Towards Informed Swarm Verification

Anton Wijs*

Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands
A.J.Wijs@tue.nl

Abstract. In this paper, we propose a new method to perform large scale grid model checking. A manager distributes the workload over many embarrassingly parallel jobs. Only little communication is needed between a worker and the manager, and only once the worker is ready for more work. The novelty here is that the individual jobs together form a so-called cumulatively exhaustive set, meaning that even though each job explores only a part of the state space, together, the tasks explore all states reachable from the initial state.

Keywords: parallel model checking, state space exploration.

1 Introduction

In (explicit-state) Model checking (MC), the truth-value of a logical statement about a system specification, i.e. design, (or directly software code) is checked by exploring all its potential behaviour, implicitly described by that specification, as a directed graph, or state space. A flawed specification includes undesired behaviour, which is represented by a trace through the corresponding state space. With MC, we can find such a trace, and report it to the developers. To show flaw (bug) absence, full exploration of the state space is crucial. However, in order to explore a state space at once, it needs to be stored in the computer's main memory, and often, state spaces are too large, possibly including billions of states. A secondary point of concern was raised in [22,23]: as the amount of available main memory gets bigger, it becomes technically possible to explore large state spaces using existing sequential, i.e. single-processor, techniques, but the time needed to do so is practically too long. Therefore new techniques are needed, which can exploit multi-core processors and grid architectures.

We envision an 'MC@Home', similar to SETI@Home [34], where machines in a network or grid can contribute to solving a computationally demanding problem. In many application areas, this is very effective. BOINC [8] has about 585,000 computers processing around 2.7 petaFLOPS, topping the current fastest supercomputer (IBM Roadrunner with 1.026 PFLOPS). However, flexible grid MC does not exist yet; current *distributed* MC methods, in which multiple machines are employed for a single MC task, need lots of synchronisation between the

* Supported by the Netherlands Organisation for Scientific Research (NWO) project 612.063.816 *Efficient Multi-Core Model Checking.*

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 422–437, 2011.

computers ('workers'), which is a serious bottleneck. MC is both computationally expensive, and cannot obviously be distributed over so-called *embarrassingly parallel* [14] processes, i.e. processes which do not synchronise with each other. In this paper, we propose a method to divide a state space reachability task into multiple smaller, embarrassingly parallel, subtasks. The penalty for doing so is that some parts of the state space may be explored multiple times, but, as noted by [22], this is probably unavoidable, and not that important, if enough processing power is available. What sets the method which we present in this paper apart from previous ones is that we distribute the work over a so-called *cumulatively exhaustive set* (Ces) of search instructions, where each individual instruction yields a strictly non-exhaustive search, in which a strict subset of the set of reachable states is explored, hence less time and memory is needed, while it is also guaranteed that the searches yielded by all instructions *together* search the whole state space. This is novel, since partial (or non-exhaustive) searches, such as random walk [40] and beam search [28,38,41] are typically very useful to detect bugs quickly, but cannot provide a guarantee of bug-absence. In our case, if all searches instructed by the (finitely-sized) Ces cannot uncover a bug, we can conclude that the state space is bug-free. We believe that a suitable method for large scale grid MC must be efficient both memory-wise and time-wise; distributed MC techniques tend to scale very well memory-wise, but disappointingly time-wise, while papers on *multi-core* MC, in which multiple cores on a single processor are employed for a single MC task, tend to focus entirely on speedup, while assuming that the state space fits entirely in the main memory of a single machine. Therefore, we wish to focus more on the combination of time and memory improvements. Our approach is built on the observation that state space explosion is often due to the fact that a system specification is defined as a set of processes in parallel composition, while those processes in isolation do not yield large state spaces. The only serious requirement which systems must meet for our method to be applicable at the moment, is that at least one process in the specification yields finite, i.e. cycle-free, behaviour; we can enforce this by performing a bounded analysis on a process yielding infinite behaviour, but then, we do not know a priori whether the swarm will be cumulatively exhaustive.

The structure of the paper is as follows: in the next Section, related work is discussed. In Section 3, preliminary notions are explained. Then, Section 4 contains a discussion on directed search techniques. After that, in Section 5, we explain the basics of our method for system with independent parallel processes. A more challenging setup with synchronising parallel processes, together with our algorithms, are presented in Section 6. Then, experimental results are given in Section 7, and finally, conclusions and future work appear in Section 8.

2 Related Work

Concerning the state space explosion problem, over the years, many techniques have been developed to make explicit-state MC tasks less demanding. Prominent examples are reduction techniques like partial order reduction [31], and directed

MC [12], which covers the whole range of state space exploration algorithms. Some of these use heuristics to find bugs quickly, but if these are inaccurate, or bugs are absent, they have no effect on the time and memory requirements.

