Iterative Variable Reordering: Taming Huge System Families*

Clemens Dubslaff
Institute of Theoretical Computer Science
Technische Universität Dresden
Dresden, Germany
clemens.dubslaff@tu-dresden.de

Andrey Morozov
Institute of Automation
Technische Universität Dresden
Dresden, Germany
andrey.morozov@tu-dresden.de

Christel Baier
Institute of Theoretical Computer Science
Technische Universität Dresden
Dresden, Germany
christel.baier@tu-dresden.de

Klaus Janschek
Institute of Automation
Technische Universität Dresden
Dresden, Germany
klaus.janschek@tu-dresden.de

For the verification of systems using model-checking techniques, symbolic representations based on binary decision diagrams (BDDs) often help to tackle the well-known state-space explosion problem. Symbolic BDD-based representations have been also shown to be successful for the analysis of families of systems that arise, e.g., through configurable parameters or following the feature-oriented modeling approach. The state space of such system families face an additional exponential blowup in the number of parameters or features. It is well known that the order of variables in ordered BDDs is crucial for the size of the model representation. Especially for automatically generated models from real-world systems, family models might even be not constructible due to bad variable orders. In this paper we describe a technique, called *iterative variable reordering*, that can enable the construction of large-scale family models. We exemplify feasibility of our approach by means of an aircraft velocity control system with redundancy mechanisms modeled in the input language of the probabilistic model checker PRISM. We show that standard reordering and dynamic reordering techniques fail to construct the family model due to memory and time constraints, respectively, while the new iterative approach succeeds to generate a symbolic family model.

1 Introduction

Model checking is an automated technique for the verification of systems [9, 5], applied in many areas of system design where reliability and correctness are key. The biggest challenge model checking faces is the well-known *state-explosion problem* that describes the exponential blow up of states in the number of system variables. One prominent approach to cope with the state-explosion problem in model checking is the use of symbolic methods, such as those based on binary decision diagrams (BDDs) [7, 25]. Such symbolic methods are most viable the more redundant or shared behaviors are present in the system description, as symbolic methods can exploit them towards concise representations. It is hence not surprising that symbolic model-checking techniques also have been successfully applied for the analysis of system families those members share lots of common behaviors. Most prominently, in feature-oriented system analysis [33], a member of the system family is characterized by its basic functionalities by means of *features*. Features can either be active or inactive, possibly leading to system families those size is

*This work is supported by the DFG through the Collaborative Research Centers CRC 912 (HAEC) and TRR 248 (see https://perspicuous-computing.science, project ID 389792660), the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), the Research Training Groups QuantLA (GRK 1763) and RoSI (GRK 1907), projects JA-1559/5-1, BA-1679/11-1, BA-1679/12-1, and the 5G Lab Germany.

A. Fehnker and H. Garavel (Eds.):
Models for Formal Analysis of Real Systems (MARS 2020)
EPTCS 316, 2020, pp. 121–133, doi:10.4204/EPTCS.316.5

© C. Dubslaff, A. Morozov, C. Baier, and K. Janschek
This work is licensed under the
Creative Commons Attribution License.

exponential in the number of features. In case family members have active features in common, they likely share the behaviors of these features, making symbolic methods effective. Despite for feature-oriented systems, also other kinds of configurable systems can benefit from symbolic methods as one easily faces an exponential blow up in the number of configuration parameters.

There are mainly two approaches to construct and analyze system families. The first, called *one-by-one approach*, constructs and analyzes the model for each family member separately. Differently, within an *all-in-one approach*, a single family model is constructed and analyzed in a single run. For system families those members share a lot of common behaviors, all-in-one approaches can benefit from symbolic representations and analysis methods by efficiently representing common parts. This usually leads to better analysis times than for one-by-one approaches [11, 33, 14, 8] or even enables an analysis [21, 15].

In this paper, we focus on all-in-one approaches where the family model is symbolically represented using reduced ordered BDDs. Such BDDs are rooted directed graphs where inner nodes are labeled by variable names, each path obeying a given variable order. It is well known that the size of the BDD significantly depends on the chosen variable order [6, 21]. In real-world applications, models subject to verification are usually automatically generated such that ad-hoc variable orders are likely to provide bigger BDD representation than possible with another variable order or even lead to BDD sizes exceeding memory constraints. Reordering algorithms such as sifting [30, 28] help to find suitable variable orders towards a small symbolic representation but are not applicable when the model cannot be constructed a priori due to insufficient memory. A solution is to dynamically reorder BDD variables [17] during model construction, which however comes at the cost of spending much time on reordering. For system families where the family model cannot be constructed due to bad variable orders and dynamic reordering techniques exceed time constraints, we present a third automated method we call *iterative variable reordering*. The basic idea is to construct parts of the system family, apply variable-reordering techniques towards smaller BDD representations, and then successively add family members until a suitable variable order for the whole family model is found. In this way, the shared behaviors between family members are step-wise incorporated into the symbolic representation. We first used this idea in [15] to enable the construction of a family model of aircraft velocity control loops given in the input language of the prominent probabilistic model checker PRISM [22]. The model of [15] was automatically generated from SIMULINK code [1] using SIMPARS [27], initially not constructible using the symbolic engine of PRISM based on *multi-terminal BDDs* [10, 18, 3]. Starting with the most basic variant of the aircraft velocity control loop, we step-wise added system variants and applied automated variable reordering capabilities presented in [21] to obtain a suitable variable order such that the whole family model could be constructed. While [15] focused on enabling a reliability analysis of the aircraft velocity control loop, we approach iterative variable reordering in a generic fashion in this paper and present an automated variant of the ad-hoc and handcrafted concepts of [15]. Specifically, our contribution is as follows:

- we formally specify iterative variable reordering along with heuristics,
- present an implementation of the algorithm that enables the automated application of iterative variable reordering on PRISM family models, and
- evaluate the approach on an even bigger instance of the velocity control loop model from [15], showing the effectiveness of our implementation.

