouping without losing affinity guarantees of individual property groups?* 3) *How to utilize property grouping efficiently in a verification portfolio to minimize redundant work across processes, and optimize work distribution?*

The rest of the chapter is organized as follow: Section 5.1 overviews our contributions to efficiently partition properties into high-affinity groups that guarantee complete utilization of available parallel workers, highlights common performance impediments in parallel verification approaches, and contrasts with related work. Section 5.2 gives background information, introduces formalisms, and reviews structural property grouping from Chapter 4. Section 5.3 describes the property grouping algorithm for parallel verification, and details heuristics that improve the overall parallel verification performance. We then describe methods to optimize localization abstraction for equivalence checking using our techniques in Section 5.4. Section 5.5 provides a large experimental evaluation of our techniques for parallel verification tasks, and details a state-of-the-art localization abstraction portfolio designed using our techniques. Section 5.6 concludes the chapter by discussing the applicability of the proposed methods to other common verification scenarios.

5.1 Introduction

Practical hardware and software verification often mandates checking a large number of properties on a given design. For example, *design-space exploration* using model checkign requires evaluating several properties for individual models in the design space. *Functional verification* involves checking a suite of low-level assertions and higher-level encompassing

properties. *Equivalence checking* compares pairwise equality of each output across two designs, yielding a distinct property per output. *Redundancy removal* requires proving many gate-equalities throughout a design, each comprising a distinct property. The redundancy removal process is the core procedure of equivalence checking, and is widely-used to boost verification scalability of functional verification tasks.

State-of-the-art tools verify multiple multiple properties by optimizing *problem orchestration*. We differentiate between three verification orchestration strategies:

1. *Atomic verification* refers to running a set of single-process verification engines (called a *strategy*) on a group of properties.

2. *Serial verification* refers to beginning one atomic verification task after another atomic verification task finishes, using only a single process.

3. *Parallel verification*, or *concurrent verification* refers to dispatching multiple atomic tasks on concurrently-running parallel processes.

Each property has a distinct minimal *cone of influence* (COI), or fan-in logic of the signals referenced in the property. Verification of a group of properties requires resources proportional to the collective COI size, which is often exponential (after lighter logic reductions). Each property adds distinct logic to the group's collective COI; *affinity* refers to the degree of common vs. distinct logic in the COI. Atomic verification of a group of low-affinity properties is thus often significantly slower than solving them one-at-a-time. Conversely, atomic verification of a high-affinity group saves considerable verification resource, as the effort expended for one property can benefit the others without significantly slowing them down [CCL+18, CN11b]. Parallel verification resource can be optimized to leverage these facts using affinity-based *property partitioning* [DBI+19], where each parallel process, or *worker*, runs the same strategy on a different property group.

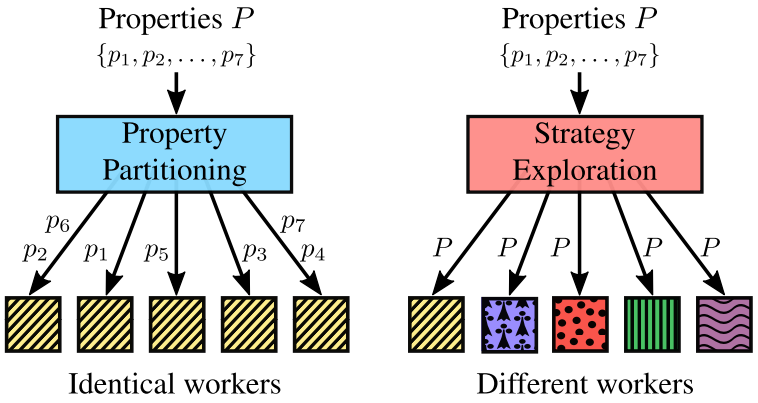


Figure 5.1: Parallel verification: property partitioning vs. strategy exploration.

An alternate way to accelerate verification is by using a parallel portfolio (*strategy exploration*), where the same property group is concurrently verified using a different strategy per worker, as depicted in Figure 5.1. However, portfolio approaches often degrade into highly-redundant work across processes, where similar algorithms address properties in nearly-identical order. Existing tools often independently use these modes in different contexts, particularly strategy exploration first running qualitatively-different strategies in available workers (e.g., BMC, IC3, interpolation) then padding differently-configured identical strategies in the remaining processes (e.g., IC3 with different heuristics). The latter yields increasingly-redundant CPU-time for diminishing gains in wall-time. These modes need not be mutually-exclusive: a strategy could partition within a worker, and partitioning could use different strategies for different groups. We explore the mutual optimization between property partitioning and strategy exploration. We specifically address the following parallel-verification performance challenges:

- *Property partitioning* $\rightarrow$

  P1 Some parallel workers are not utilized if the number of high-affinity property groups is less than available workers.

P2 Some parallel workers finish their tasks and idle (no more property partitions to dispatch) while others degrade wall-time by solving large or difficult groups, or may run on relatively slower machines.

- *Strategy exploration* →

P3 Nearly-identical algorithm strategies verify the same properties concurrently yielding redundant computation; two or more parallel workers would solve the same property, or property group at nearly the same time.

P4 A parallel worker gets stuck on the first difficult property inhibiting overall progress; easy properties may go unexplored.

P5 When using a round-robin resource-constrained approach to avoid **P4**, a parallel worker fails to solve a difficult property, or property group in the allocated time even after several repetitions with marginal, or no progress.

We improve multi-property parallel verification with respect to both wall- and CPU-time. We extend affinity-based partitioning to guarantee *complete* utilization of available processes, with provable *partition quality*. We propose methods to minimize redundant computation, and dynamically optimize work distribution. We deploy our techniques in a sequential redundancy removal framework, using *localization* to solve non-inductive properties. Our proposed six-process localization portfolio distributes properties to ensure optimum work distribution. Our techniques offer a median 2.4× speedup yielding 18.1% more property solves, as demonstrated by extensive experiments on large sequential equivalence checking, redundancy removal, and functional verification benchmarks.

5.1.1 Related Work

Despite the prevalence of parallel verification tools and multi-property testbenches, little research has addressed mutual optimization of parallel partitioning and strategy exploration. Furthermore, most approaches optimize wall-time alone without considering CPU-time, treating additional CPUs as free horsepower to fill with slightly-modified strategies without attempting to minimize redundant computation.

Methods to group properties based on COI similarity are either computationally-prohibitive [CCL+18, CN11b, DR18], or do not optimally utilize available parallel processes [DBI+19]. They may generate fewer groups than processes, or lose affinity guarantees when requiring *number of groups* as an algorithmic parameter.

Much prior work addresses ways to parallelize specific algorithms in a *single-property* context [Bra11, CK16, MGHS17]. Other work incrementally reuses information between properties to accelerate specific algorithms [KNPH06, KN12, DR17, MS07b]. These are complementary to our work, and can be used as strategies therein.

Much complementary research work has addressed sequential redundancy removal, using scalability-boosting strategies including induction [van98, BC00, MBPK05], simulation [DMJO18, MBMB09], and synergistic transformation and verification algorithms [BMP+06, MBMB09]. The benefit of parallelizing inductively-provable redundancies has been noted in [MCBJ08, PMRR19], though little work addresses parallelizing non-inductive redundancies. Localization is a powerful scalability boost to redundancy removal [MBPK05, MBMB09, BEM12] and property checking [MEB+13, MA03, AM04, CCK+02]. Prior work is focused mostly upon single-property single-process contexts [MEB+13, MA03, AM04, CCK+02], or solely upon parallel property partitioning [DBI+19]. This work is complementary to ours: we extend state-of-the-art solutions for both, to mutually-optimized parallel verification.

5.1.2 Contributions

We optimize parallel verification of multiple properties using complementary *property partitioning* and *strategy exploration*, in terms of both wall- and CPU-time.

1. We present a scalable property partitioning algorithm, extending [DBI+19] to guarantee *complete* utilization of available processes with provable partition quality.

2. We propose parallel scheduling improvements, such as resource-constrained irredundant group iteration, incremental repetition, and group decomposition to dynamically cope with more-difficult groups or slower workers.

3. We address irredundant strategy exploration of a localization portfolio in a sequential redundancy removal framework, which we have found to be the most-scalable strategy to prove non-inductive redundancies.

4. We propose improvements to *semantic group partitioning* within localization. To our knowledge, this is the first published approach to mutually-optimize property partitioning and strategy exploration within a multi-property localization abstraction portfolio.

5. Extensive evaluation on large benchmarks derived from hardware verification problems that span functional verification and equivalence checking.[1]

5.2 Preliminaries

Definition 5.2.1. The logic design under verification is represented as a *netlist* N, which is a tuple $(\langle V, E\rangle, F)$ where $\langle V, E\rangle$ is a directed graph such that

1. V is a set of vertices representing *gates*,

[1]Raw experimental results available at http://temporallogic.org/research/FMCAD20

2. $E \subseteq V \times V$ are edges representing interconnections between gates, and

3. $F : V \rightarrow types$ is a function that assigns vertices to gate *types*: constants, primary inputs, combination logic such as *AND* gates, and sequential logic such as *registers*.

A *state* is a valuation to the registers. Each register has two associated gates that represent its next-state function, and its initial-value function. Semantically, the value of the register at time "0" equals the value of the initial-value function gate at time "0", and the value of the register at time "i+1" equals that of the next-state function gate at time "i". Certain gates in the netlist are labeled as *properties* that are formed through standard synthesis of the relevant property specification language.

Definition 5.2.2. Given a netlist $N = (\langle V, E\rangle, F)$, a gate $v_i \in V$ is in the *fan-in* of gate $v_j \in V$ if and only if $(v_i, v_j) \in E$ or there exist gates $\{v_1, v_2, \ldots, v_k\} \in V$, for $k \geq 1$, such that $\{(v_i, v_1), (v_1, v_2), \ldots, (v_k, v_j)\} \in E$.

Definition 5.2.3. Given a netlist $N = (\langle V, E\rangle, F)$, a gate $v_i \in V$ is in the *fan-out* of gate $v_j \in V$ if and only if $(v_j, v_i) \in E$ or there exist gates $\{v_1, v_2, \ldots, v_k\} \in V$, for $k \geq 1$, such that $\{(v_j, v_1), (v_1, v_2), \ldots, (v_k, v_i)\} \in E$.

The *fan-in cone* of a property gate p refers to the set of all gates in the netlist which may be reached by traversing the netlist edges backward from the property gate, and is denoted *fanin(p)*. Similarly, the fan-out of gate u is the set of gates which may be reached by traversing edges forward from u. The fan-in cone of the property gate is called the *cone-of-influence* (COI) of the property. The registers and inputs in the COI of the property are called *support variables*. The number of support variables in the property's COI is the COI *size*. A *merge* of gate u onto gate v consists of moving the output edges of u onto v, then eliminating u from the netlist by treating u as a rename for v.

Definition 5.2.4. A *strongly connected component* in a netlist $N = (\langle V, E\rangle, F)$ is a set of gates $\mathcal{C} \subseteq V$ such that for every pair of gates $v_i, v_j \in \mathcal{C}$, $v_i \in fanin(v_j)$ and $v_j \in fanin(v_i)$.

Note that a primary input does not belong to any SCC, and in a *well-formed* SCC every directed cycle has at least one register gate because a netlist must be free of combinational cycles. The number of register gates in a SCC is the *weight* of the SCC.

