Distributed Extended Beam Search for Quantitative Model Checking

A.J. Wijs and B. Lisser

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{wijs,bert.lisser}@cwi.nl

Abstract. In this paper, we mainly focus on solving scheduling problems with model checking, where a finite number of entities needs to be processed as efficiently as possible, for instance by a machine. To solve these problems, we model them in untimed process algebra, where time is modelled using a special *tick* action. We propose a set of distributed state space explorations to find schedules for the modelled problems, building on the traditional notion of *beam search*. The basic approach is called distributed (detailed) beam search, which prunes parts of the state space while searching using an evaluation function in order to find near-optimal schedules in very large state spaces. Variations on this approach are presented, such as distributed *flexible*, distributed *g-synchronised*, and distributed *priority* beam search, which can also practically be used in combinations.
Keywords: directed model checking, distributed model checking, scheduling, beam search.

1 Introduction

Traditionally, model checking concerns modelling systems and checking properties, which either hold or not, in other words, the checks can be answered with either "yes" or "no". In more recent years, however, the awareness has grown that often other kinds of checks, which cannot be answered in such a manner, are as important. For these checks, one is usually interested in some measurements, such as the throughput or efficiency of a particular system. Markov Chains, for instance, have shown to be useful when one needs to do performance analysis of a system [9]. Although not common yet, sometimes scheduling problems are also addressed using model checking techniques [2, 28, 37, 41], since the tools are usually equipped with highly expressive languages, making it possible to specify complex industrial scheduling questions. Comparing the two kinds of property checks, one could label traditional model checking as *qualitative* model checking and the latter one as *quantitative* model checking [22].

Furthermore, as state explosion is a big problem in model checking, research is being done to efficiently explore state spaces to find deadlocks fast, particularly using Artificial Intelligence (AI) heuristic techniques, such as A^* [15] and genetic algorithms [19]. This approach is referred to as *directed* model checking [15]. Although mostly used for qualitative model checking, techniques like beam search [5] can be applied for quantitative model checking, in particular to solve scheduling problems.

In an earlier paper [41], we made a first attempt at solving scheduling problems, where a finite number of products needs to be processed as efficiently as possible by a

machine, by modelling them using untimed process algebra and generating state spaces from the models using a specialised toolset. Within such a state space a *minimal-time trace* represents an optimal schedule for the problem at hand.

We experienced the limits of our first attempt quite soon; state spaces tend to be very big, sometimes in the order of hundreds of gigabytes. Although we developed an on-the-fly search algorithm, which enables us to find optimal solutions while generating, we were still confronted with technological limits. Because of this we moved to a distributed setting with our minimal-time search algorithm. In [41], results of applying this distributed algorithm on finding schedules for a clinical chemical analyser can be found. The algorithm enabled us to deal with bigger problems, but still we had the impression that the technique could be improved if we were able to avoid the (many) non-promising traces and guide the search through the state space towards near-optimal schedules using a heuristic method. When looking at available pruning techniques in the literature, we found beam search [5]. Beam search is a heuristic method for combinatorial optimisation problems, which has extensively been studied in AI and operations research [27, 35]. Later this technique has been applied to scheduling problems, for example in systems designed for complex job shop [1] environments [12, 17, 31, 38, 41]. Since then new variants of beam search have been introduced, such as filtered beam search [30] and recovery beam search [12].

Using beam search proved to be very fruitful, as we were able to find near-optimal schedules for all the considered batches of tests of the clinical chemical analyser [41]. It sometimes took a lot of time, though, mostly due to the extra computation needed to evaluate states. This could be improved if we moved the beam search techniques to a distributed setting. In this paper we propose several distributed beam search variants, focussing on detailed beam search, since due to its global view when pruning, it is not obvious how a distributed algorithm should function.

Contributions We show how a technique for solving scheduling problems can be adapted to a distributed setting. The technique, beam search, is a heuristic which prunes parts of a state space while searching, in order to find near-optimal solutions. We extend the distributed technique to deal with arbitrary state spaces and make it more effective.

Structure of the paper First we will present some preliminaries. Next we describe the kind of scheduling problems we are dealing with. After that we explain the most common forms of beam search, followed by descriptions of the distributed versions we propose. We show how some of these versions perform in practice, looking at, as we call it, the Zebra Finch problem, which is a combination of several river crossing problems [14]. Finally, we discuss related work and conclude the paper.

2 Preliminaries

We use the following formalism to represent state spaces.

[1] The job shop problem is the most classic scheduling problem in the literature. In its most basic form, we have a finite set M of resources, and a number of jobs $J_1,\ldots,J_n$ which compete in using the resources in a specific order and for a finite number of time units. The problem is to allocate the resources such that the jobs are finished in minimal time.

Definition 1 (Labelled transition system). *A Labelled Transition System (LTS) is a tuple* (Σ, s_0, Act, Tr)*, where* Σ *is a finite set of states, which is usually not known a priori, but generated on-the-fly,* $s_0 \in \Sigma$ *is the initial state, Act is a given finite set of action labels and* $Tr \subseteq \Sigma \times Act \times \Sigma$ *is the transition relation. A transition* $(s,a,s') \in Tr$*, denoted* $s \xrightarrow{a} s'$*, indicates that the system can move from state s to* s' *by performing action a.*

For $T \subseteq Tr$, we define $nx(s,T) = \{s' \in \Sigma \mid \exists a \in Act.\ s \xrightarrow{a} s' \in T\}$. We define a state s to be an *endstate* iff $nx(s,Tr) = \emptyset$.

Breadth-first State Space Generation State space generation algorithms are provided with a specification as input and produce the state space which is described by that specification. A breadth-first state space generation (BFS) algorithm, as presented in Algorithm 1, starts from the initial state of the specification and names it s_0. State s_0 is placed in the set S_0. Sets S_1, S_2,... are generated iteratively.

In Algorithm 1, $expand : \Sigma \rightarrow \mathscr{P}(Tr)$ is the function that provides the interface between the state space generation algorithm and the underlying specification. For a state s, $expand(s)$ is the set of transitions which root in s. Set S_i with $i \in \mathbb{N}$ denotes the set of states in the $i+1$th level of the state space. The set *Closed* is used to perform *delayed* duplicate detection [34] when expanding states; if a state has already been expanded before, we do not need to expand it again. This is checked at the end of generating a new set, hence it is *delayed*.