To the best of our knowledge, we are the first who present a variable reordering technique that is specifically tailored for family models.

Outline. In Section 2, we provide foundations on abstract programs that formalize our models, revisit BDDs and variable reordering as well as PRISM programs. The generic algorithm of iterative variable reordering is subject of Section 3. In Section 4 we briefly recall redundancy systems and the aircraft velocity control loop model we introduced in [15] which we will use in Section 5 as example for evaluating our implementation. We close the paper with some concluding remarks in Section 6.

2 Foundations

For a given set X we denote by $\wp(X)$ its power set, i.e., the set of all subsets of X. We call a total order $(D, <_D)$ on a finite set D an *ordered domain*. For a finite set of variables *Var* we assign to each variable an ordered domain from a set $\mathscr{D}$ by a function $d\colon \mathit{Var} \to \mathscr{D}$. A variable evaluation over *Var* w.r.t. d is a function η that assigns to each variable $v \in \mathit{Var}$ a value $\eta(v) \in d(v)$. We denote by $\mathit{Eval}_d(\mathit{Var})$ the set of variable evaluations over *Var* w.r.t. d. In the following, we assume the function d to be fixed for a given set of variables *Var* and briefly write, e.g., $\mathit{Eval}(\mathit{Var})$ for the set of evaluations over *Var*.

We consider *programs* $\mathsf{Prog} = (\mathit{Var}, C, \iota)$ where *Var* is the finite set of variables on which the program is defined, C is a set of commands, and ι is an expression specifying a set of initial variable evaluations that we denote by $[\![\iota]\!] \subseteq \mathit{Eval}(\mathit{Var})$. Sometimes we use sloppy notations and replace ι by a single evaluation $\eta \in \mathit{Eval}(\mathit{Var})$ to represent the initial variable evaluation explicitly. Intuitively, a program defines a state-based semantics where states are variable evaluations in $\mathit{Eval}(\mathit{Var})$ and commands specify how to switch to from one variable evaluation to another. Every program specifies a family of systems through the initial variable evaluations, i.e., we interpret each initial variable evaluation as entry point for a system variant. This kind of interpretation is backed by well-known concepts from feature-oriented and family-based system modeling and analysis [11, 33, 14, 8].

We intentionally defined programs in a rather abstract way, as the concepts presented in this paper are applicable to a wide range of system families described by various specification formalisms. Thus, we also leave out to specify a concrete semantics of programs.

2.1 Symbolic Representations Through Binary Decision Diagrams

Binary decision diagrams (BDDs) [23, 2] were mainly developed as a universal data structure for Boolean functions, i.e., a BDD over a set of Boolean variables *Var* describes a function $f\colon \wp(\mathit{Var}) \to \{\mathsf{true}, \mathsf{false}\}$ that maps a set of variables assumed to be true to either true or false. Technically, a BDD over *Var* is a graph that is rooted, directed, and acyclic and where each node is either terminal and represents a result, i.e., either true or false, or it is a decision node. Each decision node is labeled by a Boolean variable from *Var* and has exactly two successor nodes: the 0-successor node, which stands for assigning false to the respective Boolean variable, and the 1-successor node, standing for a true assignment, respectively. We refer to the number of nodes in a BDD $\mathscr{B}$ as *size*, denoted by $\mathit{size}(\mathscr{B})$. In the context of model checking, BDDs are used to represent the characteristic function of the transition relation for state-based models.

Reduced Ordered BDDs. An *ordered BDD* [6] is a BDD over a total order $(\mathit{Var}, \prec)$ on Boolean variables where along all paths from the root to a terminal node the order $\prec$ is respected. The graph structure of an ordered BDD for a Boolean function f arises from a binary decision tree for f using the given variable ordering. By merging isomorphic subgraphs, eliminating terminal nodes with the same value, and removing any node whose two successors are isomorphic we obtain *reduced ordered BDDs*, which we simply abbreviate as *BDDs* in the following. In such BDDs, every two nodes represent different Boolean

functions. Removing redundancies from a decision tree reveals the potential of BDDs for compactly representing Boolean functions. When BDDs are used to represent family models such as programs defined above, behavior shared between family members could lead to concise BDD representations and analysis of the family. For further details on BDDs, we refer to standard textbooks such as [31, 34].

Variable Reordering. The size of a BDD crucially depends on the given variable ordering [6]. In fact, there are Boolean functions that can be represented by BDDs of linear and exponential size depending on the chosen variable order. Reordering algorithms such as sifting [30, 28], can be applied to improve the size BDD for a given Boolean function. In case BDDs are used as a representation for state-based models, reordering techniques are applied when the model is constructed. However, when the constructed model is used for BDD-based verification, dynamic reordering during the verification process can also be applied, possibly avoiding bad variable orders in those BDDs that represent intermediate results. A good heuristic for variable orderings is to first decide variables which are "most-influential", i.e., changed at the beginning of an execution of the modeled system and influence the systems behavior significantly [24].

