Chapter 7

# Model Checking with Sentential Decision Diagrams

We demonstrate the viability of symbolic model checking using Sentential Decision Diagrams (SDD), in lieu of the more common Binary Decision Diagram (BDD). The SDD data structure can be up to exponentially more succinct than BDDs, using a generalized notion of variable order called a variable tree ("vtree"). We also contribute to the practice of SDDs, giving a novel heuristic for constructing a vtree that minimizes SDD size in the context of model checking, and identifying which SDD operations form a performance bottleneck.

Experiments on 707 benchmarks, written in various specification languages, show that SDDs often use an order of magnitude less memory than BDDs, at the expense of a smaller slowdown in runtime performance.

This chapter contributes to Research Question 3:

> **Research question 3.** *Which classical decision diagrams might be effective for the analysis of quantum algorithms, if they were suitably adapted?*

In this case: can we draw upon the ideas behind Sentential Decision Diagrams to develop more powerful quantum DDs? To some extent, this has already happened: both (i) Tree Tensor Networks (TTN) and (ii) Weighted Context-Free Language Ordered Binary Decision Diagrams (WCFLOBDD) are tree-like structures which can be considered quantum extensions of SDDs. Interestingly, although the TTN predates SDDs, a straightforward "quantum SDD" has not yet been formulated or implemented. If this chapter offers any lessons to learn for TNNs and WCFLOBDDs, then perhaps it is that dynamic variable reordering (or, in this case, dynamic *tree reordering*) can help to both significantly speed up the algorithms and compress the data structure.

## 7.1 Introduction

Model checking is an automated way for verifying whether a system satisfies its specification [28]. Binary Decision Diagrams [67] (Binary DDs or BDDs) first revolutionized symbolic model checking [221]. But the representation may explode in size on certain instances [51, 97]. Later SAT and SMT based techniques, such as BMC [51] and IC3 [60], have been shown to scale even better.

DDs have the unique capability to manipulate (Boolean) functions. Due to this property, they can be used to encode a system in a bottom-up fashion [322]. In model checking, for example, this can be exploited to 'learn' transition relations on-the-fly [178], to soundly [207] implement the semantics of the underlying system. This fundamental difference from SAT also makes DDs valuable in various other applications [44, 243, 275].

To improve the performance of DDs, various more concise versions have been proposed [26, 68, 111, 229, 235, 245, 328]. One such data structure is the Sentential Decision Diagram (SDD) [96], recently proposed by Darwiche. SDDs subsume BDDs in the sense that every BDD is an SDD, and SDDs support the same functionality for function manipulation. Specifically, SDDs support the operations of conjunction, disjunction, negation operations, and existential quantification, making them suitable for the role of data structure in symbolic model checking. Since every BDD is an SDD, they need not use more memory than BDDs, and if an SDD's *variable tree* (vtree) is chosen well, then an SDD has the potential to use exponentially less memory than a BDD [58]. While SDDs have already shown potential for CAD applications [83], they have not yet been tried for model checking. And because of the relevance of DDs in many other applications, an evaluation of the performance of SDDs is also of interest in its own right.

The biggest obstacle to overcome, therefore, is to find a good vtree for the SDD. A vtree is to an SDD as a variable order is to a BDD. In BDD-based symbolic model checking, the BDD's variable order determines the size of the BDD in memory, and the choice of variable order may determine whether or not the verifier runs out of memory.

We propose in Section 7.3 three heuristics for constructing vtrees. First, we propose that the vtree is constructed in two phases. In phase 1, we build an abstract variable tree which minimizes the distance between dependent variables in the system. In phase 2, the leaves that represent integer variables are "folded out", and are replaced by "augmented right-linear" vtrees on 32 variables. These heuristics *statically* construct a vtree, before model checking, so that the vtree is preserved throughout the analysis.

We implemented our approach in the language-independent model checker model checker LTSmin [178][*], using the SDD package provided by the Automated Reasoning Group at UCLA [25]. To evaluate the method, we test the performance of SDDs on 707 benchmarks from a diverse set of inputs languages: DVE [31] (293), Petri nets [187] (357), Promela [159] (57). We test several configurations, corresponding to using all heuristics above, or only a subset, with and without dynamic vtree search (i.e., modifying the vtree during model checking). These languages represent very different domains. By obtaining good results across all domains, we gain confidence that the advantage of SDDs over BDDs is robust.

Experiments show that SDDs often use an order of magnitude less memory than BDDs on the same problem, primarily on the more difficult instances. Among instances that were solved by both SDDs and BDDs, 75% were solved with a smaller SDD, and among those instances, the SDD was on average 7.9 times smaller than the BDD. On the other hand, the SDD approach is up to an order of magnitude slower than the BDD; 82% of the instances were solved faster with BDDs, with an average speedup of 2.7. We investigate the causes of this slowdown in SDDs, but also demonstrate with larger benchmarks, that memory/time trade-off is worthwhile as we can solve larger instances with SDDs. Our analysis shows that the set intersection and union operations in SDDs are relatively fast, whereas existential quantification can be a performance bottleneck. This is despite the recent discovery that set intersection and union have exponential worst-case behaviour in theory [322].

[*]Our implementation's source code is available at https://doi.org/10.5281/zenodo.3940936

## 7.2 Background

### 7.2.1 Sentential Decision Diagrams

A Sentential Decision Diagram (SDD), recently proposed by Darwiche [96], is a data structure which stores a set $K \subseteq \{0,1\}^n$ of same-length bit vectors. In our context, a bit vector represents a state of a system, and variables are represented using multiple bits. Like Binary Decision Diagrams (BDDs) [68], SDDs are a canonical representation for each set and allow, e.g., union and intersection, with other sets. Following is a succinct exposition of the data structure.

