More Scalable LTL Model Checking via Discovering Design-Space Dependencies (D^3)

Rohit Dureja and Kristin Yvonne Rozier

Iowa State University
{dureja, kyrozier}@iastate.edu

Abstract. Modern system design often requires comparing several models over a large design space. 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 different models; however, applying model checking off-the-shelf may not scale due to the large size of the design space for today's complex systems. We exploit relationships between different models of the same (or related) systems to optimize the model-checking search. Our algorithm, D^3, preprocesses the design space and checks fewer model-checking instances, e.g., using nuXmv. It automatically prunes the search space by reducing both the number of models to check, and the number of LTL properties that need to be checked for each model in order to provide the complete model-checking verdict for every individual model-property pair. We formalize heuristics that improve the performance of D^3. We demonstrate the scalability of D^3 by extensive experimental evaluation, e.g., by checking 1,620 real-life models for NASA's NextGen air traffic control system. Compared to checking each model-property pair individually, D^3 is up to 9.4× faster.

1 Introduction

In the early phases of design, there are frequently many different models of the system under development [2, 23, 29] 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 [29, 23]. 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 [3, 23, 25, 29, 30]. Algorithms for family-based model

Thanks to NSF CAREER Award CNS-1552934 for supporting this work.

checking [13, 11] 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 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. 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 [33] 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. 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 [8, 23, 29]. We also evaluate the property-dependence analysis separately on real-life models of Boeing AIR 6110 Wheel Braking System [3] to evaluate D^3 in multi-property verification workflows.

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 [9] 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 inter-

est 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 [15, 16, 24, 31] 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 [1, 21, 22, 28], 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. The parameterized model checking problem (PCMP) [20] deals with infinite families of homogeneous processes in a system; in our case, the models are finite and heterogeneous. Specialized model-set checking algorithms [18] 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 [4, 7], SAT-solver modifications [27, 26], and property grouping [6, 5]. The inter-property dependence analysis of D^3 can be used in multi-property checking. We compare D^3 against the *affinity*[6] 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 [32, 34]. An extension to NuSMV in [13] performs symbolic model checking of feature-oriented CTL. The symbolic analysis is extended to the explicit case and support for feature-oriented LTL in [11, 12]. The work most closely related to ours is [17] 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 [14, 17].

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 [6].
4. Extensive experimental analysis using real-life benchmarks; all reproducibility artifacts and source code are publicly available.

2 Preliminaries

Definition 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 path, 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 $0 \leq i < n$. Given a LTL property φ and a LTS M, M *models* φ, denoted $M \models \varphi$, iff φ holds in all possible computation paths of M.

Definition 2. 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 3. 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.

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 1. A Boolean parameter has domain $\{true, false\}$. Fig. 1 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*.

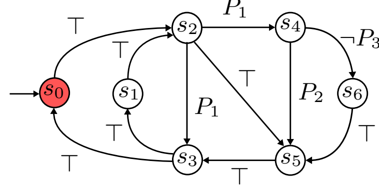


Fig. 1: An example of a combinatorial transition system M_P with parameters $P = \{P_1, P_2, P_3\}$.

Definition 4. 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 5. 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 6. 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. In Fig. 1, 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 7. A *universal model* U is a LTS that generates all possible computations paths over its atomic propositions.

Theorem 1 (LTL Satisfiability). [33] 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.[1]

Modeling a Combinatorial Transition System. Efficient modeling of a CTS 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 $M_{P(c)}$. Fig. 2 demonstrates generating a CTS from two related 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.

```
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

Fig. 2: 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.

3 Discovering Design-Space Dependencies

In this section we describe D^3. 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.

3.1 Reduction Along the Number of Instances

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

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

$c \in \mathbb{C}$, 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. 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.

Lemma 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 A and B, resp., and $s_i, s_j, s_k, s_l \in S$, i.e., a transition with guard over B does not appear before a transition with guard over A.*

As a corollary to Lemma 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 (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 $M_{P(c_1)} \models \varphi \equiv M_{P(c_2)} \models \varphi \equiv \ldots \equiv M_{P(c_k)} \models \varphi$.

