Large-Scale Directed Model Checking LTL

Stefan Edelkamp and Shahid Jabbar

University of Dortmund
Otto-Hahn Straße 14
{stefan.edelkamp,shahid.jabbar}@cs.uni-dortmund.de

Abstract. To analyze larger models for explicit-state model checking, *directed model checking* applies error-guided search, *external model checking* uses secondary storage media, and *distributed model checking* exploits parallel exploration on multiple processors.
In this paper we propose an external, distributed and directed on-the-fly model checking algorithm to check general LTL properties in the model checker SPIN. Previous attempts are restricted to checking safety properties. The worst-case I/O complexity is bounded by $O(sort(|\mathcal{F}||\mathcal{R}|)/p + l \cdot scan(|\mathcal{F}||\mathcal{S}|))$, where $\mathcal{S}$ and $\mathcal{R}$ are the sets of visited states and transitions in the synchronized product of the Büchi automata for the model and the property specification, $\mathcal{F}$ is the number of accepting states, l is the length of the shortest counterexample, and p is the number of processors. The algorithm we propose returns minimal lasso-shaped counterexamples and includes refinements for property-driven exploration.

1 Introduction

The core limitation to the exploration of systems are bounded main memory resources. Relying on virtual memory slows down the exploration due to excessive page faults. External algorithms [31] exploit hard disk space and organize the access to secondary memory. Originally designed for explicit graphs, external search algorithms have shown considerably good performances in the large-scale breadth-first and guided exploration of games [22, 12] and in the analysis of model checking problems [24][1].

A Directed explicit-state model checking [13] enhances the error-reporting capabilities of model checkers. The application of guided search for checking liveness properties is restricted to the reduction of trails [14].

Distributed explicit state model checking [9, 25] uses several processors working in parallel to enhance the exploration of larger models.

In [18] we have given a first report on combining directed, parallel and external explicit-state model checking to enhance the search for minimal counterexamples for safety errors. Under certain assumptions on the distribution of

[1] An anonymous referee has pointed us to the work of Roscoe: *Model Checking CSP* in *A Classical Mind, Essays in Honour of C.A.R. Hoare*, Prentice-Hall 1994, which also introduces to the idea of external model checking for the FDR system. Unfortunately, we haven't been able to access the reference.

the applied hash function and the number of file pointers we showed that the approach uses linear, i.e., $O(scan(|\mathcal{S}| + |\mathcal{R}|))$ I/Os. In a sequential setting, for safety explicit-state model checking state-space graphs with bounded locality we arrive at $O(sort(|\mathcal{R}|) + scan(|\mathcal{S}|))$ I/Os, which is optimal [12].

The goal of this work is to extend this work to the exploration for checking liveness properties. The main challenge for distributed and external on-the-fly model checking is that the depth-first traversal of the global state space graph as used in *Nested-DFS* (an on-the-fly variant of Tarjan's algorithm [35]) is not efficient. All attempts to solve this problem via variants of breadth-first search [7, 4, 9] arrive at a time complexity that is non-linear in the size of the model. The approach we propose in this paper is based on a translation procedure of liveness problems into safety problems [32]. The translation approach has the advantage that the underlying algorithm design to detect safety errors has not to be changed. More crucially, the approach includes a rich state description which allows to express lower bounds for cost-optimal guided search. To enhance the exploration, we additionally exploit the *never-claim* automaton structure of the temporal property to be satisfied.

The paper is structured as follows. First we briefly review explicit-state LTL model checking using Büchi automata. Then we consider distributed model checking together with its limits and possibilities. Afterwards we introduce to external model checking safety properties and delayed duplication detection. We first consider breadth-first implicit graph search. Next we turn to the guided search, recalling the algorithm *External A**. The upcoming section points out the problems in externalizing standard DFS model checking algorithms. This leads to the proposed approach for I/O efficient parallel external model checking. We provide monotone heuristics for optimal counterexample search and give empirical data for checking LTL formulae in an external and parallel variant of the SPIN model checker. Finally, we draw conclusions and indicate further research avenues.

2 Explicit-State Model Checking

In automata-based model checking, both the model to be analyzed and the specification to be checked are modeled as non-deterministic *Büchi automata*. Syntactically, Büchi automata are ordinary automata. For accepting *infinite words*, or *runs*, a different acceptance condition is applied. Let ρ be a run and $inf(\rho)$ be the set of states reached infinitely often in ρ, then a Büchi automaton accepts, if the intersection between $inf(\rho)$ and the set of final states F is not empty.

2.1 Automata-based LTL Model Checking

The desired property of the system is specified in some form of temporal logic. We briefly introduce *linear temporal logic (LTL)*. A path in model $\mathcal{M}$ is a sequence of states $\pi = S_0, S_1, \ldots$ and π^i denotes the suffix of π starting at S_i. LTL formulae have the form "Always f", where f is a *path formula*. If p is an atomic

proposition then p is a path formula. If f and g are path formulae so are $\neg f, f \vee g, f \wedge g, \mathbf{X}\ f, \mathbf{F}\ f, \mathbf{G}\ f$, and $f\ \mathbf{U}\ g$.

Transforming the model and the specification into Büchi automata assumes that systems can be modeled by automata, and that the LTL formula can be transformed into an equivalent Büchi automaton. The converse is not always possible, since Büchi automata are clearly more expressive than LTL expressions [36]. Checking correctness is reduced to checking language emptiness. More formally, the model checking procedure validates that a model represented by an automaton $\mathcal{M}$ satisfies its specification represented by an automaton $\mathcal{S}$. The task is to verify if $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$. In words: the *language accepted by the model is included in that of the specification.* We have $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$ if and only if $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$. In practice, checking language emptiness is more efficient than checking language inclusion. Büchi automata are closed under intersection and complementation [8], so that there exists an automaton that accepts $\overline{\mathcal{L}(\mathcal{S})}$ and an automata that accepts $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})}$. It is possible to complement Büchi automaton equivalent to an LTL formula, *but* the worst-case running time of such a construction is double-exponential in the size of the formula. Therefore, in practice one constructs the *never-claim* automaton for negation of the LTL formula, avoiding complementation.