In distributed algorithms such as e.g. in [3,4,5,6,10,15,27,32], multiple workers in a cluster or grid work together to perform an MC task. This has the advantage that more memory is available; in practice, though, the techniques do not scale as well as desired. Since the workers need to synchronise data quite frequently, for very large state spaces, the time spent on synchronisation tends to be longer than the time spent on the actual task. Furthermore, if one of the workers is considerably slower than the others or fails entirely, this has a direct effect on the whole process. Another development is multi-core MC. Since a few years, Moore's Law no longer holds, meaning that the speed of new processors does not double every two years anymore. Instead, new computers are equipped with a growing number of processor cores. For e.g. MC, this means that in order to speedup the computations, the available algorithms must be adapted. In multi-core MC, we can exploit that the workers share memory. Major achievements are reported in e.g. [1,20,21,26]. [26] demonstrates a significant speedup in a multi-core breadth-first search (Bfs) using a lock-free hash table. However, papers on multi-core MC tend to focus on reducing the time requirements, and it is assumed that the entire state space fits in the main memory of a single machine.

A major step towards efficient grid MC was made with Swarm Verification (SV) [22,23,24] and Parallel Randomized State Space Search [11,35], which involve embarrassingly parallel explorations. They require little synchronisation, and have been very successful in finding bugs in large state spaces quickly. Bug absence, though, still takes as much time and memory to detect than a traditional, sequential search, since the individual workers are unaware of each other's work, and each worker is not bounded to a specific part of the state space. The method we propose is based on SV, and since each worker uses particular information about the specification to guide the search, we call it *informed* SV (ISV), relating it to informed search techniques in directed MC. Similar ideas appear in related work: in [27], it is proposed to distribute work based on the behaviour of a single process. The workers are not embarrassingly parallel, though. A technique to restrict analysis of a program based on a given trace of events is presented in [16]. It has similarities with ISV, but also many differences; their technique performs slicing on a deterministic C program, and is not designed for parallel MC, whereas ISV distributes work to analyse concurrent behaviour of multiple processes based on the behaviour of a subsystem. Finally, a similar approach appears in [36], but there, it is applied on symbolic execution trees to generate test cases for software testing. Unlike ISV, they distribute the work based on a (shallow) bounded analysis of the whole system behaviour.

3 Preliminaries

Labelled Transition Systems Labelled transition systems (Ltss) capture the operational behaviour of concurrent systems. An Lts consists of transitions $s \xrightarrow{\ell} s'$,

meaning that being in a state s, an action ℓ can be executed, after which a state s' is reached. In model checking, a system specification, written in a modelling language, has a corresponding LTS, defined by the structural operational semantics of that language.

Definition 1. *A* labelled transition system (LTS) *is a tuple* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, s_{in})$, *where* $\mathcal{S}$ *is a set of states,* $\mathcal{A}$ *a set of actions or transition labels,* $\mathcal{T}$ *a transition relation, and* s_{in} *the initial state. A transition* $(s, \ell, s') \in \mathcal{T}$ *is denoted by* $s \xrightarrow{\ell} s'$.

A sequence of labels $\sigma = \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$, with $n > 0$, describes a sequence of events relating to a trace in an LTS, starting at s_{in}, with matching labels, i.e. it maps to traces in the LTS with $s_0, \ldots, s_n \in \mathcal{S}$, $\ell_1, \ldots, \ell_n \in \mathcal{A}$, with $s_0 = s_{in}$, such that $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} s_n$. Note that σ maps to a single trace iff the LTS is *label-deterministic*, i.e. that for all $s \in \mathcal{S}$, if there exist $s \xrightarrow{\ell'} s'$ and $s \xrightarrow{\ell''} s''$ with $s' \neq s''$, then also $\ell' \neq \ell''$. If the LTS is not label-deterministic, then σ may describe a set of traces. In this paper, we assume that LTSs are label-deterministic, but this is strictly not required. The set of enabled transitions restricted to a set of labels $A \in \mathcal{A}$ in state s of LTS $\mathcal{M}$ is defined as $en_{\mathcal{M}}(s, A) = \{t \in \mathcal{T} \mid \exists s' \in \mathcal{S},\ \ell \in A.\ t = s \xrightarrow{\ell} s'\}$. Whenever $en_{\mathcal{M}}(s, \mathcal{A}) = \emptyset$, we call s a *deadlock* state. For $T \subseteq \mathcal{T}$, we define $nxt(T) = \{s \in \mathcal{S} \mid \exists s' \xrightarrow{\ell} s \in T\}$. This means that $nxt(en_{\mathcal{M}}(s, \mathcal{A}))$ is the set of immediate successors of s.

System specifications often consist of a finite number of process specifications in parallel composition. Then, the process specifications describe the potential behaviour of individual system components. The potential behaviour of all these processes concurrently then constitutes the LTS of the system as a whole. What modelling language is being used to specify these systems is unimportant here; we only assume that the process specifications can be mapped to process LTSs, and that the processes can interact using *synchronisation* actions.