```
function FindUP (M_P, ĉ)
Input: M_P = CTS (Σ, S, s_o, L, δ, P, L_P)
         ĉ = partial configuration
Output: P_u = unset parameter queue
1: if all parameters are set in ĉ : return ∅
   # initially P_u is empty.
2: traverse M_P  # depth-first traversal
3:    if t ∈ δ̂ is reachable and
              L_P(t) is undefined :
   # L_P(t) is undefined when its parameter
   is NOT set in partial configuration ĉ.
4:    enqueue (p : L_P(t) is guard over p) in P_u
5: return P_u
```

(a) FindUP algorithm to find unset parameters in a partially configured CTS.

```
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
     # iterate on possible assignments to p
4:   for each p_d in ⟦p⟧ :
5:     set parameter p to p_d in ĉ
6:       # get unset parameters
7:     P_u = FindUP(M_P, ĉ)
8:     if P_u is empty :  # all parameters set
9:       add ĉ to Ĉ and return
10:    else: # set unset parameters
11:      GenPC(M_P, P_u, ĉ)
```

(b) GenPC algorithm to generate parameter configurations to be checked.

Fig. 3: Algorithms for reduction along the number of instances

Theorem 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? 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 U**nset **P**arameters) in Fig. 3(a) 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), 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. 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.*

Algorithm GenPC (**Gen**erate **P**arameter **C**onfigurations) in Fig. 3(b) 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 $[\![p]\!]$ 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 3 (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 $c_1, c_2, \dots c_k \in \mathbb{C}$ 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 $[\![B]\!] = \{d_{B_1}, d_{B_2}, \dots, d_{B_k}\}$*

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

Theorem 4 (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 there exists 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 5 (Minimality). *The minimal set of parameter configurations is $\hat{\mathbb{C}}$.*

3.2 Reduction Along the Number of Properties

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.

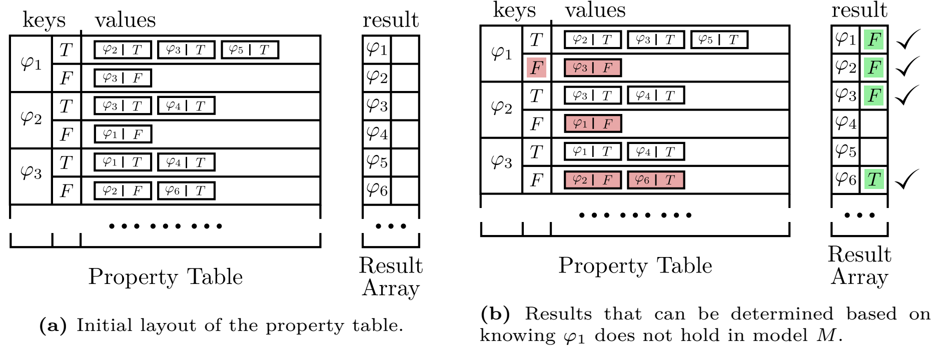
In order to check all properties in $\mathcal{P}$, a straightforward possibility is to generate a grouped property φ_g given by the conjunction of all properties $\varphi_i \in \mathcal{P}$, i.e., $\varphi_g = \bigwedge_i \varphi_i$. However, the straightforward approach may not scale [6] due to

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 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 $(\neg\varphi_1 \rightarrow \neg\varphi_2)$. Theorem 6 allows us to find dependencies automatically.

Theorem 6 (Property Dependence). For two LTL properties φ_1 and φ_2 dependence can be established by model checking with universal model U.

The dependencies learned as a result of Theorem 6 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 φ_1 is *true*, there is no counterexample to prove that φ_2 is *false*).



(a) Initial layout of the property table.

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

Fig. 4: Property table to store dependence between every LTL property pair in 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 φ_1 holds for model M then φ_2 also holds.

The pairwise property dependencies are stored in a property table as shown in Fig. 4(a). Each row in the table is a $(key, value)$ pair. For LTL properties φ_1, φ_2, and φ_3 in $\mathcal{P}$, if $(\varphi_1 \rightarrow \varphi_2)$ 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 Fig. 5 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 Fig. 4(b), 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.