The product is *synchronous*, that is each transition in one automaton implies one in the other. The property automaton is non-deterministic, such that both the successor generation and the temporal formula representation may introduce branching to the overall exploration. The construction assumes that all states in the model are accepting. If arbitrary Büchi automata are intersected, *extended acceptance conditions* are required [11].

For *checking emptiness* we have to check that the automaton accepts no word. *Accepting runs* are present in the automaton if the strongly connected components (SCCs) reachable from the initial state contain at least one accepting state. In this case, a reachable cycle contains at least one accepting state. Checking language emptiness corresponds to the validation that no such cycle exists.

2.2 Tarjan's Algorithm

For finding accepting cycles, we analyze the state space graph structure; more precisely, the strongly connected components, SCCs for short. An algorithm to compute all such components of a graph in linear time is Tarjan's algorithm [35]. The algorithm is divided into four stages. In the first stage, a DFS starting at the initial state computes the discovery and finishing times $t_d(u)$ and $t_f(u)$ for each visited state u, which corresponds to the time, when node u is entered and left. The second stage computes the inverse of the graph. In the third stage, a series of DFSs considers the nodes in order of decreasing t_f-value. The fourth and last stage outputs the nodes of each tree in the DFS forest of the third stage as a strongly connected component.

2.3 Nested DFS

On-the-fly model checking is an efficient way to perform model checking. It computes the global state transition graph during the construction of the intersection. The advantage is that only a part of the state space is constructed, which is needed in order to check the desired property.

For checking the synchronous product graph of the model and the specification for accepting cycles on-the-fly, *nested-depth-first search* has been proposed [17]. It explores the state space in a depth-first manner, stores visited states in a visited list, marks states which are on the current search stack, and invokes a secondary DFS starting at accepting states after they have been fully explored in the primary DFS. The secondary DFS explores states already visited by the primary search but not by any secondary search; states visited by the second search are *flagged* and if a state is found on the stack of first search, an accepting cycle is found. Typical implementations use 2 bits per state, one for marking, one for flagging. As with Tarjan's algorithm its worst-case is linear in the size of the intersected state transition graph, but it is capable of reporting counter-examples before the entire state space has been seen.

Property-driven or *improved nested-depth-first search* [3, 25] partitions the never-claim into SCCs. The main observation is that cycles in the state transition graph of the intersection of the system $\mathcal{M}$ and the never-claim automaton $\mathcal{N}$ is accepting only if the corresponding cycle in $\mathcal{N}$ is accepting. Therefore, these approaches use Tarjan's algorithm to analyze never-claim. An SCC in $\mathcal{N}$ is called *non-accepting* if none of its states is accepting; *fully-accepting*, if each cycle formed by states of the SCC is accepting, and *partially-accepting*, otherwise. Improved nested DFS partitions the never-claim into SCCs and applies secondary search only in case of partially accepting cycles.

3 Distributed Model Checking LTL

Liveness property validation based on DFS appears to be an inappropriate choice for distributed model checking. For distributed model checking the core reason is that in contrast to BFS, DFS appears to be inherently sequential [29]. Different attempts have been suggested to allow an efficient parallelization for model checking liveness. Unfortunately, none of the approaches guarantee a linear time complexity.

3.1 Breadth-First LTL Model Checking

A line of research tries to avoid *nested depth-first search* by studying variants of breadth-first search [5, 4, 7]. The approach presented in [5, 4] invokes a secondary search for detecting cycles from BFS *backward edges*, i.e., transitions encountered in the overall state space that link states in larger, together with (already explored) states in smaller depth. Those backward edges may potentially spawn cycles and are searched individually. If no accepting cycle is found the depth

bound is increased. The number of backward edges is reduced by similar observations as in improved nested depth-first search. The worst case time complexity is $O(|\mathcal{R}| \cdot (|\mathcal{S}| + |\mathcal{R}|))$. The approach allows *on-the-fly* model checking and is compatible with a limited form of partial order reduction. In [7], instead of backward edges, predecessor acceptance is chosen for an $O(|\mathcal{R}|^2 + |\mathcal{S}|)$ algorithm.

3.2 Explicit Fair Cycle Detection

In [9], the symbolic OWCTY[2] algorithm [15] is converted into an explicit one. Similar to Tarjan's algorithm, the approach computes the entire reachability set before extracting the cycle. Unlike Tarjan's algorithm, the order of the exploration does not matter. Next, a loop alternates between a *reachability* and *elimination phase* unless a fixpoint is reached. In the first phase, fair states are checked if they can be reached again. In the second phase, states with a determined fair status are eliminated from the search. The worst case number of iterations is bounded by the diameter d of the search space. The explicit state conversion of the approach runs in $O(d \cdot (|\mathcal{R}| + |\mathcal{S}|))$ time and has been exploited to perform distributed model checking. Cycle extraction for counter-example generation runs in linear time.

4 External Model Checking Safety

I/O-efficient model checking algorithms explicitly manage the memory hierarchy and can lead to substantial speedups compared to caching and pre-fetching heuristics of the underlying operating system, since they are more informed to predict and adjust future memory access.

