Dynamic Delayed Duplicate Detection for External Memory Model Checking

Sami Evangelista
evangeli@daimi.au.dk

DAIMI - Department of Computer Science
University of Aarhus, Denmark

Abstract. Duplicate detection is an expensive operation of disk-based model checkers. It consists of comparing some potentially new states, the *candidate* states, to previous *visited* states. We propose a new approach to this technique called *dynamic delayed duplicate detection*. This one exploits some typical properties of states spaces, and adapts itself to the structure of the state space to dynamically decide when duplicate detection must be conducted. We implemented this method in a new algorithm and found out that it greatly cuts down the cost of duplicate detection. On some classes of models, it performs significantly better than some previously published algorithms.

Model checking, or state space analysis, is a method to prove that finite state systems match their specification. Given a model of the system and a property, e.g., a temporal logic formula, it explores all the possible configurations, i.e., the state space, of the system to check the validity of the property. Despite its simplicity, its practical application is limited due to the well-known state explosion problem [19]: the state space can be far too large to be explored in reasonable time or to fit within the available main memory. Consequently, the design of methods able to cope with this problem has gained a lot of interest in the verification community.

A first family of techniques reduce the part of the state space that needs to be explored in such a way that all properties of interest are preserved. An example of such a technique is partial order reduction [4] that limits redundant interleavings by exploiting the independence of some actions.

A more pragmatic approach does not aim at reducing the size of the problem but rather at making a more subtle use of the available resources (or augment them) to extend the range of problems that can be analyzed. Many options are available. We can, for example, compress states to virtually decrease the problem size, distribute the search to benefit from the aggregate computational power and memory of a cluster of machines, or make use of external memory.

In this work we look at this last option. Using disk storage instead of main memory is indeed very tempting since it considerably increases the amount of available memory thereby making it possible to solve problems that could not be solved even with the help of sophisticated techniques such as partial order reduction. As a counterpart, disk accesses are much slower. In addition, the data structure used to store states is typically randomly accessed, involving an important runtime penalty when kept on disk. Hence, the use of disk storage requires a dedicated algorithm to be effective.

Most external memory algorithms are somehow based on the key idea of *delayed duplicate detection.* When a state is generated, the algorithm does not immediately check if the state has already been visited, i.e., if the state is a duplicate, since it would require a new disk access, and potentially the load of a disk block. Instead, the state is put into a *candidate* set that contains all the potentially new states and the comparison to the *visited* set stored on disk is delayed until it can be efficiently conducted. Hence, a large number of individual disk look-ups is replaced by a single file scan.

This paper reviews some algorithms proposed by the model checking and artificial intelligence community and introduces a new duplicate detection scheme based on breadth-first search. We refer to this scheme as *dynamic delayed duplicate detection.* Its principle is to exploit some typical properties of state spaces to decrease the cost of duplicate detection and to dynamically collect some data on the structure of the state space that will be used to decide when duplicate detection should be delayed or conducted.

Organization of the paper We recall in Section 1 some basic elements on graphs and review existing works on disk based model checking. Section 2 introduces a simple variation of hash based delayed duplicate detection [10] that will be the basis of our dynamic algorithm. Some structural properties of state spaces that can be exploited are presented in Section 3. Section 4 contains the main contribution and introduces dynamic delayed duplicate detection that is experimentally evaluated in Section 5. Finally, Section 6 concludes this paper.

1 Background

Definitions and notations We briefly give the ingredients that are relevant for understanding this paper. Figure 1 will help us illustrate these notions.

A **state space** is a directed graph (S, T, s_0) where S is a finite set of states, $T \subseteq S \times S$ is a set of transitions, and $s_0 \in S$ is the initial state. A state $s' \in S$ is a **successor** (resp. **predecessor**) of state $s \in S$, if $(s, s') \in T$ (resp. $(s', s) \in T$). We denote by $succ(s)$ (resp. $pred(s)$) the set of successors of s (resp. predecessors). The **distance** of a state s, denoted by $d(s)$ is the length of the shortest path from s_0 to s. The states at **level** k, noted $\mathscr{L}(k)$, is the set of all states which distance is k. It is defined recursively as $\mathscr{L}(0) = \{s_0\}$, $\mathscr{L}(k+1) = \{s' \in S | \exists s \in L(k) \cap pred(s')\} \setminus \cup_{i=0}^{k} \mathscr{L}(i)$. The **height** of a state space is its number of levels and its **width** is the size of its largest level, i.e., $\max_k |\mathscr{L}(k)|$. In our example, the height and width are respectively 4 and 3. The

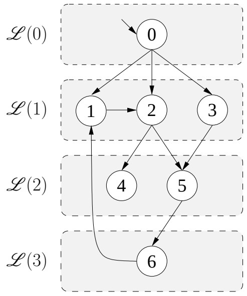


Fig. 1. A state space

average degree is the ratio $\frac{|T|}{|S|}$.
$t = (s, s')$ is a **forward transition** if $d(s') = d(s) + 1$. Otherwise it is a **backward transition** and we define its **length** as $d(s') - d(s)$. All transitions of our example are forward transitions except (1,2) and (6,1). Their lengths are respectively of 0, since 1 and 2 are on the same level, and 2.
We denote by $\mathscr{R}(k)$ the successors of states of level k, i.e., $s \in \mathscr{R}(k) \Leftrightarrow \exists s' \in \mathscr{L}(k) \cap pred(s)$. More generally, $\mathscr{R}^n(k)$ consists of all the states reachable from level k by a path of length n. Formally, $s \in \mathscr{R}^n(k) \Leftrightarrow \exists s' \in \mathscr{L}(k), (s_1, s_2), \ldots, (s_n, s_{n+1}) \in T \mid s_1 = s' \wedge s_{n+1} = s$. The successors of states of level 1 in figure 1 is the set $\mathscr{R}(1) = \mathscr{L}(2) \cup \{2\} = \{2, 4, 5\}$ and $\mathscr{R}^2(1) = \{4, 5, 6\}$. By definition, it holds for any d that $\mathscr{L}(d+1) \subseteq \mathscr{R}(d)$ since $\mathscr{R}(d)$ contains level $d + 1$ plus all the states reachable from level d by a backward transition.