Algorithm 1 Breadth-first state space generation

```
procedure bfs(s_0)
  i := 0
  S_i := {s_0}
  Closed := ∅
  while S_i \ Closed ≠ ∅ do
    S_{i+1} := ∅
    for all s in S_i \ Closed do
      S_{i+1} := S_{i+1} ∪ nx(s, expand(s))
    Closed := Closed ∪ S_i
    i := i + 1
  return Finished
```

Distributed State Space Generation Moving to a distributed setting, we no longer deal with one machine, but one manager and n clients $C_1,\ldots,C_n$, where $n \in \mathbb{N}$. For this paper, it suffices to say, that in distributed BFS state space generation, every client performs a BFS on the states it gets. After generating the set S_{i+1}, given a set S_i, how the states in S_{i+1} should be distributed over the n clients is determined by a hash function $Checksum : \Sigma \rightarrow \mathbb{N}$. For more information on distributed state space generation, the reader is referred to, for instance, [11].

The language μCRL The process algebra μCRL [21], an extension of ACP [4] with abstract data types, is a language for specifying distributed systems and protocols in an algebraic style. A μCRL specification describes an LTS, in which states represent process terms and edges are labelled with actions. This process algebra is used as input to a state space generation toolset [7], which is accompanied by symbolic reduction techniques. The toolset has also been extended for a distributed setting [6].

Based on the work from [8, 40], we use a special *tick* action, which models time progression. This is comparable to relative discrete time [1]: A *tick* action indicates that the system moves to the next time slice.

Definition 2 (minimal-time trace [41]). *Given an LTS and a transition label* a*, we say that there is a trace with execution time t ($t \in \mathbb{N}$) to a transition with label* a *iff there is a trace in the LTS starting from the starting state s_0 and reaching a transition with label* a*, such that the number of* tick *transitions occurring in this trace equals t. We define a trace from s_0 to a transition with label* a *to be* minimal-time *iff there is no other trace in the LTS from s_0 to* a *with less* tick *transitions.*

Using this definition, we can formulate a scheduling problem as a reachability problem: finding an optimal schedule to perform a batch of tasks successfully can also be seen as finding a minimal-time trace to a transition indicating successful termination in a state space containing all possible schedules as traces. That we can also in this manner deal with scheduling problems involving parallel execution of tasks, will be explained in the following section.

3 Modelling Scheduling Problems using μCRL

Scheduling problems, in this paper, are typically about processing a certain number of entities (for instance, products or jobs, in the case of jobshop scheduling). The processing is usually done by a machine, or combination of machines, which can perform tasks $t_1,\ldots,t_m \in Ta$, where Ta is a set of task labels[2], provided, that the accompanying sets of constraints $C_1,\ldots,C_m$ are met[3]. Furthermore, each task t_i has an execution time $d(t_i)$ associated with it, given by the function[4] $d : Ta \rightarrow \mathbb{N}$. In these problems, a certain goal should be reached, usually having completely processed a finite batch of entities. The question asked in scheduling is not mainly *if* this goal can be reached, but *how efficiently* this can be done.

As we perform scheduling using model checking tools, we are able to deal with complex industrial systems, the models of which tend to lead to very big, arbitrarily structured state spaces. We model tasks as transitions, meaning that performing task t_i in an execution appears as $s_j \xrightarrow{t_i} s_{j+1}$ in the LTS, where s_j and s_{j+1} are two states in the trace corresponding to the execution. In state spaces, where the traces represent schedules, we can observe the following.

[2] Later on, in our approach, action labels from *Act* represent task labels from *Ta*.

[3] To keep things general, we do not fix these constraints to a specific notation here. Suffice it so say that they can deal with time and data.

[4] Since execution times are here represented using natural numbers, we use discrete time.

A function *progress*: $\Sigma \rightarrow \mathbb{N}$ can be constructed, which can access the state variables of a state s, using the underlying μCRL specification of the LTS (similar to *expand(s)* in section 2) and quantifies the progress made to reaching some predetermined goal, for instance having completely processed a given batch of entities. In general, say we have $c_0, c_{end} \in \mathbb{N}$, $\forall s \in \Sigma . c_0 \leq progress(s) \leq c_{end}$ and $progress(s_0) = c_0$, in other words, c_0 is the initial (no) progress and c_{end} represents having reached the goal. We do not claim any monotonicity of this function, as in general one can imagine tasks which provide negative progress, which, for instance, is the case in our example in section 7.

Building on Definition 1, we can now distinguish two kinds of endstates.

Definition 3 (termination and deadlock). *A state s is a* termination *state iff it is an endstate and* $progress(s) = c_{end}$. *A state s is a* deadlock *state iff it is an endstate and* $progress(s) \neq c_{end}$.

The intuition behind this, is that we can distinguish two kinds of endstates: one where the predetermined goal is reached, and one where it is not.

The general structure of a μCRL model of a scheduling problem can be described as consisting of a process (or processes), which is an alternative composition of all tasks t_i, each followed by a sequence of *tick* actions, to indicate the execution time. The tasks t_i can only be executed if the accompanying conditions C_i are met, written in the model as conditions for the actions representing the tasks, and, once executed, a task has an effect on the progress of the processing (as expressed by function *prog*). So this model can execute all available tasks as long as the constraints are satisfied. Which tasks to execute and when is decided non-deterministically; there are no built-in priorities.

Besides that, we introduce a special action called *finished*. We use this action in such a way that it can be executed iff it leads to a termination state.

Sometimes, a system consists of several processes running in parallel, and the scheduling problem involves the parallel execution of tasks. In μCRL, it is possible to model multiple processes in parallel and have them work with time correctly. For this it must be enforced that all *tick* actions are synchronised; only if all processes can do a *tick* action, a *tick* action occurs. Explaining in detail how this can be achieved is outside the scope of this paper, since it involves a detailed explanation of μCRL. The interested reader is referred to [8, 40]. We can note here, that having several processes in the structure mentioned earlier, means that we can still relate a schedule to a path in the state space. For this, we need to interpret a sequence of tasks, not containing any *tick* actions, as a set of tasks happening at the same time. Consider, for example, the sequence $a{\cdot}b{\cdot}tick$ in a trace, where '$\cdot$' is the sequential composition operator of μCRL. Due to the structure of the processes, we know that a and b originate from different processes; if not, they would be seperated by at least one *tick* action (assuming that the execution of each task takes at least one time unit). Furthermore, we can interpret $a{\cdot}b{\cdot}tick$ as a and b happening at the same time, which makes sense, considering that they happen in the same time unit (i.e. between the same two *tick* actions). If we do this, then we do not differentiate $a{\cdot}b{\cdot}tick$ from $b{\cdot}a{\cdot}tick$. Note, that this relates to the notion of independent actions for partial order reduction [32]. Using this terminology in our case, given a solution path, we abstract away the particular action arrangement of independent actions.