The standard model for comparing the performance of external algorithms consists of a single processor, a small internal memory that can hold up to M data items, and an unlimited secondary memory. The size of the input problem (in terms of the number of records) is abbreviated by N. Moreover, the *block size* B governs the bandwidth of memory transfers. It is often convenient to refer to these parameters in terms of blocks, so we define $m = M/B$ and $n = N/B$. It is usually assumed that at the beginning of the algorithm, the input data is stored in contiguous blocks on external memory, and the same must hold for the output. Only the number of block reads and writes are counted, computations in internal memory do not incur any cost. The single disk model for external algorithms has been invented by [2]. An extension of the model considers D disks that can be accessed simultaneously. When using multiple disks in parallel, the technique of *disk striping* can be employed to essentially increase the block size by a factor of D. Successive blocks are distributed across different disks.

It is convenient to express the complexity of external-memory algorithms using a number of frequently occurring primitive operations. The simplest operation is *scanning*, which means reading a stream of records stored consecutively on

[2] Acronym for *One Way to Catch them Young*

secondary memory. In this case, it is trivial to exploit disk- and block-parallelism. The number of I/Os is *scan(N)* $= \Theta(\frac{N}{DB}) = \Theta(\frac{n}{D})$. Another important operation is external *sorting*. The proposed algorithms fall into two categories: those based on the *merging* paradigm, and those based on the *distribution* paradigm. The algorithms' complexity is *sort(N)* $= \Theta(\frac{N}{DB} \log_{M/B} \frac{N}{B}) = \Theta(\frac{n}{D} \log_m n)$.

4.1 External BFS

Recall the standard internal-memory BFS algorithm: it visits each node $v \in V$ of the input problem graph G in a one-by-one fashion, as stored in a FIFO queue. After a node v is extracted, its adjacency list (the sets of neighbors in G) is examined, and those of them that haven't been visited so far are inserted into the queue in turn. In external search the internal queue is substituted with a file. Naively running the standard internal-BFS algorithm in the same way in external memory will result in $\Theta(|\mathcal{S}|)$ I/Os for unstructured accesses to the adjacency lists, and $\Theta(|\mathcal{R}|)$ I/Os for finding out whether neighboring nodes have already been visited. The explicit external graph algorithm of [27] improves on the latter complexity for the case of undirected graphs, in which duplicates are constrained to be located in adjacent levels. After the preprocessing step the graph is stored in adjacency-list representation, it generating the multi-set of neighbors for each BFS-level followed by a duplicate elimination phase. Duplicate elimination is realized via external sorting followed by an external scan. External BFS requires $O(|\mathcal{S}| + sort(|\mathcal{R}|))$ time, where $O(|\mathcal{S}|)$ is due to the external representation of the graph and the initial reconfiguration time to enable efficient successor generation.

An implicit variant of the above algorithm algorithm [27] for explicit BFS-search in implicit graphs has been coined to the term *delayed duplicate detection* for *frontier search* [21]. It assumes an undirected search graph. The algorithm maintains BFS layers on disk. Layer $L(i-1)$ is scanned and the set of successors are put into a buffer of size close to the main memory capacity. If the buffer becomes full, internal sorting followed by a duplicate elimination scanning phase generates a sorted duplicate-free state sequence in the buffer that is flushed to disk. The outcome of this phase are k sorted files. In the next step, *external merging* is applied to unify the files into $L(i)$ by a simultaneous scan. The size of the output files is chosen such that a single pass suffices. Duplicates are eliminated. Since the files were sorted, the complexity is given by the scanning time of all files. One also has to eliminate $L(i-1)$ and $L(i-2)$ from $L(i)$ to avoid re-computations; that is, nodes extracted from the external queue are not immediately deleted, but kept until after the layer has been completely generated and sorted, at which point duplicates can be eliminated using a parallel scan. The process is repeated until $L(i-1)$ becomes empty, or the goal has been found. The total execution time is $O(sort(|\mathcal{R}|) + scan(|\mathcal{S}|))$ I/Os. The I/O optimality of External BFS is based on the work of [1], who gave a matching lower bound for external sorting.

External BFS has been successfully applied to fully explore the 15-Puzzle using 1.4 terabytes of hard disk in about three weeks [22]. The algorithm shares similarities with the internal *frontier search* algorithms [23] that were used

for solving multiple sequence alignment problem, an idea that goes back to Hirschberg [16].

4.2 External A*

*External A** [12] maintains the search space on disk. The priority queue data structure is represented as a list of buckets. In the course of the algorithm, each bucket $L(i, j)$ will contain all states u with path length $g(u) = i$ and heuristic estimate $h(u) = j$. We will later discuss how such estimates can be derived in real-time minimum-cost reachability analysis. As same states have same heuristic estimates, it is easy to restrict duplicate detection to buckets of the same h-value. By an assumed undirected, unweighted state space problem graph structure, we can restrict aspirants for duplicate detection further. If all duplicates of a state with g-value i are removed with respect to the levels i, $i-1$ and $i-2$, then no duplicate state will remain for the entire search process. For breadth-first-search in explicit graphs, this is in fact the algorithm of [27]. We consider each bucket as a different file that has an individual internal buffer. A bucket is *active* if some of its states are currently expanded or generated. If a buffer becomes full, then it is flushed to disk.

Since External A* simulates A* and changes only the order of elements to be expanded that have the same f-value, completeness and optimality are inherited from the properties of A*. The I/O complexity for External A* in an implicit unweighted and undirected graph with monotone estimates is bounded by $O(sort(|\mathcal{R}|)+scan(|\mathcal{S}|))$, where $|\mathcal{S}|$ and $|\mathcal{R}|$ are the number of nodes and edges in the explored subgraph of the state space problem graph. It has been shown [12] that the lower bound for the delayed duplicate detection is $\Omega(sort(|\mathcal{S}|))$ I/Os.