A breadth-first search (BFS) explores a state space by maintaining a set of already visited states and a FIFO queue filled with states to be further explored. Each state dequeued is expanded and those of its successors that do not belong to the visited set are inserted into it and enqueued to be later expanded. Applying a FIFO strategy ensures that levels of the state space will be processed one by one: if a state s of level k is dequeued then all states of levels $l < k$ have been processed and belong to the visited set. A transition (s, s') generates a **duplicate** s' if s' already belongs to the visited set when s is processed. Obviously, any backward transition will generate a duplicate when applying BFS.

Related work Dill and Stern [2] were the first work to propose the use of external memory as a way to enhance the capabilities of explicit state model checkers. Their breadth-first search algorithm stores all the states of the current level in a RAM hashtable while previous levels are stored on disk. The search for duplicates occurs each time a BFS level has been completed or the RAM hashtable becomes full. The disk file is then read and duplicates in memory are deleted. Remaining states are written on disk and inserted in the memory queue. Grouping disk lookups is the strategy of most disk based model checkers. It allows to read or write whole blocks of states at once, while checking for each state individually would likely require to reload a new block from disk. This technique is known as *delayed duplicate detection* (DDD): the resolution is postponed until it may be efficiently conducted.

Della Penna et al. [18,17] proposed two algorithms that benefits from a property usually exhibited by communications protocol: backward transitions are usually short. Hence, in BFS, they usually lead to a recently visited state. The first one, [18], is a cache based algorithm that only keeps recent states in memory while the queue is on disk. The second one, [17] is a variation of the initial disk-based algorithm of [2]. Instead of systematically comparing the set of candidate states to the visited set, it is only checked against some blocks of states chosen randomly according to their age. We found in [21] a similar process called layered duplicate detection.

Bao and Jones observed in [1] that duplicate detection is the most time consuming operation of the algorithms of [2] and [17]. They proposed a new algorithm, based on a partitioned hash table, that mimics a distributed search and behaves better than these two.

Hammer and Weber [5] designed a hybrid algorithm that adapts itself to the size of the state space: as long as memory is sufficient it remains a pure RAM based algorithm while it slowly shifts to a disk based algorithm as memory becomes scarce.

In [7], an I/O efficient solution is presented for directed model checking. States are organized into buckets according to some heuristics. Files are associated to buckets in order to store overflowing states. This distribution greatly eases the search for duplicates and the parallelization of the algorithm [8].

Korf introduced in [9] and [10] the principles of sorting-based (SDDD) and hash-based delayed duplicate detection (HDDD) both based on a generic algorithm called frontier search [12] that only stores states to be expanded (the frontier). This algorithm cannot really be applied in the context of model checking as it requires the ability to compute the predecessors of a state, an operation that is impossible when the state space is given implicitly as an initial state and a successor function. After each expansion phase SDDD sorts resulting files in order to detect duplicates while HDDD avoids the complexity of sorting by distributing states onto multiple files using two orthogonal hash functions. Not only is it more efficient than SDDD but it can also be efficiently parallelized [11].

In structured duplicate detection [20] an abstraction of the system is used to determine when to load/unload states from/to disk. This technique has a strong potential but heavily relies on the quality of the abstraction. This problem may be overcome by partitioning the edges [23] or by automatically extracting an appropriate abstraction from the system description [22].

2 A variation of hash-based duplicate detection

Hash based delayed duplicate detection (HDDD) is a very successful strategy for external memory graph search, already applied to state spaces with more than 10^{12} states [11]. Unfortunately, in its basic version, it is based on the generic frontier search that cannot be applied in the context of implicitly given graphs that we have in model checking. We present in this section a simple breadth first search (BFS) variation of the algorithm of Korf [9]. We refer to this algorithm as BFS-HDDD. Exploring the state space in breadth-first order has several advantages. The most obvious one is its ability to report safety violations of minimal lengths. Secondly, as opposed to, e.g., depth first search, BFS can be easily parallelized. This requires some synchronizations [8], but if the load is well balanced among processors (or nodes of the network), which is the case in the algorithm of [9], it is likely that latency will be negligible. Last, it is possible with BFS to exploit some interesting properties of state spaces to reduce duplicate detection times and fasten the search. This is perhaps the most interesting property of BFS for us since it is an important component of our new algorithm.

The BFS-HDDD algorithm (see figure 2) partitions the queue of states to visit into N files $\mathcal{Q}_1, \ldots, \mathcal{Q}_N$ and the visited set into N files $\mathcal{V}_1, \ldots, \mathcal{V}_N$. In a first step,

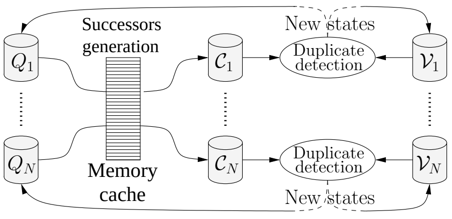


Fig. 2. An iteration of BFS-HDDD.

queued states are processed, their successors generated and inserted into a memory cache. If this one becomes full its content is flushed to the candidate set, also partitioned in a set of N files $\mathcal{C}_1, \ldots, \mathcal{C}_N$. A first hash function is used to map states to the appropriate candidate file ensuring that duplicates will be inserted into the same file. Once all queued states have been expanded the cache is flushed to candidate files and the duplicate detection phase begins. In the second step each partition is processed one by one. The content of a candidate file is hashed to memory using a second hash function, thus detecting duplicates in this file. Then, the states of the corresponding visited file are read one by one and deleted from memory. Remaining states in memory are therefore new and can be written in the visited file as well as in the queue file so that they can be processed by the algorithm at the next iteration. Once all partitions are processed the algorithm can move to the next BFS level. Before that, the candidate set is emptied.

Partitioning the state space has two advantages. First it is helpful to parallelize the algorithm as showed in [11]. Moreover, it virtually multiplies by N the number of candidates that may reside in memory allowing us to perform a single duplicate detection per level. This is in constrat with algorithms of [2] and [17] which also perform a detection when the cache is full. Hence, BFS-HDDD should behave much better with large models for which only a small fraction of the state space can be kept in RAM. However, duplicate detection still remains a costly operation, especially if the graph has a large height. Detections are very cheap at the beginning of the search when the visited set is small but on the last levels each one entails to read from disk a large portion of the state space. More generally, if H is the height of the graph, the number of states read from the visited files during duplicate detections will exactly be $\sum_{s \in S}(H - d(s))$. Some interesting properties usually exhibited by state spaces can however help us to reduce the cost of this operation and design a new algorithm that behaves better than BFS-HDDD.