Having created a μCRL model, it is possible, using the μCRL toolset, to generate a state space from it. This state space incorporates all possible behaviour of the system

described by the model. Somewhere in this state space there is at least one minimal-time trace to a finish. Given Definition 2, we use the *finished* action as transition a, in order to formulate a minimal-time trace to a termination. In [41], this modelling approach is applied on a clinical chemical analyser, and a specific minimal-cost search is explained (the search is mentioned again later in this paper in section 6).

4 Beam Search

Beam search [5] is similar to breadth-first search as it progresses level by level. At each level, it uses a heuristic evaluation function to estimate the promise of encountered states. The β most promising states are selected for further examination. Because of this aggressive pruning, the generation time is a linear function of β and is thus heavily decreased. When $\beta \rightarrow \infty$, beam search behaves as breadth-first search [39].

The beam search approach is a branch-and-bound technique where only the β most promising states at each level of the search tree are selected for further branching. This β is the so-called *beam width*, which is fixed to a value before searching. Other states are discarded, so searching can be done relatively quickly. Because of this, using the beam search technique does not guarantee finding an optimal solution, since wrong decisions can be made while pruning. To limit the possibility of wrong decisions one can increase the beam width, at the cost of an increase in computational effort.

Clearly the evaluation function used to select states is very important. In the past, two types of evaluation functions have been used: *priority* and *total cost evaluation functions*. A priority evaluation function calculates a priority for each task, while a total cost evaluation function calculates an estimate of the total cost of the best schedule that can be found continuing from the partial schedule represented by the state. Priority evaluation functions have a local view of the problem, since they only consider the next task to be scheduled, while total cost evaluation functions have a global view, taking the complete schedule into account. These types of functions lead to two classic beam searches, namely priority and detailed beam search, using a priority and a total cost evaluation function, respectively.

In a detailed beam search, at each level up to β nodes are selected to continue, regardless of what their parent states are, therefore it could be the case, that some nodes have multiple selected children, while others have none. A total cost evaluation function allows comparison of states from different executions as it shows the progress each execution is making (i.e. it has a global view). This in contrast to priority evaluation functions, which only allow comparison of alternatives, which are part of the same trace up to that point.

In [39], detailed and priority beam search were extended for usage on arbitrary state spaces, as opposed to highly structured trees. In the following section we present extended detailed beam search, as implemented in the μCRL state space generator. For a detailed comparison between the basic notion of this beam search and the extension and eventual adaptation to the μCRL toolset setting, the reader is referred to [39].

5 Extended Detailed Beam Search

In this section we first present the extended detailed beam search in its sequential form. After that we adapt it to a distributed setting. From now on, whenever detailed beam search is mentioned, we refer to the search extended for arbitrarily structured state spaces.

5.1 Sequential Detailed Beam Search

A user of the μCRL toolset can perform a detailed beam search, i.e. a beam search using a total cost evaluation function. The user can provide a function using constants and variables from the model, combining them using mathematical operators.

Algorithm 2 shows in pseudo-code the detailed beam search algorithm as used in the μCRL toolset. The evaluation function is called $f : \Sigma \rightarrow \mathbb{N}$. This function is decomposed to $f(s) = g(s) + h(s)$, where $g(s)$ represents the cost taken to reach s from the root of the tree, which is defined as $g(s) = g(s') + cost(a)$ if $s' \xrightarrow{a} s$. The function $cost : Act \rightarrow \mathbb{N}$ assigns weights to actions that can, for instance, denote the time needed to perform different jobs in a scheduling problem. These are usually fixed to certain values before searching starts. Since the range of *cost* is non-negative numbers, if $s \xrightarrow{a} s'$, then $g(s') \geq g(s)$, for any action a. The $h(s)$ function is an estimation of the cost it would take to efficiently complete the schedule continuing from s. Here, we consider *admissible* heuristics, i.e. for all states s, $h(s)$ is an underestimation of the real minimal cost needed to complete the schedule. The function $getf_{max} : \mathscr{P}(\Sigma) \rightarrow \Sigma$, given a set of states, returns one of the states that has the highest f value. It thus computes $f(s) = g(s) + h(s)$ for each member of the set. Contrary to Algorithm 1, here, all S_i and *Closed* contain pairs of states and corresponding g-values. Finally, the functions $unify(X)$ and $update(X,Y)$ are defined as follows: $unify(X) = \{\langle s,g \rangle \in X \mid \forall \langle s',g' \rangle \in X.s = s' \implies g \leq g'\}$ and $update(X,Y) = \{\langle s,g \rangle \in X \mid \neg \exists g' \leq g.\langle s,g' \rangle \in Y\}$. These functions are used to perform a delayed duplicate detection, where revisiting of a state is allowed if it is reached via a path with a lower cost than the g-cost assigned to it so far.

Note, that no additional stopping condition appears in Algorithm 2, i.e. it appears as though we exclude searching for something in particular, for instance the violation of a property. This, however, can in practice be done on top of any search. Relating back to section 3, we can perform a detailed beam search and at the same time check, whether a transition $s \xrightarrow{a} s'$ is found, such that $a = finished$. Once this is the case, the search can be stopped and a trace from s_0 to s' can be returned, which corresponds to a schedule. This approach is used in our experiments (see section 7).

5.2 Distributed Detailed Beam Search

Because of the global view of total cost evaluation functions, designing a distributed version of detailed beam search is non-trivial. Clients should not select states for further exploration in isolation of each other, but have to communicate.