*Parallel External A** [18] is a parallel variant of External A* based on queues of working requests. In the exploration stage, each processor flushes the successors with a particular g and h value to an individual file. It has its own hash table and eliminates some duplicates already in main memory. If the output buffer exceeds memory capacity the processor writes the hash table to disk. In a first sorting stage, it sorts its own files. The number of file pointers needed is restricted by the number of flushed buffers. In the distribution stage, a single processor distributes all states in the pre-sorted files into different files according to the hash value's range. As all input files are sorted this is a mere scan. In the second sorting stage, processors externally sort the partially sorted files to find further duplicates. The output of this phase are sorted and partitioned buffers. Using the hash index as the sorting key the concatenation of files is totally sorted.

5 Problems with Externalizing DFS

External depth-first search relies on an external stack data structure. The search stack is small compared to the overall search but in the worst-case it can become large. For an external stack, the buffer is just an internal memory array of $2B$

elements that at any time contains the $k < 2B$ elements most recently inserted. We assume that the stack content is bounded by at most N elements. A *pop* operation incurs no I/O, except for the case where the buffer has run empty, where $O(1)$ I/O to retrieve a block of B elements is sufficient. A *push* operation incurs no I/O, except for the case where the buffer has run full, where $O(1)$ I/O is to retrieve a block of B elements is needed. Insertion and deletion take $1/B$ I/Os in the amortized sense.

The I/O complexity for external DFS for explicit (possible directed) graphs has been shown to be $O(|\mathcal{S}| + |\mathcal{S}|/M \cdot \mathit{scan}(|\mathcal{R}|))$ [10]. There are $|\mathcal{S}|/M$ stages where the internal buffer for the visited state set becomes full, in which case it is flushed and duplicates are eliminated from the external adjacency list representation by a file scan. Visited successors in the unexplored adjacency lists are marked not to be generated again, such that all states in the internal visited list can be eliminated for good. As with External BFS in explicit graphs, value $O(|\mathcal{S}|)$ I/Os is due to the unstructured access to the external adjacency list. Computing SCCs in explicit graphs has the same I/O complexity as DFS, i.e. $O(|\mathcal{S}| + |\mathcal{S}|/M \cdot \mathit{scan}(|\mathcal{R}|))$ I/Os. For implicit graphs as generated for model checking liveness, no access to an external adjacency list is needed, so that the world should look better. Dropping the term of $O(|\mathcal{S}|)$ I/O as with External BFS, however, is a challenge. The major problem for external DFS exploration in implicit graphs is that unseen adjacencies cannot been modeled and there is no time for performing delayed duplicate detection. For implicit graphs this is not available, as we cannot access the search graph that we have not seen so far.

6 Large-Scale Model Checking Liveness

We decided to build our external model checker on top of the *liveness as safety model checking* approach [32]. It proposes to convert a liveness model checking problem into a safety model checking problem by roughly doubling the state vector size and *guessing* the seed of a fairness cycle. More precisely, the proposed extension stores with the current state s a previously seen state s' together with two flags *start-cycle* and *closed-cycle*. The first flag is set to prevent future overwriting of the stored state. The second flag indicates that a second occurrence of s' has been found. Unless the seed of the cycle has not been guessed s equals s'. The initial state is spawned to two states, one attached to (*false*,*false*) and the other attached to *true*,*false*). If $\mathcal{S}$ and $\mathcal{R}$ are the set of states and the set of transitions of the synchronous product of the model and the (never-claim) specification, then $\mathcal{S}$ is searched at most $|\mathcal{S}|$ times, yielding a time complexity of $O(|\mathcal{S}| \cdot (|\mathcal{S}| + |\mathcal{R}|))$.

The most important observation is that based on this extension the exploration algorithms themselves have not (or only in a minor way) to be changed. For example, in [32] the authors show how to extend models using so-called observers and applying the same model checker. In [33] the authors showed that for fairness constraints of the form $\mathbf{F}p$ we have that

$$\rho = (S_1 \dots S_{l-1})(S_l \dots S_{k-1})^\omega$$

is a run in the state space $\mathcal{S}$ if and only if

$$\rho' = (S_0, S_0, 0, 0) \dots (S_{l-1}, S_{l-1}, 0, 0)((S_l, S_l, 1, 0) \dots (S_{k-1}, S_l, 1, 0))^{\omega}(S_l, S_l, 1, 1)$$

is a run in the extended state space $\mathcal{S}'$.

As this construction does not yet record Büchi automaton acceptance conditions for explicit-state model checking, as suggested by [32], we work with a slightly different state description. State pairs in the first phase are called *primary* states, states pairs in the secondary phase are called *sencondary* states. We drop Boolean variables completely as we distinguish primary from secondary states by comparing the state vectors of the state pair. Moreover, we spawn secondary children only at accepting primary states.

Without any heuristic the algorithm executes external breadth-first search, where each iteration can actually be seen as a snapshot in *bounded automata-based model checking*. Bounded model checking [6] uses a propositional SAT solver for the symbolic exploration of model checking problems. It exploits the SATPLAN exploration idea of [20] using a rising search horizon k to generate Boolean formulae encoding the overall exploration problem up the BFS-level k. In bounded automata-based model checking we use a similar approach, but without using BDDs nor SAT-formulae. To avoid traversing the full state space in Tarjan's algorithm, we analyze the cross product graph up to some threshold depth value k. If we find a counter-example already in depth k we terminate, otherwise we increase k. The bounded semantics for this strategy are the same as in BMC [6]: $\pi \models_k^i p$ if and only if $p \in L(p(i))$, $\pi \models_k^i \neg p$ if and only if $p \notin L(p(i))$, $\pi \models_k^i f \wedge g$ if and only if $\pi \models_k^i f$ and $\pi \models_k^i g$, $\pi \models_k^i f \vee g$ if and only if $\pi \models_k^i f$ or $\pi \models_k^i g$, $\pi \models_k^i \mathbf{G}f$ *is always false*, $\pi \models_k^i \mathbf{F}f$ if and only if $\exists j, i \leq j \leq k : \pi \models_k^j f$, $\pi \models_k^i \mathbf{X}f$ if and only if $i < k$ and $\pi \models_k^{i+1} f$, and $\pi \models_k^i f \mathbf{U} g$ if and only if $\exists j, i \leq j \leq k : \pi \models_k^j g$ and $\forall n, i \leq n < j : \pi \models_k^n f$.