BDDs over Program Variables. In our setting, we use BDDs as representation for state-based models of programs, which are defined on possibly non-Boolean variables. However, every element of a finite domain can be represented by a bit vector. Given a finite domain D, we enumerate the domain by a function $\#\colon D \to \{0,\ldots,|D|-1\}$. Then for an element $x \in D$ we use the bit representation of $\#(x)$ with length $k = \lceil \log(|D|-1) \rceil$, i.e., introduce k fresh Boolean variables to represent elements of the domain D. In this way, the interpretation of BDDs on Boolean variables can easily be lifted to BDDs over variables *Var* of programs. We hence might refer to variables and functions in this general meaning rather than referring to Boolean variables and Boolean functions only. When reordering program variables, we assume that the bit-wise order of the variables is maintained (i.e., bits are not *exploded*, see [21]).

2.2 PRISM Programs

We introduced programs in a generic fashion as tuples $\mathsf{Prog} = (\mathit{Var}, C, \iota)$. For our case study, we rely on an instance of such programs given by the input language of the probabilistic model checker PRISM [22]. In PRISM, variables of *Var* are assigned to bounded intervals of integers with the standard total order on integers as ordered domain. Possible transitions between states, i.e., variable evaluations, are given by *guarded commands* [13] collected in C. A command has the form

$$\texttt{[action] guard} \;\rightarrow\; \mathtt{p}_1\texttt{:update}_1 \;\texttt{+}\; \mathtt{p}_2\texttt{:update}_2 \;\texttt{+}\; \ldots \;\texttt{+}\; \mathtt{p}_n\texttt{:update}_n \;.$$

Here, `guard` is a Boolean expression over arithmetic constraints on variable evaluations, e.g., $(\mathtt{x} = 1) \wedge (\mathtt{y} \leq 5)$ for $\mathtt{x}, \mathtt{y} \in \mathit{Var}$ is fulfilled in every state where variable `x` has value `1` and `y` has a value from the domain $d(\mathtt{y})$ of `y` that is smaller or equal than `5`. In case the guard evaluates to `true` in some state, the command is enabled, leading to a transition into a successor state by updating variables according to *updates*. An update describes how variables change depending on the current variable evaluation, e.g., `x'=1+y` changes the value of `x` to the increment of the value of `y`. Each update $\mathtt{update}_i$ is chosen with probability $\mathtt{p}_i$ for $i \in \{1,\ldots,n\}$. That is, in every state fulfilling `guard` the evaluations of the expressions $\mathtt{p}_1$, $\mathtt{p}_2$, …, $\mathtt{p}_n$ constitute a probability distribution, i.e., must sum up to 1. PRISM programs describe stochastic state-based models such as Markov decision processes (MDPs) or discrete Markov chains (DTMCs). For further details on such models, we refer to standard textbooks such as [29, 5].

Family Models in PRISM. In PRISM, initial variable evaluations describe the entry points for system variants in families, specified through a guard encapsulated in an `init` block. Guards are, as for commands, Boolean expressions over arithmetic constraints on variable evaluations. All variable evaluations that fulfill the guard in the `init` block will be considered as initial states. For instance, when considering variables $Var = \{\mathtt{x},\mathtt{y}\}$ with domains $d(\mathtt{x}) = \{0,1\}$ and $d(\mathtt{y}) = \{4,5,6\}$, the following `init` block specifies two initial states η_1 and η_2 where $\eta_1(\mathtt{x}) = \eta_2(\mathtt{x}) = 1$, $\eta_1(\mathtt{y}) = 4$, and $\eta_2(\mathtt{y}) = 5$:

$$\texttt{init (x=1)} \ \wedge \ (\mathtt{y} \leq 5) \ \ \texttt{endinit} \ .$$

The described family model would have two members, one system starting in η_1 and one starting in η_2. In this paper, we also consider `init` blocks in conjunctive normal form (CNF) over single variable evaluations. The above block could, e.g., be represented in CNF by

$$\texttt{init (x=1)} \ \wedge \ \texttt{(y=4)} \vee \texttt{(y=5)} \ \ \texttt{endinit} \ .$$

Variable Ordering in PRISM. For its symbolic engines, PRISM uses algorithms that rely on Multi-terminal binary decision diagrams [10, 18, 3] (MTBDDs). MTBDDs extend BDDs by allowing for terminal nodes with real values rather than true and false. In PRISM they are used as data structure for representing the transition probability matrix of stochastic models. To this end, the performance and memory consumption of an analysis by PRISM crucially depends on the variable order inside the MTBDD that represents the model. In PRISM, the variable order coincides with the order of variable declarations in the program, enabling reordering methods directly on the source-code level[1]. For family models in PRISM, [14] showed that a suitable variable order could significantly improve the analysis speed using a hand-crafted approach for tuning variable orders. In [21] automated reordering techniques for PRISM models have been presented, also showing that the size of the symbolic representation of the feature-oriented family model of [14] could be reduced even further and yield analysis speed ups.

3 Iterative Variable Reordering

For this section, let us fix a program $\mathsf{Prog} = (Var, C, \iota)$ on which we explain our approach of *iterative variable reordering*. We assume $[\![\mathsf{Prog}]\!]$ to denote the state-based semantics of the program Prog. Furthermore, we assume that there is a uniquely defined symbolic representation $[\![\mathsf{Prog}]\!]_\pi$ of $[\![\mathsf{Prog}]\!]$ that depends on a *variable order* $\pi = (Var, \prec)$, i.e., a total order on the set of variables *Var* the program Prog is defined on. For instance, when Prog is a PRISM program, $[\![\mathsf{Prog}]\!]_\pi$ could be the MTBDD representation of the program's semantics in terms of an MDP or DTMC w.r.t. the variable order π.