Say we have a manager and n clients to do a distributed detailed beam search. As described in 2, we have a hash function $Checksum : \Sigma \rightarrow \mathbb{N}$, which is used to distribute

Algorithm 2 Detailed beam search for state space generation

```
procedure detbs (s_0, β)
  s_0.g := 0
  i := 0
  S_i := {⟨s_0, s_0.g⟩}
  Closed := ∅
  while S_i ≠ ∅ do
    S_{i+1} := ∅
    while |S_i| > β do
      S_i := S_i \ {(s,g) ∈ S_i | s = getf_max(S_i)}
    for all s ∈ S_i do
      for all s --a--> s' ∈ expand(s) do
        s'.g := s.g + cost(a)
        S_{i+1} := S_{i+1} ∪ {⟨s', s'.g⟩}
    Closed := unify(Closed ∪ S_i)
    S_{i+1} := update(unify(S_{i+1}), Closed)
    i := i + 1
  return Finished
```

generated states over the clients for future exploration. Say the LTS consists of levels S_0, S_1, etc. As detailed beam search is done in a breadth-first manner, each level of states S_i gets distributed over the n clients before exploration, leading to the subsets $S_i^1, \ldots, S_i^n$, such that $S_i^1 \cup \ldots \cup S_i^n = S_i$ for all levels i, where S_i^j is the subset of S_i designated to client j by the hash function.

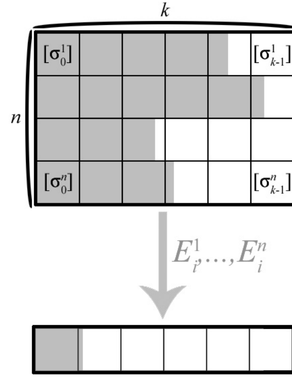

Fig. 1. Distributing, partitioning and selecting

Now, we define function $p_f : \mathscr{P}(\Sigma) \to \mathscr{P}(\mathscr{P}(\Sigma))$, which is used at each level i by each client j. For practical reasons, we say, that k is an upper limit of f. Now, p_f distributes the states from a set S_i^j over k equivalence classes $[\sigma_0^j], \ldots, [\sigma_{k-1}^j]$, such that $\forall u \in \{0, 1, \ldots, k-1\}. \forall s \in [\sigma_u^j]. f(s) = u$.

We refer to a selection of γ states from a set S_i^j using evaluation function f as $sel_\gamma^f(S_i^j) = [\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma']$, with $r \in \mathbb{N}$ and $r < k-1$, such that $|[\sigma_0^j] \cup \ldots \cup [\sigma_r^j]| < \gamma$, $[\sigma'] \subseteq [\sigma_{r+1}^j]$ and $|[\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma']| = \gamma$. In practice, $[\sigma'] \subseteq [\sigma_{r+1}^j]$ is composed according to a so-called tie-breaking rule. In the remainder of this paper, we denote sel_γ^f as sel_γ.

The goal to achieve now for the algorithm is the following:

$$\forall i. sel_{\gamma_{i,1}}(S_i^1) \cup \ldots \cup sel_{\gamma_{i,n}}(S_i^n) = sel_\beta(S_i) \tag{1}$$

Here, β is the beam width and $\gamma_{i,1}, \ldots, \gamma_{i,n} \in \mathbb{N}$, such that $\gamma_{i,1} + \ldots + \gamma_{i,n} = \beta$. If we could assume that $\gamma_{i,1} = \ldots = \gamma_{i,n}$, then there would be no problem moving the sequential beam search algorithm to a distributed setting. Then, however, besides assuming that

the states of each level are evenly distributed over the clients, we also have to assume that the β most promising states of a level are evenly distributed. This we cannot assume in general. Instead, we can move to a more general situation where the $\gamma_{i,j}$s are unequal to each other. In order to achieve this, extra communication is necessary.

Being in level i, let every client j first determine $sel_\beta(S_i^j)$, this to be prepared for the worst case scenario where all β most promising states end up at a single client. This is illustrated in the top part of Figure 1, where each row in the diagram represents a client, and each column represents an equivalence class. Having constructed the equivalence classes, $sel_\beta(S_i^j)$ is determined, which, in Figure 1, is highlighted in grey for each client. Once this is done, the clients send a set of tuples, each consisting of an evaluation value and the number of states in $sel_\beta(S_i^j)$ that have this evaluation value to the manager. To put it more formal, the following is sent by each client j, being in level i of the LTS:

$$E_i^j = \{(r, |[\sigma_r^j] \cap sel_\beta(S_i^j)|) | 0 \leq r \leq k-1 \wedge |[\sigma_r^j] \cap sel_\beta(S_i^j)| \neq 0\} \tag{2}$$

All the sets E_i^j sent by the clients are used by the manager to determine a final selection of β states. This is illustrated in the bottom part of Figure 1. First E_i is created as $E_i = \{(j,e,t) | (e,t) \in E_i^j\}$, where e and t correspond to the first and second element in the tuples calculated in (2). Similar to p_f, we define a function $p_e : \mathscr{P}(\mathbb{N}^3) \rightarrow \mathscr{P}(\mathscr{P}(\mathbb{N}^3))$, which allows us to distribute the elements of the set E_i over k equivalence classes $[e_0], \ldots, [e_{k-1}]$, such that $\forall u \in \{0,1,\ldots,k-1\}. \forall (j,e,t) \in [e_u]. e = u$.

For selecting the β best states, we define a function $T_j : \mathscr{P}(\mathbb{N}^3) \rightarrow \mathbb{N}$, which returns the number of states from client j represented in the given evaluation set E; more specific, $T_j(E) = 0 + \sum_{(j,e,t) \in E'} t$, with $E' = \{(j',e',t') \in E \mid j' = j\}$. We define $T : \mathscr{P}(\mathbb{N}^3) \rightarrow \mathbb{N}$ as the total number of states represented in the given evaluation set E, so $T(E) = \sum_{j=1}^{n} T_j(E)$. We refer to a selection of β triples from E_i as $evsel_\beta(E_i) = [e_0] \cup \ldots \cup [e_r] \cup [e']$, with $r \in \mathbb{N}$ and $r < k-1$, such that $T([e_0]) + \ldots + T([e_r]) < \beta$, $[e'] = evsubsel_{\beta-(T([e_0])+\ldots+T([e_r]))}([e_{r+1}])$. Here, $evsubsel_{\beta'}([e_u]) = \{(j_0,e_0,t_0)\} \cup \ldots \cup \{(j_{w-1},e_{w-1},t_{w-1})\} \cup \{(j_w,e_w,t'_w)\}$, where $(j_0,e_0,t_0),\ldots,(j_w,e_w,t_w) \in [e_u], t'_w \leq t_w$ and $t_0 + \ldots + t_{w-1} + t'_w = \beta'$. In practice, $[e']$ is composed according to a tie-breaking rule.