Theorem 1. *For problem graphs the external BFS LTL model checking algorithm finds the shortest counterexample with an accepting seed state. Its I/O complexity is* $O(\mathrm{sort}(|\mathcal{F}||\mathcal{R}|) + l \cdot \mathrm{scan}(|\mathcal{F}||\mathcal{S}|))$*, where* l *is the length of the shortest counterexample.*

Proof. Since each state is expanded at most once, all sortings can be done in time $O(sort(|\mathcal{F}||\mathcal{R}|))$ I/Os. Filtering, evaluating, and merging are all available in scanning time of the buckets in consideration. The I/O complexity for predecessor elimination depends on the number of buckets that are referred to during file subtraction/reduction. The number of buckets is bounded by the number of layers and thus the length of the shortest counterexample. Consequently, the I/O complexity for large-scale LTL model checking is bounded by $O(sort(|\mathcal{F}||\mathcal{R}|) + l \cdot scan(|\mathcal{F}||\mathcal{S}|))$ I/Os.

6.1 Heuristics for Safety Model Checking

For defining heuristics for safety model checking, we assume that the global state space is generated based on the asynchronous compositions of local state spaces

$\mathcal{P}_i$, $i \in \{1, \ldots, n\}$, called processes. In other words, each global system state is partitioned into n local states. The state of a local process $\mathcal{P}_i$ is called its *program counter*, $i \in \{1, \ldots, n\}$, pc_i for short.

The *FSM distance heuristic* is defined as the sum for each $\mathcal{P}_i$ of the distance between the local state of $\mathcal{P}_i$ in s and the local state of $\mathcal{P}_i$ in s', i.e.,

$$H_M(s, s') = \sum_{i=1}^{n} D_i(pc_i(s), pc_i(s')),$$

where $D_i(pc_i(s), pc_i(s'))$ denotes the shortest path from $pc_i(s)$ to $pc_i(s')$ in the automaton representation of $\mathcal{P}_i$. The values for D_i are computed prior to the search.

6.2 Trail-directed Heuristics

The FSM distance heuristic assumes that both states s and s' are known to the exploration module. It has mainly been used in *trail-directed search*, where a counter-example to an existing error state is to be shortened. It has also been applied to the verification of liveness properties where the prefix path to the start of the cycle and the accepting cycle itself are shortened in sequence. For this case the distance in the never-claim automaton $\mathcal{N}$ is included as follows

$$H'_M(s, s') = \max\left\{H_M(s, s'), D_{\mathcal{N}}(pc_{\mathcal{N}}(s), pc_{\mathcal{N}}(s'))\right\}.$$

As the product of different processes is asynchronous, it is not difficult to see [26] that the FSM distance is *monotone*, i.e., $H_M(s) - H_M(s') \leq 1$ for each pair (s, s') with s' being the direct successor of s. Monotone heuristics guarantee the optimality of counterexample paths in heuristic search exploration algorithms like A* [28]. It is also not difficult to see that the maximum of two monotone heuristics is monotone. Hence, $H'_M(s, s')$ is also a monotone heuristic for shortening liveness trails.

6.3 Heuristic for LTL Properties

In the extended search space $\mathcal{S}'$ we search for shortest lasso-shaped counterexamples, without knowing the start of the cycle beforehand. We used the monotone heuristic

$$H_a(s) = \min_{s' \in F_{\mathcal{N}}} \{D_{\mathcal{N}}(pc_{\mathcal{N}}(s), pc_{\mathcal{N}}(s'))\}$$

for finding accepting states in the original search space.

States in the extended search are abbreviated by tuples (s, s'), with s recording the start state of the cycle s' being the current search state. If we reach an accepting state, we immediately switch to secondary search. Therefore, we observe two distinct cases: primary search, accepting state not yet reached, secondary search, accepting state once found. The state $s = s'$ reached in secondary search is the goal. As it is a successor of a secondary state, we can distinguish the situation from reaching such a state for the first time.

For all $e = (s, s')$ in the extended search space $\mathcal{S}'$, let $H_a(e) = H_a(s)$ and $H_M(e) = H_M(s, s')$. Now we are ready to define a heuristic for liveness

$$H(e) = \begin{cases} H_a(s) & \text{if } s = s' \\ H'_M(s, s') & \text{if } s \neq s' \end{cases} \tag{1}$$

Lemma 1. *Let $h^*(e)$ be the shortest lasso-shaped counterexample with an accepting seed state starting at e. Then $H(e)$ is a lower bound on $h^*(e)$.*

Proof. As each counterexample has to contain at least one accepting state in the never-claim, for primary states e we have that $H = H_a(e)$ is a lower bound. For secondary states $e = (s, s')$, we have

$$H(e) = H'_M(s, s') = \max\{H_M(s, s'), D_{\mathcal{N}}(pc_{\mathcal{N}}(s), pc_{\mathcal{N}}(s'))\},$$

a lower bound to close the cycle and the lasso in total.