5.2.1 Affinity Analysis

Property grouping algorithms represent support variable information as a *Boolean bitvector* per property [CCQ16]. Every support variable in the netlist is indexed to a unique position in the bitvector, set to "1" if and only if the support variable is in the COI of the property. The length of such a bitvector is equal to the total number of support variables in the netlist, and all bitvectors have the same length. The COI size of the property is the number of bits set to "1". These bitvectors may be compared to determine relative property *affinity*. Properties p_1, p_2 with bitvectors bv_1, bv_2 respectively have

$$0 \leq \text{affinity}(p_1, p_2) = 1 - \frac{\text{hamming}(bv_1, bv_2)}{\text{length}(bv_1)} \leq 1.0$$

where $\text{hamming}(bv_1, bv_2)$ is the Hamming distance between bv_1 and bv_2, and $\text{length}(bv_1)$ is the number of support variables in the netlist [DBI+19]. The *distance* between p_1, p_2 equals the Hamming distance between their bitvectors, i.e., $\text{dist}(p_1, p_2) = \text{hamming}(bv_1, bv_2)$. A *group* g is a set of properties, with a single property g^* therein representing its *center*. The *quality* $Q(g)$ of a group is the minimum affinity between any property in g vs. its center g^*:

$$Q(g) = \min(\{\text{affinity}(p, g^*) \mid \forall p \in g\})$$

It is desirable that efficient property partitioning algorithms guarantee group quality to be greater than a specifiable threshold.

function structural_grouping (P, N, l,t)

Input: P = set of properties, N = netlist, l = desired grouping level,
t = affinity threshold

Output: G = high-affinity property groups

1: **Groups** $G = P$ *# each property in singleton group*

2: **if** $l \geq 1$: grouping_level_1 (G, N) *# identical COI*

3: **if** $l \geq 2$: grouping_level_2 (G, N, t) *# large SCCs in COI*

4: **if** $l \geq 3$: grouping_level_3 (G, N, t) *# Hamming distance*

5: **return** G *# return high-affinity groups*

Figure 5.2: Algorithm to group properties based on structural affinity [DBI+19].

5.2.2 High-Affinity Property Grouping

Three-leveled grouping procedure of Chapter 4 [DBI+19] (reproduced in Figure 5.2) utilizes support bitvectors of properties to generate high-affinity groups. The algorithm takes the desired grouping level (l) and affinity threshold (t). It groups properties based upon: a) *Level-1*: identical bitvectors (identical support variables); b) *Level-2*: common large SCCs (containing t% netlist support variables) in the COI; and c) *Level-3*: small Hamming distance between support bitvectors, scalably identified by equivalence-classing *mapped* bitvectors using threshold-aware mapping functions. Higher grouping levels yield progressively fewer but larger high-affinity property groups.

Straightforward grouping approaches such as pairwise comparison are computationally prohibitive [CCQ16], requiring at least quadratic resource with respect to number of properties. Despite being conceptually a quadratic-resource algorithm, bitvector equivalence-classing [DBI+19] consumes near-linear runtime and memory in practice, enabling scalable

online partitioning with provable quality bounds [DBI+19]. Bitvectors are computed during a linear sweep of the netlist, and have size proportional to the number of SCCs plus non-SCC support variables. SCC computation has linear runtime [Tar72]. With efficient implementation, this entire process consumes a few seconds on netlists with millions of support variables and properties: e.g. computing bitvectors in topological netlist order, and garbage-collecting bitvectors as soon as all fan-out references have been processed [CCQ16].

A priori knowledge of solvers may dictate the ideal grouping level. For example, BDD-based reachability is highly sensitive to COI size, and thus may prefer level=1. BMC may prefer level=3 with lower affinity. Localization may prefer level=1, =2, or =3 depending on subsequent solvers. In many contexts, the caller can set level=3 and allow Figure 5.2 to determine group count and size, especially when using the techniques of Section 5.3.2 and Section 5.4.3 to decompose difficult groups. The structural grouping procedure of Figure 5.2 generates groups with provable affinity bounds.

Theorem 5.2.1 ([DBI+19]). *The level-1 grouping procedure (Figure 4.5) generates high-affinity property groups G such that $\forall g \in G : Q(g) = 1.0$.*

Theorem 5.2.2 ([DBI+19]). *Given affinity threshold t, the level-2 grouping procedure (Figure. 4.6) generates property groups G such that $\forall g \in G : Q(g) \geq t$.*

Theorem 5.2.3 ([DBI+19]). *Given affinity t and word size n, the level-3 grouping procedure (Figure 4.10) generates property groups G such that $\forall g \in G : Q(g) \geq 2 * t + \hat{t} - 2$, where $\hat{t} = 1 - \lfloor (1-t) * n \rceil \div n$. For $\hat{t} = t$, we have $Q(g) = 3 * t - 2$.*

Note that desired number of property groups is not an algorithmic parameter; affinity analysis determines the optimal number of groups respecting configurable quality bounds. For more details on leveled grouping, we refer the reader to Chapter 4 [DBI+19].

5.3 Grouping for Parallel Verification

Many organizations have large clusters of computers for load-balancing of tasks such as verification. The maximum number of available workers for a given task (n) is often known, e.g. the maximum number of organizational job submissions allowed per user, minus how many that user wishes to reserve for other tasks. Existing scalable grouping algorithms [DBI+19] may generate fewer high-affinity groups than n (**P1**). While partitioning a high-affinity group may yield redundant CPU-time (similar effort expended on nearly-identical COIs), it may benefit wall-time due to disparate difficulty of properties therein: e.g. one may be inductive, and another require deep sequential analysis. Traditional clustering algorithms can be configured to produce $\geq n$ groups, though are computationally prohibitive for online use and may not yield affinity guarantees if n does not align with the given netlist.

5.3.1 Property Grouping Algorithm

Figure 5.5 shows our extension to leveled grouping [DBI+19] (Figure 5.2), guaranteeing generation of at least $\mathsf{min}(n, |P|)$ provable-affinity groups. Each property is returned as a singleton if there are fewer than n properties. Otherwise, grouping is performed in three levels that iteratively generate fewer, larger groups. Later levels are skipped if the number of generated groups becomes less than n at any level. The algorithm then rebalances as needed by fine-grained affinity analysis: subdividing large or lower-affinity groups to generate at least $\mathsf{min}(n, |P|)$ property groups. The general grouping for parallel verification algorithmic flow is shown in Figure 5.3. As discussed in Section 5.3.2, this procedure is beneficial even after initial partitioning to subdivide a difficult group into provably high-affinity subgroups.

The rebalancing algorithm is shown in Figure 5.6. Figure 5.4 shows the high-level algorithmic flow for rebalancing high-affinity property groups. It subdivides groups based on the grouping level l_c that generated fewer groups than n. For level-1, quality is already 100%

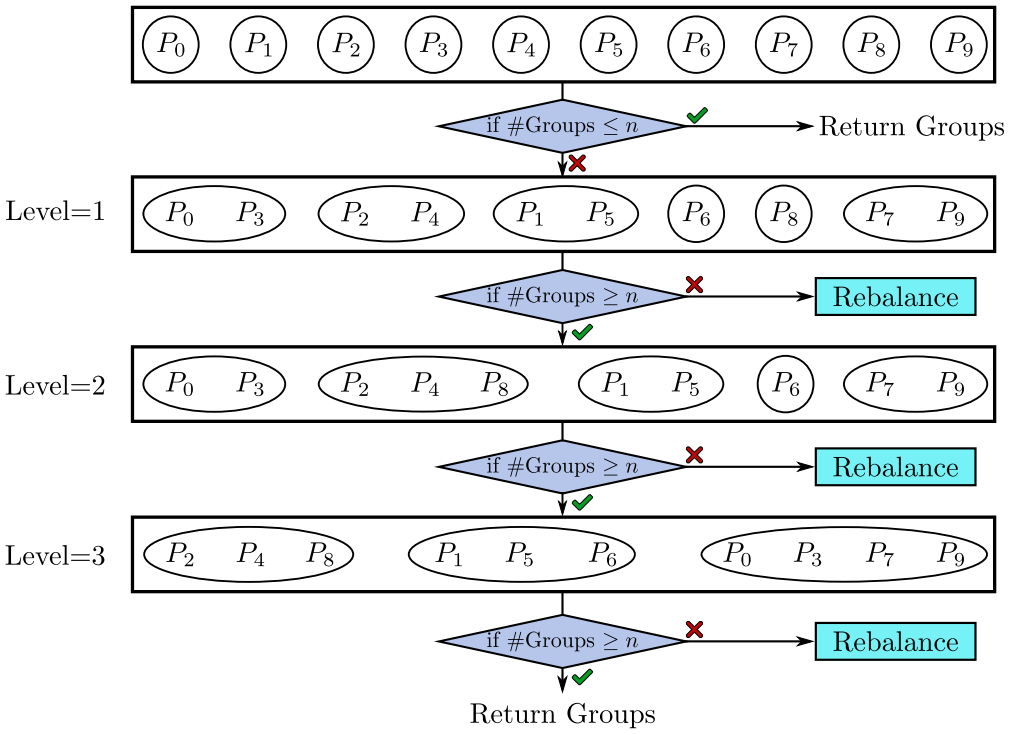


Figure 5.3: General algorithmic flow for property grouping

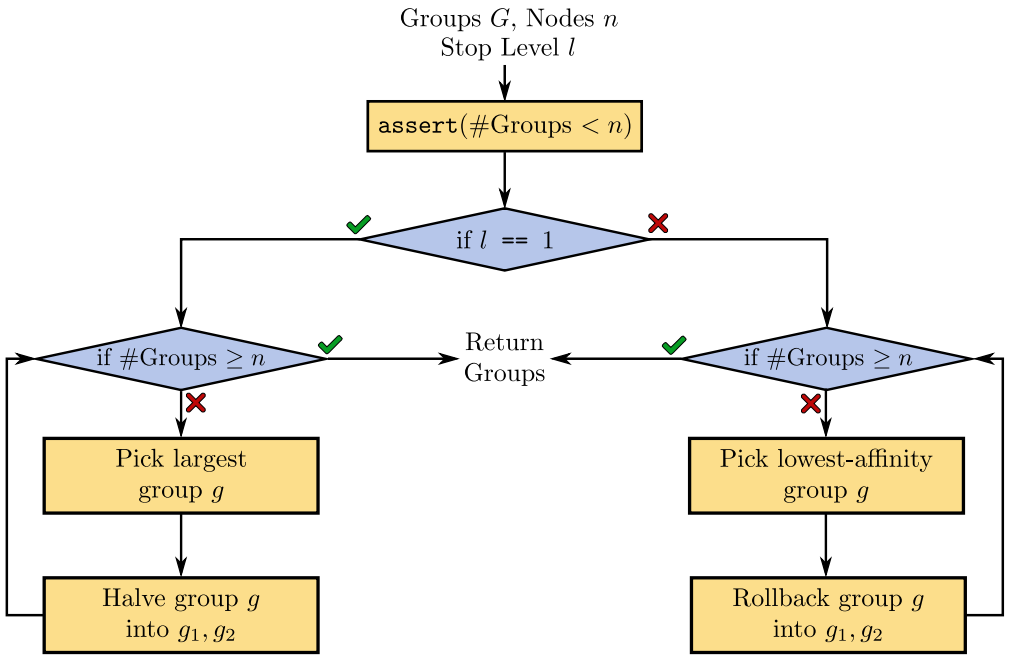


Figure 5.4: General algorithmic flow for rebalancing high-affinity groups

function structural_grouping_parallel (P, N, l, t, n)

Input: P = set of properties, N = netlist, l = desired grouping level,
t = affinity threshold, n = number of parallel workers