Next, we will highlight how a system LTS can be derived from a given set of process LTSs and a so-called *synchronisation function*. System behaviour can be described by a finite set Π of $n > 0$ process LTSs $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, s_{in,i})$, for $1 \leq i \leq n$, together with a partial function $\mathfrak{C} : \mathcal{A}^s \times \mathcal{A}^s \to \mathcal{A}^f$, with $\mathcal{A}^s = \bigcup_{1 \leq i \leq n} \mathcal{A}_i$ and $\mathcal{A}^f$ a set of actions representing successful synchronisation, describing the potential synchronisation behaviour of the system, i.e. it defines which actions $\ell, \ell' \in \bigcup_{1 \leq i \leq n} \mathcal{A}_i$ can synchronise with each other, resulting in an action $\ell'' \in \mathcal{A}^f$. We write $\mathfrak{C}(\{\ell, \ell'\}) = \ell''$, to indicate that the order of ℓ and ℓ' does not matter.[1] Furthermore, we assume that each action ℓ is always only involved in at most one synchronisation rule, i.e. for each ℓ, there are no two distinct ℓ', ℓ'' such that both $\mathfrak{C}(\{\ell, \ell'\})$ and $\mathfrak{C}(\{\ell, \ell''\})$ are defined. Definition 2 describes how to construct a system LTS from a finite set Π of process LTSs.

Definition 2. *Given a set* Π *of* $n > 0$ *process* LTSs $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, s_{in,i})$, *for* $1 \leq i \leq n$, *and synchronisation function* $\mathfrak{C} : \mathcal{A}^s \times \mathcal{A}^s \to \mathcal{A}^f$, *with* $\mathcal{A}^s =$

[1] In practice, synchronisation rules can also be defined for more than two parties, resulting in *broadcasting* rules. In this paper, we restrict synchronisation to two parties. Note, however, that the definitions can be extended to support broadcasting.

$\bigcup_{1 \leq i \leq n} \mathcal{A}_i$ *and* $\mathcal{A}^f$ *a set of actions representing successful synchronisation, we construct a system* Lts $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, s_{in})$ *as follows:*

- $s_{in} = (s_{in,1}, \ldots, s_{in,n})$;
- *Let* $z_1 = (s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n) \in \mathcal{S}$ *with* $i \neq j$.
  - *If for some* $\mathcal{M}_i \in \Pi$, $s_i \xrightarrow{\ell} s_i'$ *with* $\ell \in \mathcal{A}_i$, *and there does not exist* $\ell' \in \mathcal{A}^s$ *such that* $\mathfrak{C}(\{\ell, \ell'\}) = \ell''$, *for some* $\ell'' \in \mathcal{A}^f$, *then* $z_2 = (s_1, \ldots, s_i', \ldots, s_j, \ldots, s_n) \in \mathcal{S}$. *In this case,* $\ell \in \mathcal{A}$ *and* $z_1 \xrightarrow{\ell} z_2 \in \mathcal{T}$;
  - *If for some* $\mathcal{M}_i \in \Pi$, $s_i \xrightarrow{\ell} s_i'$ *with* $\ell \in \mathcal{A}_i$, *and for some* $\mathcal{M}_j \in \Pi$ $(i \neq j)$, $s_j \xrightarrow{\ell'} s_j'$ *with* $\ell' \in \mathcal{A}_j$, *and* $\mathfrak{C}(\{\ell, \ell'\}) = \ell''$, *for some* $\ell'' \in \mathcal{A}^f$, *then* $z_2 = (s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n) \in \mathcal{S}$. *In this case,* $\ell'' \in \mathcal{A}$ *and* $z_1 \xrightarrow{\ell''} z_2 \in \mathcal{T}$.

4 Directed Lts Search Techniques

The two most basic Lts exploration algorithms available in model checkers are Bfs and depth-first search (Dfs). They differ in the order in which they consider states for exploration. In Bfs, states are explored in order of their distance from s_{in}. Dfs gives priority to searching at increasing depth instead of exploring all states at a certain depth before continuing. If at any point in the search, the selected state has no successors, or they have all been visited before, then Dfs will backtrack to the parent of this state, and explore the next state from the parent's set of successors, according to the ordering function. Bfs and Dfs are typical *blind* searches, since they do not take additional information about the system under verification into account. In contrast to this are the *informed* searches which do use such information. Examples of informed searches are *Uniform-cost search* [33], also known as *Dijkstra's search* [9], and A^* [19].

All searches, both blind and informed, mentioned so far are examples of *exhaustive* searches, i.e. in the absence of deadlocks, they will explore all states reachable from s_{in}. Another class of searches is formed by the *non-exhaustive* searches. These searches prune the Lts on-the-fly, completely ignoring those parts which are deemed uninteresting according to some heuristics. At the cost of losing completeness, these searches can find deadlocks in very large Ltss fast, since they can have drastically lower memory and time requirements, compared to exhaustive searches. A blind search in this category is random walk, or simulation, in which successor states are chosen randomly. An informed example is *beam search*, which is basically a Bfs-like search, where in each iteration, i.e. depth, only up to $\beta \in \mathbb{N}$, which is given a priori, states are selected for exploration. For the selection procedure, various functions can be used; in classic beam search, a (state-based) function as in A^* is used, in *priority* beam search [38,39,41], a selection procedure based on transition labels is employed, while *highway search* [13] uses random selection, and is therefore a blind variant of beam search.