Lemma 2. *The estimator H is monotone, i.e., $H(e) - H(e') \leq 1$ for all successor states e' of e.*

Proof. Consistency is a local property. As both H_a and H'_M are monotone [26] and only one of them is true at a time, the only thing we have to show that H is monotone are the transitions between the different cases. The only problematic situation is the transition in case of reaching an accepting state. Here we have that a predecessor e with an evaluation of $H(e) = H_a(e) = 0$ spawns successors e' with evaluation values of $H_M(e') > 0$. However, this incurs no problem as $H(e) - H(e') \leq 1$ still preserves monotonicity.

The gap between H_M and H_a at accepting states may indicate that there is some option for applying an improved search estimate.

The next result shows that, given a monotone heuristic estimate, our approach terminates with an minimal-length counterexample where the lasso seed is accepting. If one allows seed states also to be non-accepting, there are potentially shorter counterexamples. This is possible if the accepting state is reachable only via a non-accepting seed. In this case the path from the seed to the accepting state would appear twice in the corresponding counterexample found in our algorithm starting the secondary search from an accepting seed state. Note that this subtlety does not effect completeness, a lasso with accepting seed exists if and only if an lasso with an accepting cycle exists.

7 External Guided Exploration

The model checking algorithm for directed external LTL search is an extension External A* and traverse the bucket file list along growing $f = g + h$ diagonals. In each external state we store (packed) original state vector pairs (s, s') with $s = s'$.

Figure 1 (left) depicts a prototypical execution of the guided exploration. For primary nodes (illustrated using two white half circles), we apply the heuristic

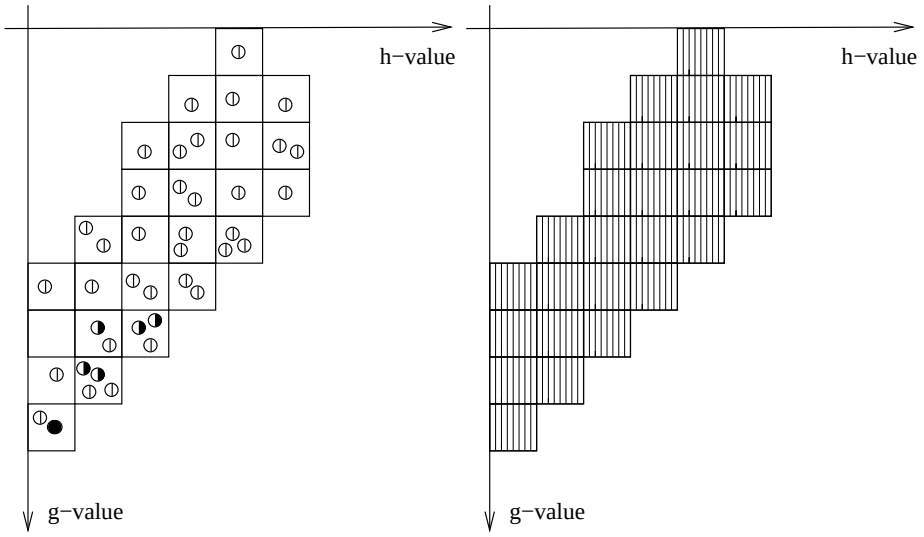

Fig. 1. Directed model checking LTL (left), distribution among several processors (right).

H_a, while for secondary nodes (illustrated using cycles half white/half black) we apply the estimate H_M. Once a terminal state with $s = s'$ (illustrated using two black half circles) is reached we have found an accepting cycle.

Figure 1 (right) illustrates how to perform parallel exploration[3]. The internal work for exploration a bucket is uniformly distributed among the set of available processors, that individually expand and sort individual files as described above.

Theorem 2. *For problem graphs the external, parallel and guided LTL model checking algorithm finds the shortest counterexample with an accepting seed state. Its I/O complexity is* $O(\text{sort}(|\mathcal{F}||\mathcal{R}|)/p + l \cdot \text{scan}(|\mathcal{F}||\mathcal{S}|))$*, where* l *is the length of the shortest counterexample.*

Proof. The proof is analogous to Theorem 1. Additionally, the parallelism divides the sorting efforts.

The main advantage of directed search is that the set of expanded states $\mathcal{S}$ (and subsequently $\mathcal{R}$) is smaller than with blind search.

The solution path is reconstructed by backward chaining starting with the final state. There are two main options. Either for a state in depth g we intersect the set of possible predecessors with the buckets of depth $g - 1$. Any state that is in the intersection is reachable on an optimal solution path, so that we can recur. As generating the predecessor state can be problematic in software model

[3] For a full treatment of the parallel execution of External A* we refer the reader to [19]. As the paper is not printed yet, the reviewers can obtain a copy of the work at `http://ls5-www.cs.uni-dortmund.de/~jabbar/vmcai06.pdf`

checking domains, we may store with each state its predecessor on a shortest path, doubling the required disk space. The time complexity is bounded by the scanning time of at most l buckets in consideration and surely in $O(scan(|\mathcal{F}||\mathcal{S}|))$.

8 Experiments

We implemented external LTL property validation on top of our experimental model checker IO-HSF-SPIN [18], the recent extension the directed model checking SPIN-derivate HSF-SPIN. The inputs are Promela-files and the output is a trail file in SPIN's format. The Promela language scope of IO-HSF-SPIN is not as large as in SPIN[4] as it lacks some features like fully dynamic process creation and embedded `c`-code, but sufficiently strong even for larger models that we have in our benchmark set.

As with its ancestors, in IO-HSF-SPIN Promela models are compiled into self-contained model checking units. The experiments for single-processor were conducted on a Pentium-4 PC, 3 GHz with 2 gigabytes of main memory and 180 gigabytes of hard disk. We exploit disk parallelism by RAID 0 using two hard disk. For multi-processor experiments we chose a Sun Enterprise System with four 750 MHz processors working with 8 gigabyte RAM and 30 gigabyte shared hard disk space. In this case, we worked with a single hard disk, so that no form of disk parallelism was exploited.