1: **Level** $l_c = 0$ *# current grouping level*
2: **Groups** G = singletons(P) *# initialize to singleton groups*
3: **if** $|G| \leq n$: **return** G *# fewer properties than workers*
4: **if** $l \geq 1$: grouping_level_1 (G, N), $l_c = 1$ *# identical COI*
5: **if** $l \geq 2$ **and** $|G| \geq n$: *# else fewer groups than workers*
6: grouping_level_2 (G, N, t), $l_c = 2$ *# large SCCs in COI*
7: **if** $l \geq 3$ **and** $|G| \geq n$: *# else fewer groups than workers*
8: grouping_level_3 (G, N, t), $l_c = 3$ *# Hamming distance*
9: **if** $|G| < n$: *# fewer groups than available workers*
10: rebalance (G, N, l_c, t, n) *# distribute groups, see Figure 5.6*
11: **assert** $(|G| \geq n)$ *# guaranteed to hold*
12: **return** G *# return high-affinity groups*

Figure 5.5: Property grouping guaranteed to generate at least min(n, $|P|$) high-affinity groups for n parallel workers.

```
function rebalance (G, N, l_c, t, n)
Input: G = property groups, N = netlist, l_c = highest grouping level,
       t = affinity threshold, n = number of parallel workers
1: if l_c == 1 :   # divide large level-1 groups in half
2:   halve_groups (G, n)  # see Figure 5.7
3: else # rollback minimal-quality level-2 & level-3 groups
4:   rollback_groups (G, N, l_c, t, n)  # see Figure 5.8
```

Figure 5.6: Algorithm to subdivide high-affinity groups for n workers.

so division is based on number of properties in the group (Figure 5.7). Groups with the most properties are halved until at least $\mathsf{min}(n, |P|)$ groups are generated. Finer-grained analysis may be integrated if desired, e.g. considering affinity of combinational gates in the combinational fan-in of these properties. Group rollback for higher levels is more intricate (Figure 5.8), with the goal of *improving* group quality. A group with minimal quality is conservatively subdivided until at least $\mathsf{min}(n, |P|)$ groups are generated. A minimal-quality group is split to yield smaller, higher-quality subgroups (Figure 5.9). This process has negligible runtime, reuses precomputed support bitvectors and requires only a few milliseconds on the largest netlists with thousands of properties.

The rebalancing procedure generates groups with quality bounds per Theorems 5.2.1, 5.2.2 and 5.2.3. Note that arbitrarily subdividing level-2,-3 groups without careful affinity consideration might violate affinity thresholds, because the quality of group g is measured with respect to its center property g^*. Assume that we generate subgroups g_0 and g_1 from g. If g^* is in g_0, we trivially have $Q(g_0^*) \geq Q(g^*)$ for any properties subgrouped with g^*. However, no such claim can be made about g_1; its properties might have been nearer to g^* than to each other. It is thus desirable to subdivide the most-distant property g_1^* from g^*

function halve_groups (G, n)

Input: G = property groups, n = number of parallel workers

1: **while** $|G| < n$:

2: **Group** g = pick largest non-singleton group from G

3: $G = (G \setminus g) \cup$ halve_group (g) *# see below*

function halve_group (g)

Input: g = property group

1: **return** {first half of g, second half of g} *# split in half*

Figure 5.7: Algorithm for subdividing large level-1 groups in half.

to improve vs. risk degrading the resulting quality of both subgroups. Moreover, simply rolling back a higher level group to lower-level subgroups risks generating more groups than necessary, e.g., one level-2 group rolled back to ten level-1 groups. The algorithm in Figure 5.5 generates *a minimal number* $|G|$ of high-affinity groups with provable affinity bounds, where $|G| \geq \mathsf{min}(n, |P|)$.

Theorem 5.3.1. *Given a group g, the* rollback_group *procedure subdivides g into two disjoint subgroups g_0 and g_1 such that $Q(g_0) \geq Q(g)$ and $Q(g_1) \geq Q(g)$.*

Proof. The algorithm returns two 100% affinity groups when properties in g generate at most two level-1 subgroups. Otherwise, the greatest-Hamming-distance property $g_1^* \in g$ from g's center property g^* is identified. Subgroup g_0 inherits g^* as its center, and g_1 inherits g_1^* as its center. Remaining properties in g are added to g_0 vs. g_1 to minimize distance from g_0^* vs. g_1^*, ensuring provable quality bounds. □

function rollback_groups (G, N, l_c, t, n)

Input: G = property groups, N = netlist, l_c = highest grouping level,
t = affinity threshold, n = number of parallel workers

1: **while** $|G| < n$:
2: **Group** g = pick minimal-quality non-singleton group from G
3: $G = (G \setminus g) \cup$ rollback_group (g, N, l_c, t) *# see below*

function rollback_group(g, N, l_c, t)

Input: g = property group to rollback, N = netlist, l_c = highest grouping level,
t = affinity threshold,

1: **Groups** G = singletons(g) *# split g to singletons*
2: grouping_level_1 (G, N) *# level-1*
3: **if** $|G|$ == 1 : $G =$ halve_group ($g \in G$) **return** G *# $|G|$ == 2*
4: **else if** $|G|$ == 2 : **return** G *# g had two 100% quality subgroups*
5: rollback_group_level (G, N, t, 2) *# level-2, see Figure 5.9*
6: **if** $|G|$ == 2 : **return** G
7: **if** l_c == 3 : rollback_group_level (G, N, t, 3) *# level-3, see Figure 5.9*
8: **return** G *# $|G|$ == 2*

Figure 5.8: Algorithm for subdividing minimal-quality groups.

function rollback_group_level (**Groups** G, **Netlist** N, **Affinity** t, **Level** l)

Input: G = singleton property groups, N = netlist, t = affinity threshold,
l = grouping level,

Groups $G_c = G$ *# local copy of G*

Group g_0, $g_1 = \emptyset$ *# temporary groups, initially empty*

if l == 2 : grouping_level_2 (G_c, N, t) *# level-2*

else : grouping_level_3 (G_c, N, t) *# level-3*

if $|G_c|$ == 1 : *# G_c is one group containing all properties in G*

  $g_0 = g \in G$ containing center property g_c^*

  # extract most-distant property into distinct subgroup

  $g_1 = g \in G$ s.t. dist(g_0^*, g^*) == $max(\{\text{dist}(g_0^*, g_i^*) \mid \forall g_i \in G\})$

  for each group $g \in G$: *# merge groups to minimize distance*

    if dist(g_0^*, g^*) $\leq$ dist(g_1^*, g^*) : add properties in g to g_0

    else : add properties in g to g_1

  $G = \{g_0, g_1\}$ *# note $Q(g_0), Q(g_1) \geq Q(g_c)$, see Theorem 5.3.1*

else : $G = G_c$ *# $|G| \geq 2$*

Figure 5.9: Algorithm to subdivide minimum-quality non-singleton group by rolling back to at least two lower level subgroups.

Corollary 5.3.1.1. *Given affinity t and grouping level l, the grouping for parallel verification procedure (Figure 5.5) generates groups G such that $\forall g \in G$: a) $Q(g) = 1.0$ if $l = 1$, b) $Q(g) \geq t$ if $l = 2$, and c) $Q(g) \geq 3 * t - 2$ if $l = 3$.*

Proof. The proof follows per Theorem 5.2.1, Theorem 5.2.2 and Theorem 5.2.3 when no rebalancing occurs. Otherwise, rebalancing divides group g in to smaller groups based on: (i) $l = 1$, level-1 subgroups are generated and $Q(g) = 1.0$ per Theorem 5.2.1; (ii) $l = 2$, levels-1 or 2 subgroups are generated and $Q(g) \geq t$ per Theorem 5.2.2 and Theorem 5.3.1; and (iii) $l = 3$, levels-1, 2 or 3 subgroups are generated and $Q(g) \geq 3 * t - 2$ per Theorem 5.2.3 and Theorem 5.3.1. □

Theorem 5.3.2. *Given groups G over a set of properties P, and workers n with $|G| < n$ and $|P| \geq n$, rebalancing generates property groups G' such that $|G'| = n$.*

Proof. Both the halve_group and rollback_group procedures subdivide a non-singleton group g into exactly two subgroups, and iterate until $|G'| \geq n$. Therefore, the number of groups increases by exactly one in every iteration, unless all groups become singleton which cannot happen until $|G'| = |P| \geq n$. □

Corollary 5.3.2.1. *Given a set of properties P and n workers, the grouping for parallel verification procedure (Figure 5.5) generates groups G from P such that $|G| \geq \mathsf{min}(n, |P|)$.*

Proof. The proof trivially holds when $\geq n$ groups or $|P| \leq n$ singletons are generated without rebalancing. Otherwise, the proof holds per Theorem 5.3.2 when rebalancing occurs. □

5.3.2 Group Distribution Heuristics

We propose three heuristics to optimally utilize parallel workers, used on-the-fly by a *manager* routine that dispatches property groups and dynamically adjusts dispatch ordering based upon feedback from parallel workers. When partitioning is supported by an engine

within a strategy (e.g. a localization engine [DBI+19]), there might be multiple managers partitioning an identical or overlapping set of properties. It is sometimes beneficial to use a hierarchy of managers: the *root* might use lower-affinity partitioning onto parallel strategies, with higher-affinity partitioning within a strategy.

5.3.2.1 Iteration order (I)

Figure 5.5 orders groups deterministically, and thus distributed managers within a strategy will likely verify common properties in the same order. This results in redundant CPU-time, where two or more strategies may solve the same property at nearly the same time (**P3**). The root manager could instead dispatch disjoint properties to different workers, though there are motivations for building intelligence into distributed managers working on the entire property set, such as enabling incrementality and data sharing across properties [KNPH06, KN12, DR17, MS07b]. To minimize redundant work, the manager may be augmented with options to iterate common groups in different orders: 1) smallest to largest COI (*forward*); 2) largest to smallest COI (*backward*); and 3) *random* to heuristically minimize concurrent solving of the same group while more groups than workers remain unsolved. If all properties are of comparable difficulty, running two identical strategies with opposite group ordering effectively halves wall-time with almost no redundant CPU-time. This approach can yield superlinear irredundant speedup when different strategies are tailored for easier vs more-difficult properties: a lighter strategy can iterate *forward* heuristically addressing easier properties first (the heavier strategy worker would be slower for these), while the heavier strategy can iterate *backward* addressing more-difficult properties first (the lighter strategy worker might be unable to solve these).

function get_next_group (**Groups** G, **Netlist** N, **Level** l_c, **Affinity** t)

Input: G = property groups N = netlist, l_c = highest grouping level, t = affinity threshold

Output: g = unsolved property group

1: **Group** g = pick unsolved or inactive group from G

2: **if** g == null : **return** null *# all group are solved or active*

3: **if** unsolved(g) **and** inactive(g) : **return** g *# dispatch group*

4: **if** unsolved(g) : *# decompose (new groups are unsolved and inactive)*

5: **if** l_c == 1 : $G = (G \setminus g) \cup$ halve_group(g) *# see Figure 5.7*

6: **else** $G = (G \setminus g) \cup$ rollback_group(g, N, l_c, t) *# see Figure 5.8*

7: **else** remove g from G *# group is already solved*

8: **goto** 1 *# pick next group to dispatch*

Figure 5.10: *Manager* routine to dispatch unsolved groups using decomposition.

5.3.2.2 Controlled repetition (R)

Each worker solves groups one-at-a-time. Encountering a difficult group inhibits overall progress (**P4**). Easier groups might follow, which when solved might speed-up incremental verification of the previous difficult group. Furthermore, solving easy properties sooner benefits other workers, allowing them to focus on fewer difficult groups. It is thus beneficial to impose time-limits per group within certain *fast* strategies. The manager must be capable of pruning already-solved properties (possibly solved by different workers), and repeating groups up to a configurable maximum allowed repetitions (to reduce redundant CPU-time). It may be beneficial to increase resource limits between repetitions, possibly after n repetitions with no progress. Engine incrementality is fairly important when imposing time-limits and repetition, to minimize redundant CPU-time.