Each client j receives a width $\gamma_{i,j} = T_j(evsel_\beta(E_i))$, which it uses to obtain $sel_{\gamma_{i,j}}(S_i^j)$. Since $sel_{\gamma_{i,j}}(S_i^j) \subseteq sel_\beta(S_i^j)$, this set can be constructed from memory. For this approach only one extra communication round is necessary. Memory-wise, a distributed detailed beam search with beam width β is comparable with a sequential detailed beam search with beam width $n.\beta$, but, of course, on the whole, there is more memory available in a distributed setting than in a sequential one.

One advantage of detailed beam search is that if a level contains up to β states, for all states s in the level, $h(s)$, which can be computationally expensive, does not have to be calculated. To achieve this in the distributed version, the manager gets from every client the number of newly generated states. The sum of these numbers equals the complete size of the next level. If it sends this number together with the next `continue` command, the clients know whether or not to prune (see Algorithms 3 and 4).

In general, distributed state space generation algorithms benefit from symmetry. If all clients have to do a similar amount of work, than little to no idle time occurs in any of the clients and therefore no processing power is wasted. However, if we allow unequal $\gamma_{i,j}$s, then the workload of the clients can be very unequal at times. It makes no sense

to have clients idle, while they could very well expand states. Exploring more states than originally asked for can in practice, where the accuracy of the evaluation function[5] and the minimally necessary beam width are in general not known, only be seen as an improvement in accuracy[6]. For this reason we decided to create a variant where the manager does not provide every client j with $\gamma_{i,j}$, but a single $\gamma_i = max(\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,n})$ is provided to all clients. In this way every client expands the same amount of states[7], and we know that the β most promising states are selected[8].

Algorithms 3 and 4 show what the clients and the manager do in a distributed detailed beam search, respectively. The selection procedure of the manager in order to obtain γ_i is done in *calculateLimit*(). Matching send and receive functions can be identified by their names. Note, that duplicate detection is now performed by each client after having received the new set of states to be expanded. This works thanks to the *Checksum* function, which ensures that a state s is always assigned to the same client j. During the generation, a client can receive the following commands from the manager:

- `continue`: In the next step, receive new states in S_i and expand them.
- `finish`: Stop the search algorithm.

6 Other Beam Search Variants

In general, minimal-time traces to a transition a are not necessarily shortest traces to this transition. This fact means that when first encountering a in a BFS, we cannot claim having found a minimal-time trace. This can, however, be achieved by searching a state space using minimal-cost, or minimal-time, search [41], which can be seen as uniform-cost search [24], where the costs are modelled using additional actions. There, compared to BFS, the sets S_i do not comprise of states which are i transitions removed from s_0, but, using a total cost function g, in each iteration, S_i is transformed into $\hat{S}_i = \{\langle s, g\rangle \in S_i \mid \forall \langle s', g'\rangle \in S_i.g \leq g'\}$, $\hat{S}_i$ is expanded, leading to $\hat{S}_{i+1}$, and finally, $S_{i+1} = \hat{S}_{i+1} \cup (S_i \setminus \hat{S}_i)$. This technique can be combined with beam search, resulting in *g-synchronised* beam search, which is presented in [39] as an instance of *G-synchronised* beam search, where G can be any reasonable function. Compared to regular beam search, now only states with equal g-values are considered at the same time and states are selected purely on their h-value. It can be seen as *greedy search*, as described in [36], on top of minimal-cost, or uniform-cost, search. In each iteration,

[5] Of course, an important problem is to find a very good evaluation function. This is however beyond the scope of this paper, where we assume a given function, its accuracy unknown.

[6] There are results where a bigger beam width does not correspond to a higher accuracy, such as in [29, 41] and in section 7.2. However, this phenomenon mainly occurs when using relatively small beam widths (compared to the size of the state space), and can therefore be ignored for bigger cases.

[7] The exception to this is when a client has less states available than it is told to expand.

[8] One could argue that another approach is to redistribute the β selected states over the clients, in order to balance the workload. However, then we go against the distribution of the hash function, which means that clients will no longer be able to perform duplicate detection, leading possibly to redundant work.

Algorithm 3 Distributed detailed beam search - Client Instantiator

```
procedure ddbsclient(CLIENTNUMBER,{clientnumbers},s_0,β)
  s_0.g := 0
  i := 0
  if Checksum(s_0) = CLIENTNUMBER then
    S_i := {⟨s_0, s_0.g⟩}
  else
    S_i := ∅
  Closed := ∅
  SendToClientsNextLevel(S_i)
  (command, levelsize) := RecvFromMgr()
  if command ≠ finish then
    repeat
      S_i := update(unify(RecvFromClientsNextLevel()), Closed)
      S_{i+1} := ∅
      if levelsize > β then
        while |S_i| > β do
          S_i := S_i \ {(s,g) ∈ S_i | s = getf_max(S_i)}
        SendToMgrEvalInfo(I), with I as (2), sel_β(S_i^j) = S_i
        γ_i := RecvFromMgrLimit()
        while |S_i| > γ_i do
          S_i := S_i \ {(s,g) ∈ S_i | s = getf_max(S_i)}
      for all s ∈ S_i do
        for all s --a--> s' ∈ expand(s) do
          s'.g := s.g + cost(a)
          S_{i+1} := S_{i+1} ∪ {⟨s', s'.g⟩}
      Closed := unify(Closed ∪ S_i)
      SendToClientsNextLevel(unify(S_{i+1}))
      SendToMgrSizeNextLevel(|unify(S_{i+1})|)
      i := i + 1
      (command, levelsize) := RecvFromMgr()
    until command = finish
  return Finished
```

Algorithm 4 Distributed detailed beam search - Manager Instantiator

```
procedure ddbsmanager({clientnumbers},s_0,β)
  levelsize := 1
  SendToClients(continue, levelsize)
  repeat
    if levelsize > β then
      SendToClientsLimit(calculateLimit(RecvFromClientsEvalInfo()))
    levelsize := RecvFromClientsSizeNextLevel()
    if levelsize = 0 then
      SendToClients(finish, 0)
    else
      SendToClients(continue, levelsize)
  until levelsize = 0
  return Finished
```

first, the current S_i is transformed to $\hat{S}_i$ like in minimal-cost search, as described earlier. Then, h is applied on $\hat{S}_i$, in order to keep up to β states, as is done in greedy search. Greedy search from [36] corresponds to beam search with a constant g[9]. This variant not only allows finding minimal-time solutions within the beam before any other solutions. If one uses additional actions to model costs, it also removes the necessity to store the g-value of every state, since revisiting a state necessarily means having found a less efficient trace compared with a previous trace to the state.