Recall that the purpose of our reordering approach is to enable the construction of a symbolic representation $[\![\mathsf{Prog}]\!]_\pi$ of system families that are not completely constructible ad-hoc from a given π. In this paper, "constructability" is understood with respect to given memory and time constraints. The presented method can be only effective in case there is a variable order ρ such that the complete family model $[\![\mathsf{Prog}]\!]_\rho$ is constructible. Note that even though, our presented method does not ensure to yield a constructible family model representation $[\![\mathsf{Prog}]\!]_\rho$. This is mainly due to intermediate model representations that might be not constructible.

As prerequisites to our presented method, we rely on the following assumptions:

[1] Recall that we considered commands to be unordered in general programs. We hence always provide a variable order explicitly for PRISM programs.

Algorithm 1: Iterative variable reordering

input : Program Prog $= (Var, C, \iota)$; variable order $\pi = (Var, \prec)$ and initial variable evaluation $\eta \in [\![\iota]\!]$ as in assumption (1)
output: Variable order $\rho = (Var, \prec')$

1 Construct $\mathscr{B} = [\![(Var, C, \eta)]\!]_\pi$
2 $\rho := reorder(\mathscr{B})$
3 **forall** $v \in Var$ **do**
4 $\quad E(v) := \{\eta(v)\}$
5 $\quad G(v) := \{\eta'(v) : \eta' \in [\![\iota]\!]\}$
6 **while** $E \neq G$ **do**
7 $\quad$ Pick the π-minimal $v \in Var$ where $E(v) \neq G(v)$
8 $\quad E(v) := E(v) \cup \{\min_{d(v)}(G(v) \setminus E(v))\}$
9 $\quad$ Construct $\mathscr{B} = [\![(Var, C, \iota \wedge cnf(E))]\!]_\rho$
10 $\quad \rho := reorder(\mathscr{B})$
11 **return** ρ

(1) We have given an initial variable evaluation $\eta \in [\![\iota]\!]$ and a variable ordering π where $[\![(Var, C, \eta)]\!]_\pi$ is constructible.

(2) We have a reordering method $reorder(\cdot)$ that accepts a symbolic representation $[\![\mathscr{P}]\!]_\alpha$ of a program $\mathscr{P}$ and a variable order α and returns a variable order β such that $size\big([\![\mathscr{P}]\!]_\alpha\big) \geq size\big([\![\mathscr{P}]\!]_\beta\big)$.

The above assumptions intuitively provide a base for the iteration (1) and a method to iterate (2).

We specify iterative variable reordering as pseudo code in Algorithm 1, where for some evaluation domain $E\colon Var \to \wp(\mathscr{D})$ we define its representation as conjunctive normal form (CNF) by

$$cnf(E) \;=\; \bigwedge\nolimits_{v \in Var} \bigvee\nolimits_{x \in E(v)} (v = x) \;\; .$$

Choosing the CNF representing E is a design decision that could be replaced by any other equivalent representation by Boolean expressions over arithmetic constraints on variables and domains. The algorithm is instantiated with the initial variable order π and initial evaluation η from which we know by assumption (1) that the symbolic representation $[\![(Var, C, \eta)]\!]_\pi$ is constructible. The actual construction is performed in line 1, directly followed by a reordering step to initialize the variable order ρ (line 2). Then, the evaluation domain E based on η is specified in line 4, while in line 5 the *goal* evaluation domain G provides the sets of all values a variable can evaluate to in an initial variable evaluation of ι. The algorithm then iteratively adds values for variables v to the evaluation domain E that are used in an initial variable evaluation of ι until the goal evaluation domain G is reached (line 6-10). This is done by always selecting v as the variable that is minimal w.r.t. ρ. After constructing $[\![(Var, C, \iota \wedge cnf(E))]\!]_\rho$, i.e., the symbolic representation of the program Prog with the additional option evaluating v, the symbolic representation is reordered using $reorder(\cdot)$ and ρ is updated to a new variable order β for which

$$size\big([\![(Var, C, \iota \wedge cnf(E))]\!]_\rho\big) \geq size\big([\![(Var, C, \iota \wedge cnf(E))]\!]_\beta\big)$$

due to assumption (2). The algorithm terminates always in case the intermediate symbolic representations could be constructed (see line 9) as E is strictly increasing, G is finite, and in each iteration step and for all $v \in Var$ we have $E(v) \subseteq G(v)$ as an invariant.

3.1 Optimization Heuristics

The basic iterative variable reordering algorithm provided by Algorithm 1 can be extended by various optimization heuristics. Such heuristics could enable or fasten the process of finding a suitable variable reorder ρ that allows for the construction of $[\![\mathsf{Prog}]\!]_\rho$. We focus here on the following variations:

Step size: For a given integer $n \in \mathbb{N}$ called *step size*, we repeat line 7 and 8 n times, i.e., increase the evaluation domain E by n elements.

ρ-min/max selection: In line 7, we choose the ρ-minimal or ρ-maximal $v \in \mathit{Var}$ instead the π-minimal.

The ratio behind the step-size heuristics is that programs with many variables might waste a lot of time with reordering while adding more family members in one iteration while they could also provide a suitable variable order where the symbolic representation is already constructible. The two latter heuristics provide a form of adaptivity to the selection of variables. For instance, when π already has most influential variables at the beginning of the order, adding family members for influential variables could lead to symbolic representations not constructible anymore or, the other way around, less influential variables in front of the variable order could only lead to small changes in the reordered order. Adding variants to variables usually increases their influence on the operational behavior. It is likely that then, this variable appears more in front of the reorder, following the rule of "most influential" variables [24] in case for BDDs as symbolic representation. Choosing ρ-maximal variables adds variants to those variables that are not as influential, somehow balancing their effect on the size of the symbolic representation.