5.3.2.3 Decomposition (D)

Some groups are more difficult than others, either because they are large (e.g., many properties), or because individual properties therein are more difficult (e.g., having a very-deep counterexample). Some workers might be slower than others, possibly due to varying machine load. A common wall-time degradation occurs when fewer difficult groups than workers remain, and previously-active workers become idle (**P2**). This heuristic decomposes unsolved groups and dispatches them to idle workers, to accelerate convergence despite imposing some redundant CPU-time. Rather than redundantly dispatching an entire unsolved group, this heuristic utilizes the algorithms of Figure 5.7 and Figure 5.8 to subdivide unsolved groups to smaller and higher-affinity groups, eventually becoming singletons. Smaller groups are easier for idle workers to redundantly solve (**P5**), benefiting but not preempting active workers (which might be on the verge of solves). The corresponding manager with decomposition is shown in Figure 5.10. A group is *inactive* when no worker is currently verifying it. Solved properties and groups are discarded; groups with *unsolved* properties are subdivided and redundantly dispatched. Singleton groups are not redundantly dispatched, being *inactive* after the first dispatch.

5.4 Localization for Redundancy Removal

Industrial hardware designs are often rife with redundancy, e.g. to boost the performance of semiconductor devices, and to implement features such as error resilience, security, initialization logic and post-silicon observability. Verification testbenches yield additional netlist redundancies, due to *input constraints* restricting the set of stimulus applied to the design, and due to redundancies arising between the design and synthesized properties. Equivalence checking can be viewed as verifying a *composite netlist* comprising two designs as per Figure 5.11. *Sequential redundancy removal* [van98, BC00, MBPK05, BMP+06, MBMB09,

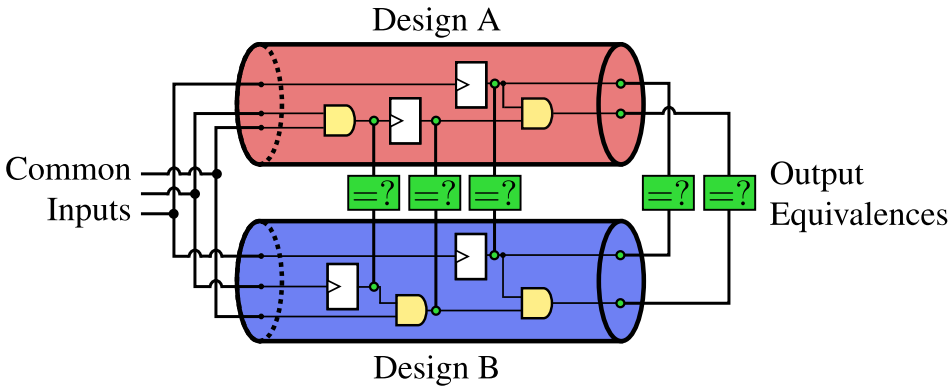


Figure 5.11: Sequential equivalence checking uses redundancy removal to eliminate gate-equivalences between two logic designs. Each speculated gate-equality requires verifying a property called a *miter* (depicted as green box *=?*).

MCBJ08, CBMK11] (Figure 5.12) is the process of proving that equivalence-classes of gates evaluate to equal or opposite values in all reachable states; each speculated redundancy entails solving a property called a *miter*. When a miter is proven, the corresponding redundant gates can be merged. This COI reduction is highly beneficial to verification scalability, and is the core procedure of sequential equivalence checking (SEC).

Various heuristics control the scope of equivalence-class candidates affecting runtime vs. reduction (Figure 5.12 Step 1): e.g. whether to consider only registers vs. all gate types; whether to prune classes to reflect *corresponded signal names* or require per-class candidates spanning both designs in an equivalence-checking context (Figure 5.11) [BEM12, MBPK05]. A *speculatively-reduced netlist* (Steps 2-3) accelerates verification of the miters. Techniques such as BMC and guided simulation are typically used to falsify miters; then induction proves the easier miters; and finally multi-engine strategies prove the difficult miters or find difficult counterexamples (Steps 4,5). Failed proofs (falsified miters or inconclusive results) cause a *refinement* of the equivalence classes to separate unproven miters' gates, then another expensive proof iteration is performed. Our goal is to minimize inconclusive proofs to achieve maximum netlist reduction with minimal wall- *and CPU*-time, using a parallel localization

function redundancy_removal (N)

Input: N = netlist

1: Guess the *redundancy candidates* - sets of equivalence classes of gates in N, where gate u in class $Q(u)$ is suspected equivalent to every other gate v in the same equivalence class.

2: Select a representative gate $R(Q(u))$ from each class $Q(u)$.

3: Construct the *speculatively-reduced netlist* by replacing source gate u of every edge $(u, v) \in E$ by $R(Q(u))$. Additionally, for each gate v, add a *miter* property asserted when $v \not\equiv R(q(v))$.

4: Attempt to prove that each miter is unassertable.

5: If a miter cannot be proven unassertable, refine the equivalence classes to separate the corresponding gates, and goto Step 2.

6: For all unassertable miters, merge the corresponding gates onto the representative to eliminate redundancy.

Figure 5.12: Generic sequential redundancy removal framework [MBMB09].

portfolio. Note that even if a testbench has only a single property, redundancy removal will often create thousands of miters. The large number of miters often tremendously benefit from parallel processing, as noted for combinational redundancy removal [PMRR19] and induction [MCBJ08]. These miters are distributed throughout the netlist, making affinity partitioning particularly beneficial. Since practical netlists comprise a diversity of logic, different miters benefit from different strategies.

The proof or counterexample of a property often only depends on a small subset of logic in its COI. *Localization* [MEB+13, MA03, AM04, CCK+02] is a powerful abstraction method to reduce COI size by replacing irrelevant gates by *cutpoints* or unconstrained primary inputs. Since cutpoints can simulate the behavior of the original gates and more, the abstracted netlist over-approximates the behavior of the original netlist: abstract proofs

imply original proofs, but abstract counterexamples might be spurious. Abstraction *refinement* eliminates cutpoints deemed responsible for spurious counterexamples, re-introducing previously-eliminated logic. It is desirable that the abstract netlist be as small as possible to enable scalable verification, while being immune to spurious counterexamples.

Localization is often essential to solve non-inductive miters, leveraging speculative reduction to abstract nearly all logic except for differently-implemented yet functionally-equivalent logic *between* speculated equivalences [MBPK05, MBMB09]. Without localization, the COI of a miter may be very large despite speculative reduction. This large COI size may choke even fairly-scalable provers such as IC3. While the benefits of localization for sequential redundancy removal are well-known [BMP+06], prior work considered only single-process miter verification, aside from use of a standard parallel model-checking portfolio to solve miters [BEM12]. Ours is the first to optimize a parallel localization portfolio in this (or any multi-property) context, using property partitioning and irredundant scheduling procedures (Figs. 5.5 and 5.10), along with the following complementary strategies tailored for easier vs. difficult properties. Note that substrategies in either may be employed by the other.

5.4.1 Fast-and-Lossy Localization

Fast-and-Lossy localization (Figure 5.13) attempts to quickly discharge easier property groups, using timeouts to skip difficult groups. If the group is not solved within the allotted time, verification data (e.g., the current abstract netlist and achieved BMC depth) is saved for incremental reuse to accelerate later repetition. Skipped groups can be repeated as-is, or rebalanced (Figure 5.10) after several repetitions of no progress. Note that repeating a group as-is may likely proceed further upon repetition, by incrementally skipping earlier processing and since a different worker might have solved some properties therein. *Fast-and-Lossy* localization uses counterexample-based refinement sometimes with quick proof-based abstraction (PBA), possibly yielding larger abstract netlists that are more-difficult to prove

but with less time expended in BMC itself [AM04] for faster performance on easier groups. When ready to prove (i.e., no refinements occur for n consecutive BMC steps), abstracted groups are passed to a sequence of lighter reduction engines then IC3 [Bra11, EMB11]) under a modest time-limit (e.g. $\leq 300s$) which can be increased across repetitions (**R**).

5.4.2 Aggressive Localization

Aggressive localization (Figure 5.14) is aimed at solving difficult properties, where *Fast-and-Lossy* may fail due to larger-than-necessary abstractions, insufficient reductions prior to IC3, or small group time-limits. *Aggressive* never repeats groups, so either imposes no time limit whatsoever, or a large time-limit as shown applied to semantically-partitioned (Sec. 5.4.3) sub-groups but iterated and increased until the group is solved. *Aggressive* typically uses a hybrid of counterexample-based refinement and PBA run after every unsatisfiable BMC result, to yield smaller abstractions than the former alone to accelerate subsequent proofs at the expense of more runtime spent in BMC itself [AM04]. When ready to prove (i.e., no refinements occur for n consecutive BMC steps), abstracted groups are passed to a sequence of heavy reduction engines (including nested induction-only sequential redundancy removal across *all gates*, which might be too expensive to converge on large netlists before localization) followed by IC3 [Bra11, EMB11]).

5.4.3 Semantic Partitioning

Semantic partitioning [DBI+19] refers to re-partitioning a group whose *unabstracted COI* was high-affinity, yielding sub-groups of high affinity with respect to *abstract COI* as correlates to subsequent verification complexity. Abstract COI information is mined onto support bitvectors on a per-property basis as cutpoints are refined (Figure 5.14 Step 4), considering minimized counterexamples for individual properties despite incrementally using the same

```
function fast_lossy_localization (g, n, T)
Input: g = property group, n = inactivity-limit for BMC, T = timeout
1: Netlist L = load_incremental_abstraction(g) # initially empty
2: unsigned k = load_incremental_bmc_depth(g) # initially k = 0
3: while elapsed_time() ≤ T and unsolved(g) :
4:   localize_bmc (g, L, k, unchanged)  # see below
       # check if netlist unchanged for last n bmc steps
5:   if unchanged < n : k = k + 1, goto 4  # increment depth
6:   run_proof_strategy(L, g, T - elapsed_time())
7: save_incremental_data (G, k, L)  # timeout: save incremental data
function localize_bmc (g, L, k, unchanged)
Input: g = property group, N = netlist, k = current BMC depth
1: bool stop = 0  # some properties fail at depth k
2: while not stop :  # loop until all properties pass at depth k
3:   Gates c = {}, stop = 1  # cutpoints to refine, initially empty
4:   for each Property p ∈ g :
5:     Result r = run_bmc(L, p, k)  # run bmc with depth k
6:     if r == unsat : continue  # property passes
7:     if cex not spurious : report_solved(p, cex), continue
8:     stop = 0  # property fails
9:     Gates d = cutpoints_to_refine(), c = c ∪ d
10:  if not stop : refine_abstraction(L, c), unchanged = 0
11:  else unchanged + = 1  # no change in abstraction
```

Figure 5.13: *Fast-and-Lossy* localization strategy with incremental repetition of high-affinity property groups.

function aggressive_localization (**Group** g, **unsigned** n, **bool** pba, **bool** semantic, **Affinity** t, **Timeout** T, **Multiplier** m)

Input: g = property group, n = inactivity-limit for BMC, pba = enable/disable PBA, semantic = enable/disable partitioning, t = affinity threshold, T = timeout, m = timeout multiplier

1: **Netlist** L = initial_abstraction(g) *# initially empty*

2: **unsigned** $k = 0$ *# bmc depth*

3: localize_bmc (g, L, k, unchanged) *# see Figure 5.13*

4: **if** semantic : collect_support_info (...) *# see Section 5.4.3*

5: **if** pba : minimize L using proof-based abstraction

 # check if netlist unchanged for last n bmc steps

6: **if** unchanged $< n$: $k = k + 1$, **goto** 3 *# increment depth*

7: **Groups** $\hat{G}$ = semantic ? structural_grouping (g, L, 3, t) : G

*# Sort via **(I)** mode (Section 5.3.2): forward, backward, or random*

8: Sort $\hat{G}$ by abstract COI size

9: **for each** unsolved group $\hat{g} \in \hat{G}$:

10: **while** elapsed_time() $\leq$ T **and** unsolved($\hat{g}$) :

11: run_proof_strategy(L, $\hat{g}$, T - elapsed_time())

12: **if** unsolved groups remain : $T = T \times m$, **goto** 9

Figure 5.14: *Aggressive* localization strategy with semantic partitioning, counterexample- and proof-based abstraction for property groups.