Suppose we wish to store the set $K \subseteq \{0,1\}^n$ of length $n$ bit vectors. If $n = 1$, then $K$ is one of four base cases, $K \in \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$. Otherwise, if $n \geq 2$, then we decompose $K$ into a union of simpler sets, as follows. Choose an integer $1 \leq a < n$. Split each string into two parts: the first $a$ bits and the last $n-a$ bits, called the prefix and the postfix.

For a string $p \in \{0,1\}^a$, let $Post(p) \subseteq \{0,1\}^{n-a}$ be the set of postfixes $s \in \{0,1\}^{n-a}$ with which $p$ can end. More precisely:

$$Post(p) = \{s \in \{0,1\}^{n-a} \mid ps \in K\} \tag{7.1}$$

where $ps$ is the concatenation of $p$ with $s$.

Next, define an equivalence relation on the set $\{0,1\}^a$ where $p_1 \sim p_2$ when $Post(p_1) = Post(p_2)$. Suppose that this equivalence relation partitions the set $\{0,1\}^a$ into the $\ell$ sets $P_1, \ldots, P_\ell$. Let $\overline{p_1} \in P_1, \ldots, \overline{p_\ell} \in P_\ell$ be arbitrary representatives from the respective sets. Then we can represent $K$ as

$$K = \bigcup_{i=1}^{\ell} P_i \times Post(\overline{p_i}) \tag{7.2}$$

The sets $P_i, P_j$ are disjoint for $i \neq j$, because these sets form a partition of $\{0,1\}^a$, so the Cartesian products $P_i \times Post(\overline{p_i}), P_j \times Post(\overline{p_j})$ are also disjoint for $i \neq j$. Hence we have decomposed $K$ into a disjoint union of smaller sets.

To build an SDD for $K$, recursively decompose the sets $P_1, \ldots, P_\ell$ and $Post(\overline{p_1}), \ldots, Post(\overline{p_\ell})$, until the base case is reached, where the bit vectors have length 1. To this end, choose two integers $1 \leq a_1 < a$ and $a \leq a_2 < n$ and decompose the

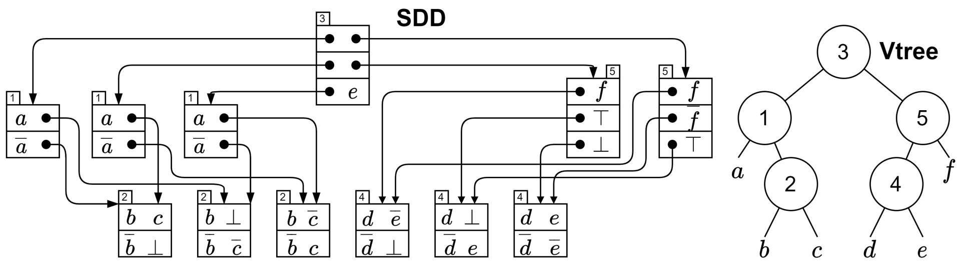

Figure 7.1: An SDD and the vtree it is normalised to. Rectangles are SDD nodes, labelled with the vtree node they are normalized to.

sets $P_i$ with respect to $a_1$ and decompose the sets $Post(\overline{p_i})$ with respect to $a_2$. In the literature, the sets $P_i$ are called the *primes*, the sets $Post(\overline{p_i})$ the *subs*, and the sets $P_i \times Post(\overline{p}_i)$ are called the *elements* of $K$.

**Example 7.1.** Figure 7.1 shows an SDD for a set of length-6 bitstrings, normalized to the vtree on the right. A rectangle represents a set. A box within a rectangle represents an element of that set, with its prime on the left and its sub on the right. The box represents the Cartesian product of its prime and sub. In particular, the SDD nodes' labels indicate which vtree node they are normalized to. A $\top$ represents $\{0,1\}$, and $\bot$ represents the empty set.

**Variable trees.** In the construction of the SDD, the choice of the integer $a$ determines how the top level of the SDD decomposes into tuples. The left and right partitions are then further decomposed, recursively, resulting in a full binary tree, called a *variable tree*, or "vtree" for short. This binary tree is full in the sense that each node is either a leaf, labelled with a single variable, or else it is an internal node with two children (internal nodes do not correspond to variables). An SDD's variable tree, then, determines how the smaller sets decompose and therefore completely determines the SDD. SDDs are called a *canonical representation* for this reason. Consequently, searching for a small SDD reduces to searching for a good vtree.

A left-to-right traversal of a vtree induces a variable order. When we say that an algorithm for vtree optimization preserves the variable order, we mean that the algorithm produces a new vtree with the same induced variable order. A *right-linear* vtree is one in which every internal node's left child is a leaf. BDDs are precisely those SDDs with a right-linear vtree, hence SDDs generalize BDDs.

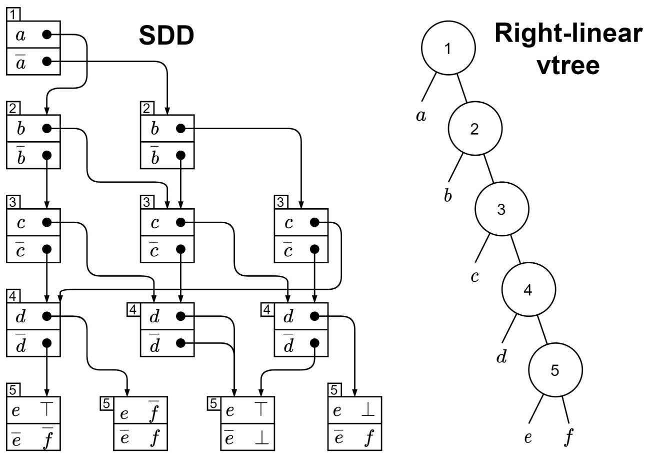

Figure 7.2: An SDD normalized to a right-linear vrtee.