4 Redundancy System Models

To support our implementation and evaluation of the iterative variable reordering, this section is devoted to a brief summary of the approach of [15] and the models investigated in this paper.

Redundancy systems are systems where components might be replicated to increase fault tolerance of the overall system. For instance, some component could be protected by triplicating the component and process the replicas' results through a majority voting mechanism towards a single output, implementing the well-known *triple modular redundancy (TMR)* principle. Analyzing each instance of the redundancy system easily becomes infeasible as the number of possible protection combinations grows exponentially in the number of protectable components. In [15] we proposed to use family-based approaches for the analysis of redundancy systems, motivated by the fact that introducing redundancies in components also introduces common behaviors. Such common behaviors are likely to be concisely representable using symbolic techniques. We illustrated the benefits of our approach by a reliability analysis of redundancy systems modeled in SIMULINK [32], a widely used framework for model-based system design. A SIMULINK design comprises blocks that describe the behavior of the system, connected by arrows standing for control and data flow. An example of a SIMULINK design is provided in Figure 1, borrowed from the SIMULINK example set [1].

4.1 SIMULINK Models with Redundancy

To introduce redundancy mechanisms into SIMULINK models, we consider syntactic transformation rules that describe how to obtain protected blocks from non-protected ones. In this paper, we consider the following redundancy mechanisms [15]:

(comparison) The block is duplicated and both outputs are compared. In case their output differs a dedicated failure state is reached. Otherwise, the output is the one of both blocks.

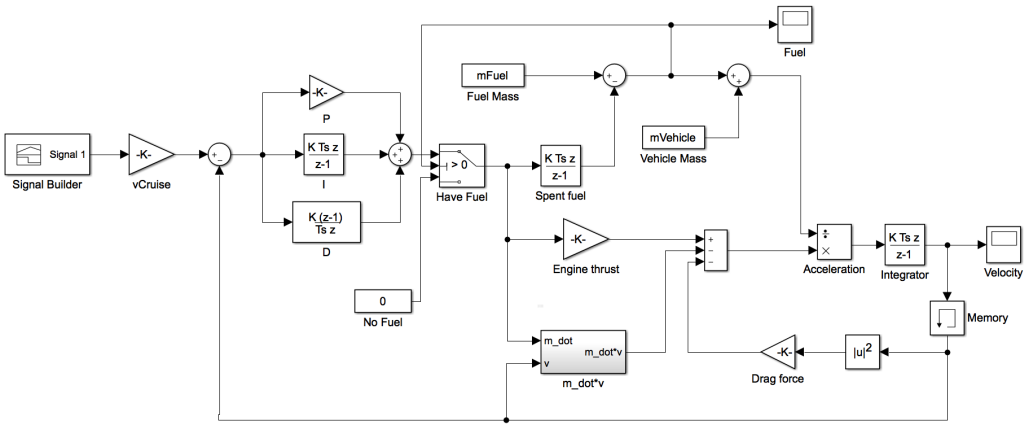

Figure 1: The SIMULINK aircraft velocity control loop (VCL) model

(voting) Following the principle of triple modular redundancy principle, the block is triplicated and the output is based on a majority decision.

For SIMULINK models with protection mechanisms, we introduced a discrete Markov chain (DTMC) family semantics [15] that can be automatically generated using OPENERRORPRO [26]. The general workflow of our approach towards a DTMC family out from SIMULINK models is depicted in Figure 2.

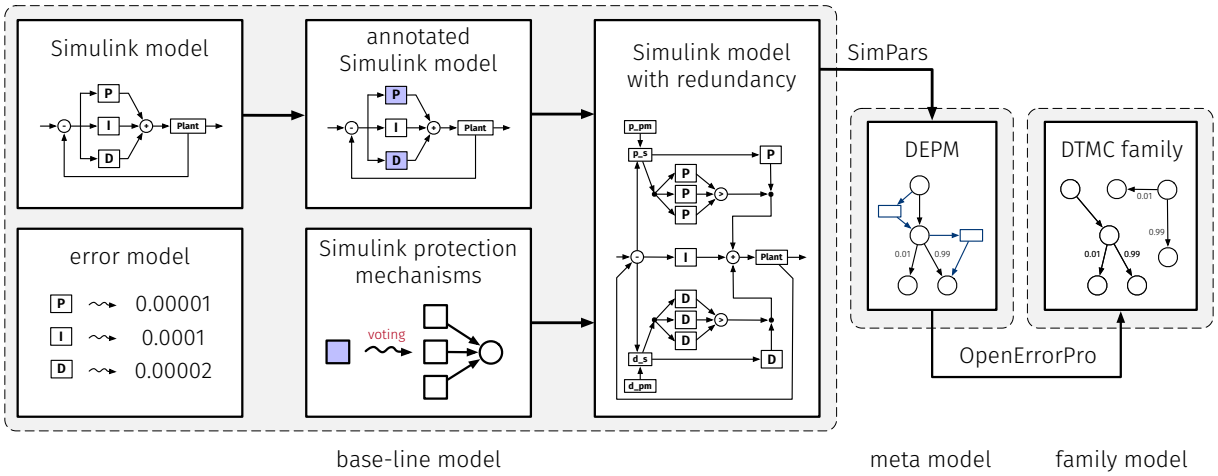

Figure 2: Schema of the approach to obtain DTMC families from annotated SIMULINK models [15]