BMC instance for the entire group. The group is partitioned into smaller, high-localized-affinity subgroups (Step 7) before attempting to prove.

5.4.3.1 Improvements to semantic partitioning vs. [DBI+19]

Per-property abstract-COI bloat may arise during counterexample analysis, because the group must be mutually refined to be free of spurious counterexamples. Eager partitioning (as soon as any diverged abstract COI occurs) *could* circumvent this ambiguous bloat, though often severely hurts performance since intermediate abstract-COI differences often reconverge. In practice, lazy partitioning deferred until modest BMC time limits are exceeded is far superior (particularly since BMC often benefits from level=3 lower affinity), retaining high-affinity atomic verification benefits. Abstract-COI ambiguities can be largely corrected during proof analysis, by analyzing a distinct proof per property. Incremental data should be saved when semantically re-partitioning, to minimize restart penalty.

Difficult sub-groups are susceptible to delaying easier later sub-groups. Subgroups should be ordered as per **(I)** mode (Section 5.3.2): *forward*, *backward*, and *random*, configured differently in parallel strategies for better portfolio performance with less redundant CPU-time. Subgroups are verified in the chosen order using controlled repetition (**R**) and large *Aggressive* time-limits (Steps 9–11). We recommend T $\geq$ 1h multiplying 2$\times$ at each iteration (Step 12) and overriding to *unlimited* when a single sub-group remains.

5.5 Experimental Analysis

In this section, we report on the extensive experimental analysis of our techniques within the post-induction proof strategy of a sequential redundancy removal framework (Figure 5.12). We briefly detail our benchmarks, summarize the setup used for the experiments, and end with experimental results and a discussion of results.

5.5.1 Benchmarks

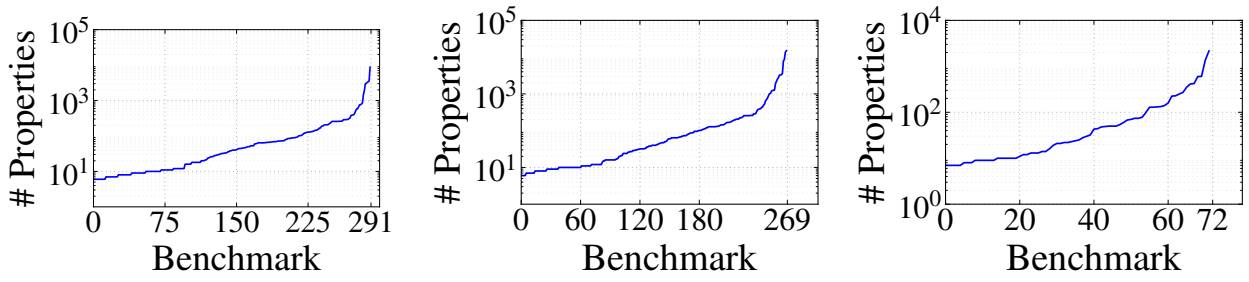
To eliminate *noise* such as different counterexamples yielding different equivalence-classes (Step 5, Figure 5.12), we snapshot the speculatively-reduced netlist after ten minutes of induction, before the final iteration of a six-hour eight-process semi-formal bug-hunting [NGB+16] and localization portfolio to eliminate most incorrect and easier [CBMK11] miters. The following experiments are run on these snapshotted netlists (pruning those with fewer miters than processes), yielding three benchmark sets.

5.5.1.1 Equivalence Checking Benchmarks

We evaluate our techniques on two sequential equivalence checking (SEC) benchmark sets containing properties ranging from a few hundreds to thousands. Benchmark set **B1** (Figure 5.15a) are the most-difficult 291 of 1822 proprietary SEC benchmarks, where initial equivalence classes comprise original properties and *name corresponded* register pairs. Benchmark set **B2** (Figure 5.15b) has 269 netlists derived from the former, including a large equivalence class for registers without name correlation.

5.5.1.2 HWMCC Benchmarks

We evaluate our techniques on selected benchmarks from the Hardware Model Checking Competition (HWMCC). Set **B3** has 72 netlists from the SINGLE property HWMCC 2017 benchmarks, comprising a large initial equivalence class of all registers. Though these benchmarks only have one functional property, the equivalence classing for redundancy removal generates benchmarks with several properties, often in the thousands.



(a) Set B1 **(b)** Set B2 **(c)** Set B3

Figure 5.15: Number of properties per benchmark set used for evaluation of our techniques

5.5.2 Localization Portfolio

We select our localization portfolio (Table 5.1) from extensive evaluation of 36 single-process localization configurations and 30 subsequent proof strategies, exploring options such as enabling vs. disabling PBA [AM04]; different levels of property grouping vs. no grouping [DBI+19]; enabling vs. disabling semantic partitioning (Section 5.4.3); and different policies for group iteration **(I)**, repetition **(R)**, and decomposition **(D)** (Section 5.3.2). The best-performing collection is chosen, maximizing *complementary* unique solves. *Aggressive* localization (Section 5.4.2) primarily uses both counterexample- and proof-based abstraction, yielding smallest abstract netlists solved with a single-process heavy strategy of combinational rewriting; input elimination [BM05, EM13, GBI+19] which is especially powerful after localization due to inserted cutpoints; min-area retiming [KB01]; a nested induction-only gate-based sequential redundancy removal; then IC3. *Fast-and-Lossy* localization (Section 5.4.1) uses counterexample-based refinement mainly with no or lighter PBA for faster BMC, yielding larger abstract netlists solved using light combinational rewriting, input elimination, then IC3. The *Aggressive* strategy is fastest for difficult properties, while the *Fast-and-Lossy* strategy is fastest for easier properties.

We compare four 6-process localization portfolios derived from Table 5.1. The localization configuration and subsequent solving strategy of each process is identical across port-

Table 5.1: Six-process complementary localization portfolio.

#	Localization Strategy	Grouping Level	Semantic	Iteration (I)	Repetition (R)	Decomposition (D)
S1	*Fast-and-Lossy*	Level-1	✗	Forward	✓	✗
S2	*Fast-and-Lossy*	Level-1	✗	Reverse	✓	✓
S3	*Fast-and-Lossy*	Level-3	✓	Forward	✓	✓
S4	*Aggressive*	Level-1	✗	Forward	✗	-
S5	*Aggressive*	Level-1	✗	Reverse	✗	-
S6	*Aggressive*	Level-3	✓	Forward	✗	-

folios, except for adherence to the illustrated scheduling differences as discussed below. For greater portfolio value, each process includes localization configuration differences beyond the illustrated scheduling distinction in Table 5.1. **S1** only performs counterexample-based refinement; **S2** and **S3** also perform PBA. **S2** vs. **S3** perform hybrid counterexample-based refinement with light PBA (modest time limit) after every unsatisfiable BMC step vs. only before the subsequent solving strategy, respectively. Abstract-netlist gates remaining after PBA are considered *committed* and cannot be eliminated in later PBA steps [MEB+13] in **S2**, but not **S3**. **S3** utilizes a minimal unsatisfiable core to further reduce the abstract netlist. **S4-S6** are identical to **S1-S3**, respectively, without imposed time-limits and modulo the above-mentioned post-localization solving strategy differences. To highlight our individual contributions, we compare four variants of this portfolio:

1. `base`: No property grouping or incremental repetition of properties; all processes iterate properties in forward order. This represents a standard state-of-the-art localization portfolio approach *without property grouping*, e.g., before [DBI+19].

2. `base+g` extends `base` with affinity property grouping, including semantic partitioning in one *Fast-and-Lossy* and one *Aggressive* strategy. This represents a state-of-the-art localization portfolio *with property grouping*, e.g., as per [DBI+19] though with our semantic refinement improvements of Section 5.4.3.

3. `best-d` extends `base+g` with incremental repetition **(R)** and irredundant iteration order **(I)**, to reduce CPU-time.

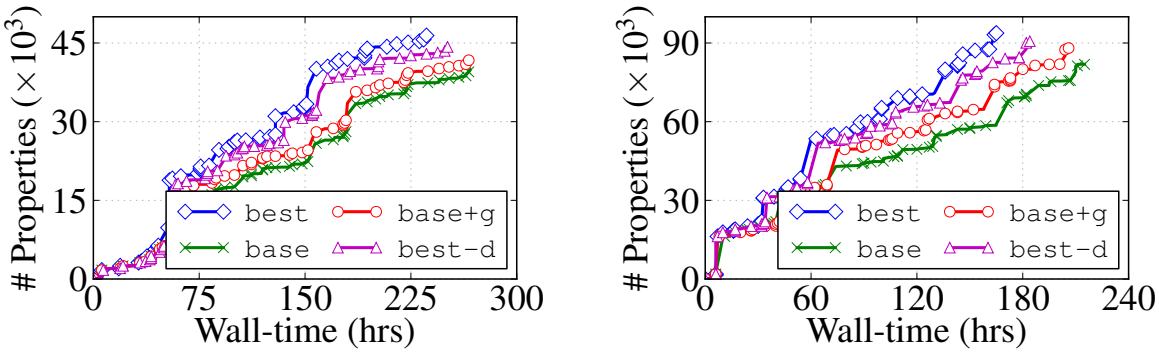
4. `best` extends `best-d` with decomposition **(D)**.

Processes S1-S6 in the portfolio are generic online localization strategies. Multi-property localization *without affinity-partitioning* generally yields poor/noncompetitive performance [3], eroding most of its scalability benefit, especially for difficult miters. (Recall that these benchmarks pre-filter easier miters, using induction and semi-formal bug-hunting.) Therefore, both `base` and `base+g` are highly-competitive 6-process localization portfolios, for online "first-run-of-a-testbench." Industrial verification tools may use more processes for large testbenches, and may post-process data from prior/ongoing runs to accelerate future results. This level of sophisticated benchmark-specific orchestration is valuable, though does not readily benefit "first-run-of-a-testbench" and introduces noise in experiments hence are not used herein. We optimize runtime of a generic 6-process localization portfolio for "first-run-of-a-testbench" without per-benchmark customization.

5.5.3 Experiment Setup

Our experiments run on a computing grid with identical x86 Linux nodes. Each benchmark run uses a 6-process portfolio (Table 5.1); each process **S1**-**S6** runs on a single identical CPU core on the same host-machine. Each process eagerly cancels solved properties across all processes in that portfolio, to reduce redundant computation. Our techniques are implemented within *RuleBase: Sixthsense Edition* [MBP+04].

While most prior research and competitions focus solely upon optimizing wall-time, our techniques additionally benefit CPU-time. Traditionally, Fast-and-Lossy (unlike Aggressive) processes terminate early, leaving unsolved difficult properties. In these experiments, `base` and `base+g` augment Fast-and-Lossy processes to naively repeat identically-configured



(a) Set B1 **(b)** Set B2

Figure 5.16: Number of properties solved vs. wall-time for **B1** and **B2**; 6-hour time limit.