**Example 7.2.** Figure 7.2 shows an SDD normalised to a right-linear vtree for the same set $K$ as in Example 7.1. The label in a round node indicates the vtree node it is normalised to. Although the vtrees are different, they induce the same variable order $A < B < C < D$. This SDD has six nodes, whereas the SDD in Figure 7.1 has seven nodes. We see, therefore, that the variable tree can affect the size of the SDD, independently of the variable order.

**Formal semantics** An SDD node $t$ formally represents a set of strings denoted by $\langle t \rangle$, as follows. The node may be of one of two types:

1. $t$ is a terminal node labelled with a set $A \in \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$, in which case $\langle t \rangle = A$.
2. $t$ is a decomposition node, and $t = \{(p_1, s_1), \ldots, (p_\ell, s_\ell)\}$, where $p_i$ and $s_i$ are other SDD nodes. Then $\langle t \rangle = \bigcup_{i=1}^{\ell} \langle p_i \rangle \times \langle s_i \rangle$

#### 7.2.1.1 Queries and transformations

Because SDDs are a canonical representation, equivalence of two SDDs can be checked in constant time, provided that they obey the same variable tree. SDDs support polynomial-time model counting and model enumeration. Whether SDDs support polynomial-time conjunction and disjunction was an open question which was recently answered in the negative, unfortunately, by Van den Broeck and Darwiche [322].[†] However, they find empirically that this worst-case behaviour rarely manifests in practice.

### 7.2.2 Model checking and reachability

Model checkers verify whether the behavior of a system $M$ conforms to a specification $\varphi$. To support reactive systems, the property $\varphi$ is expressed in a temporal logic, such as, LTL [266], CTL [85] or the modal $\mu$-calculus [189]. The formal check involves deciding semantic entailment, i.e., $M \models \varphi$, and can be implemented as a fixpoint computation on the semantic interpretation of the system [28], typically a transition system (Kripke structure). Since the fixpoint computations reduce to reachability, we only need to focus on solving reachability to evaluate and compare the performance of various decision diagrams in model checking.

The reachability procedure is a repeated image computation with the symbolic transition relation $R$ until a *fixpoint* is reached. Starting from $A := S_0$, we compute $A_{i+1} := A_i \cup R(A_i)$, until this computation converges, i.e., an iteration adds no more states. In each iteration, the set $A_i$ contains a set of system states.

In symbolic model checking, we store the set $A_i$ using a decision diagram, e.g., an SDD. In order to encode the relation $R$, the vtree will contain, for every variable $x \in X$, a “primed copy” $x' \in X'$. The SDD encoding the transition relation will contain one satisfying assignment for each transition $(s, t) \in R$ (and no more). We therefore denote this SDD as $R(X, X')$; and similarly the SDDs representing sets as $A(X)$. For reachability with SDDs, we need to compute the image function of the transition relation. Equation 7.3 shows how the image is computed using elementary manipulation operations on SDDs (conjunction, existential quantification and variable renaming). The conjunction constrains the relation to the transitions starting in $A$; the existential quantification yields the target states only and the renaming relabels target states in $X$ instead of $X'$ variables.

$$\text{Image}(R, A) \triangleq (\exists X\colon R(X, X') \wedge A(X))[X' := X] \tag{7.3}$$

[†]More precisely, they show that exponential blowups may occur when disjoining two SDDs if the output SDD must obey the same vtree, but it remains an open question whether the blowup is unavoidable when the vtree is allowed to change.

In our context, we have the specification of the system we wish to model check. Consequently, we know in advance which variables and actions exist, (e.g., in a Petri net the variables are the places and the actions are the transitions), and therefore we know which read/write dependencies exist between actions and variables.[‡] We use this information to design heuristics for SDDs. The integers in our systems are at most 16-bit, so each integer is represented by 16 primed and unprimed variables in the relation (for details, see Section 7.5).

## 7.3 Vtree heuristics

We propose three heuristics for constructing vtrees based on the read/write dependencies in the system being verified. Sec. 7.3.1 proposes that the vtree is constructed in two phases, and Sections 7.3.2 and 7.3.3 propose heuristics to be used in the two respective phases. Since the vtree structure can influence SDD size independent of the variable order, as shown in the previous section, we assume a good variable order and give heuristics which construct a vtree which preserves that variable order. The input to our vtree construction is therefore a good variable order, provided by existing heuristics [224], along with the known read/write dependencies between those variables, derived from the input that we wish to model check. We then construct the vtree using the heuristics explained below, preserving the variable order. The resulting vtree is then preserved throughout the reachability procedure, or, if dynamic vtree search is enabled, serves as the initial vtree.

Our application requires that we store relations rather than states, so our vtrees will contain two copies of each variable, as explained in Sec. 7.2.2.

### 7.3.1 Two-phase vtree construction

Any vtree heuristic will have to deal with the fact that the system's variables are integers, whereas an SDD (or a BDD) encodes each bit as an individual variable, so that an integer is represented by multiple variables. Therefore, our first heuristic is that we propose that a vtree be constructed in two phases.

[‡]While for some specification languages this information has to be estimated using static analysis, there are ways to support dynamic read/write dependencies [223], but for the sake of simplicity, we do not consider them here.

In the first phase, one builds a small vtree called an "abstract variable tree", containing all of the system's variables as leaves. In the second phase, we "fold out" the leaves corresponding to variables that represent more than one bit, i.e., whose domain is larger than $\{0, 1\}$. Folding out a leaf means replacing it by a larger vtree on several variables. For example, a leaf representing a 32-bit integer variable will be folded out into a vtree on 64 variables (namely 32 state variables and 32 primed copies). Figure 7.3 illustrates this process. In this example, a system with four 4-bit integer variables, $p, x, y, z$, may lead, during phase 1, to the abstract variable tree in the top of the picture. In phase 2, the leaves may be folded out into, in this case, vtrees on 8 bits, producing either the bottom left or right vtree.