Priority beam search is performed using a priority evaluation function $f : Tr \rightarrow \mathbb{Z}$, which assigns priorities to transitions. Therefore, priority beam search works on transitions, not states. A fixed number of outgoing transitions is selected *per state*, which makes the adaptation to a distributed setting straightforward. Since each selection does not consider the outgoing transitions of other states, communication with other clients is not needed. We can take the standard distributed state space generation algorithm, and insert an evaluation and selection step at the point where a state is expanded. Priority beam search for state spaces is described in more detail in [39].

Two other (related) variants are *flexible* priority beam search and *flexible* detailed beam search, introduced in [39, 41]. Flexible priority beam search behaves as regular priority beam search, but at each state it also selects any transition which has the same priority as the least competent member of the usually selected set. In other words, tie-breaking is avoided, by making the beam dynamic in size. The benefit of this approach is that there are no selection criteria other than the evaluation function used. This not only leads to more insight in the effectiveness of the function, but in practice it may also mean that smaller beam widths can be used, compared to non-flexible beam search (see, for instance, the results in section 7). The drawback is that the memory requirement is no longer linear in the maximum search depth, since β is only a guideline for the beam width. This search can be implemented in a distributed setting, since the local view characteristic is not lost. Similarly, in flexible detailed beam search we achieve at each level closure on the worst evaluation value still selected. The algorithm described in section 5.2 can be made flexible by redefining some functions. First we say that function $sel_\gamma(S_i^j)$ selects at least γ states, where $sel_\gamma(S_i^j) = [\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma_{r+1}^j]$, with $r \in \mathbb{N}$ and $r < k-1$, such that $|[\sigma_0^j] \cup \ldots \cup [\sigma_r^j]| < \gamma$ and $|[\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma_{r+1}^j]| \geq \gamma$. Likewise, we redefine $evsel_\beta(E_i) = [e_0] \cup \ldots \cup [e_r] \cup [e_{r+1}]$, with $r \in \mathbb{N}$ and $r < k-1$, such that $T([e_0]) + \ldots + T([e_r]) < \beta$ and $T([e_0]) + \ldots + T([e_{r+1}]) \geq \beta$.

7 Experimental Results

In this section we will show some experimental results of trying to solve instances of what we call the *Zebra Finch problem*. We based this problem on a combination of several *river crossing problems* [14], such as *five jealous husbands* and *soldiers and children*. First we describe the problem and then we provide the results obtained using the techniques described in this paper.

[9] It should be noted, that in the literature greedy search is sometimes given a different meaning. At least one other greedy search exists, which corresponds to detailed beam search with $\beta = 1$ (e.g. [43]).

7.1 The Zebra Finch Problem

Zebra Finches (*Taeniopygia guttata*) are small birds living in Central Australia [42]. They are found in large colonies of pairs inhabiting open steppes with scattered bushes and trees. These birds can react aggressively towards each other, for instance when a jealous male bird tries to keep other male birds away from his mate. When young birds reach an age where they can live outside the nest they are quickly adopted by the group.

We consider a group consisting of n pairs and m young, sitting in a tree on an open steppe. They want to migrate to some bushes up ahead, but they have to travel in smaller groups, since there are some hawks flying in the distance, which can spot a group of more than k adult finches. Once a group has reached the bushes, at least one of the Zebra Finches needs to fly back, in order to signal that a new group can travel. On top of this there are two other conditions:

Fig. 2. A pair of Zebra Finches

1. Considering the jealous nature of the male Zebra Finches, no female finch may ever be either in the tree, the travelling group or the bushes in the presence of other male birds, unless her partner is also present.
2. The young in the colony have to be guided by at least one adult finch, so the travelling group cannot consist of only young finches. In limiting the group size, two young are equivalent to one adult.

Finally some costs are related to the travelling from tree to bushes and back:

- A group consisting of only adults needs 1 time unit to travel the distance, independent of the size of the group;
- If the number of young in the group does not exceed the number of adults, the time needed to travel is 2 time units (each adult needs to take care of at most one young);
- When, in the group, the number of young exceeds the number of adults, the travel takes 3 time units, since at least one adult takes care of more than one young.

We model the problem allowing all possible actions at all times. It demonstrates the techniques' ability to deal with arbitrary state spaces; problem instances lead to state spaces containing both cycles (while forming the group and when birds fly away and back again), and deadlocks (violations of the 'jealous male' condition).

7.2 Results

In Table 1 we present some results we found for instances of the Zebra Finch problem. We used minimal-cost search, g-synchronised detailed beam search and its flexible variant, where for the last two cases we defined the h for each state as the number of finches still in the tree, thereby encouraging fast removal and discouraging the returning of finches. Problem instances are described by providing n, m and k. For each search, the total execution time of the result found is given. Furthermore, the number of states searched to find the solution and the time needed to find it is provided. Searches not

performed are marked with hyphens, and where the results could not be obtained due to technical reasons, dots are written. When a search is done in a distributed setting, an asterisk is placed after the number of states. Sequential searches were performed using a machine with a 64bit Athlon 2.2Ghz processor, 1 GB of memory and running Suse 9.3, while 16 of these machines together performed the distributed searches.

The minimal-cost search tells us that as the problem instances get bigger, the state spaces grow very rapidly. The beam searches on the other hand show a much nicer increase in states from instance to instance. Looking at the (50,50,10) instance though, we see an unwanted effect in the regular g-synchronised beam search, already briefly referred to in section 5.2, namely that increasing β not necessarily means getting a better result. This might be due to pruning sometimes not being done only based on f, but also on other criteria, simply because more than β states turn out to be promising enough. Although this mainly has a noticeable effect in smaller instances, it is undesired and does not occur in its flexible variant. The fact, by the way, that a much bigger beam width was also needed for the flexible search in comparison with previous instances may indicate that the evaluation function can still be improved.