S1-**S3** with identical resource limits per group (whereas `best-d` and `best` add incremental-repetition **(R)** with resource-doubling across repetitions), until all properties are solved or global timeout. This naive repetition is wasteful in practice, yielding highly-redundant CPU-time for marginal benefit. However, disabling naive repetition in these experiments yielded 3.2% fewer solves in `base` and `base+g` vs. `best-d` and `best`, which arguably unfairly penalized them as state-of-the-art solutions *before our contributions*. Therefore, **S1**-**S6** in each portfolio continue working until all processes terminate, hence CPU-time is approximately 6× wall-time in these experiments.

5.5.4 Experimental Results

5.5.4.1 Proprietary Benchmarks

Figure 5.16 shows the number of properties solved vs. wall-time for benchmark sets **B1** and **B2**. The `best` portfolio is the clear winner, solving 18.1% (15.3%) more properties in 17.2% (22.9%) less time for set **B1** (set **B2**, respectively) compared to `base`. Affinity-grouping significantly improves performance of `base+g` over `base`. Level-3 grouping with our semantic partitioning improvements (Section 5.4.3) benefits *Aggressive* localization strat-

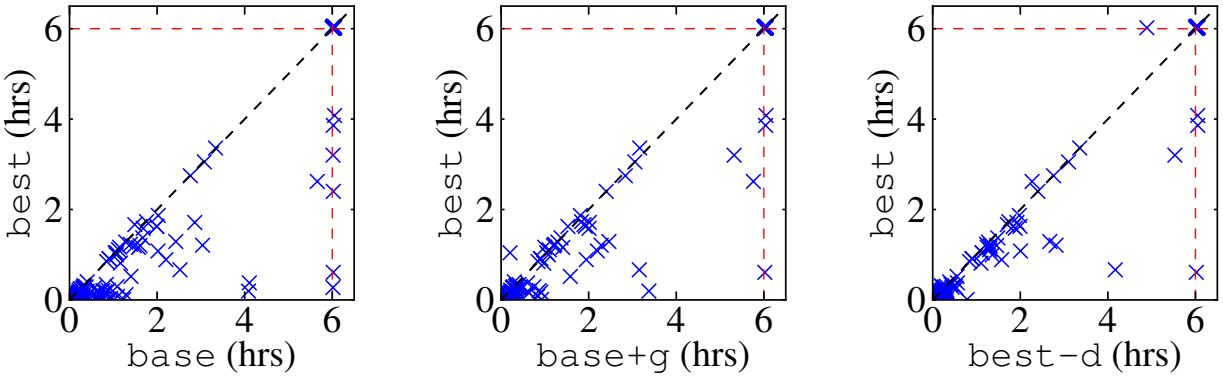


Figure 5.17: best vs. baselines for **B1** (points below diagonal are in favor).

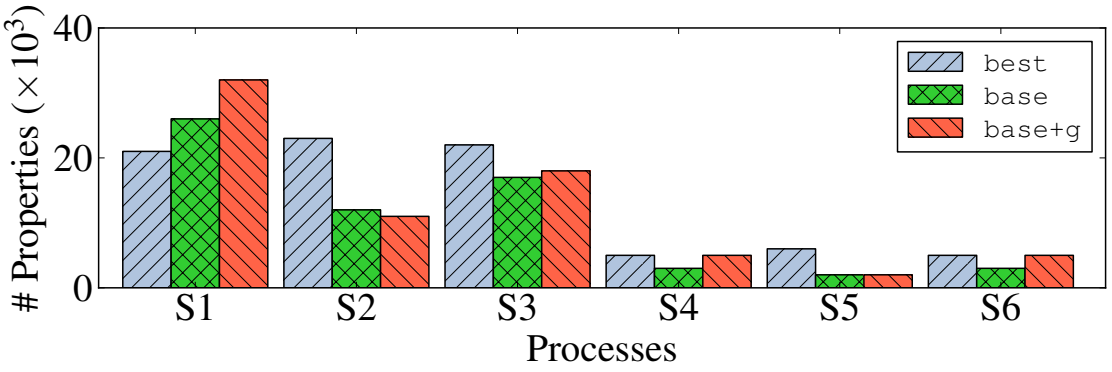


Figure 5.18: Number of properties solved on **B2** per process of Table 5.1.

egy, atomically solving properties in fewer, larger high-abstract-affinity groups compared to level-1,-2. Incremental repetition and irredundant iteration allows `best-d` to solve 8.1% more properties than `base+g`, less-severely hindered by difficult groups. `best` yields additional solves through decomposition of difficult groups after five incremental repetitions of no progress, solving all properties in 4 vs. 6 benchmarks in **B1** vs. **B2** that time out with other portfolios. Figure 5.17 details per-**B1**-benchmark runtimes of `best`, yielding a median speedup of 2.4×, 2.0× and 1.5× vs. `base`, `base+g`, and `best-d`, respectively.

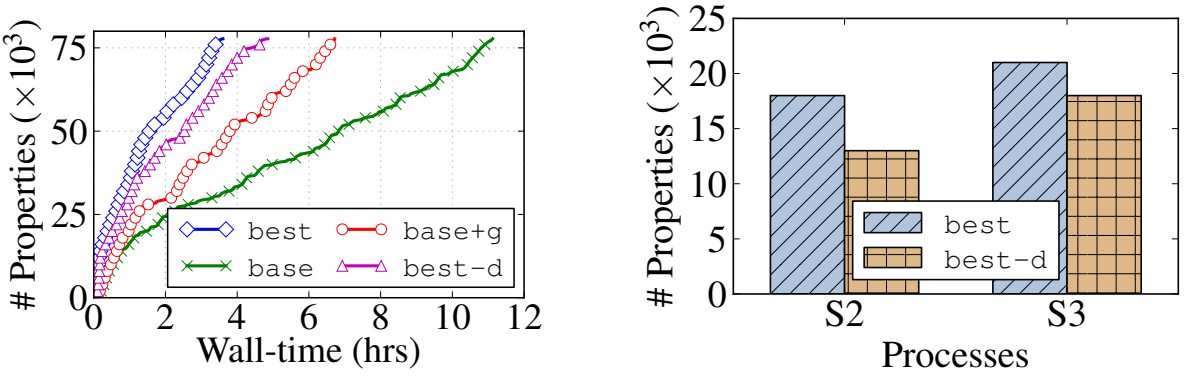
Figure 5.18 shows the distribution of properties solved per process (Table 5.1) within these portfolios. The percentage solved by each *Fast-and-Lossy* (and *Aggressive*) process is nearly

Table 5.2: Utility of aggressive strategy processes in a portfolio.

Portfolio	Set B1		Set B2	
	#Solved	Time (h)	#Solved	Time (h)
3× *Fast-and-Lossy*, 3× *Aggressive*	**46,844**	**236**	**93,806**	**165**
6× *Fast-and-Lossy* (modified `best`)	41,702	275	91,639	184

uniform in `best`, showing near-optimal irredundant work distribution. In contrast, without **(I)** and **(R)**, portfolios `base` and `base+g` have highly-uneven distributions due largely to parallel processes addressing the same groups concurrently. While the number of solved (easier) miters is considerably larger with *Fast-and-Lossy*, we emphasize how critical the *Aggressive* solution of difficult miters is to the overall redundancy removal process. If any are left unsolved, Figure 5.12 Step 5 will forgo attempting to merge the corresponding gates, thereby weakening netlist reductions, risking unsolved SEC, and hurting runtime by requiring yet another expensive proof iteration with refined equivalence classes [MBPK05] – where fan-out miters often become more-difficult than those unsolved in prior iterations. Table 5.2 shows the number of properties solved by `best`, and a modified `best` portfolio with all *Fast-and-Lossy* strategy processes where processes **S4**-**S6** are identical to processes **S1**-**S3** respectively, but without imposed time-limits and iterating groups in opposite order. Without *Aggressive* processes in the portfolio, the modified `best` portfolio solves 10.9% (2.31%) fewer properties in 16.5% (11.51%) more time for set **B1** (set **B2**). Therefore, the *Aggressive* strategy solution of difficult miters is vital for overall performance of the redundancy removal process.

To further highlight the value of decomposition **(D)**, Figure 5.19b illustrates an additional **big** benchmark containing 77728 properties partitioned into 9958 level-1 and level-2, and 2991 level-3 high-affinity groups. Figure 5.19a shows the number of properties solved by each portfolio vs. time. `best` is 3.0× faster than `base`. Figure 5.19b shows the number



(a) Properties vs. time

(b) Work distribution

Figure 5.19: Number of properties solved vs. wall-time for **big**: (a) by all portfolios; (b) per process of Table 5.1 within `best` and `best-d`.

of properties solved by two *Fast-and-Lossy* processes of `best` and `best-d`; decomposition enables **S2** and **S3** in `best` to collectively solve 25.2% more properties than `best-d`.

5.5.4.2 HWMCC Benchmarks

Figure 5.20 shows the number of properties solved by each portfolio for benchmark set **B3**. The `best` portfolio is again the winner, solving 3054 more properties in less time than the `base` portfolio. Incremental repetition and irredundant iteration is particularly beneficial in this set: several benchmarks have counterexamples that are discovered in earlier group repetitions, enabling *Aggressive* and later *Fast-and-Lossy* repetitions to direct resource upon more-difficult but provable miters.

5.6 Summary and Discussion

We focus upon boosting the scalability of multi-property parallel verification, with application to sequential redundancy removal using a localization portfolio. Our contributions optimize both wall-time and CPU-time, orchestrating via complementary strategy explo-

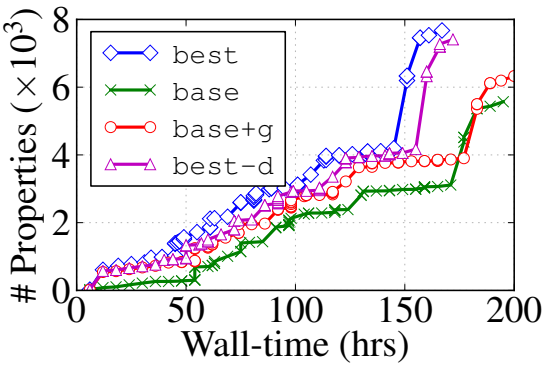


Figure 5.20: Number of properties solved vs. wall-time for **B3**.

ration and property partitioning. We extend scalable affinity-based property partitioning to guarantee *complete* utilization of available processes with provable partition affinities. We propose improvements to the scheduling of parallel processes, such as resource-constrained irredundant iteration, incremental repetition, and decomposition of difficult groups. We deliver a carefully-optimized localization portfolio, self-tailoring to irredundantly address a range of property difficulties through a synergistic balance of *Fast-and-Lossy* vs. *Aggressive* configurations. We propose improvements to *semantic group partitioning* within localization, boosting scalability by enabling the BMC within localization to benefit from larger and slightly-lower affinity groups, then optimally sub-dividing those groups before solving the localized properties. To our knowledge, this is the first published approach to optimize both property partitioning and strategy exploration within a multi-property localization portfolio. Experiments confirm that this solution works well across large suites of benchmarks.

The methods presented in this chapter are not limited to localization portfolios. The proposed heuristics are general, and can be used to accelerate any parallel multi-property verification task. Note that our mutually-optimized partitioning vs. strategy-exploration orchestration offers broad insights early in an ongoing verification-tool run, whereas traditional orchestration typically explores only easier (smaller-COI) properties or only a subset of

strategies early in the run. Optimized parallel verification is vital for maximizing verification throughput for design-space exploration using model-checking. The concurrent verification of high-affinity property groups ensures that incremental algorithms can reuse information across runs, and the control on the number of high-affinity property groups guarantees full-utilization of available parallel processes. Strategy exploration enables the use of different verification algorithms for different types of properties across parallel workers, while property partitioning with heuristics helps minimize redundant work across workers.