For a search L, let us define its scope $\text{REACH}_{\mathcal{M}}(L)$ in a given LTS $\mathcal{M}$ as the set of states in $\mathcal{M}$ that it will explore. For all exhaustive L, we have $\text{REACH}_{\mathcal{M}}(L) = \mathcal{S}$, while for all non-exhaustive L, $\text{REACH}_{\mathcal{M}}(L) \subset \mathcal{S}$. Let us consider two searches L_1 and L_2, such that $\text{REACH}_{\mathcal{M}}(L_1) \cup \text{REACH}_{\mathcal{M}}(L_2) = \mathcal{S}$. Then, we propose to call $\{L_1, L_2\}$ *cumulatively exhaustive* on $\mathcal{M}$. Such *cumulatively exhaustive sets* (CESs) are very interesting for SV; the elements can be run independently in parallel, and once they are all finished, the full LTS will have been explored.

Some existing searches lead to CESs. *Iterative deepening* [25] uses depth-bounded DFS in several iterations, each time relaxing the bound. Each iteration can be seen as an individual search, subsequent searches having increasing scopes. Iterative searches form a class, which includes e.g. IDA* [25]. Another class leading to CESs consists of random searches like random walk and highway search. However, all these are not suitable for grid computing. Iterative searches form CESs containing an exhaustive search. If $\mathcal{M}$ is bug-free, then eventually this search is performed, which is particularly inefficient. With random searches, there is no guarantee that after n searches, all reachable states are visited. If $n \to \infty$, the set will eventually be cumulatively exhaustive, but performing the searches may take forever. Moreover, the probabilities to visit states in a random walk are not uniformly distributed, but depend on the graph structure [30].

We want to derive CESs with a bounded number of non-exhaustive elements from a system under verification. Preferably, all scopes have equal size, to ensure load-balancing, but this is not necessary (in SV, workers do not synchronise, hence load-balancing is less important [22,23]). To achieve this, we have developed a search called *informed swarm search* (ISS) which accepts a guiding function, and we have a method to compute a set of guiding functions $f_0, f_1, \ldots, f_n$ given a system specification, such that $\{\text{ISS}(f_0), \text{ISS}(f_1), \ldots\}$ is a CES. The guiding functions actually relate to traces through the LTS of a subsystem π of the system under verification, which are derived from an LTS exploration of π. Such an LTS can in practice be much smaller than the system LTS. For now, we require that π yields finite behaviour, i.e. that its LTS is cycle-free.

ISS only selects those transitions for exploration which either do not stem from π, or which correspond with the current position in the given trace through the LTS of π. This is computationally inexpensive, since it only entails label comparison. The underlying assumption is that labels can be uniquely mapped to process LTSs; given a label, we know from which process it stems. If a given set of LTSs does not meet this requirement, some label renaming can fix this.

5 Systems with Independent Processes

In this section, we will explain our method for constructing CESs for very basic specifications which consist of completely independent processes in parallel composition. We are aware of the fact that such specifications may practically not be very interesting. However, they are very useful for our explanation.

Figure 1 presents two LTSs of a beverage machine, and a snack machine, respectively. Both are able to dispense goods when one of their buttons is pressed.

There is no interaction between the machines. We have $\Pi = \{\mathcal{M}_b, \mathcal{M}_s\}$, with $\mathcal{M}_b$ and $\mathcal{M}_s$ the LTSs of the beverage machine and the snack machine, respectively.

If we perform a DFS through $\mathcal{M}_b$, and we record the encountered traces whenever backtracking is required, we get the following set: $\{\langle push\_button(1), get\_coffee\rangle, \langle push\_button(2), get\_tea\rangle\}$. Note that all reachable states of $\mathcal{M}_b$ have been visited. We use these two traces as guiding principles for two different searches through $\mathcal{M}_{bs}$, which is the LTS obtained by placing the two LTSs in parallel composition. Algorithm 1 presents the pseudo-code of our ISS, which accepts a trace σ and a set of transition labels $\mathcal{A}^{ex}$ to guide the search. In our example, each ISS, with one of the two traces and $\mathcal{A}^{ex} = \mathcal{A}_b$ as input, focusses on specific behaviour of the beverage machine within the bigger system context.

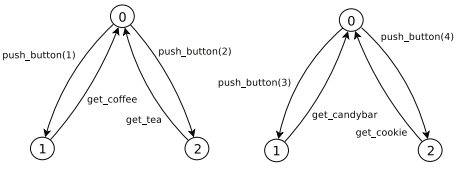

Fig. 1. Two LTSs of a beverage and a snack machine

Fig. 2 shows which states will be visited in $\mathcal{M}_{bs}$ if we perform an ISS based on $\langle push\_button(1), get\_coffee\rangle$. Alg. 1 explains how this is done. Initially, we put s_{in} in *Open*. Then, we add all successors reached via a transition with a label not in $\mathcal{A}^{ex}$ in *Next*, and add all successors reached via a transition labelled $\sigma(i)$ in *Step*. Here, $\sigma(i)$ returns the $(i+1)^{th}$ element in the trace σ; if i is bigger than or equal to the trace length, we say that $\sigma(i) = \perp$, where $\perp$ represents 'undefined', and $\{\perp\}$ is equivalent to $\emptyset$. For now, please ignore the next step concerning $\mathcal{F}_i$; it has to do with feedback to the manager, and will be explained later. Finally, s_{in} is added to *Closed*, i.e. the set of explored states, and the states in *Next* which have not been explored constitute the new *Open*. Then, the whole process is repeated. This continues until $Open = \emptyset$. Then, the contents of *Step* is moved to *Open*, and the ISS moves on to the next step in σ. In this way, the ISS explores all traces $\gamma = \langle \alpha_0, \sigma(0), \alpha_1, \ldots, \alpha_n, \sigma(n), \alpha_{n+1}\rangle$, with n the length of σ and the α_i traces