Note that this never produces a right-linear vtree, even if both the abstract variable tree and the integer's vtree are right-linear.

### 7.3.2 Abstract variable tree heuristic

To design the abstract variable tree, we adopt the intuition that, if a system reads variable $x$ in order to determine the value to write to variable $y$, then, all other things equal, it would be better if the distance in the vtree between $x$ and $y$ was small.

To capture this intuition, we define a penalty function $p(T)$ (Equation 7.4) on a vtree $T$, which we call the *minimum distance* penalty. We then try to minimize the value of this penalty function by applying local changes to the vtree. The penalty function will simply be the sum, over all read-write dependencies in the system, of the distance between the read and write variable in the vtree. By distance in the vtree, denoted $d_T$, we mean simply the length of the unique path in the abstract variable tree from the leaf labelled with variable $x$ to the leaf labelled with variable $y$. Here $A$ is the set of a system's actions, $R_a$ is the set of variables read by action $a$, and $W_a$ is the set of variables written to by action $a$.

$$p(T) = \sum_{a \in A} \sum_{(r,w) \in R_a \times W_a} d_T(r, w) \tag{7.4}$$

This heuristic is inspired by various variable ordering heuristics, such as Cuthill-Mckee [90] and the bandwidth and wavefront reduction algorithms [224], all of which try to minimize the distances between dependent variables.

### 7.3.3 Folding out vtrees for large variable domains

We consider two options for folding out an abstract variable. First, such a variable may be folded out into a right-linear vtree, i.e., to a BDD. Figure 7.3 (left) shows this approach. Second, we introduce the "augmented right-linear vtree", which is a right-linear vtree except that each pair of corresponding state and prime bit is put under a shared least common ancestor. The idea is that a state and primed bit are more closely correlated to one another than to other bits. Figure 7.3 (right) shows this approach.

Note that in both options, the resulting vtree is not a right-linear vtree.

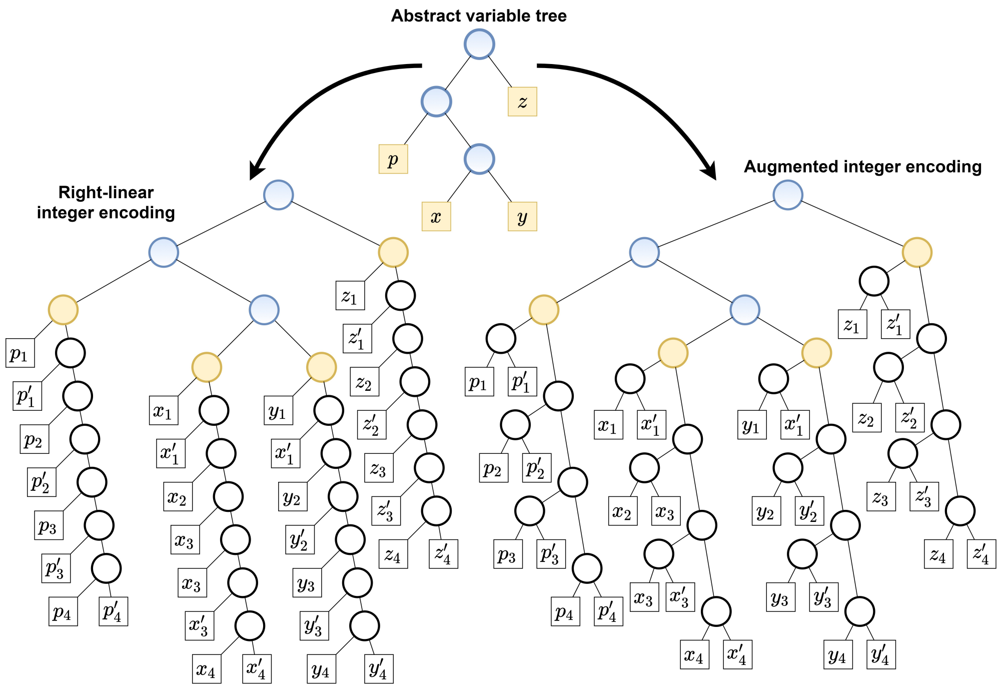
Abstract variable tree
Right-linear integer encoding
Augmented integer encoding
$p$
$x$
$y$
$z$
$p_1$ $p'_1$ $p_2$ $p'_2$ $p_3$ $p'_3$ $p_4$ $p'_4$
$x_1$ $x'_1$ $x_2$ $x_3$ $x_3$ $x'_3$ $x_4$ $x'_4$
$y_1$ $x'_1$ $y_2$ $y'_2$ $y_3$ $y'_3$ $y_4$ $y'_4$
$z_1$ $z'_1$ $z_2$ $z'_2$ $z_3$ $z'_3$ $z_4$ $z'_4$

Figure 7.3: An abstract variable tree (top) with four 4-bit integers can be folded out in two ways: To a vtree with right-linear integer encoding (left) or with augmented integer encoding (right).

## 7.4 Related work

Many algorithms exist for static variable ordering for BDDs, e.g., the Cuthill-McKee algorithm [90], Sloan [297] and MINCE [9]; see [276] for a survey. These algorithms exploit event-locality: the observation that most events involve only very few of a system's variables. By using an order in which variables appear next to each other when they appear together in many events, the size of a BDD can decrease relative to variable orders which do not take this into account. Finding an optimal order is NP-Hard [56], so these algorithms heuristically minimize certain distance metrics [292].