Furthermore, it is interesting to note that for smaller instances, the distributed algorithm performs worse than the sequential version, which can be seen in the (50,50,20) case, where we performed both a sequential and a distributed search. The S_i sets in the state space are all relatively small, making the communication overhead of the distributed algorithm noticeable. This seems to be directly related to the argument found in the literature against distributed beam search in a more traditional setting [5], mentioned in more detail in section 8. Besides that, note that the result obtained with the distributed search is better than the one of the sequential search, even though the beam widths are equal. This is due to tie-breaking, which, in a distributed environment, can happen at multiple places in a single level, instead of only at one point. In the flexible search, where tie-breaking is avoided altogether, this behaviour does not appear.

The (100,100,50) and the (100,100,80) case have a big difference in execution time, while the number of states in the latter case is even lower. However, although the number of expanded states is lower in the (100,100,80) case, the number of encountered and evaluated states is much higher. This is directly related to the maximum size of the travelling group k.

The last two cases could not be solved using flexible beam search. The main reason for this is that in many levels all states had to be expanded, since no states could be pruned based on f. This shows that the flexible variant can point to the necessity to design a better evaluation function, in this case for instance one, that also takes the number of finches in the group into account. Finally, as stated earlier in section 6, for the flexible search, overall β is more stable compared to the non-flexible search. This means in general, that, given some search results, it is easier to determine β for a new flexible search, than for a new non-flexible one.

8 Related Work

Concerning scheduling, quite some research has been done in the field of timed automata. In a paper by Niebert, et al. [28], the problem of minimum-time reachability for timed automata is considered. In several papers by Behrmann, et al. (e.g. [2]), linearly

Table 1. Zebra Finch problem results

Instance			minimal-cost search			g-synch. detailed BS				Flex. g-synch. det. BS			
n	m	k	result	# states	time	β	result	# states	time	β	result	# states	time
10	5	5	19	228,737	00:00:29	400	19	58,272	00:00:14	400	19	67,804	00:00:18
10	10	5	21	513,123	00:01:07	400	21	65,605	00:00:18	400	21	85,633	00:00:24
10	10	8	10	2,020,061	00:04:28	450	10	48,669	00:00:19	400	10	69,550	00:00:21
50	50	5	121	18,157,429	00:48:13	1,000	121	641,315	00:04:49	400	121	298,065	00:02:31
50	50	10	41	475,744,120 *	05:13:26	1,000	43	637,285	00:07:28	-	-	-	-
50	50	10	-	-	-	1,500	44	946,660	00:13:37	-	-	-	-
50	50	10	-	-	-	4,000	43	2,139,347	…	4,000	42	2,365,102	…
50	50	20	-	-	-	5,000	24	3,478,600	01:14:00	1,500	22	1,649,203	…
50	50	20	-	-	-	5,000	20	3,095,782 *	02:01:05	4,000	20	2,579,479 *	01:48:16
100	100	10	-	-	-	5,000	87	6,009,134 *	01:39:52	4,000	87	5,318,589 *	06:22:54
100	100	20	-	-	-	5,000	41	5,884,895 *	00:42:48	4,000	42	5,433,733 *	04:02:26
100	100	50	-	-	-	20,000	17	27,366,213 *	02:57:21	4,000	18	41,611,293 *	06:16:29
100	100	80	-	-	-	20,000	10	19,107,091 *	ca. 24h	…	…	…	…
200	200	50	-	-	-	50,000	35	135,964,662 *	ca. 36h	…	…	…	…

priced timed automata are introduced as an extension of timed automata with prices on both transitions and locations. They consider the minimum-cost reachability problem and an algorithmic solution is offered. In [37], an approach specific for SPIN is presented using a depth-first search algorithm.

There are many papers on solving job-shop scheduling problems, for instance [10]. Most approaches, however, are specifically designed for job-shop problems, while the techniques described in this paper are also meant for other, industrial systems.

Distributed state space generation has appeared in various forms and in various settings, we will just mention a few here. An early approach not limited to any specific input language was proposed in [11]. In [13], a distributed generation algorithm is presented for the MURϕ verifier. Based on this technique a distributed UPPAAL has been developed [3]. An implementation of a distributed state space exploration algorithm based on the SPIN model checker [26] exists. In [18], a method is described to generate LTSs in a distributed way by means of the CADP model checker. All these approaches, however, focus on exhaustive state space generation and not on heuristically pruning parts of the state space on-the-fly in order to solve a particular kind of problem. In [23], a distributed, external version of A^* is developed, combining the fields of distributed, directed and external model checking.

Attempts to create a distributed beam search can be found outside of model checking [5]. In those settings one usually works with search trees which have a much lower average branching factor (the number of outgoing transitions per state) compared to an average state space. Because of this, small beam widths, usually not bigger than 10, can be used, making a distributed beam search counter-productive due to the communication overhead (a similar result can be found in section 7.2). In model checking, however, we wish to deal with arbitrary state spaces, where the average branching factor can be much higher, thereby, for bigger instances, making a distributed beam search effective.

Relating our extensions of beam search to other work, in [16], best-first search is extended to k-best-first search, allowing to compensate for inaccuracies in the evaluation function by selecting in each iteration more than only the best state. Essentially, the difference between k-best-first search and beam search is the decision to keep states not selected in one iteration for the next iteration. This makes k-best first search a complete search, but it also means its memory requirement is higher, since there is no pruning done. A trade-off can, however, be achieved, by using inadmissible heuristics, such that fewer states are expanded, but the solution will be near-optimal. This trade-off is also used for weighted A^* [33] and linear-space best-first search [25], where the h-function is multiplied by some factor. Moreover, in the latter, the memory requirement is linear in the size of the search depth. Our extension of g-synchronised beam search can probably best be compared with filtered beam search [30], in the sense that in each iteration, the current set of states undergoes two phases; in filtered beam search, first a priority beam search is applied, and on the outcome of that, detailed beam search is used, this to lessen the computational complexity. In g-synchronised beam search, we first postpone some states, and then prune states from the remaining set.

In [20], the development of heuristics is the main focus, making it nicely connecting to this paper, in the sense that we start with the assumption of having a heuristic function. Their objective is to model check Java programs with heuristics constructed using the properties to check, the structure of the programs and additional input of the user. They use a number of search algorithms, one of which is beam search. Their beam search, however, seems to deviate from the traditional notion, in that $f(s) = h(s)$, making it practically a linear space greedy search. Furthermore, they include duplicate detection, but do not consider other extensions in order to deal more efficiently with arbitrary state spaces, such as a flexible search.