We choose a small internal buffer size for buffered reading and writing consisting of only 1,997 states. We applied internal (hash table based) and external (delayed) duplicate detection within the next bucket to expand. Duplicate elimination with respect to visited states in previous buckets is not done. This reduces the number of scans to linear-time complexity by the cost of some redundant states. The heuristic we applied takes a combination of H_a (for primary search) and H_M (for secondary search).

When comparing to SPIN it should be noted that this model checker was invoked with partial order reduction. Actually, as indicated by [26], partial order reduction preserves completeness but not optimality. It may lead to non-optimal counterexamples.

In our first set of experiments we use an elevator simulation protocol[5]. Table 1 shows the exploration results. We denote the number of expanded states, the number of states inserted to the hash table, the CPU time consumed and the length of the counterexample obtained. The sizes of the counterexamples are divided into the prefix and cycle length.

We compare the results of the exploration of External BFS and External A* as implemented in IO-HSF-SPIN with Nested-DFS as implemented in SPIN, Distribution 4.2. Due to the statistic information provided by SPIN instead of the number of expanded and inserted states, we give the number of stored states and explored transitions[6]. SPIN and IO-HSF-SPIN return counterexamples that

[4] The SPIN code we started with was SPIN 3.4

[5] Derived from `www.inf.ethz.ch/personal/biere/teaching/mctools/elsim.html`

[6] The counterexamples are produced with the options `-t -p`

I/O-HSF-SPIN	Expanded	Inserted	Time	Length
External A*	2,090,933	2,275,778	1m18s	$67 + 34$
External BFS	2,642,575	2,827,073	2m3.96s	$67 + 34$
SPIN 4.2	Transition	Stored	Time	Length
Nested DFS	33,900	11,149	0m0.064s	$109 + 100$

Table 1. LTL Model Checking with External A*, External BFS and Internal Nested DFS for 2-Elevator protocol

I/O-HSF-SPIN	Expanded	Inserted	Time	Length
External A*	178	369	0m1.318s	$15 + 5$
External BFS	1,343	1,427	0m0.787s	$15 + 5$
SPIN 4.2	Transition	Stored	Time	Length
Nested DFS	155,963	8,500	1m47s	$18 + 5$

Table 2. LTL Model Checking with External A*, External BFS and Internal Nested DFS for SGC protocol

start at accepting states[7]. We observe that SPIN's counterexamples are in general longer than the ones in IO-HSF-SPIN[8].

From the results of our first experiments we do not see a large gain of External A* compared to External BFS in the number of expanded and inserted states. The established counterexample lengths match. In the time, however, we see that External A* is considerably faster. There a different reason for the difference in ratios for the number of expansions and CPU time. First, as there are less buckets in External BFS (one for every layer) compared to External A*, there are more I/Os needed for external sorting. The other reason is that the number of generated nodes that fall into the buckets that are not considered for expansion (with counterexample length larger than the optimum) are larger for External BFS.

SPIN's exploration is remarkably good, as it requires only 6 milliseconds for generating an optimized trail. The number of stored nodes for Nested-DFS is much smaller as compared to blind BFS and A* LTL property search. The established counterexample is longer.

In the second experiment we take a larger protocol, as used in [37], a Promela model of a procedure with related processes. In Table 2 we see an opposite behavior as compared to the previous experiment. External search performed

[7] Without the predefined bound on the search depth, SPIN tends to find very long counterexamples, e.g. with 9998 steps. We therefore chose an iterative depth-first search strategy `-i` for SPIN. As this option may be caught in a depth anomaly [26] we also checked option `-DREACH`, which should return optimal traces. However, the results we obtained with this setting were not better than with `-i`.

[8] This is not neccessarily due to their non-optimality, but probably relying on a different measurement for steps, as SPIN is likely to put some additional increment on synchronized never-claim transitions.

I/O-HSF-SPIN	Expanded	Inserted	Time	Length
External A*	2,298	127,813	0m6.108s	196 + 2
External BFS	2,298	47,118	0m13.549s	196 + 2
SPIN 4.2	Transition	Stored	Time	Length
Nested DFS	-out-of-mem-	-out-of-mem-	–	–

Table 3. LTL Model Checking with External A*, External BFS and Internal Nested DFS for 64-Dining Philosopher

a much smaller number of expansions than internal iterated Nested DFS. The reason is that iterative improvement strategy takes a long time to decrease the counterexample length to a feasible low number. The behavior of External BFS compared to External A* is also opposite to the above. Now the number of expansion is smaller in External A* is much smaller due to its good guidance, but External BFS CPU time is superior. The reason for this is that the distribution of the heuristic estimate is fine-grained such that many internal buckets have to be allocated but never used.

In the third set of experiments we choose the scalable Dining Philosophers protocol with 64 philosophers. The LTL property we checked for was

```
[]  (philosopher[1]@eat -> <>  philosopher[2]@eat)
```

realizing the *response* property that always if the first philosopher eats, so does the second. Table 3 shows our results. Coincidently, the number of expanded nodes for guided and unguided external search match. The number of inserted nodes is, however, smaller for External BFS. We explain this behavior by absence of external duplicate removal for unexplored buckets. In agreement with this argument, External BFS took more time to perform external delayed duplicate detection. SPIN, unfortunately, ran out of memory. It found counterexample in very large depth, but was unable to shorten the trail. Even provided with a depth bound of 300 it was unable to terminate its iterated improvement strategy, due to the limits of main memory, which in our case was 2 gigabytes. Manually adapting the search depth to the optimum of 212 allowed SPIN to complete its exploration finding a counterexample with a acceptance cycle seed at depth 207.