We are not aware of any work which attempts to statically construct vtrees. Works in this direction are Darwiche [96] and Oztok and Darwiche [249], which show that a CNF formula with bounded treewidth, or CV-width, respectively, can be compiled into a polynomial-size SDD if a vtree is known which reflects the tree decomposition of the formula. Unfortunately, finding the tree decomposition is NP-Hard [19], and indeed even hard to approximate [353], although this can be done in linear time when the treewidth is bounded [53].

Choi and Darwiche [83] introduce *dynamic vtree search*, that is, they modify the vtree in between SDD operations, aiming to reduce the SDD size. This is useful even when space is not a bottleneck, because, all other things equal, algorithms run faster on small SDDs than on large ones. When dynamic vtree search is enabled in our experiments, we use the implementation of Choi and Darwiche.

## 7.5 Experiments

We evaluate our methods on 707 benchmarks, consisting of 293 DVE models, 357 Petri Net models and 57 Promela models. The benchmarks are provided[§] by Meijer and van den Pol [224], who collected them from the BEEM database [258] and the 2015 Petri Net Model Checking Competition [188], selecting those problems that can be handled by the static variable ordering algorithms. To utilize these different specification languages, we extended[¶] the LTSmin model checker [178] with an SDD package [25]. The language-independent interface of LTSmin [178] represents all variables as 16-bit integers. Both the BDD and the SDD packages benchmarked here utilize the same

[§]The benchmarks are available at `https://github.com/utwente-fmt/BW-NFM2016`

[¶]Again, the implementation is available at https://doi.org/10.5281/zenodo.3940936 interface. As explained in the introduction, we are interested in evaluating the performance of SDDs with respect to BDDs, to study the feasibility of SDDs for model checking and identify bottlenecks in SDD implementations.

To implement the abstract variable tree heuristic described in Sec. 7.3.2, we perform a vtree search before executing the reachability procedure in the model checker. We greedily perform local modifications on the vtree until either (i) no possible modification improves the value of the penalty function $p$ (Equation 7.4), or (ii) our budget of $n$ local modifications is exhausted, where $n$ is the number of program variables, or (iii) 30 seconds have elapsed. The local modifications are restricted to *vtree rotations*, which is an operation which inverts the roles of parent and child for a particular pair of adjacent internal vtree nodes. This operation preserves the induced variable order.

We experiment with two BDD configurations and four SDD configurations, listed in Table 7.1. Each configuration is tested on all 707 benchmarks, with a timeout of 10 minutes (600 seconds) per benchmark. Experiments are done on a Linux machine with two 2.10 GHz Intel Xeon CPU E5-2620 v4 CPUs, which each have 20 MB cache and 16 cores, with 90GB of main memory. The BDDs are implemented using the Sylvan multicore BDD package (version 1.4.0) [327]. The SDDs are implemented using the SDD package (version 2.0) [25]. Before vtree construction, we apply the Cuthill-Mckee (BCM) [90] variable reordering heuristic provided by LTSmin [224]. The input to our vtree construction is the variable order found by BCM, along with the known read/write dependencies between those variables. We perform a small case study of six problems from the DVE set with a timeout of 3 hours instead of 10 minutes.

Table 7.1 lists the parameter settings used for benchmarks. The first row denotes the BDD configuration against which we compare our method. The SDD configurations are such that, in each next row, we "turn on" one more vtree heuristic. The BCM in the variable order column refers to the Cuthill-Mckee ordering heuristic [90], which is performed once, before vtree construction and model checking. The column "Augmented int" records whether integer leaves in the abstract variable tree are folded out to a right-linear or an augmented right-linear representation. The "vtree heuristic" denotes whether the abstract variable tree was right-linear ("No") or was constructed using the minimum distance penalty ("Min. distance"). The row BDD refers to the single-threaded implementation, whereas the row BDD(32) uses the multi-threaded (32-core) implementation [327], which we include only for reference as SDDs [25] are still single-threaded.

Dynamic vtree search [83] is the SDD analog of dynamic variable reordering, that is, it modifies the vtree during execution, aiming to reduce the size of the SDD. The column "Dynamic vtree search" records whether vtree search was enabled during reachability analysis. In that case, at most half the time is spent on vtree search.

| Configuration name | Variable order | Two-phase construction | Augmented int | vtree heuristic | Dynamic vtree search |
|---|---|---|---|---|---|
| BDD | BCM | (N/A) | (N/A) | (N/A) | No |
| BDD(32) | BCM | (N/A) | (N/A) | (N/A) | No |
| SDD(r) | BCM | Yes | No | No | No |
| SDD(r,a) | BCM | Yes | Yes | No | No |
| SDD(d,a) | BCM | Yes | Yes | Min. distance | No |
| SDD(d,a)+s | BCM | Yes | Yes | Min. distance | Yes |

Table 7.1: The parameter settings used in the experiments.

The memory usage is computed as follows. A BDD node takes up 24 bytes. An SDD node takes up 72 bytes, plus 8 bytes per element (an SDD node has at least two elements). That this measure is fair for SDDs was verified by Darwiche in personal communication. We take this approach instead of relying on the memory usage as reported by the operating system, which is not indicative of the size of the SDD / BDD due to the use of hash tables. For each benchmark, we will report the peak memory consumption, that is, the amount of memory in use when the BDD / SDD was at its largest during the reachability procedure.

### 7.5.1 Results

Figure 7.4 and 7.5 show the results of the experiments. In each plot, a blue "×" represents a DVE instance, a green "△" a Petri net instance, and a red "+" a Promela instance. Instances on the black diagonal line are solved using an equal amount of time or memory by both approaches, instances on a gray diagonal line are solved by one method with an order of magnitude advantage over the other. All graphs are formatted such that benchmarks that were solved better by the more advanced configuration appear as a point below the diagonal line; otherwise, if the less advanced configuration performed better, then it appears above the diagonal line. The horizontal and vertical lines of instances near the top left of the figures, represent instances where one or both methods exceeded the timeout limit.