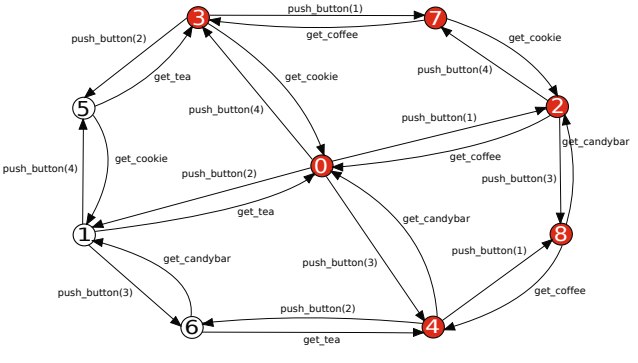

Fig. 2. A search through $\mathcal{M}_{bs}$ with $\mathcal{M}_b$ restricted to $\langle push\_button(1), get\_coffee\rangle$

Algorithm 1. BFS-based Informed Swarm Search

Require: Implicit description of $\mathcal{M}$, exclusion action set $\mathcal{A}^{ex}$, swarm trace σ
Ensure: $\mathcal{M}$ restricted to σ is explored
$i \leftarrow 0$
$Open \leftarrow s_{in}$; $Closed, Next, Step, \mathcal{F}_i \leftarrow \emptyset$
while $Open \neq \emptyset \vee Step \neq \emptyset$ **do**
  if $Open = \emptyset$ **then**
    $i \leftarrow i + 1$
    $Open \leftarrow Step \setminus Closed$; $Step, \mathcal{F}_i \leftarrow \emptyset$
  end if
  for all $s \in Open$ **do**
    $Next \leftarrow Next \cup nxt(en_{\mathcal{M}}(s, \mathcal{A} \setminus \mathcal{A}^{ex}))$
    $Step \leftarrow Step \cup nxt(en_{\mathcal{M}}(s, \{\sigma(i)\}))$
    $\mathcal{F}_i \leftarrow \mathcal{F}_i \cup \{\ell \mid \exists s' \in \mathcal{S}.(s \xrightarrow{\ell} s') \in en_{\mathcal{M}}(s, \mathcal{A}^{ex})\}$
  end for
  $Closed \leftarrow Closed \cup Open$
  $Open \leftarrow Next \setminus Closed$; $Next \leftarrow \emptyset$
end while

containing only labels from $\mathcal{A} \setminus \mathcal{A}_b$. If we perform an ISS for every trace through $\mathcal{M}_b$, we will visit all reachable states in $\mathcal{M}_{bs}$. Figure 2 shows what the ISS with $\langle push\_button(1), get\_coffee \rangle$ explores; out of the 9 states, 6 are explored, meaning that 33% of $\mathcal{M}_{bs}$ could be ignored. The ISS using $\langle push\_button(2), get\_tea \rangle$ also explores 6 states, namely 0, 3 and 4 (the states reachable via behaviour from $\mathcal{M}_s$), and 1, 5, and 6. In this way, some states are explored multiple times, but we have a CES of non-exhaustive searches through $\mathcal{M}_{bs}$.

6 Systems with Synchronising Processes

Next, we consider parallel processes which synchronise. In such a setting, things get slightly more complicated. Before we continue with an example, let us first formally define a subsystem and the LTS it yields.

Definition 3. *Given a set Π of $n > 0$ process* LTS*s $\mathcal{M}_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, s_{in}i)$, for $1 \leq i \leq n$, and a synchronisation function $\mathfrak{C} : \mathcal{A}^s \times \mathcal{A}^s \to \mathcal{A}^f$, we call a subset $\pi \subseteq \Pi$ of* LTS*s a* subsystem *of Π. We can derive a synchronisation function $\mathfrak{C}_\pi : \mathcal{A}^s_\pi \times \mathcal{A}^s_\pi \to \mathcal{A}^f_\pi$ from $\mathfrak{C}$ as follows: $\mathcal{A}^s_\pi = \bigcup_{\mathcal{M}_i \in \pi} \mathcal{A}_i$, and for all $\ell, \ell' \in \mathcal{A}^s_\pi$, if $\mathfrak{C}(\{\ell, \ell'\}) = \ell''$, for some $\ell'' \in \mathcal{A}^f$, we define $\mathfrak{C}_\pi(\{\ell, \ell'\}) = \ell''$ and $\ell'' \in \mathcal{A}^f_\pi$.*