First, blocks in the SIMULINK model are annotated with the types of protections that should be considered, e.g., comparison or voting. In Figure 2, we annotated voting protections for the P and D block (indicated by shaded blocks). Then, the annotated model is transformed to a SIMULINK model that includes redundancies by replacing every block where protection should be taken into account with a switch block that depends on a fresh switch variable, followed by the actual blocks for the redundancy mechanisms. The role of the switch variable is to select the redundancy mechanism, directing the control flow, e.g., to the triplicated blocks in case of the voting mechanism. The SIMULINK model with redundancy stands for a family of models with different protection combinations. An error model furthermore assigns to each SIMULINK block the probability for some fault occurring in this block, introducing stochastics into

the model required for a reliability analysis. In [15] we presented a DTMC family semantics as PRISM family model generated using the tools SIMPARS [27] and OPENERRORPRO [26]. For this, we employed a meta-model representation as Dual-graph Error Propagation Model (DEPM) [26] (see also Figure 2).

4.2 The Velocity Control Loop Model

We illustrate our approach of iterative variable reordering described in Section 3 on a PRISM program that stems from an aircraft velocity control loop (VCL) with redundancies described in [15]. Our model is a simplified version of the aircraft model borrowed from the SIMULINK example set [1] that itself is based on a long-haul passenger aircraft flying at cruising altitude and speed, adjusting the fuel flow rate to control the aircraft velocity. Figure 1 shows the SIMULINK model where all blocks are amendable for protection mechanisms. In [15] we applied three different kinds of redundancy mechanisms on eight of these blocks, leading a PRISM DTMC family model with $4^8 = 65\,536$ family members, each member standing for one protection combination. The PRISM family model generated by OPENERRORPRO [26] could not immediately be constructed by PRISM's symbolic engine. We hence had to apply two (hand-crafted) optimizations to enable a reliability analysis of the VCL redundancy system: *reset value optimization* [19, 16] and *iterative variable reordering*. While reset value optimization speeded up the analysis time, iterative variable reordering was the technique that enabled the construction of the model.

In this paper, we use a variant of the VCL model of [15] as starting point where neither reset value optimization nor hand-crafted iterative variable reordering has been applied to and where 13 blocks could be protected with either voting or comparison mechanisms.

5 Implementation and Evaluation

In this section, we report on an implementation of the iterative variable reordering for family models expressed in the input language of PRISM that follows Algorithm 1 and includes the optimization heuristics detailed in Section 3.

5.1 Implementation

Our implementation is done in PYTHON and takes as input a PRISM family model, a parameter specifying the variable selection heuristics (π-minimal, ρ-minimal, or ρ-maximal), and the step size of the iteration. The advantage of the heuristics is that the construction of the model can be parallelized trying different heuristics on multi-core systems to eventually obtain a suitable variable order that renders the whole system family to be constructible even when some heuristics are slow or not successful.

Remark. PRISM provides a faster model construction for non-family models, i.e., models without an `init` block, than for family models as evaluating the `init`-block expression relies on SAT solving, an NP-complete problem [20]. Hence, we implemented the initial construction step in line 1 of Algorithm 1 for a model representation without `init` block. For later constructions (line 9 of Algorithm 1) generated an `init` block for $\iota \wedge \mathit{cnf}(E)$ as indicated.

5.2 Evaluation

We evaluated our implementation on the PRISM DTMC family generated from a SIMULINK VCL redundancy model (see Section 4) where 13 blocks are marked as protectable by comparison or voting

Table 1: Performance of the heuristics within 20 minutes

variable selection	step size	iteration	combinations	states	nodes
π-minimal	1	22	177 147	$8.67{\cdot}10^{11}$	97 386
	2	12	531 441	$6.19{\cdot}10^{12}$	130 321
	3	7	118 098	$6.29{\cdot}10^{11}$	101 766
	4	5	59 049	$1.25{\cdot}10^{11}$	88 054
ρ-minimal	1	22	177 147	$8.67{\cdot}10^{11}$	97 386
	2	12	531 441	$7.37{\cdot}10^{12}$	153 189
	3	7	118 098	$8.17{\cdot}10^{11}$	138 287
	4	5	59 049	$1.19{\cdot}10^{11}$	163 882
ρ-maximal	1	12	4 096	$9.47{\cdot}10^{10}$	126 338
	2	10	59 049	$6.83{\cdot}10^{11}$	139 991
	3	5	7 776	$1.25{\cdot}10^{11}$	251 944
	4	5	59 049	$9.71{\cdot}10^{11}$	225 711

mechanisms. Thus, we applied iterative variable reordering on a family model with $3^{13} = 1\,594\,323$ protection variants, more than 24 times the number of protections considered in [15]. Note that we did not apply reset value optimization [19, 16] as in [15]. All experiments are carried out on a Linux server system[2] with a memory bound of 30 GB of RAM, running the PRISM version presented in [21].

Heuristic Comparison. First, we evaluate the different heuristics implemented. Table 1 shows the situation of the implementation as snapshot after 20 minutes. As expected, the higher the step size, the less iterations could be performed by the iterative variable reordering. On our model, π-minimal and ρ-minimal selection heuristics perform best with step size 2. For both heuristics, the complete family model with its $3^{13} = 1\,594\,323$ protection variants was constructed within half an hour. It is not surprising that step size 2 is a good choice as we considered two redundancy mechanisms for each block, i.e., in each step all redundancies for each block contribute to the family model at once.