3 Some structural properties of state spaces

State spaces, as opposed to random graphs, have some typical properties [14] that can be exploited in automated verification. For instance, disk based model checkers can exploit transition locality [18,17]. Tools can also decide which reduction technique to apply depending on a partial knowledge of the graph [16].

The BEEM database [15] is a precious tool to analyze such properties. It contains more than 50 parametrized models and 300 actual instances of various families. Three observations, that have some consequences in BFS, can be made. The reader may consult [3] for further details on the data provided in this section.

Observation 1: Low proportion of backward transitions First, as already shown in [18], most transitions of state spaces are forward transitions. The average rate of backward transitions we computed is around 20%. If we only consider communication protocols this rate goes down to approximately 15%.

Observation 2: Few typical lengths for backward transitions A closer look at the backward transitions also reveals a non uniform distribution of their length as already pointed out in [14]. There are usually a few typical lengths and most backward transitions have one of these lengths. For example, we observed that on 95% of the database instances 5 lengths covered more than 50% of all backward transitions. Even one single length cover more than 50% of backward transitions in 53% of the instances.

Observation 3: Regular evolution of levels We measured the progression of rate $\frac{|\mathscr{L}(l+1)|}{|\mathscr{L}(l)|}$, that we shall call the *level progression rate* (or more simply progression rate), and found out that the size of levels evolve in a rather regular way and there are usually no huge variations of this rate between close levels. When this is not the case we however noticed that corresponding levels are rather small, meaning that the number of states involved is negligible. Some simple models, e.g, the tower of Hanoi, do not have such a property but if we look at more interesting ones like communication protocols, this observation is often valid. The progression rate generally follows a three step scenario. First levels are characterized by a high rate: levels grow quickly at the beginning of the search. Then the progression rate quickly collapses to a value close to 1 and during a long period stays around this value while tending to decrease. Finally, on last levels, the rate drops down to 0.

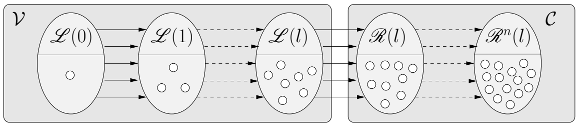
4 Dynamic delayed duplicate detection

We propose in this section, *dynamic delayed duplicate detection* (or DDDD for short), as an alternative to existing duplicate detection schemes. DDDD is based on two key ideas. First, it exploits the structural properties usually exhibited by state spaces that we have discussed in the previous section. We thus obtain a specialized algorithm, especially designed for state spaces having those properties. Second, we dynamically collect data on the graph structure so that the algorithm can adapt itself on-the-fly to its particular characteristics. Thus, even if the model is not, a priori, suited, the algorithm will progressively change its strategy to fit with the model.

4.1 Principle

A breadth first search algorithm based on the DDDD discipline works basically as the algorithm presented in section 2. The only difference is the following one: instead of systematically comparing the candidate set to the visited set at each level, we only perform a duplicate detection when we consider it to be necessary. This decision will mainly be based on data collected by the algorithm during the search. The general principle of DDDD is also the one of [17] and is motivated by the first observation made in section 3: when we expand the states of level l, it is likely that most of the states reached will not belong to the visited set. Thus, looking for duplicates may be almost useless. Instead we store $\mathscr{R}(l)$ on disk in a candidate set that will be used later during the next duplicate detection. The states expanded at the next BFS level will be those of $\mathscr{R}(l)$ and their successors, i.e., $\mathscr{R}^2(l)$, will also be written in the candidate set, and so on. Only when we decide to perform duplicate detection will the candidate states be hashed to memory and the visited states will be read in order to delete duplicates in memory as done by the BFS-HDDD algorithm. Remaining states in memory are inserted to the visited set and those which are on the "front" of the candidate set, i.e., the states of $\mathscr{R}^n(l)$ (if the detection occurs at level $l + n$) minus the duplicates removed, are later expanded.

The figure below presents a snapshot of visited and candidate sets during the execution of our algorithm. Visited states belongs to $\mathscr{L}(0) \cup \cdots \cup \mathscr{L}(l)$ while the candidate set contains all states reachable from level l via a path of length n or less. The latter is actually a multi-set since a state may belong to $\mathscr{R}(l)$, $\mathscr{R}^2(l)$, $\ldots$, and $\mathscr{R}^n(l)$.



Though this strategy is expected to decrease I/Os it has a cost since a state may be reexpanded during the search: any target of a backward transition (or one of its descendant) is likely to be revisited. Since the expansion of a duplicate necessarily leads us to other duplicates, the proportion of duplicates visited may quickly grow even if the graph has few backward transitions. For instance, if 90% of transitions of forward transitions, we can expect to approximately have 10% of duplicates on level $l + 1$, then $1 - 0.9^2 = 19\%$ on level $l + 2$ and more generally, a proportion of $1 - 0.9^n$ duplicates on level $l + n$.

4.2 The algorithm

The BFS-DDDD algorithm (see figure 3) partitions the visited and candidate sets as well as the BFS queue into N files $\mathcal{V}_1, \ldots, \mathcal{V}_N$, $\mathcal{C}_1, \ldots, \mathcal{C}_N$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_N$. A unique hash function h is used to map states to these files. The only global data structure to reside in memory is the memory cache $\mathcal{C}ache$ implemented by a chained hash table. States overflowing from $\mathcal{C}ache$ are stored in some temporary files $\mathcal{T}_1, \ldots, \mathcal{T}_N$[1]. Those are candidate states and could be directly written in candidate files but we prefer to avoid it as it would involve the possibility to have multiple instances of the same state in a candidate file. Since it is likely

[1] Thereafter we shall use the term of visited, candidate and temporary sets when speaking of states written in the corresponding files. We write $\mathcal{V} = \cup_i \mathcal{V}_i$, $\mathcal{C} = \cup_i \mathcal{C}_i$ and $\mathcal{T} = \cup_i \mathcal{T}_i$.

that candidate files will be read several times (especially with the optimization described in Section 4.4) this seems preferable.

The search begins with the insertion of the initial state in the appropriate queue file. Each level l is then processed in two steps[2].

Procedure EXPAND first reads states from the queue and inserts their successors into the cache. If $\mathcal{Cache}$ becomes full (lines 4-8) a state s'' is chosen, written to the appropriate temporary file if not stored yet and deleted from $\mathcal{Cache}$. Once this expansion phase terminates unstored states residing in the cache are written in temporary files (lines 11-14). At this point these will trivially contain all the states of $\mathscr{R}(l)$ and may contain several occurrences of the same state. It may happen if a state is removed from the cache and reached again later within the same expansion phase.