Figure 7.4 (top) shows that the simplest SDD configuration, SDD(r), tends to outperform BDDs in terms of memory, especially on more difficult instances, but not in terms of time, often taking an order of magnitude longer. As a result of the 600 second timeout, BDDs manage to solve many instances that SDDs fail to solve. The middle row shows that using augmented integer vtrees is not perceptibly better than using right-linear vtrees. The bottom row shows that using our minimum distance heuristic improves both running time and diagram size. Finally, Figure 7.5 (top) shows that adding dynamic vtree search yields even smaller diagrams, and on difficult instances, tends to be slightly faster.

Figure 7.5 (bottom) compares SDD(d,a)+s to BDDs. It shows that memory usage is often an order of magnitude lower than that of BDDs on the same benchmark. The effect is even more pronounced than in Figure 7.4 (top). Unfortunately, this configuration often uses more time than BDDs do.

We see that two of our heuristics improve the performance of SDDs, at least as far as memory is concerned, namely, (i) using two-phase vtree construction instead of a BDD (Figure 7.4, top) and (ii) using the minimum abstract variable tree distance heuristic (Figure 7.4, bottom).

Table 7.2 and Table 7.3 show the relative speedup and memory improvements of all pairs of methods. A cell indicates the ratio by which the column method outperformed the row method, restricted to those benchmarks on which it outperformed the row method. For example, Table 7.2, shows that, on the 264 benchmarks that both SDD(r) and BDD solved, SDD(r) uses less memory than BDD on 172 benchmarks, and on those benchmarks, uses 3.1 times less memory on average, whereas on the remaining 92 instances, BDD used 5.5 times less memory than SDD(r).

From the tables we see that no method dominates another, even in just one metric. However, we do see that SDD(d,a)+s often uses much less memory than BDD (namely on 75% of benchmarks that both methods solved), and then on average by a factor 7.9. Strikingly, Figure 7.5 reveals that this advantage becomes progressively larger as the instances become more difficult. We also see that each heuristic helps a little bit, with the exception of augmented integer vtrees. That is, in the sequence BDD, SDD(r), SDD(d,a), SDD(d,a)+s, each next configuration, compared with the previous, is better on more instances than it is worse (There is one exception: SDD(d,a)+s is often smaller but slower than SDD(d,a)).

The results of the case study with a timeout of 3 hours are shown in Table 7.4, and are

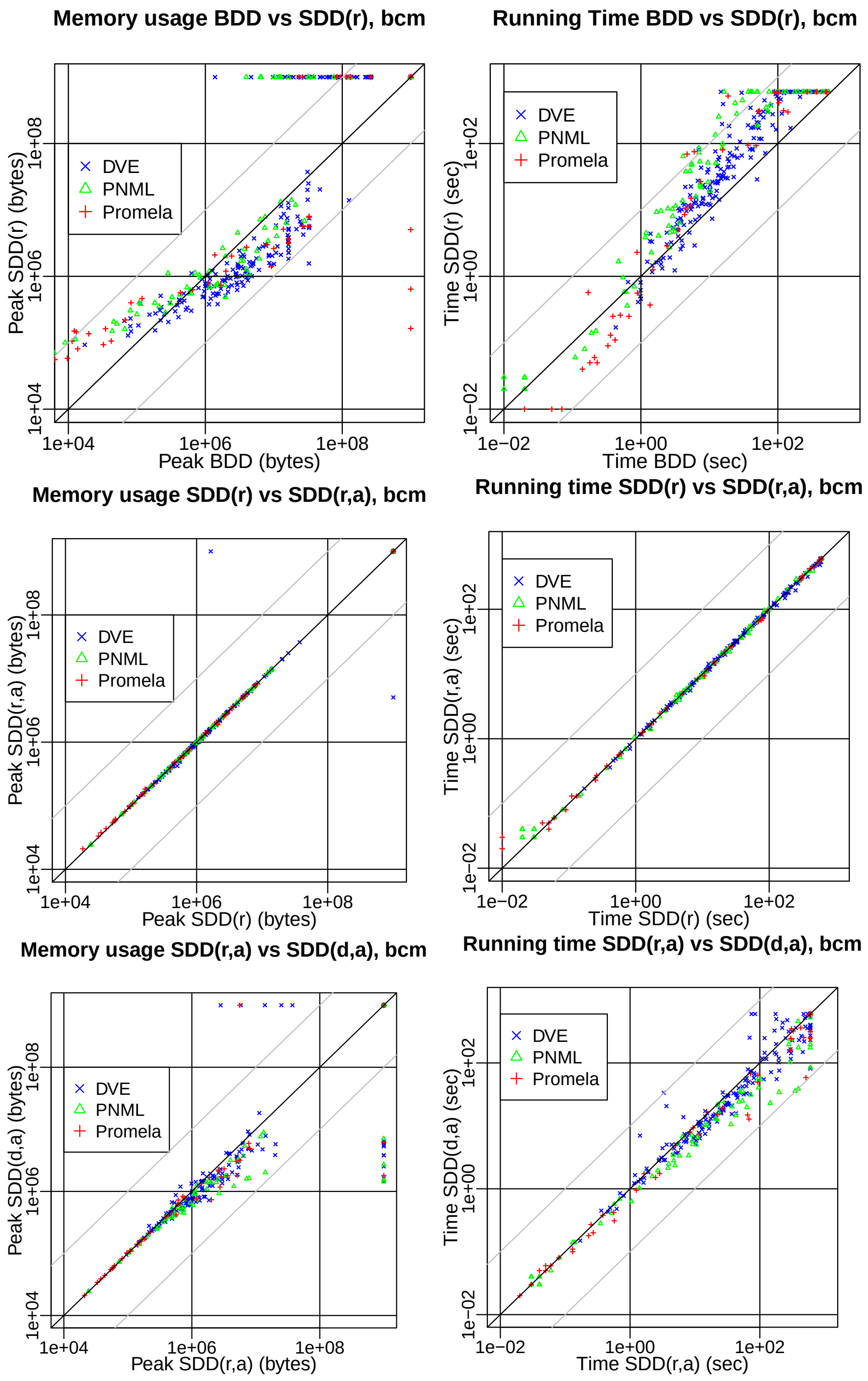

Figure 7.4: Memory (left) and time (right) consumption comparisons of various SDD configurations.