Iteration Statistics. For step size of 2 and using π-minimal variable selection, we detail the statistics of every iteration until the whole family model has been constructed in Table 2. We shaded the iteration that was reached after 45 minutes of computation time (cf. the shaded row in Table 1). In the column `combinations`, the family size of the intermediate family is listed. Column `#nodes` shows the number of MTBDD-nodes used to represent that family model before applying reordering (cf. line 10 of Algorithm 1) and thereafter. The last row indicates the time required to construct the model (cf. line 9 of Algorithm 1) and to perform reordering (cf. line 10 of Algorithm 1).

Alternative Approaches. Clearly, there are alternative methods that could be used to enable an analysis of large-scale system families when the family model cannot be constructed in an ad-hoc fashion. The first is to switch from an all-in-one approach to a one-by-one approach, i.e., analyzing every family member in isolation. However, even the smallest family member (see Table 2, first iteration) required more than 7 seconds for the construction of the model. One could hence expect at least $3^{13} \cdot 7$ seconds of

[2] 2 × Intel Xeon E5-2680 (Octa Core, Sandy Bridge) running at 2.70 GHz with 384 GB of RAM; Turbo Boost and Hyper Threading disabled; Debian GNU/Linux 9.1.

Table 2: Step-wise statistics of ρ-maximal

			#nodes		time [s]	
iteration	combinations	states	before	after	model	reorder
0	1	113 891	24 816	23 359	7.40	2.67
1	3	347 401	24 576	23 329	7.81	2.87
2	9	$1.06 \cdot 10^6$	24 545	23 319	7.36	2.93
3	27	$3.93 \cdot 10^6$	48 413	33 647	8.48	4.76
4	81	$1.19 \cdot 10^7$	35 185	33 664	8.30	4.55
5	243	$4.05 \cdot 10^7$	52 497	40 431	9.18	6.12
6	729	$1.54 \cdot 10^8$	69 535	48 888	13.76	8.07
7	2 187	$6.65 \cdot 10^8$	94 022	55 835	23.57	11.00
8	6 561	$3.00 \cdot 10^9$	73 242	62 636	28.55	9.75
9	19 683	$1.85 \cdot 10^{11}$	132 279	76 518	73.55	18.14
10	59 049	$1.25 \cdot 10^{11}$	95 740	79 420	76.40	14.63
11	177 147	$8.67 \cdot 10^{12}$	120 565	97 386	155.03	19.08
12	531 441	$6.19 \cdot 10^{12}$	205 962	130 321	360.38	33.37
13	1 594 323	$4.87 \cdot 10^{13}$	256 662	160 798	723.02	41.08
$\sum$					1 502.80	179.00

construction time for all family members, which would correspond to more than one year of construction time on our test system. Second, one could apply dynamic reordering techniques as implemented in the probabilistic model checker STORM [12]. With this method, the variable order on the MTBDD is dynamically changed during the model-construction process. However, even with smaller families, e.g., the VCL family model from [15] where only 8 protectable blocks were considered, model construction did not finish within days. We hence conclude that, at least for the VCL family models we generated, iterative variable reordering is the only technique we considered and yielded a successful model construction.

6 Concluding Remarks

In this paper we presented iterative variable reordering as a generic method to cope with the problem of constructing family models of huge system families. While our implementation supports models specified in the input language of the probabilistic model checker PRISM, the formally defined algorithm is applicable to a wide range of analysis tools that use variable-order sensitive symbolic representations for their models and support some kind of reordering mechanism. For instance, the verification of large feature-oriented system families according to [11] those verification is based on the BDD engine of NUSMV also could benefit from our approach. By supporting system families given as PRISM programs, our implementation can be directly applied on feature-oriented systems specified in PROFEAT [8]. Our methods could be also viable for family-based parameter synthesis [4]. Experiments on such system families are left for future work.

Acknowledgements. We would like to thank Joachim Klein who extended STORM with support for family models, used to validate that dynamic reordering techniques are not viable for the VCL example.

References

[1] *Verify Model Using Simulink Control Design and Simulink Verification Blocks (accessed 19/02/2019)*. https://mathworks.com/help/slcontrol/ug/model-verification-using-simulink-control-design-and-simulink-verification-blocks-.html.

[2] S. B. Akers (1978): *Binary Decision Diagrams*. IEEE Transactions Computers 27(6), pp. 509–516, doi:10.1109/TC.1978.1675141.

[3] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo & F. Somenzi (1997): *Algebraic Decision Diagrams and Their Applications*. Formal Methods in System Design 10(2/3), pp. 171–206, doi:10.1023/A:1008699807402.

[4] C. Baier & C. Dubslaff (2018): *From Verification to Synthesis under Cost-Utility Constraints*. ACM SIGLOG News 5(4), pp. 26–46, doi:10.1145/3292048.3292052.

[5] C. Baier & J.-P. Katoen (2008): *Principles of Model Checking*. MIT Press.

[6] R. E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers 35, pp. 677–691, doi:10.1109/TC.1986.1676819.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill & L. J. Hwang (1992): *Symbolic Model Checking: 10^{20} States and Beyond*. Inform. and Comp. 98(2), pp. 142–170, doi:10.1016/0890-5401(92)90017-A.

[8] P. Chrszon, C. Dubslaff, S. Klüppelholz & C. Baier (2018): *ProFeat: feature-oriented engineering for family-based probabilistic model checking*. Formal Aspects of Computing 30(1), pp. 45–75, doi:10.1007/s00165-017-0432-4.

[9] E. Clarke, O. Grumberg & D. Peled (2000): *Model Checking*. MIT Press.