The MERGE procedure decides whether it will perform duplicate detection or postpone it to a future level (line 1) and then processes partitions one by one.

States of the temporary file are first hashed to memory in table $\mathcal{H}$, hence detecting duplicates in this file (line 4).

If duplicate detection is delayed (lines 5-6), the content of $\mathcal{H}$ is written back to the candidate file in order to be processed during the next duplicate detection.

Otherwise (lines 7-11) the states of the candidate file are also hashed to memory. The visited states of this partition are read from disk and deleted from $\mathcal{H}$. States in $\mathcal{H}$ are therefore new and written to the visited file. Note that a boolean value is associated to the states of $\mathcal{H}$ in order to identify states in front of the candidate set that will be expanded at the next level (the ones in the temporary file) from those of previous levels which have already been expanded (the ones of the candidate file).

The last step (lines 12-13) consists of writing in the queue file the front states of the candidate set identified as new so that they can be expanded later.

4.3 Deciding when to perform duplicate detection

One question still remains: when should we perform or postpone duplicate detection. Delaying detection comes at the cost of possibly revisiting some duplicates while performing it requires to read the whole visited and candidate sets from disk. The underlying principle of the decision procedure is therefore to delay the detection as long as it estimates that the number of duplicates visited so far is too small to justify a duplicate detection. It is of course impossible to know the number of duplicates in the candidate set as it would require to actually perform the detection, but we can still estimate it from our knowledge of the graph structure.

Visiting a duplicate is an expensive task as it implies many costly operations. First, the state must be written (read) to (from) the queue file and the candidate file. Then it is expanded[3] and its successors are written to temporary files to

[2] It is actually possible to merge both procedures to save some disk accesses but we separated them for sake of clarity.

[3] An expansion implies several non trivial operations that represent the most time consuming tasks of RAM based model checkers: computation of enabled actions,

```
BFS-DDDD ()
for $i \in 1..N$ do
    $\mathcal{V}_i := \emptyset$ ; $\mathcal{C}_i := \emptyset$ ; $\mathcal{Q}_i := \emptyset$
$\mathit{Cache} := \emptyset$ ; $\mathcal{Q}_{h(s_0)}.\mathit{write}(s_0)$
while $\exists i \in 1..N$ with $\mathcal{Q}_i \neq \emptyset$ do
    EXPAND() ; MERGE()
```

```
MERGE ()
$\mathit{detection} := \mathit{doDetection}()$
for $i \in 1..N$ do
    $\mathcal{H} := \emptyset$ ; $\mathcal{Q}_i := \emptyset$
    for $s \in \mathcal{T}_i$ do $\mathcal{H}.\mathit{insert}(s, \mathit{true})$
    if $\neg\mathit{detection}$ then
        for $(s, \_) \in \mathcal{H}$ do $\mathcal{C}_i.\mathit{write}(s)$
    else
        for $s \in \mathcal{C}_i$ do $\mathcal{H}.\mathit{insert}(s, \mathit{false})$
        for $s \in \mathcal{V}_i$ do $\mathcal{H}.\mathit{delete}(s)$
        for $(s, \_) \in \mathcal{H}$ do $\mathcal{V}_i.\mathit{write}(s)$
        $\mathcal{C}_i := \emptyset$
    for $(s, \mathit{exp}) \in \mathcal{H}$ do
        if $\mathit{exp}$ then $\mathcal{Q}_i.\mathit{write}(s)$
```

```
EXPAND ()
for $i \in 1..N$ do $\mathcal{T}_i := \emptyset$
for $i \in 1..N, s \in \mathcal{Q}_i, s' \in \mathit{succ}(s)$ do
    if $s' \notin \mathit{Cache}$ then
        if $\mathit{Cache.isFull}()$ then
            $s'' := \mathit{Cache.choose}()$
            if $\neg s''.\mathit{stored}$ then
                $\mathcal{T}_{h(s'')}.\mathit{write}(s'')$
            $\mathit{Cache.delete}(s'')$
        $\mathit{Cache.insert}(s')$
        $s'.\mathit{stored} := \mathit{false}$
for $s \in \mathit{Cache}$ do
    if $\neg s.\mathit{stored}$ then
        $\mathcal{T}_{h(s)}.\mathit{write}(s)$
    $s.\mathit{stored} := \mathit{true}$
```

Fig. 3. The BFS-DDDD algorithm based on dynamic delayed duplicate detection.

be later read again. To estimate both alternatives, the algorithm assigns a cost to each of these basic operations: ec for expansions, rc for read accesses and wc for write accesses[4]. We can thus approximate the cost of visiting a duplicate by $ec + (rc + wc) \cdot (2 + deg)$ where deg is the average degree of the graph. Note that, due to cache effect, not all the successors may be written in the temporary files, but we will still use the average degree as an over approximation.

Since duplicate detection implies the read of the whole visited and candidate files we can therefore estimate that delaying detection should be preferred if:

$$(|\mathcal{C}| + |\mathcal{V}|) \cdot rc > |\mathit{duplicates}| \cdot (ec + (rc + wc) \cdot (2 + deg)) \qquad (1)$$

where $|\mathit{duplicates}|$ is the total number of duplicates in candidate and temporary files.

This is naturally a rather coarse approximation. What really matters in our sense is that the decision procedure makes its choice on the basis of:

- the size of the candidate and visited sets: looking for duplicates is very cheap on the first levels and its cost increase as we go deeper into the graph.
- the proportion of backward transitions, which has a direct impact on the number of duplicates: numerous backward transitions will naturally introduce many useless state revisits which in turn mean additional disk accesses.

generation of successors and insertion of the successors into the cache usually via an encoding into bit strings.

[4] In our implementation we arbitrarily set $rc = 1$, $wc = 2$ and $ec = 2$ which is clearly not the best solution. We propose in section 4.5 a method to set these parameters.