Table 7.2: Average ratio of memory usage on instances that the column method solved using less memory than the row method. Numbers in brackets indicate the number of benchmark instances on which the column method outperformed the row method. We leave out BDD(32), since its memory use is equal to BDD.

| Memory | BDD | SDD(r) | SDD(r,a) | SDD(d,a) | SDD(d,a)+s |
|---|---|---|---|---|---|
| BDD | 1 | 3.1 (172) | 3.1 (173) | 4.4 (193) | 7.9 (220) |
| SDD(r) | 5.5 (92) | 1 | 1.02 (138) | 1.5 (186) | 2.3 (216) |
| SDD(r,a) | 5.7 (91) | 1.01 (128) | 1 | 1.5 (182) | 2.3 (218) |
| SDD(d,a) | 6.2 (82) | 1.1 (75) | 1.1 (79) | 1 | 1.6 (212) |
| SDD(d,a)+s | 6.9 (72) | 1.2 (48) | 1.1 (46) | 1.1 (62) | 1 |

Table 7.3: Average ratio of time used on instances that the column method solved using less time than the row method. Numbers in brackets indicate the number of benchmark instances on which the column method outperformed the row method.

| Time | BDD | BDD(32) | SDD(r) | SDD(r,a) | SDD(d,a) | SDD(d,a)+s |
|---|---|---|---|---|---|---|
| BDD | 1 | 6.1 (321) | 2.2 (45) | 1.9 (46) | 1.9 (54) | 2.1 (51) |
| BDD(32) | 1.2 (11) | 1 | 2.4 (21) | 2.1 (21) | 2.1 (22) | 2.1 (23) |
| SDD(r) | 3.8 (219) | 17.2 (243) | 1 | 1.04 (121) | 1.8(187) | 2.1 (144) |
| SDD(r,a) | 3.7 (218) | 17.0 (243) | 1.1 (145) | 1 | 1.7 (189) | 2.1 (149) |
| SDD(d,a) | 2.8 (221) | 13.2 (253) | 1.6 (74) | 1.5 (72) | 1 | 2.0 (78) |
| SDD(d,a)+s | 2.7 (240) | 13.3 (269) | 1.4 (120) | 1.3 (115) | 1.4 (196) | 1 |

included in Figure 7.5 (bottom left). In this problem set, SDDs consumed 31 times less memory on average.

### 7.5.2 SDD runtime profiles

Figure 7.6 breaks up the time spent on the most difficult instances into the three SDD operations (Existential quantification, intersection and union of sets), and the glue code which connects LTSmin to the SDD package (LTSmin's Partitioned Next-State Interface (PINS), is described in [178]). For a few instances (`pouring`, `lup`, and `peg_solitaire`) the glue code takes the most runtime. We verified that in these cases the input system contains few locality (the PINS dependency matrix is dense), which is not what LTSmin was designed for, so we will not discuss these instances further.

In the SDD packages, we see that the main bottleneck is existential quantification and vtree search, whereas intersection and especially union take less time. The vtree search pays off, because in this setting the model checking procedure performed best as we discussed in the previous section. It comes as no surprise that existential quan-

Table 7.4: Time and peak memory consumption (kB) on six large problems. Times are in seconds.

| Metric | Name | synapse.7 | telephony.4 | szymansky.5 | sched_world.3 | sokoban.1 | telephony.7 |
|---|---|---|---|---|---|---|---|
| Space | SDD(d,a)+s | 6804 | 20893 | 24587 | 45694 | 1838 | 19453 |
| | BDD | 88905 | 131214 | 165836 | 259544 | 261113 | 262210 |
| Time | SDD(d,a)+s | 836 | 1078 | 4918 | 1789 | 858 | 2295 |
| | BDD | 73 | 19 | 42 | 41 | 28 | 36 |

tification is a bottleneck, because quantification for multiple variables is NP-Hard and all variables are bit-blasted in LTSmin. It is, in some sense, good news: There is a lot performance to be gained by eliminating this bottleneck, and there is good hope that this is possible, because NP-Hard problems can often be made tractable in many easy instances by developing good heuristics. The SDD package currently employs no special heuristics to perform existential quantification.

We were unable identify unique characteristics in terms of their number of variables or their number of actions for the outlier instances in which set intersection is the bottleneck (e.g., `elevator2.3`).

## 7.6 Conclusion and future work

Sentential Decision Diagrams are a viable alternative to Binary Decision Diagrams for use in symbolic model checking. The size in memory of an SDD is determined by its vtree, and choosing this vtree was the foremost challenge when implementing SDD-based model checking. This challenge was satisfactorily met with novel heuristics. Our experiments show that the novel heuristics yield SDDs that are often an order of magnitude smaller than BDDs on the same problem, and this advantage becomes larger on more difficult instances. These results are robust to the difficulty of the instance, as the performance of BDDs on our benchmark set spans five orders of magnitude, from 10kB of memory to 300MB of memory. That SDDs are slower contrasts with findings of Choi and Darwiche [83]. We suggest three avenues for research in the near future.

1. Combine variable order heuristics with vtree structure heuristics
2. Use more advanced data structures than SDD
3. Eliminate the speed bottlenecks in SDDs; specifically, develop (heuristics for) faster existential quantification algorithms.