CHAPTER 6. CONCLUSION AND DISCUSSION

The process of *design-space exploration* presents a systematic methodology of discovering and evaluating design choices for a system under development. Design-space exploration must be performed carefully due to the large number of design alternatives to be explored to determine which design configurations are 'optimal', i.e., meet design specifications. The different competing systems arise out of a need to weigh different design choices, to check core capabilities of system versions with varying features, or to analyze a future version against previous ones in the product line. Every unique combination of choices yields competing systems that differ in terms of assumptions, implementations, and configurations. Formal verification techniques, like *model checking*, are growing increasingly vital for the development and verification of software and hardware systems. These techniques provide high-levels of safety assurance by guaranteeing that the designed system behaves according to the specification, and does not do anything that is outside the specified behavior. Model checking can aid system development by systematically comparing the different models in terms of *functional correctness*, however, applying model checking off-the-shelf may not scale due to the large size of the design space for today's complex systems. The designer faces a tradeoff: restrict design-choice combinations, or resort to time-honored but inherently-incomplete techniques of simulation or testing. For some systems this is an acceptable risk, but unacceptable for safety-critical systems whose failure might endanger human life. In this dissertation, we present scalable algorithms for *design-space exploration* using model checking that enable exhaustive comparison of all competing models in large design spaces.

6.1 Contribution Review

Model checking a design space entails checking multiple models and properties. We present algorithms that automatically prune the design space by finding inter-model relationships and property dependencies (Chapter 2). We observe that sequential enumeration of the design space generates models with small incremental differences. Typical model-checking algorithms do not take advantage of this information, and end up re-verifying "already-explored" state spaces across models. We evaluate our methodology on case-studies from NASA and Boeing; our techniques offer up to 9.4× speedup compared to traditional approaches. We present algorithms that learn and reuse information from solving related models in sequential model-checking runs (Chapter 3). Extensive experiments show that information reuse boosts runtime performance of sequential model-checking by up to 5.48×. Model-checking design spaces tasks often mandates checking several properties on individual models. State-of-the-art tools do not optimally exploit subproblem sharing between "nearly-identical" properties. We present a near-linear runtime algorithm for partitioning properties into provably high-affinity groups for individual model-checking tasks (Chapter 4). Our techniques significantly improve multi-property model-checking performance, and often yield >4.0× speedup. The verification effort expended for one property in a group can be directly reused to accelerate the verification of the others. Building upon these ideas, we optimize parallel verification to maximize the benefits of our proposed techniques. We propose methods to minimize redundant computation, and dynamically optimize work distribution when checking multiple properties for individual models (Chapter 5). Our methods offer a median 2.4× speedup for complex parallel verification tasks with thousands of properties.

6.2 Future Work

6.2.1 Design-Space Reduction

We plan to examine extending D^3 to other logics besides LTL, and its applicability to other types of transition systems, like families of Markov processes. We also plan to investigate further reduction in the search space by extending D^3 to re-use intermediate model checking results across several models. In a nutshell, D^3 is a front-end design-space preprocessing algorithm. Improved model checking back-ends that utilize available information can help reduce the overall amortized performance. Finally, since checking families of models is becoming commonplace, we plan to develop more industrial-sized SMV model sets and make them publicly available as research benchmarks.

6.2.2 Incremental Verification

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. We plan to extend FuseIC3 to checking liveness properties by using it as a safety checker[CS12]. 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.

6.2.3 Multi-Property Verification

Future work includes improved ordering and compaction of support bitvector bits to improve performance, e.g., support variables present in every property can be projected out of the bitvectors. Dynamic trie matching that discounts differences in very small SCCs in COI for properties, may improve level-2 grouping. Extending level-3 grouping to work with packed bitvectors may speed up grouping: large SCCs for which any distinction exceeds threshold require identical valuations in grouping, and smaller SCCs are either unpacked to multiple bits or treated with finer-grained map. Clever data structures, such as MA_FSA [DMWW00], and branch-and-bound traversal [WF74] can search for fairly-high-affinity bitvectors that differ in only a few n-bit segments, thereby reducing level-3 asymmetry. Extending semantic partitioning to cases where refinement occurs during a proof engine run is a promising research direction. We plan to investigate how semantic information from BMC and IC3 can be used to perform property grouping.

6.2.4 Parallel Orchestration

Our mutually-optimized partitioning vs. strategy-exploration orchestration offers broad insights early in an ongoing verification-tool run, whereas traditional orchestration typically explores only easier (smaller-COI) properties or only a subset of strategies early in the run. Exploring how this insight may enable dynamic benchmark-specific customized orchestration *during* an ongoing run is a promising future direction, e.g. dynamically adjusting which strategy is used per process and partition. Exploring these techniques across a broader set of engines, and exploring incrementality of strategies across localization and equivalence-class refinements, are additional promising research directions.

BIBLIOGRAPHY

[AI08] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[AM04] Nina Amla and Ken L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 260–274, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[App] Austin Appleby. SMHasher and MurmurHash3. https://github.com/aappleby/smhasher.

[ASH07] Christopher W. Anderson, Joost R. Santos, and Yacov Y. Haimes. A risk-based input–output methodology for measuring the effects of the august 2003 northeast blackout. *Economic Systems Research*, 19(2):183–204, 2007.

[AV06] David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Symposium on Computational Geometry (SCG)*, pages 144–153, New York, NY, USA, 2006. ACM.

[AvW+13] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491, May 2013.

[BBDEL96] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *DAC*, 1996.

[BC00] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 409–426, Berlin, Heidelberg, Oct 2000. Springer Berlin Heidelberg.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. pages 117–148, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BCFM00] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630 – 659, 2000.

[BCFP+15] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal design and safety analysis of AIR6110 wheel brake system. In *Computer-Aided Verification*, 2015.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^20 states and beyond. 98(2):428–439, 1990.

[BCM18] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, pages 727–762. Springer International Publishing, Cham, 2018.

[BDBK09] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the ground up*, volume 71. Springer Science & Business Media, 2009.

[BDK+14] Christel Baier, Clemens Dubslaff, Sascha Klüppelholz, Marcus Daum, Joachim Klein, Steffen Märcker, and Sascha Wunderlich. Probabilistic model checking and non-standard multi-objective reasoning. In *FASE*, 2014.

[BDSAB15] Shoham Ben-David, Baruch Sterin, Joanne M Atlee, and Sandy Beidu. Symbolic model checking of product-line requirements using SAT-based methods. In *ICSE*, volume 1, pages 189–199, 2015.

[BEM12] Robert Brayton, Niklas Een, and Alan Mishchenko. Using speculation for sequential equivalence checking. In *International Workshop on Logic and Synthesis (IWLS)*, Jun 2012.

[BF93] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering (TSE)*, 19(1):3–12, January 1993.

[Bie15] Armin Biere. HWMCC. http://fmv.jku.at/hwmcc15/, 2015.

[BLBM07] Christophe Bauer, Kristen Lagadec, Christian Bès, and Marcel Mongeau. Flight control system architecture optimization for fly-by-wire airliners. *Journal of Guidance, Control, and Dynamics*, 30(4):1023–1029, jul 2007.

[BM05] Jason Baumgartner and Hari Mony. Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies. In *Correct Hardware Design and Verification Methods*, Oct 2005.

[BM10] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification (CAV)*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BMP+06] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *2006 International Conference on Computer Design*, pages 259–266, Oct 2006.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Bra12] Aaron R Bradley. Understanding IC3. In *SAT*, pages 1–14, 2012.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[CBMK11] M. Case, J. Baumgartner, H. Mony, and R. Kanzelman. Optimal redundancy removal without fixedpoint computation. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 101–108, Oct 2011.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer-Aided Verification*, 2014.

[CCG+09] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer. Speeding up model checking by exploiting explicit and hidden verification constraints. In *DATE*, 2009.

[CCH+12] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):589–612, jun 2012.

[CCK+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In Mark D. Aagaard and John W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 33–51, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[CCL+17] G. Cabodi, P. E. Camurati, C. Loiacono, M. Palena, P. Pasini, D. Patti, and S. Quer. To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking. *STTT*, 2017.

[CCL+18] G. Cabodi, P. E. Camurati, C. Loiacono, M. Palena, P. Pasini, D. Patti, and S. Quer. 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 (STTT)*, 20(3):313–325, Jun 2018.

[CCQ16] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties. *Software: Practice and Experience*, 46(4):493–511, 2016.

[CCS+13a] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering (TSE)*, 39(8):1069–1089, aug 2013.

[CCS+13b] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 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.

[CDT13] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *ASE*, 2013.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[CGM+10] Gianpiero Cabodi, Luz Amanda Garcia, Marco Murciano, Sergio Nocco, and Stefano Quer. Partitioning interpolant-based verification for effective unbounded model checking. *TCAD*, 29(3), 2010.

[CGMT13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with IC3. In *Formal Methods in Computer-Aided Design*. IEEE, oct 2013.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 93–107, 2013.

[CHS+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE*, 2010.

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *International Conference on Software Engineering (ICSE)*, page 321–330, New York, NY, USA, 2011. Association for Computing Machinery.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[CIM+11] Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental Formal Verification of Hardware. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 135–143, 2011.

[CK16] Sagar Chaki and Derrick Karimi. Model checking with multi-threaded IC3 portfolios. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 517–535, Berlin, Heidelberg, Jan 2016. Springer Berlin Heidelberg.

[CKV06] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. Coverage metrics for temporal logic model checking. *FMSD*, 28(3):189–212, 2006.

[CM10] M. Chen and P. Mishra. Functional test generation using efficient property clustering and learning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):396–404, March 2010.

[CN11a] G Cabodi and S Nocco. Optimized model checking of multiple properties. In *DATE*, 2011.

[CN11b] G. Cabodi and S. Nocco. Optimized model checking of multiple properties. In *2011 Design, Automation Test in Europe*, pages 1–4, March 2011.

[CS12] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 52–59, 2012.

[CZ17] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Comput. Surv.*, 50(1):11:1–11:36, April 2017.

[DASBW15] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wąsowski. Family-based model checking without a family-based model checker. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, pages 282–299. Springer International Publishing, Cham, 2015.

[DBI+19] Rohit Dureja, Jason Baumgartner, Alexander Ivrii, Robert Kanzelman, and Kristin Yvonne Rozier. Boosting verification scalability via structural grouping and semantic partitioning of properties. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, Oct 2019.

[DBK+20] Rohit Dureja, Jason Baumgartner, Robert Kanzelman, Mark Williams, and Kristin Y. Rozier. Accelerating Parallel Verification via Complementary Property Partitioning and Strategy Exploration. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, Haifa, Israel, September 2020. IEEE/ACM.

[DG18] Dennis Dams and Orna Grumberg. *Abstraction and Abstraction Refinement*, pages 385–419. Springer International Publishing, Cham, 2018.

[DJJ+15] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. PROPhESY: A probabilistic parameter synthesis tool. In *Computer-Aided Verification*, 2015.

[DJJ+16] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. Parameter synthesis for probabilistic systems. *MBMV*, 2016.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(7):1165–1178, July 2008.

[DLP+19] Rohit Dureja, Jianwen Li, Geguang Pu, Moshe Y. Vardi, and Kristin Y. Rozier. Intersection and rotation of assumption literals boosts bug-finding. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, pages 180–192, Cham, 2019. Springer International Publishing.

[DMJO18] K. Debnath, R. Murgai, M. Jain, and J. Olson. SAT-based redundancy removal. In *Design, Automation and Test in Europe (DATE)*, pages 315–318, Mar 2018.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1):3–16, 2000.