```
1: array results  # initially empty
function CheckRP (M, P, T)
 Input: M = LTS (Σ, S, s0, L, δ)
        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 ⊨ φ, else F
5:   set S = {(φ : outcome)}
6:   while S is not empty :
7:     (p : result) = pop element from S
8:     results[ p ] = result  # update result
9:     S = S ∪ unchecked properties
               dependent on (p : result) in D
10: return
```

Fig. 5: CheckRP algorithm to check LTL properties against a model.

```
function D^3 (M_P, P)
 Input: M_P = CTS (Σ, S, s0, L, δ, P, L_P)
        P = set of LTL properties
1: configuration set Ĉ  # initially empty
2: parameter queue P_u = FindUP(M_P, _)
3: Ĉ = GenPC(M_P, P_u, _)  # See § 3.1
   # generate property table, see § 3.2
4: property table D  # initially empty
5: for every property pair (φ1, φ2) in P :
6:   check if φ1 and φ2 are dependent
7:   add entry to D
   # check configured instances
8: for each c in Ĉ :
9:   generate instance M_P(c)  # See § 2
10:  array results  # initially empty
11:  CheckRP(M_P(c), P, D)  # See § 3.2
12:  return results
```

Fig. 6: Discovering Design-Space Dependencies (D^3) algorithm.

H1: Maximum Dependence. 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. If $\mathcal{U} \subseteq \mathcal{P}$ are unchecked properties in table D, 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.

H2: Property Grouping. 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* [6, 5] 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 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 φ_i and φ_j are checked individually against model M.

4 Experimental Analysis

Our revised model checking procedure D^3 is shown in Fig. 6. 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.

4.1 Benchmarks

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

1) *Air Traffic Controller (ATC) Models:* 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 [23]. In the formulation of [23], the checking problem for each model is split in to five phases.[2]. 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 [10] for compositional modeling, though we do not rely on its features when using the generated model-set.

2) *Boeing Wheel Braking System (WBS) Models:* are a set of seven real-world nuXmv models representing possible designs for the Boeing AIR 6110 wheel braking system [3]. Each model in the set is checked against ∼200 LTL properties. However, the seven models are not generated from a CTS. We evaluate D^3 against this benchmark to evaluate performance on multi-property verification workflows, and compare with existing work on property grouping [6].

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 no resource conflict with other jobs will occur.

4.3 Experimental Results

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 1 summarizes the results for complete verification of the ATC design space: 191 LTL properties for each of 1,620 models.

[2]For a detailed explanation we refer the reader to [23]

Table 1: Timing results of 1,620 models for each phase using individual-model checking, and D^3. 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×	
II	UNSAT	29 (19)	12.76	5.17	2.5×	4.5×
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×	-

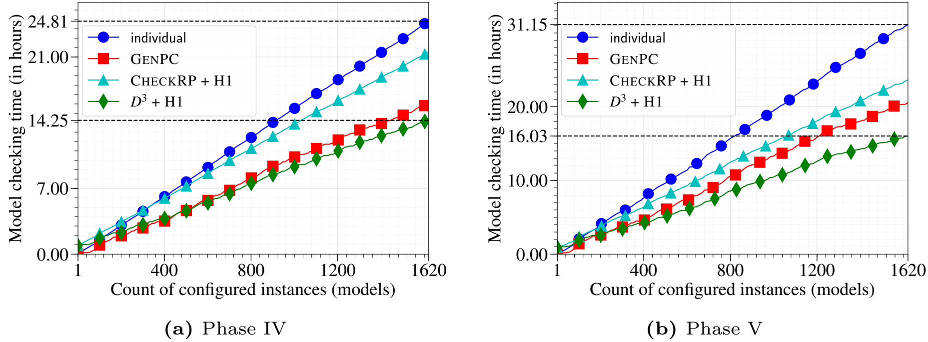
Compared to individual model checking, wherein every model-property pair is checked one-by-one, verification of the ATC design space using D^3 is 4.0× faster. It reduces the the 1,620 models in the design space to 1,028 models. D^3 takes roughly three hours to find dependencies between LTL properties 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 Fig. 7.

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 D^3 also increases.

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 [23].