Note that the LTS of a subsystem π, which can be obtained with Definition 2, describes an over-approximation of the potential behaviour of π within the bigger context of Π. This is because in π, it is implicitly assumed that all synchronisation with external processes (which are in Π, but not in π) can happen whenever a process in π can take part in it. For the ISS, we have to choose $\mathcal{A}^{ex}$ more carefully, and we need to post-process traces σ yielded from π; we must take synchronisation between π and the larger context into account. For this, we define a relabelling function $R : \mathcal{A}_\pi \to \mathcal{A}^f$ as follows: $R(\ell) = \ell$ if there exists no ℓ' such that $\mathfrak{C}(\{\ell, \ell'\})$ is defined, and $R(\ell) = \ell''$ if there exists an ℓ' such that $\mathfrak{C}(\{\ell, \ell'\}) = \ell''$. Then, we say that $\mathcal{A}^{ex} = \{R(\ell) \mid \ell \in \mathcal{A}_\pi\}$ and $\sigma'(i) = R(\sigma(i))$ for all defined $\sigma(i)$, such that $\mathcal{A}^{ex}$ and σ' are applicable in the system LTS, because

actions from π which are forced to synchronise are relabelled to the results of those synchronisations. In this case, σ' relates to a single trace iff $\mathfrak{C}$ is injective.

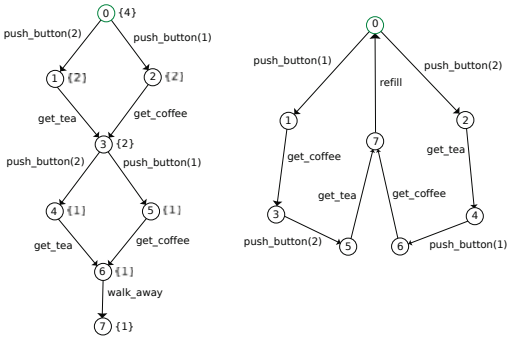

Fig. 3. LTSs of a user of a beverage machine, and a beverage machine, respectively

Fig. 3 shows two LTSs: one of a modified beverage machine $\mathcal{M}_b$, and one of a user $\mathcal{M}_u$ (for now, please ignore the numbers between curly brackets). For the parallel composition $\mathcal{M}_{ub}$, we define: $\mathfrak{C}(\{push\_button, push\_button\}) = button\_pushed$[2], $\mathfrak{C}(\{get\_coffee, get\_coffee\}) = take\_coffee$, $\mathfrak{C}(\{get\_tea, get\_tea\}) = take\_tea$.

First, if $\pi = \{\mathcal{M}_u\}$, observe that a DFS through $\mathcal{M}_u$ does not give us the full set of traces through $\mathcal{M}_u$; if we consider the transition ordering from left to right, a DFS will first provide trace $\langle push\_button(2), get\_tea, push\_button(2), get\_tea, walk\_away\rangle$. Then, it will backtrack to state 3, and continue via 5 to 6.

Since 6 has already been explored, the DFS produces trace $\langle push\_button(2), get\_tea, push\_button(1), get\_coffee\rangle$ and backtracks to 0. Note that this new trace does not finish with *walk_away*. Continuing from 0, the search will finally produce the trace $\langle push\_button(1), get\_coffee\rangle$.

Algorithm 2. Trace-counting DFS

Require: Implicit description of cycle-free $\mathcal{M}$
Ensure: $\mathcal{M}$ and $tc : \mathcal{S} \to \mathbb{N}$ are constructed
  $Closed \leftarrow \emptyset$
  $tc(s_{in}) \leftarrow dfs(s_{in})$
dfs(s) =
  if $s \notin Closed$ **then**
    $tc(s) \leftarrow 0$
    for all $s' \in nxt(en_{\mathcal{M}}(s, \mathcal{A}))$ **do**
      $tc(s) \leftarrow tc(s) + dfs(s')$
    end for
    if $nxt(en_{\mathcal{M}}(s, \mathcal{A})) = \emptyset$ **then**
      $tc(s) \leftarrow 1$
    end if
    $Closed \leftarrow Closed \cup \{s\}$
  end if
  return $tc(s)$

Figure 4 presents $\mathcal{M}_{ub}$. If we use these three traces (after relabelling with R) as guiding functions as in Alg. 1, and define $\mathcal{A}^{ex}$ as mentioned earlier, none of the searches will visit (the marked) state 5! The reason for this is that although in $\mathcal{M}_u$, multiple traces may lead to the same state, in $\mathcal{M}_{ub}$, the corresponding traces may not. This is due to synchronisation. Since the different traces in $\mathcal{M}_u$ synchronise with different traces in $\mathcal{M}_b$ which do not lead to the same state, also in $\mathcal{M}_{ub}$, the resulting traces will lead to different states. One solution is to fully explore $\mathcal{M}_u$ with a DFS without a *Closed* set. However, this is very inefficient, since the complete reachable graph from a state s needs to be explored n times, if n different traces reach s. Instead, we opt for constructing a *weighted* LTS, where each state is assigned a value indicating the number of traces that can be explored from

[2] When transition labels have parameters, they can synchronise iff they have the same parameter values. Then, the resulting transition also has these parameter values.

that state. In Figure 3, these numbers are displayed between curly brackets. The advantage of this is that we can avoid re-exploration of states, and it is in fact possible to uniquely identify traces through $\mathcal{M}_u$ by a trace ID $\in \mathbb{N}$.