Finally, in [43], beam search is extended to a complete search, by using a new data structure, called a beam stack. With this it is possible to achieve a range of searches, from depth-first search ($\beta = 1$) to BFS ($\beta \to \infty$). Considering our extensions for arbitrary state spaces, it would be interesting to try to combined these two approaches.

9 Conclusions

We presented a distributed version of detailed beam search, used in a model checking setting. Due to the global view of detailed beam search, creating this version was non-trivial. In practice it shows that for bigger problem instances, the distributed algorithm pays off. We developed a variant called g-synchronised beam search, which considers the states sorted by increasing g. It does not need the storage of g-values of all states when using additional cost actions, since reopening is never necessary. Furthermore, we observed that sometimes increasing β does not lead to finding better results, due to the sometimes cutting away of states which are promising enough. To avoid this unwanted behaviour, we created a (distributed) flexible variant of beam search.

Future work Usage in practice indicated that modelling time using a sequence of *tick* actions leads to state space explosions very quickly. The searches could be adapted to deal with $tick(t)$ actions, where $t \in \mathbb{N}$ denotes a number of time units delayed at once. Furthermore, as the construction of a suitable f is a big problem when using heuristics, it might be interesting to try to quantify the effectiveness of a given function.

Acknowledgements We thank the anonymous reviewers of MoChArtIV for their constructive comments, and Mohammad Torabi Dashti for the help in designing the distributed detailed beam search algorithm.

References

1. J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monograph. Springer, 2002.
2. G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, and J.M.T. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *Proc. TACAS'01*, volume 2031 of *LNCS*, pages 174–188, 2001.
3. G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking - How the Search Order Matters. In *Proc. CAV'00*, volume 1855 of *LNCS*, pages 216–231, 2000.
4. J. Bergstra and J. Klop. Algebra of Communicating Processes with Abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.
5. R. Bisiani. Beam Search. In *Encyclopedia of Artificial Intelligence*, pages 1467–1568. Wiley Interscience Publication, 1992.
6. S. Blom and S. Orzan. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. In *Proc. of PDMC 2002*, volume 68 (4) of *ENTCS*. Elsevier, 2002.
7. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *Proc. CAV 2001*, volume 2102 of *LNCS*, pages 250–254, 2001.
8. S.C.C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In *Proc. PSI 2003*, volume 2890 of *LNCS*, pages 178–192, 2003.
9. G. Bolch, S. Greiner, H. de Meer, and K. Shridharbhai Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, 2nd Edition*. Wiley, 2006.
10. P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(6):107–127, 1994.
11. G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.
12. F. Della Croce and V. T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *Journal of the Operational Research Society*, 53:1275–1280, 2002.
13. D. Dill. The Murϕ Verification System. In *Proc. CAV'96*, volume 1102 of *LNCS*, pages 390–393, 1996.
14. H.E. Dudeney. *Amusements in Mathematics*, chapter 9, pages 112–114. Dover Publications, Inc., 1958.
15. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed Explicit-state Model Checking in the Validation of Communication Protocols. *STTT*, 5:247–267, 2003.
16. A. Felner, S. Kraus, and R.E. Korf. KBFS: K-Best-First Search. *AMAI*, 39(1-2):19–39, 2003.
17. M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, CMU, 1983.
18. H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *Proc. SPIN'01*, volume 2057 of *LNCS*, pages 217–234, 2001.
19. P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Proc. TACAS'02*, volume 2280 of *LNCS*, pages 266–280, 2002.
20. A. Groce and W. Visser. Heuristics for Model Checking Java Programs. *STTT*, 6(4):260–276, 2004.

21. J.F. Groote and M.A. Reniers. *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.
22. Michael Huth and Marta Kwiatkowska. Quantitative Analysis and Model Checking. In *Proc. LICS'97*, pages 111–127. IEEE Computer Society, 1997.
23. S. Jabbar and S. Edelkamp. Parallel External Directed Model Checking With Linear I/O. In *Proc. VMCAI'06*, volume 3855 of *LNCS*, pages 237–251, 2006.
24. R.E. Korf. Uniform-cost Search. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1461–1462. New York, NY: Wiley-Interscience, 1992.
25. R.E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.
26. F. Lerda and R. Sista. Distributed-Memory model checking with SPIN. In *Proc. SPIN'99*, volume 1680 of *LNCS*, pages 22–39, 1999.
27. B.T. Lowerre. *The HARPY speech recognition system*. PhD thesis, CMU, 1976.
28. P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *Proc. MED 2000*. IEEE, 2000.
29. S. Oechsner and O. Rose. Scheduling Cluster Tools Using Filtered Beam Search and Recipe Comparison. In *Proc. 2005 Winter Simulation Conference*, pages 2203–2210. IEEE, 2005.
30. P.S. Ow and E.T. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:35–62, 1988.
31. P.S. Ow and S.F. Smith. Viewing scheduling as an opportunistic problem-solving process. *Annals of Operations Research*, 12:85–108, 1988.
32. D. Peled, V. Pratt, and G. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *DIMACS series in discrete mathematics and theoretical computer science*. AMS, 1996.
33. I. Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1:193–204, 1970.
34. A.W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice-Hall, 1994.
35. S. Rubin. *The ARGOS Image Understanding System*. PhD thesis, CMU, 1978.
36. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
37. T.C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *Proc. 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 1–17, 2003.
38. I. Sabuncuoglu and M. Bayiz. Job shop scheduling with beam search. *European Journal of Operational Research*, 118:390–412, 1999.
39. M. Torabi Dashti and A.J. Wijs. Pruning State Spaces Using Extended Beam Search. To be published, `http://www.cwi.nl/~wijs/beamsearch.pdf`, June 2006.
40. A.J. Wijs and W.J. Fokkink. From χ_t to μCRL: Combining Performance and Functional Analysis. In *Proc. ICECCS'05*, pages 184–193. IEEE Computer Society Press, 2005.
41. A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving Scheduling Problems by Untimed Model Checking. In *Proc. FMICS'05*, pages 54–61. ACM Press, 2005. Extended version as CWI technical report SEN-R0608, `http://db.cwi.nl/rapporten/abstract.php?abstractnr=2034`.
42. R.A. Zann. *The Zebra Finch - A Synthesis of Field and Laboratory Studies*. Oxford University Press Inc., 1996.
43. R. Zhou and E.A. Hansen. Beam-Stack Search: Integrating Backtracking with Beam Search. In *Proc. ICAPS'05*, pages 90–98. AAAI, 2005.