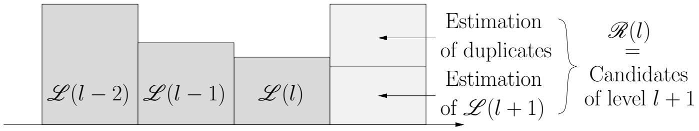

$\mathscr{L}(l-2)$
$\mathscr{L}(l-1)$
$\mathscr{L}(l)$
Estimation of duplicates
Estimation of $\mathscr{L}(l+1)$
$\mathscr{R}(l)$ = Candidates of level $l+1$

Fig. 4. Estimating the number of duplicates in the candidate set.

- the average degree of the graph. This third factor is perhaps less intuitive but still should be considered. A high degree weighs down the cost of visiting duplicates insofar as such a revisit may, in the worst case, lead to approximately *deg* read/write accesses to temporary files.

Some parameters of formula (1) are available, like $|\mathcal{V}|$, $|\mathcal{C}|$ or the average degree *deg* that can be estimated from our partial exploration of the graph. Computing $|duplicates|$ is more problematic as it requires to actually perform duplicate detection which is exactly what we want to avoid. Instead, we approximate it using observation 3 from the previous section: the size of levels does usually progress in a regular way. Thus, we can roughly forecast the size of a level from previous ones. We can then reasonably assume that the difference between what we expected and the actual size of the candidate set can be explained by the presence of duplicate candidates.

This forecast process can be illustrated with the help of figure 4. We notice a decrease in previous levels $l-2$, $l-1$ and l. By making the hypothesis that this trend will continue we forecast the size of level $l+1$. We then deduce from this forecast an estimation of the number of duplicates.

Thereafter, we shall denote by $lpr_i = \frac{|\mathscr{L}(i+1)|}{\mathscr{L}(i)}$ the level progression rate of level i, l the level of the last detection and $l{+}n$ the current level. The estimation of some value v will be denoted by $\overline{v}$. The number of duplicates stored in candidate and temporary files is estimated as follows.

$$\overline{|duplicates|} \;=\; \sum_{i=1}^{n} cr_{l+i} \cdot \min(0, |\mathscr{R}^i(l)| - \overline{|\mathscr{L}(l+i)|}) \tag{2}$$

A correction rate cr_{l+i}, which purpose will be made clear thereafter, is used to over-approximate our estimation.

The size of some level $l+i$ is then estimated from the last level we actually measured, i.e., level l, by combining it with the successive (estimated) progression rates $\overline{lpr_l}$, …, $\overline{lpr_{l+i-1}}$.

$$\overline{|\mathscr{L}(l+i)|} = |\mathscr{L}(l)| \cdot \prod_{j=l}^{l+i-1} \overline{lpr_j} \tag{3}$$

This simplifies our task since the level progression rate can be estimated from the data previously collected on the graph structure and the size of $\mathscr{R}^i(l)$ is available.

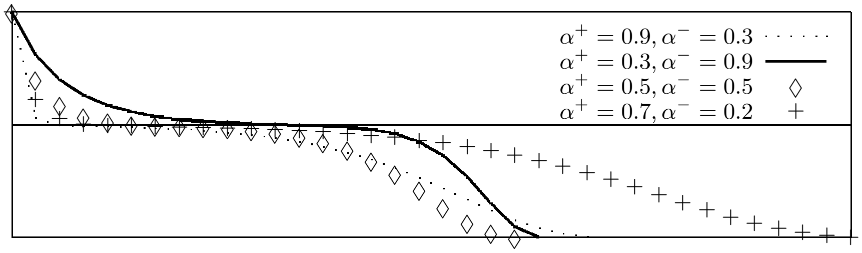


Fig. 5. Curve used to estimate the progression of levels

Estimation of $|\mathscr{R}^i(l)|$ For any $i < n$, $\mathscr{R}^i(l)$ is stored in candidate files and we can hence easily know its size. $\mathscr{R}^n(l)$ is stored in temporary files and as the cache may not be large enough, $\mathcal{T}$ is actually a multi-set of $\mathscr{R}^n(l)$. Computing the actual size of $\mathscr{R}^n(l)$ would require to merge temporary files, i.e., remove duplicate in these, before calling *doDetection*, which is a non trivial operation. The solution we implemented is to only merge a few files in order to have an idea of the average multiplicity of each state of $\mathscr{R}^n(l)$ in $\mathcal{T}$ and hence, a more accurate estimation of $|\mathscr{R}^n(l)|$. Note that if the cache is large enough to contain $\mathscr{R}^n(l)$ then $|\mathcal{T}| = |\mathscr{R}^n(l)|$ and this operation is not necessary.

Estimation of the level progression rate Using observation 3 made in Section 3, we will assume that the progression rate evolves in a regular way and can be captured through the following formula.

$$lpr_{i+1} = \begin{cases} lpr_i \cdot (lpr_i + \epsilon)^{-\alpha^+} \ \textbf{if}\ lpr_i \geq 1 \\ lpr_i \cdot (lpr_i - \epsilon)^{+\alpha^-} \ \textbf{else} \end{cases} \tag{4}$$

where $\alpha^+ \in [0,1]$ (resp. $\alpha^- \in [0,1]$) determines how fast a progression rate greater than 1 (less than 1) drops down to 1 (0); and ϵ is used, first to ensure that the rate will eventually reach 1 and later 0, and second to determine how long the progression rate will stay around 1. In our implementation, we set its value to 0.01.

We use two different values α^+ and α^- as, in general, the progression rate decreases faster when it is above 1 than below. Therefore it is preferable that $\alpha^+ > \alpha^-$. In our implementation, we set $\alpha^+ = 0.7$ and $\alpha^- = 0.2$ as, on the average, these gave us the best results. To give to the reader an idea of the progression induced by this formula, we have plotted in figure 5 the curves for some values of α^- and α^+.

The progression rate of the current level can then be forecasted from the previous one using formula (4). If a detection occurred on the current level k, the forecast is made from the actual rate since we measured $\mathscr{L}(k-1)$ and $\mathscr{L}(k)$. Otherwise, it is based on the forecasts made on previous levels.

Correcting the estimation It is clear that the quality of our estimations will degrade as we move away from the level of the last detection. As a defensive approach we will over-approximate our estimation of the number of duplicates by a correction factor that grows exponentially as we delay detection:

$$cr_{l+i} = (1+\beta)^{i-1} \tag{5}$$

Hence, we will avoid long series of levels without detection. As on the first levels following a detection the estimation is usually satisfactory (see experience 1 in Section 5), β should be set to a very small value, e.g, 0.02 in our implementation.