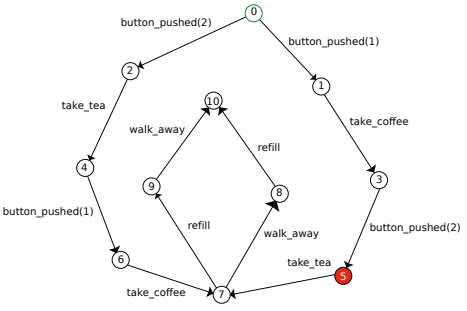

Fig. 4. The LTS of a beverage machine and a user in concurrency

Alg. 2 presents our trace counting search (in which *Closed* is a global variable), which not only explores a full LTS, but also constructs a function $tc : \mathcal{S} \to \mathbb{N}$ indicating the number of traces one can follow from a state. Deadlock states get weight 1, and other states get a weight equal to the sum of the weights of their immediate successors. In DFS, a state s is placed in *Closed* once all states reachable from s have been explored, hence at that moment, we know the final weight of s. This allows us to reuse weight information whenever we visit an explored state. Note that $tc(s_{in})$ equals the number of possible traces through the LTS. Alg. 3 shows how to reconstruct a trace, given its ID between 0 and $tc(s_{in})$. It is important here that for each state, its successor states are always ordered in the same way. In the weighted LTS, each trace from s_{in} to a state s represents a range of trace IDs from *lower*, the maintained lower-bound, to $lower + tc(s)$. Starting at s_{in}, the algorithm narrows the matching range down to the exact ID. At each state, it explores the transition with a matching ID interval to the next state, and continues like this until a deadlock state is reached.

The method works as follows: first, an explicit weighted LTS of a subsystem π is constructed. Whenever a worker is ready for work, he contacts a manager, who then selects a trace ID from the given set of IDs, and constructs the associated trace σ using the weighted LTS. This trace, after relabelling with R, is used by the worker to guide his ISS. Next, we discuss the $\mathcal{F}_i$ in Alg. 1. For each $\sigma(i)$, set $\mathcal{F}_i$ is constructed to hold all labels from $\mathcal{A}_\pi$ after relabelling which are encountered in $\mathcal{M}$ while searching for $\sigma(i)$. Since $\mathcal{M}_\pi$ is an over-approximation of the potential behaviour of π in $\mathcal{M}$,[3] the set of trace IDs is likely to contain many false positives, i.e. behaviour of π which cannot be fully followed in $\mathcal{M}$.

Algorithm 3. Trace Reconstruction

Require: Cycle-free $\mathcal{M}$, $tc : \mathcal{S} \to \mathbb{N}$, $\texttt{ID} \in \mathbb{N}$
Ensure: Trace with given ID is constructed in σ
  $i, lower \leftarrow 0$
  $crt \leftarrow s_{in}$
  for all $(crt \xrightarrow{\ell} s) \in en_{\mathcal{M}}(crt, \mathcal{A})$ **do**
    if $lower + tc(s) > \texttt{ID}$ **then**
      $crt \leftarrow s$
      $\sigma(i) \leftarrow \ell$; $i \leftarrow i + 1$
    else
      $lower \leftarrow lower + tc(s)$
    end if
  end for

[3] Note that only 2 of the 4 traces through $\mathcal{M}_u$ can be followed completely in $\mathcal{M}_{ub}$ by a swarm search.

These $\mathcal{F}_i$ provide invaluable feedback, allowing the manager to prune non-executable traces from the work-list. This is essential to on-the-fly reduce the number of Isss drastically. The manager performs this pruning by traversing the weighted Lts, similar to Algorithm 3, and removing all ranges of trace IDs corresponding to traces which are known to be false positives from a maintained trace set. E.g. if a worker discovers that after an action a of π, the only action of π which can be performed is b, then the manager will first follow a in the weighted Lts of π from s_{in} to a state s, and remove all ID ranges corresponding to the immediate successors of s which are not reached via a transition labelled b from the trace set. The manager will also remove the ID of the trace followed by that worker, if the worker was able to process it entirely. From that moment on, the manager will select trace IDs from the pruned set. This allows for embarrassingly parallel workers, and the manager can dynamically process feedback and provide new work in the form of traces. Furthermore, only little communication is needed, and if a worker fails due to a technical issue, the work can easily be redone.

Note that trace-counting Dfs, like Dfs, runs in $O(|\mathcal{S}| + |\mathcal{T}|)$, and the trace reconstruction search and the ID pruning algorithm run in $O(n + (n * b))$, with n the length of the largest trace through the weighted Lts, and b the maximum number of successors of a state in the Lts. Finally, the complexity of Iss depends on the Lts structure; it is less than $O(|\mathcal{S}| + |\mathcal{T}|)$, as it is non-exhaustive, but also not linear in the length of the longest trace through the Lts, like e.g. beam search, as it prunes less aggressively (not every Bfs-level has the same size).