[10] E. M. Clarke, M. Fujita, P. C. McGeers, K. L. McMillan, J. C.-Y. Yang & X.-J. Zhao (1993): *Multi-terminal binary decision diagrams: An efficient data structure for matrix representation*. In: Proc. International Workshop on Logic & Synthesis, doi:10.1023/A:1008647823331.

[11] A. Classen, P. Heymans, P.-Y. Schobbens & A. Legay (2011): *Symbolic model checking of software product lines*. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, pp. 321–330, doi:10.1145/1985793.1985838.

[12] C. Dehnert, S. Junges, J.-P. Katoen & M. Volk (2017): *A Storm is Coming: A Modern Probabilistic Model Checker*. In: 29th Int. Conf. on Computer Aided Verification (CAV), LNCS 10427, Springer, pp. 592–600, doi:10.1007/978-3-319-63390-9_31.

[13] E. W. Dijkstra (1975): *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. Commun. ACM 18(8), pp. 453–457, doi:10.1145/360933.360975.

[14] C. Dubslaff, C. Baier & S. Klüppelholz (2015): *Probabilistic Model Checking for Feature-Oriented Systems*. Transactions on Aspect-Oriented Software Development 12, pp. 180–220, doi:10.1007/978-3-662-46734-3_5.

[15] C. Dubslaff, K. Ding, A. Morozov, C. Baier & K. Janschek (2019): *Breaking the Limits of Redundancy Systems Analysis*. In: Proceedings of the 29th European Safety and Reliability Conference, Research Publishing, Singapore, pp. 2317–2325, doi:10.3850/978-981-11-2724-3_0618-cd.

[16] C. Dubslaff, A. Morozov, C. Baier & K. Janschek (2020): *Reduction Methods on Probabilistic Control-flow Programs for Reliability Analysis*. CoRR abs/2004.06637. Available at https://arxiv.org/abs/2004.06637.

[17] E. Felt, G. York, R. Brayton & A. Sangiovanni-Vincentelli (1993): *Dynamic variable reordering for BDD minimization*. In: Proceedings of EURO-DAC 93 and EURO-VHDL 93- European Design Automation Conference, pp. 130–135, doi:10.1109/EURDAC.1993.410627.

[18] M. Fujita, P.C. McGeer & J.C.-Y. Yang (1997): *Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation*. Formal Methods in System Design 10(2-3), pp. 149–169, doi:10.1023/A:1008647823331.

[19] H. Garavel & W. Serwe (2006): *State space reduction for process algebra specifications*. Theoretical Computer Science 351(2), pp. 131 – 145, doi:10.1016/j.tcs.2005.09.064. Algebraic Methodology and Software Technology.

[20] M. R. Garey & D. S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Company.

[21] J. Klein, C. Baier, P. Chrszon, M. Daum, C. Dubslaff, S. Klüppelholz, S. Märcker & D. Müller (2018): *Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic Büchi automata*. Intern. Journal on Software Tools for Technology Transfer 20(2), pp. 179–194, doi:10.1007/s10009-017-0456-3.

[22] M. Kwiatkowska, G. Norman & D. Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: Proc. 23rd International Conference on Computer Aided Verification (CAV'11), LNCS 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[23] C. Y. Lee (1959): *Representation of Switching Circuits by Binary-Decision Programs*. Bell System Technical Journal 38(4), pp. 985–999, doi:10.1002/j.1538-7305.1959.tb01585.x.

[24] S. Malik, A.R. Wang, R.K. Brayton & A. Sangiovanni-Vincentelli (1988): *Logic verification using binary decision diagrams in a logic synthesis environment*. In: Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on, pp. 6–9, doi:10.1109/ICCAD.1988.122451.

[25] K. L. McMillan (1993): *Symbolic Model Checking*. Kluwer, doi:10.1007/978-1-4615-3190-6.

[26] A. Morozov, K. Ding, M. Steurer & K. Janschek (2019): *OpenErrorPro: A New Tool for Stochastic Model-Based Reliability and Resilience Analysis*. In: 30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019, pp. 303–312, doi:10.1109/ISSRE.2019.00038.

[27] A. Morozov, K. Janschek, T. Krüger & A. Schiele (2016): *Stochastic Error Propagation Analysis of Model-Driven Space Robotic Software Implemented in Simulink*. In: Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering, MORSE '16, Association for Computing Machinery, New York, NY, USA, pp. 24–31, doi:10.1145/3022099.3022103.

[28] S. Panda & F. Somenzi (1995): *Who Are the Variables in Your Neighborhood*. In: Proc. Computer-Aided Design (ICCAD'95), IEEE, pp. 74–77, doi:10.5555/224841.224862.

[29] M. L. Puterman (1994): *Markov Decision Processes*. Wiley, doi:10.1002/9780470316887.

[30] R. Rudell (1993): *Dynamic variable ordering for ordered binary decision diagrams*. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)., pp. 42–47, doi:10.1109/ICCAD.1993.580029.

[31] F. Somenzi (1999): *Binary Decision Diagrams*. In: Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences, IOS Press, pp. 303–366.

[32] The MathWorks Inc. (2018): *MATLAB and Statistics Toolbox Release 2018b*. Natick, Massachusetts, United States.

[33] T. Thüm, S. Apel, C. Kästner, I. Schaefer & G. Saake (2014): *A Classification and Survey of Analysis Strategies for Software Product Lines*. ACM Comput. Surv. 47, pp. 1–45, doi:10.1145/2580950.

[34] I. Wegener (2000): *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Monographs on Discrete Mathematics and Applications. SIAM, doi:10.1137/1.9780898719789.