4.4 Performing partial duplicate detections

A strategy allowing to reduce revisits is, when detection is delayed, to select a subset of disk states and perform a *partial duplicate detection*. The question is how to select these states in such a way that it helps us to delete many duplicates while still not asking for too much work with respect to a full duplicate detection.

In [17], it was suggested to select states randomly according to the locality principle. As previously suggested by Pelánek in [14], our algorithm rather exploits the second statistical fact described in section 3: backward transitions usually have a few typical lengths. Hence, using our knowledge of the graph we know in which part of the file we should look to delete many duplicates. To this end, the algorithm records for each partition p and level l the position in the visited or candidate file of the first state of level l in partition p. States of a given level can then be recovered using a seek operation in the appropriate file. After each full duplicate detection we then select the k most typical lengths observed so far (k being a user defined parameter) and use these to select stored states during the next partial duplicate detections.

4.5 Extensions

We discuss in this section two possible extensions to the method.

Sampling the state space BFS-DDDD assigns a cost to each basic operation (wc for a write access, rc for a read access and ec for a state expansion) to decide when to perform duplicate detection. For ideal performance these should be tuned according to the specific characteristics of the model. The value rc and wc should indeed reflect the size of the state vector while ec should be set according to the state generation speed. A possible way to address this problem is to perform a first "training run" using only RAM, as in [6], to collect some data on the model that can be used for a second run using BFS-DDDD. Not only is it useful to tune these parameters, but it can also give us some precious knowledge of the graph structure, e.g., on the length of backward transitions, that could later be exploited by BFS-DDDD.

Profiting from a static analysis of the model Backward transitions are often triggered by some specific higher level transitions in the system specification. For instance, `end loop` statements often close cycles and hence are the source of backward transitions. On the opposite, some actions, such as variable incrementations, will never generate backward transitions. It could thus be interesting to perform a static analysis of the model prior to state space exploration to identify actions that could potentially lead to such transitions. These data can then be used by BFS-DDDD to estimate the probability of a newly generated candidate to be a duplicate, depending on the actions associated to the incoming arcs of the state. For states marked as "probably duplicate" it may be interesting to delay their expansion to the next duplicate detection (if the detection revealed that it is actually not a duplicate) rather than expanding them at the next expansion phase. Once again, a first training run may be useful to identify more accurately those actions in the models.

5 Experiments

The BFS-DDDD algorithm has been integrated into the ASAP verification tool [13]. We report in this section the results of a series of experiments. All models are taken from the BEEM database. Some additional data on experience 3 and 4 may be found in [3].

Experiment 1 One may wonder how the algorithm used to estimate the number of duplicates works in practice. To this end, we first selected 204 instances (all instances with at most 5,000,000 states) and compared their BFS level graphs with the graphs forecasted during the search. We measured the error rate

$$\text{err} = \frac{\sum_{i \in \{1,\dots,n\}} \text{abs}\left(\sum_{j=l_{i-1}+1}^{l_i} \overline{|\mathscr{L}(j)|} - |\mathscr{L}(j)|\right)}{\sum_i |\mathscr{L}(i)|}$$

where $l_0 = -1$ and $l_1, \dots, l_n$ denote levels where detections were performed. The principle of this rate is to observe for each slice $[l_i + 1, l_{i+1}]$ of levels closed by a duplicate detection the distance between the number of states estimated and the number of states actually measured. If detections are performed on each level, using BFS-HDDD, we obtain an average error rate of 0.10 which basically means that our method to evaluate duplicates is viable: it is possible to accurately evaluate a BFS level based on prior levels. With BFS-DDDD the average goes up to around 0.18 which is still rather good. Figure 6 presents some comparisons between the actual BFS level graph (plain curve) and the graph forecasted (dotted curve) using BFS-DDDD. We drew a vertical line for each level with a detection. The graph of `cambridge.4` has a regular bell shape and a large proportion of backward transitions which leads to frequent detections. Our estimation is thus excellent and we can estimate that detection frequency is almost optimal. On the contrary, for `firewire_link.2`, we often over-approximate levels. As a consequence, detections are too largely spaced and too many states are revisited.

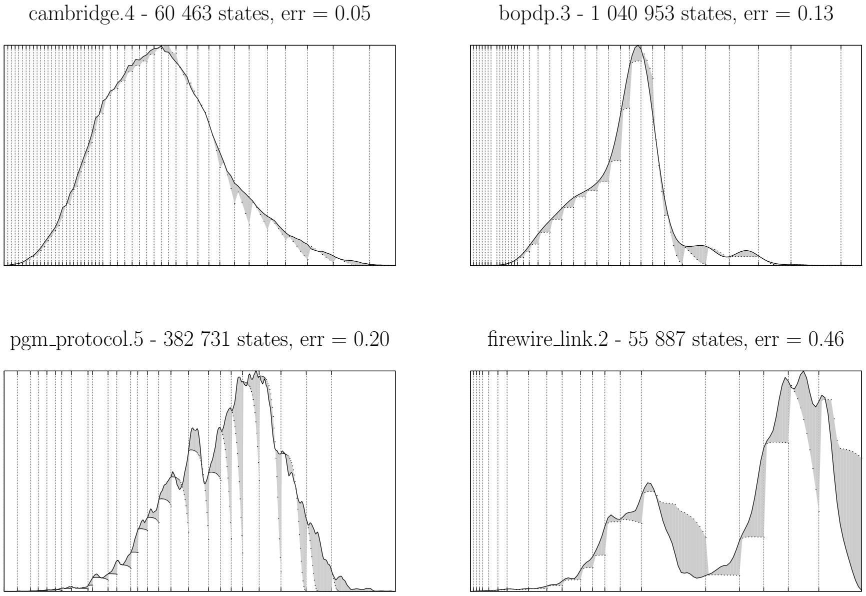


Fig. 6. Experiment 1: comparison of the BFS level graph with our forecast.

However, the opposite situation is much more frequent: we generally tend to under-approximate levels (and forecast too much duplicates) and perform unnecessary detections. Hence, our strategy is perhaps not aggressive enough.