[Dow97] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, March 1997.

[DR17] Rohit Dureja and Kristin Y. Rozier. FuseIC3: An algorithm for checking large design spaces. In *Formal Methods in Computer-Aided Design (FMCAD)*, Vienna, Austria, October 2017. IEEE/ACM.

[DR18] Rohit Dureja and Kristin Yvonne Rozier. More Scalable LTL Model Checking via Discovering Design-Space Dependencies (D^3). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 309–327, Cham, 2018. Springer International Publishing.

[DR20a] Rohit Dureja and Kristin Yvonne Rozier. Formal framework for safety, security, and availability of aircraft communication networks. *Journal of Aerospace Information Systems*, 17(7):322–335, 2020.

[DR20b] Rohit Dureja and Kristin Yvonne Rozier. Incremental design-space model checking via reusable reachable state approximations. *Formal Methods in System Design*, 2020.

[EK00] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE*, 2000.

[EKVY07] K. Etessami, M. Kwiatkowska, M. Y. Vardi, and M. Yannakakis. Multi-objective model checking of markov decision processes. In *Tools and Algorithms for Construction and Analysis of Systems*, 2007.

[EM13] Niklas Eén and Alan Mishchenko. A fast reparameterization procedure. In *International Workshop on Design and Implementation of Formal Tools and Systems*, 2013.

[EMB11] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient Implementation of Property Directed Reachability. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, 2011.

[Fix08] Limor Fix. *Fifteen Years of Formal Property Verification in Intel*, pages 139–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[FKN+11] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In *Tools and Algorithms for Construction and Analysis of Systems*, 2011.

[GBI+19] R. K. Gajavelly, J. Baumgartner, A. Ivrii, R. L. Kanzelman, and S. Ghosh. Input elimination transformations for scalable verification and trace reconstruction. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2019.

[GBS+06] T. Glokler, J. Baumgartner, D. Shanmugam, R. Seigler, G. V. Huben, B. Ramanandray, H. Mony, and P. Roessler. Enabling large-scale pervasive logic verification through multi-algorithmic formal reasoning. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 3–10, Nov 2006.

[GCM+16] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Yvonne Rozier. Model checking at scale: Automated air traffic control design space exploration. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification (CAV)*, pages 3–22, Cham, 2016. Springer International Publishing.

[GGKM18] E. Goldberg, M. Güdemann, D. Kroening, and R. Mukherjee. Efficient verification of multi-property designs (The benefit of wrong assumptions). In *Design, Automation Test in Europe (DATE)*, pages 43–48, March 2018.

[GNP18] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. *Compositional Reasoning*, pages 345–383. Springer International Publishing, Cham, 2018.

[Gon85] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293 – 306, 1985.

[GR16] Alberto Griggio and Marco Roveri. Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 35(6):1026–1039, Jun 2016.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[HC13] Hung-Yi Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7, May 2013.

[He10] Xiuqiang He. *Efficient Techniques for Design Space Exploration and Optimization of Distributed Real-Time Embedded Systems*. PhD thesis, 2010.

[HHZ11] Ernst Moritz Hahn, Tingting Han, and Lijun Zhang. Synthesis for PCTL in parametric markov decision processes. In *NFM*, 2011.

[JH19] Phillip Johnston and Rozi Harris. The boeing 737 MAX saga: lessons for software organizations. *Software Quality Professional*, 21(3):4–12, 2019.

[JMN+14] Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. On modelling and verifying railway interlockings: Tracking train lengths. *Science of Computer Programming*, 96(3), 2014.

[JPM+12] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. Modeling and verification of a dual chamber implantable pacemaker. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 188–203, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[KB01] Andreas Kuehlmann and Jason Baumgartner. Transformation-based verification using generalized retiming. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification (CAV)*, pages 104–117, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999.

[KGN+09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in intel core i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[KJCH19] Jintaek Kang, Dowhan Jung, Kwanghyun Chung, and Soonhoi Ha. Fast performance estimation and design space exploration of manycore-based neural processors. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[KJS11] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. In Radu Calinescu and Ethan Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, pages 33–54, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[KN12] Zurab Khasidashvili and Alexander Nadel. Implicative simultaneous satisfiability and applications. In *HVC*, 2012.

[KNPH06] Zurab Khasidashvili, Alexander Nadel, Amit Palti, and Ziyad Hanna. Simultaneous sat-based model checking of safety properties. In Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, editors, *HVC*, 2006.

[KNPQ13] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Compositional probabilistic verification through multi-objective model checking. *Inf. Comput.*, 232, 2013.

[Koo14] Phil Koopman. A case study of toyota unintended acceleration and software safety. 2014.

[LDP+18] Jianwen Li, Rohit Dureja, Geguang Pu, Kristin Yvonne Rozier, and Moshe Y. Vardi. SimpleCAR: An efficient bug-finding tool based on approximate reachability. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 37–44, Cham, 2018. Springer International Publishing.

[LGHT08] M. Lukasiewycz, M. Glass, C. Haubelt, and J. Teich. Efficient symbolic multi-objective design space exploration. In *Asia and South Pacific Design Automation Conference*, pages 691–696, March 2008.

[LPP+13] C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, S. Ricossa, D. Vendraminetto, and J. Baumgartner. Fast cone-of-influence computation and estimation in problems with multiple properties. In *Design, Automation Test in Europe (DATE)*, pages 803–806, March 2013.

[LXX+09] Peng Liu, Bingjie Xia, Chunchang Xiang, Xiaohang Wang, Weidong Wang, and Qingdong Yao. A networks-on-chip architecture design space exploration – the lib. *Computers and Electrical Engineering*, 35(6):817 – 836, 2009. High Performance Computing Architectures.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 2–17, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[MAW+12] Steven Miller, Elise Anderson, Lucas Wagner, Michael Whalen, and Matts Heimdahl. *Formal Verification of Flight Critical Software*. 2012.

[MBMB09] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton. Speculative reduction-based scalable redundancy identification. In *Design, Automation and Test in Europe (DATE)*, pages 1674–1679, Apr 2009.

[MBP+04] Hari Mony, Jason Baumgartner, Viresh Paruthi, Robert Kanzelman, and Andreas Kuehlmann. Scalable automated verification via expert-system guided transformations. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 159–173, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[MBPK05] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. In *Design Automation Conference (DAC)*, pages 463–466, Jun 2005.

[MCBJ08] Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. In *International Conference on Computer-Aided Design*, 2008.

[MCG+15] Cristian Mattarei, Alessandro Cimatti, Marco Gario, Stefano Tonetta, and Kristin Y. Rozier. Comparing different functional allocations in automated air traffic control design. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, sep 2015.

[McM03] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[ME04] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Networked Systems Design and Implementation (NSDI)*, NSDI'04, page 12, USA, 2004. USENIX Association.

[MEB+13] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla. GLA: Gate-level abstraction revisited. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1399–1404, March 2013.

[MGHS17] Matteo Marescotti, Arie Gurfinkel, Antti E. J. Hyvärinen, and Natasha Sharygina. Designing parallel PDR. In *Formal Methods in Computer-Aided Design (FMCAD)*, page 156–163, Austin, Texas, Oct 2017. FMCAD Inc.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, IJCAI'71, page 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[MNR+13] Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, and Helen Treharne. Defining and model checking abstractions of complex railway models using CSP||B. In *HVC*, 2013.

[MS07a] Joao Marques-Silva. Interpolant learning and reuse in sat-based model checking. *Theoretical Computer Science*, 174(3):31 – 43, 2007.

[MS07b] Joao Marques-Silva. Interpolant learning and reuse in sat-based model checking. *Electronic Notes in Theoretical Computer Science*, 174(3):31 – 43, 2007. Proceedings of the Fourth International Workshop on Bounded Model Checking (BMC 2006).

[NGB+16] Pradeep Kumar Nalla, Raj Kumar Gajavelly, Jason Baumgartner, Hari Mony, Robert Kanzelman, and Alexander Ivrii. The art of semi-formal bug hunting. In *International Conference on Computer-Aided Design (ICCAD)*, New York, NY, USA, 2016. ACM.

[NM] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proceedings ED&TC European Design and Test Conference*. IEEE Computer Society Press.

[PCC11] S. Padmanabhan, Y. Chen, and R. D. Chamberlain. Optimal design-space exploration of streaming applications. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 227–230, Sep. 2011.

[Pel18] Doron Peled. *Partial-Order Reduction*, pages 173–190. Springer International Publishing, Cham, 2018.

[PI17] Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In Leonardo de Moura, editor, *International Conference on Automated Deduction (CADE)*, pages 26–41, Cham, 2017. Springer International Publishing.

[Pim17] A. D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design and Test*, 34(1):77–90, Feb 2017.

[PJ09] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(Part 2):3336 – 3341, 2009.

[PMRR19] V. N. Possani, A. Mishchenko, R. P. Ribas, and A. I. Reis. Parallel combinational equivalence checking. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Oct 2019.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society.

[psl10] IEEE standard for property specification language (psl). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010.

[QDJ+16] Tim Quatmann, Christian Dehnert, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter synthesis for markov models: Faster than ever. In *ATVA*, 2016.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[RM05] Lior Rokach and Oded Maimon. *Clustering Methods*, pages 321–352. Springer US, 2005.

[Roz16] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 8–26, Cham, 2016. Springer International Publishing.

[RS10] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. *VaMoS*, 10, 2010.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[RV07] Kristin Y Rozier and Moshe Y Vardi. LTL satisfiability checking. In *SPIN*, 2007.

[SB11] Fabio Somenzi and Aaron R Bradley. IC3: Where Monolithic and Incremental Meet. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 3–8, 2011.

[SKB+16] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software. *Formal Aspects of Computing*, 2016.

[SS09] Horst Schirmeier and Olaf Spinczyk. Challenges in software product line composition. In *HICSS*. IEEE, 2009.

[SSZ11] Anirban Sengupta, Reza Sedaghat, and Zhipeng Zeng. Multi-objective efficient design space exploration and architectural synthesis of an application specific processor (asp). *Microprocessors and Microsystems*, 35(4):392 – 404, 2011.

[STF16] M. Y. Siraichi, C. Tonetti, and A. Faustino da Silva. A design space exploration of compiler optimizations guided by hot functions. In *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–12, Oct 2016.

[Str09] Ofer Strichman. Regression verification: Proving the equivalence of similar programs. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 63–63, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[sva18] IEEE standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. In *SIAM Journal on Computing*, 1972.

[Tho12] M. Thompson. Tools and techniques for efficient system-level design space exploration. jan 2012.

[van98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Design, Automation and Test in Europe (DATE)*, pages 618–623, Feb 1998.

[ver06] IEEE standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[VG09] Y. Vizel and O. Grumberg. Interpolation-sequence based model checking. In *Formal Methods in Computer-Aided Design*, pages 1–8, Nov 2009.

[War12] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, jan 1974.

[YBO+98] Bwolen Yang, Randal E Bryant, David R O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K Ranjan, and Fabio Somenzi. A performance study of bdd-based model checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 255–289, 1998.

[YCY20] B. Yuan, H. Chen, and X. Yao. Toward efficient design space exploration for fault-tolerant multiprocessor systems. *IEEE Transactions on Evolutionary Computation*, 24(1):157–169, Feb 2020.

[YDR09] Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.

[YFB+19] Guowei Yang, Antonio Filieri, Mateus Borges, Donato Clun, and Junye Wen. Advances in symbolic execution. volume 113 of *Advances in Computers*, pages 225 – 287. Elsevier, 2019.

[ZBG20] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Network-level design space exploration of resource-constrained networks-of-systems. *ACM Trans. Embed. Comput. Syst.*, 19(4), June 2020.