For Item 2, the Zero-Suppressed Sentential Decision Diagram [245], and the Tagged Sentential Decision Diagram [111,328], look like promising next steps, because both do well when representing sparse sets, which is the case in the current application. Using Uncompressed Sentential Decision Diagrams looks promising in theory, but recent work suggests that much still needs to be done before they can be considered a tractable data structure [322].

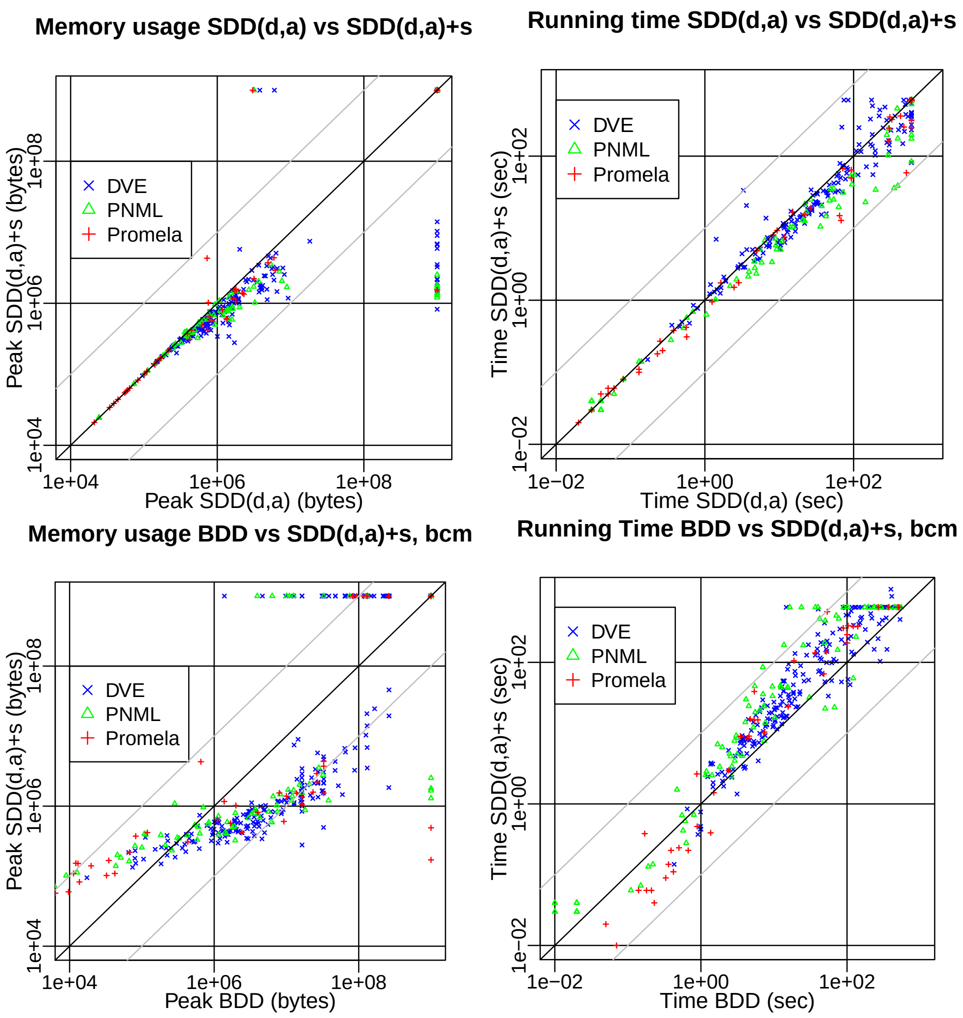

Figure 7.5: Memory (left) and time (right) consumption comparisons (continued).

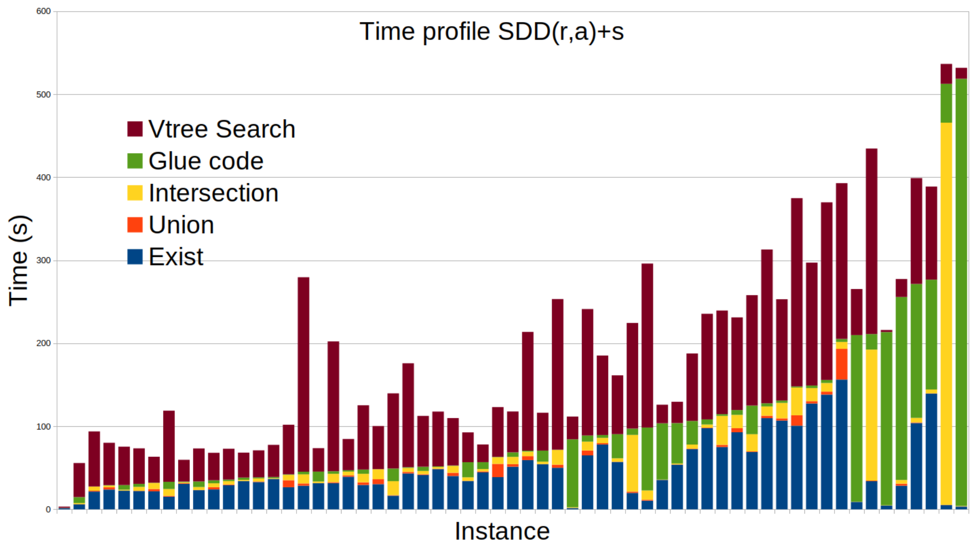

Figure 7.6: The time profile of SDD configuration SDD(d,a)+s, split among the three primary SDD set operations of Intersection, Union and Existential quantification, and the glue code which connects LTSmin to SDD. Shown are the DVE and PNML instances that were solved in more than 100 seconds. The instances are sorted by the time taken by the SDD operations, plus glue code.