Experiment 2 BFS-DDDD was then compared to BFS-HDDD. We selected all the non trivial (with more than 500.000 states) instances of the database and measured the execution times with both algorithms. We plotted the results in figure 7. Each point corresponds to an instance. Data collected confirm our initial expectations: BFS-DDDD is especially interesting for long graphs, i.e., with a large height. This is however not the only parameter: the proportion of backward transitions also plays an important role. For example, in the case of `cambridge.7`, it is not so interesting to use BFS-DDDD: even though its graph is long, it has a high proportion of backward transitions ($> 50\%$) which leads us to perform many duplicate detections; whereas we observe good performances for `leader_election.5` which has the opposite characteristics. Rates observed go from 1.10 (`reader_writer.3`) to 0.05 (`iprotocol.6`). The average is around 0.6. This does not look like an important improvement but, as we shall see it, the major interest of our algorithm is that is scales much better than BFS-HDDD to large state spaces. Moreover the next experiment shows that partial detections often allow to delete many duplicates at a low cost.

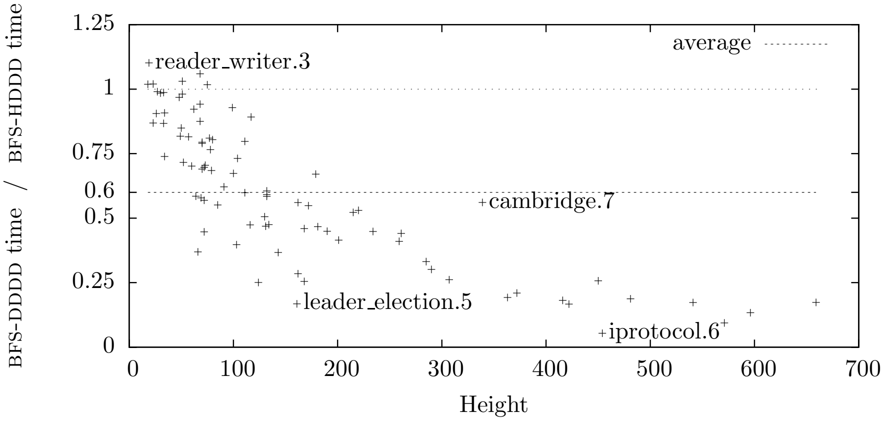


Fig. 7. Experiment 2: comparison of BFS-DDDD and BFS-HDDD

Experiment 3 We then compared our algorithm to PART [1], based on a partitioned hash table. This choice is mostly motivated by the fact that, according to our experiments and the ones of Bao and Jones, it outperforms the algorithms of [2] and [17] that can be considered, roughly speaking, as parents of BFS-DDDD.

To compare these algorithms we tried to select a representative set of models according to these parameters: average degree, width and height, proportion of backward transitions, distribution of backward transition lengths. As we previously saw these may have a certain impact on the performance of our algorithm.

Table 1 presents the results of this experiment. For each instance we performed one run with PART and several runs with BFS-DDDD, first without partial detection and then with partial detections with different values for parameter k (number of lengths considered). Each run was given the same amount of memory, that is, the ability to keep at most $16{\cdot}10^6$ states in memory. This represents, in most cases, a small fraction of the state space. Execution times are expressed in the form *hours:minutes* for PART and as a fraction of this time for BFS-DDDD. Best times have been written in bold. The last row indicates average values.

We noticed that PART is very sensitive to the graph structure: it is best suited to wide and short graphs. In this case queues associated with partitions are filled with many states meaning that few partition swaps will occur although disk queues will be accessed more frequently. Therefore, we can basically make the same observations as in the previous experiment: BFS-DDDD is comparatively better on long graphs (e.g., `brp2`, `iprotocol`, `rether`), with preferably, few backward transitions. Loading/unloading partitions is a major time consuming operation of PART for these kinds of graphs, especially if the state vector is large, e.g., for `rether`. `telephony` is typically the worst input we can think of for BFS-DDDD. It is short and has 16 levels with more states than the cache can hold. Since its average degree is high this leads to a huge amount of disk accesses in temporary files. To a lesser extent, the same remark also applies to `elevator`.

Table 1. Experiment 3: comparison of BFS-DDDD and PART

Instance	States	PART	BFS-DDDD	BFS-DDDD + partial DD		
				k=2	k=4	k=8
anderson	538 M	17 : 56	0.80	**0.28**	**0.28**	**0.28**
bakery	403 M	7 : 36	0.65	**0.47**	**0.47**	**0.47**
brp2	145 M	20 : 08	**0.20**	**0.20**	0.21	0.22
cambridge	255 M	32 : 28	0.62	0.44	**0.28**	0.30
collision	972 M	25 : 34	0.71	**0.67**	0.69	0.69
elevator	833 M	17 : 23	1.16	0.86	0.85	**0.82**
iprotocol	706 M	31 : 01	0.35	0.30	**0.29**	0.32
lann	421 M	23 : 28	**0.47**	**0.47**	0.49	0.52
leader_filters	431 M	7 : 24	**0.38**	**0.38**	**0.38**	**0.38**
lup	379 M	8 : 21	1.03	**0.29**	0.31	0.32
peterson	142 M	3 : 20	0.76	**0.71**	0.73	0.77
rether	151 M	54 : 08	0.07	0.07	0.07	0.08
telephony	534 M	12 : 57	1.01	0.98	**0.95**	**0.95**
train-gate	478 M	26 : 34	**0.24**	**0.24**	0.25	0.31
			0.60	0.45	0.45	0.46

Performing partial detections is especially interesting when backward transitions have very few typical lengths. `anderson` is a caricatural case as all its backward transitions have the same length. Therefore partial detections helped us to divide the execution time by almost three. This also applies to `lup` which graph has two lengths that cover more than 90% of backward transitions. For graphs that do not have typical lengths, e.g., `train-gate` or `brp2`, this optimization does not bring any improvement. Hence, it should always be turned on: at worst, we will not gain anything.

Experience 4 The last experiments were done with 6 real life protocols: `brp`, `brp2` (timed version of `brp`), `cambridge`, `iprotocol`, `pgm_protocol` and `rether`. Our goal was to evaluate how PART, BFS-HDDD and BFS-DDDD behave as the graph gets larger and the height increases.

Figure 8 presents our results. We gave each algorithm the same amount of memory and for BFS-DDDD, we set parameter k to 4 as it gave us the best results in previous experiment. For some fixed parameters we progressively increased another parameter (on the x-axis, see [3] for details) and recorded the search speed as $\frac{\text{nodes of the graph}}{\text{search time}}$ (on the y-axis). We remark that PART is more efficient for small graphs although there is no huge difference. However, above a certain point BFS-DDDD becomes more interesting. In addition, even though there generally is a moment where the speed decreases for all algorithms, this instant occurs later for BFS-DDDD. At last, many communication protocols have in common that their BFS level graphs are terminated by a long series of very small levels - that possibly correspond to the termination phase of the protocol. This property also explains why BFS-DDDD evolves better on such models.