For distributed execution on the multi-processor machine we again choose the Dining Philosopher example (see Table 4), now scaled to 128 philosophers. First, we note that disk space consumption is considerably large. The single processor version could not finish its exploration. One file for the set generated states became larger than 2 gigabytes and was killed by the operating system. The reason that the multi-processor versions could finalize their implementation, is early duplicate detection in intermediate files. The length of the produced counterexamples match and the observed speed-up is noticeable.

I/O-HSF-SPIN	Time	Secondary Memory	Length
1 processor	–	–	–
2 processors	5m53.96s	4.7 gigabytes	$388 + 2$
3 processors	4m7.13s	5.28 gigabytes	$388 + 2$

Table 4. LTL Model Checking with External A* for 128-Dining Philosopher

9 Conclusion

In this work we have combined directed, external and parallel approaches to compute optimal counterexamples for LTL properties in explicit-state model checking. The I/O complexity of $O(sort(|\mathcal{F}||\mathcal{R}|)/p + l \cdot scan(|\mathcal{F}||\mathcal{S}|))$ is a drastic improvement to simulating DFS as done for computing strongly connected components in explicit graphs with Tarjan's algorithm, as it avoids unstructured access to the adjacency lists. Different to *NestedDFS* the approach provides an optimality guarantee on the length of the counterexample.

The search space is generated using state pairs of active and cycle seed state, which supports the design of monotone LTL heuristics for directed model checking. Primary and secondary search states are examined together in one common file. The underlying exploration algorithm extends External A* to allow accepting cycles to be found. As with External A*, the approach can be effectively be parallelized. Duplicate detection is delayed. Up to synchronization mechanism for work distribution, no communication between the individual processes is needed, which in large problems allows almost linear speed-ups in a distributed environment.

With this research, we hope to have pushed the limits of practical model checking where the internal memory does not limit the number of realistic models that can be verified. With our support of pause-and-resume the size of the secondary storage can be resized without harming the correctness of the model checking process. Combining this with our approach presented in [19] on parallel external guided safety model checking, we now put our focus on larger industrial-sized models, which means targeting towards state spaces requiring terrabytes of storage.

A challenge for future research will be to reduce the (sequential) time complexity to $O(sort(|\mathcal{R}|) + scan(|\mathcal{S}|))$ as for safety model checking.

Acknowledgments The work is supported by *Deutsche Forschungsgemeinschaft* (DFG) in the projects *Heuristic Search* (Ed 74/3) and *Directed Model Checking* (Ed 74/2).

References

1. A. Aggarwal and J. S. Vitter. Complexity of sorting and related problems. In *International Colloquim on Automata, Languages and Programming (ICALP)*, number 267 in LNCS, pages 467–478, 1987.
2. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Journal of the ACM*, 31(9):1116–1127, 1988.
3. J. Barnat, L. Brim, and I. Cerna. Property driven distribution of nested DFS. In *International Workshop on Verification and Computational Logic (VCL)*, pages 1–10, 2002.
4. J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model checking. In *International Conference on Automated Software Engineering (ASE)*, pages 106–115, 2003.
5. J. Barnat, L. Brim, and J. Chaloupka. From distribution memory cycle detection to parallel model checking. *Electronic Notes in Theoretical Computer Science*, 133:21–39, 2005.
6. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In *Advances in Computers (volume 58)*. Academic Press, 2003.
7. L. Brim and I. Cerna. Accepting predecessors are better than back edges in distributed LTL model-checking. In *Formal methods in Computer-Aided Design (FM-CAD)*, pages 352–366, 2004.
8. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Conference on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1962.
9. I. Cerna and R. Palanek. Distributed explicit fair cycle detection. In *Model Checking Software (SPIN)*, pages 49–73, 2003.
10. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.
11. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
12. S. Edelkamp, S. Jabbar, and S. Schroedl. External A*. In *German Conference on Artificial Intelligence (KI)*, pages 226–240, 2004.
13. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2-3):247–267, 2004.
14. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial order reduction and trail improvement in directed model checking. *International Journal on Software Tools for Technology*, 6(4):277–301, 2004.
15. K. Fisler, R. Fraer, G. Kamhi, Y. Vardi, and Y. Ynag. Is there a best symbolic cycle detection algorithm. In *TACAS*, pages 420–434, 2001.
16. D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
17. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. *The SPIN Verification System*, pages 23–32, 1972.
18. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 313–329, 2005.
19. S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 237–251, 2006.

20. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *AAAI*, pages 1194–1201, 1996.
21. R. E. Korf. Best-first frontier search with delayed duplicate detection. In *AAAI*, pages 650–657, 2004.
22. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, 2005.
23. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *AAAI*, pages 910–916, 2000.
24. L. Kristensen and T. Mailund. Path finding with the sweep-line method using external storage. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 319–337, 2003.
25. A. Lluch-Lafuente. Simplified distributed ltl model checking by localizing cycles. Technical report, Institute of Computer Science, University of Freiburg, 2002.
26. A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, Institute of Computer Science, University of Freiburg, 2003.
27. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 687 – 694, 1999.
28. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
29. J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.
30. S. Safra. On the complexity of omega-automata. In *Annual Symposium on Foundations of Computer Science*, pages 319–237. IEEE Computer Society, 1998.
31. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
32. V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):185–204, 2004.
33. V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. In S. A. Smolka and J. Srba, editors, *INFINITY '05*, Electronic Notes in Theoretical Computer Science, 149(1), pages 79–96. Elsevier, 2006.
34. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3):217–237, 1983.
35. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, (1):146–160, 1972.
36. P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.
37. W. Zhang. Model checking operator procedures. In *Workshop on Model Checking Software (SPIN)*, pages 200–215, 1999.