2) *Boeing Wheel Braking System (WBS) Models.* LTL Properties for each of the seven models are checked using four algorithms:

i. Single : properties are checked one-by-one against the model,
ii. CheckRP : properties are checked using inter-property dependencies,
iii. CheckRP + Maximum Dependence (H1) : unchecked property with the maximum dependent properties as per inter-property dependencies is checked,



(a) Phase IV

(b) Phase V

Fig. 7: 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). D^3 outperforms individual-model checking in all phases.

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

Fig. 8 summarizes the results. 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.

Analysis. For H2, we limited our experiments to pairwise groupings, however, larger groupings may be possible (trade-off required between property interdependencies and groupings). It took ∼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 will start reaping benefits. Nevertheless, CheckRP is suited for multi-property verification in large design spaces.

5 Conclusions and Future Work

We present an algorithm, 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 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

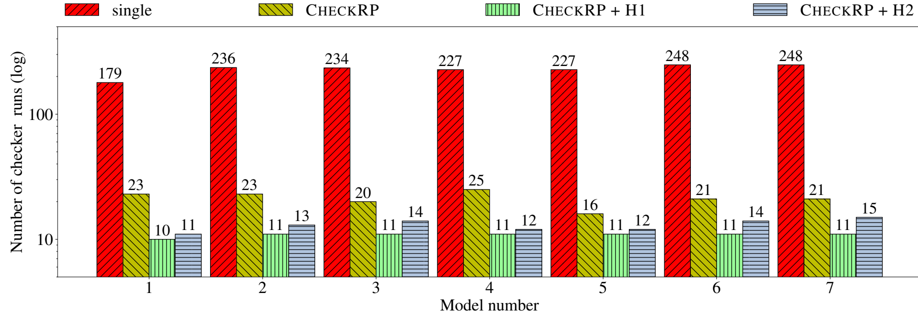


Fig. 8: Number of calls made to the model checker to verify all properties in the set for a model. Every call to the checker verifies one property: single or grouped. For CheckRP, multiple property results are determined (based on inter-property dependencies) on every checker run. Heuristics H1 and H2 improve performance of CheckRP.

Table 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

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. 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 preprocessing algorithm, and future work involves tying in an improved model checking back-end and utilizing available information to 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 Supporting Artifact

The artifact for reproducibility of our experiments [19] is publicly available under the MIT License, and supports all reported results of Section 4. It includes

1. *Benchmarks:* NASA's NextGen Air Traffic Control System [23] and Boeing's Wheel Braking System [3] (Section 4.1).
2. *Scripts:* Python scripts to run D^3 on the two benchmarks (Figure 6).
3. *Datasets:* Ready-to-use datasets generated during our analysis (Section 4.3)

The artifact supports the following usage scenarios.

1. Verify the benchmarks using both individual-model checking and model checking with D^3, or run the complete experimental analysis to reproduce the results reported in Tables 1 and 2.
2. Study and evaluate the benchmarks and source code for D^3, sub-algorithms (GenPC and CheckRP), and heuristics (H1 and H2).
3. Introduce extensions to D^3 and experiment with new heuristics.

Please refer to the README files in the artifact for further information. Every README inside a directory details the directory structure, usage of contained files with respect to the evaluation, and step-by-step instructions on how to the use the contained scripts to regenerate the experimental analysis.

Data Availability Statement. The benchmarks evaluated, source code, and datasets generated during our experimental analysis are available in the Springer/ Figshare repository: `https://doi.org/10.6084/m9.figshare.5913013`. Theorem proofs and extended results are available on the paper's accompanying website: `http://temporallogic.org/research/TACAS18/`.

References

1. Baier, C., Dubslaff, C., Klüppelholz, S., Daum, M., Klein, J., Märcker, S., Wunderlich, S.: Probabilistic model checking and non-standard multi-objective reasoning. In: FASE (2014)
2. Bauer, C., Lagadec, K., Bès, C., Mongeau, M.: Flight control system architecture optimization for fly-by-wire airliners. J. Guidance, Control, & Dynamics 30(4) (2007)
3. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal design and safety analysis of AIR6110 wheel brake system. In: CAV (2015)
4. Cabodi, G., Camurati, P., Garcia, L., Murciano, M., Nocco, S., Quer, S.: Speeding up model checking by exploiting explicit and hidden verification constraints. In: DATE (2009)
5. Cabodi, G., Camurati, P.E., Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S.: To split or to group: from divide-and-conquer to sub-task sharing for verifying multiple properties in model checking. STTT (2017)
6. Cabodi, G., Nocco, S.: Optimized model checking of multiple properties. In: DATE (2011)

7. Cabodi, G., Garcia, L.A., Murciano, M., Nocco, S., Quer, S.: Partitioning interpolant-based verification for effective unbounded model checking. TCAD 29(3) (2010)
8. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV (2014)
9. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: FMCAD (2013)
10. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: A tool for checking the refinement of temporal contracts. In: ASE (2013)
11. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. JSTTT 14(5) (2012)
12. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. TSE 39(8) (2013)
13. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE (2011)
14. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE (2010)
15. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.P., Ábrahám, E.: PROPhESY: A probabilistic parameter synthesis tool. In: CAV (2015)
16. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.P., Ábrahám, E.: Parameter synthesis for probabilistic systems. MBMV (2016)
17. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wąsowski, A.: Family-based model checking without a family-based model checker. In: Model Checking Software
18. Dureja, R., Rozier, K.Y.: FuseIC3: An algorithm for checking large design spaces. In: FMCAD (2017)
19. Dureja, R., Rozier, K.Y.: More Scalable LTL Model Checking via Discovering Design-Space Dependencies (Artifact). `https://doi.org/10.6084/m9.figshare.5913013` (2018)
20. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE (2000)
21. Etessami, K., Kwiatkowska, M., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of markov decision processes. In: TACAS (2007)
22. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: TACAS (2011)
23. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: CAV (2016)
24. Hahn, E.M., Han, T., Zhang, L.: Synthesis for PCTL in parametric markov decision processes. In: NFM (2011)
25. James, P., Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: On modelling and verifying railway interlockings: Tracking train lengths. Science of Computer Programming 96(3) (2014)
26. Khasidashvili, Z., Nadel, A.: Implicative simultaneous satisfiability and applications. In: HVC (2012)
27. Khasidashvili, Z., Nadel, A., Palti, A., Hanna, Z.: Simultaneous sat-based model checking of safety properties. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC (2006)
28. Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Compositional probabilistic verification through multi-objective model checking. Inf. Comput. 232 (2013)

29. Mattarei, C., Cimatti, A., Gario, M., Tonetta, S., Rozier, K.Y.: Comparing different functional allocations in automated air traffic control design. In: FMCAD (2015)
30. Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Defining and model checking abstractions of complex railway models using CSP||B. In: HVC (2013)
31. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.P.: Parameter synthesis for markov models: Faster than ever. In: ATVA (2016)
32. Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. VaMoS 10 (2010)
33. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: SPIN (2007)
34. Schirmeier, H., Spinczyk, O.: Challenges in software product line composition. In: HICSS. IEEE (2009)