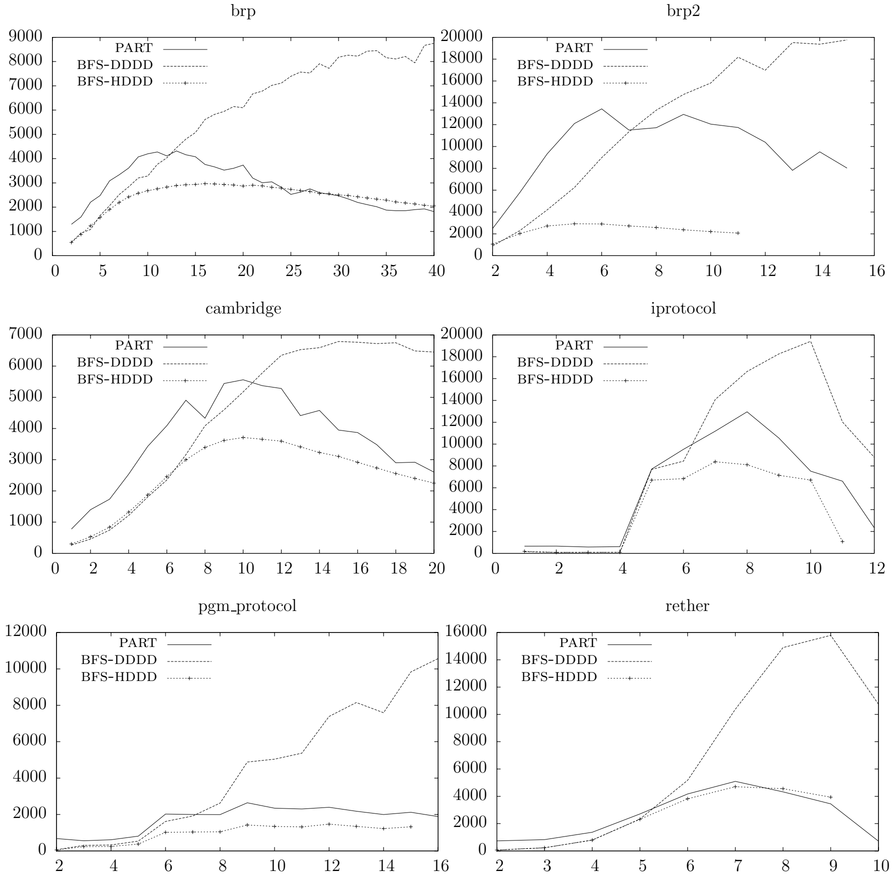


Fig. 8. Experiment 4: Comparison of PART, BFS-HDDD and BFS-DDDD.

6 Conclusion

We proposed in this work an adaptive duplicate detection scheme for external memory model checking that borrows several ideas from the literature: [10,14,17]. Its principle is to collect during the search some data that help us to determine when detection should be performed or postponed. We evaluated this method on several models of different families and complexities and find out that the new algorithm is especially well suited to communication protocols with long graphs and few backward transitions that are quite common in model checking.

Besides the extensions described in Section 4.5, we plan to refine the way lengths are selected during partial duplicate detections. Our scheme assumes that these lengths do not evolve during the search. This assumption is apparently invalid in many cases where lengths are function of the level.

References

1. T. Bao and M. Jones. Time-efficient model checking with magnetic disk. In *Proc. of TACAS*, LNCS vol. 3440, p. 526–540. Springer, 2005.
2. D. L. Dill and U. Stern. Using magnetic disk instead of main memory in the Murϕ verifier. In *Proc. of CAV*, LNCS vol. 1427, p. 172–183. Springer, 1998.
3. S. Evangelista. Dynamic delayed duplicate detection for external memory model checking. Technical report, DAIMI, University of Aarhus, Denmark, 2008. `http://daimi.au.dk/~evangeli/doc/dddd.pdf`.
4. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, LNCS vol. 1032. Springer, 1996.
5. M. Hammer and M. Weber. "To store or not to store reloaded": Reclaiming memory on demand. In *Proc. of FMICS*, LNCS vol. 4346, p. 51–66. Springer, 2006.
6. G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of the Third Spin Workshop*, 1997.
7. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Proc. of VMCAI*, LNCS vol. 3385, p. 313–329. Springer, 2005.
8. S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *Proc. of VMCAI*, LNCS vol. 3855, p. 237–251. Springer, 2006.
9. R. E. Korf. Delayed duplicate detection: Extended abstract. In *Proc. of IJCAI*, p. 1539–1541. Morgan Kaufmann, 2003.
10. R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proc. of AAAI*, p. 650–657. AAAI Press/The MIT Press, 2004.
11. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *Proc. of AAAI*, p. 1380–1385. AAAI Press/The MIT Press, 2005.
12. R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *J. ACM*, 52(5):715–748, 2005.
13. L.M. Kristensen and M. Westergaard. The ascoveco state space analysis platform. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, p. 1–6, 2007.
14. R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN*, LNCS vol. 2989, p. 5–22. Springer, 2004.
15. R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proc. of SPIN*, LNCS vol. 4595, p. 263–267. Springer, 2007.
16. R. Pelánek. Model classifications and automated verification. In *Proc. of FMICS*, LNCS. Springer, 2007.
17. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in the disk based Murphi verifier. In *Proc. of FMCAD*, LNCS vol. 2517, p. 202–219. Springer, 2002.
18. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting transition locality in automatic verification. In *Proc. of CHARME*, LNCS vol. 2144, p. 259–274. Springer, 2001.
19. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, p. 429–528. Springer, 1998.
20. R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. of AAAI*, p. 683–689. AAAI Press/The MIT Press, 2004.
21. R. Zhou and E. A. Hansen. A breadth-first approach to memory-efficient graph search. In *Proc. of AAAI*. AAAI Press/The MIT Press, 2006.
22. R. Zhou and E. A. Hansen. Domain-independent structured duplicate detection. In *Proc. of AAAI*. AAAI Press/The MIT Press, 2006.
23. R. Zhou and E. A. Hansen. Edge partitioning in external-memory graph search. In *Proc. of IJCAI*, p. 2410–2417, 2007.