7 Experiments

All proposed algorithms are implemented as an extension of LTSmin [7]. The advantage of using this toolset as a starting point is that it has interfaces with multiple popular model checkers, such as DiVinE [2] and mCRL2 [17]. However, DiVinE is based on Kripke structures, where the states instead of the transitions are labelled. Future work, therefore, is to develop a state-based ISV.

We have two `bash` scripts for performing real multi-core ISVs and simulating ISVs, in case not enough processors are available. We do not yet support communication between workers and the manager over a network. The functionality of the manager is implemented in new tools to perform the pruning on the current trace ID set, and select new trace IDs from the remaining set. All intermediary information is maintained on disk; Initially, the user has to create a subsystem specification based on the system specification, which is used for trace-counting, leading to a weighted Lts on disk. The selection tool can select available trace IDs from that Lts, and write explicit traces into individual files. Relabelling can also be applied, given the synchronisation rules. The IDs are currently selected such that they are evenly distributed over the ID range. The trace files are accepted by the Iss in LTSmin, applied on the system specification. Finally, the written feedback can be applied on a file containing the current trace ID set.

Table 1. Results for two protocol specifications. ISV n indicates an ISV with n workers. # π-*traces*: estimated # ISSs needed. (1 for single BFS)# ISSs: actual # ISSs needed. *max. states*: largest # states explored by an ISS, or # states explored by BFS. *max. time*: longest running time of an ISS, or running time of BFS.

case	**search**	**results**			
		# π-*traces*	# ISSs	*max. states*	*max. time*
DRM (1nnc, 3ttp)	BFS	1	1	13,246,976	19,477 s
	ISV 10	$1.31 * 10^{13}$	7,070	70,211	177 s
	ISV 100	$1.31 * 10^{13}$	9,900	70,211	175 s
1394 (3 link ent.)	BFS	1	1	137,935,402	105,020 s
	ISV 10	$3.01 * 10^{9}$	1,160	236,823	524 s
	ISV 100	$3.01 * 10^{9}$	1,400	236,823	521 s

We performed a number of experiments using μCRL [18] specifications of a DRM protocol [37] and the Link Layer Protocol of the IEEE-1394 Serial Bus (Firewire) [29] with three parallel link protocol entities. In the first case, we performed trace-counting on the two iPod devices in parallel composition. In the second case, we isolated one of the link protocol entities for the trace-counting, and bounded its infinite behaviour, only allowing the traversal through cyclic behaviour up to 2 times. The experiments were performed on a machine with two dual-core AMD OPTERON (tm) processors 885 2.6 GHz, 126 GB RAM, running RED HAT 4.3.2-7. Creating the weighted LTSs took no more than a few minutes; the DRM weighted LTS contained 962 states, the 1394 weighted LTS 73 states. We simulated the ISVs, executing ISSs in sequence. This influences the outcome slightly, since we process feedback each time n new ISSs have been performed; we do this to approach a real ISV, where, on the one hand, feedback can be processed as it becomes available, but on the other hand, many ISSs can be launched in parallel at the same time, hence when a certain amount of feedback has been processed. Therefore, updating the remaining set of traces after each individual ISS is not really fair. The need to simulate was the main reason that we have not looked at larger LTSs yet. Table 1 presents the results. For smaller instances, we validated that the swarm was cumulatively exhaustive, by writing the full state vectors of states to disk. Observe that initially, the first analyses produced a large over-approximation of the number of ISSs needed (see "#π-*traces*"). The quality of the estimation has an effect on the ISV efficiency, even with feedback. This also has an effect on the difference in efficiency when changing n (the number of parallel workers); when the over-approximation is large, many ISSs may be launched which can not process the given trace entirely, since it is a false positive. As n is increased, the probability of launching such ISSs gets higher, as more ISSs are launched before any new feedback is given. On the other hand, increasing n still reduces the overall execution time, because there is more parallellism.

The ISVs take more time[4] compared to a BFS. This seems to indicate that the method is not interesting. However, keep in mind that already the first few ISSs

[4] Note that the overall execution time takes at most *max. time* $*$ (#ISSs/n) seconds.

reach great depths, and they go into several directions. Therefore, even though we did no bug-hunting, in many cases, it is to be expected that like SV, our ISV will find bugs much quicker than a single search. This could perhaps even be improved by using a DFS-based version of Alg. 1. We plan to do an empirical comparison between such a DFS-based ISV and SV. For full exploration, ISV does not provide a speedup, but this was not intended; instead, observe that the maximum number of states explored in an ISS is much smaller than the LTS size (in the DRM case about $\frac{1}{2}$% of the overall size, in the 1394 case about $\frac{1}{6}$%). A better trade-off could be realised by guiding each ISS with a set of subsystem traces; for the DRM case, an ISS following 10 traces would still explore no more than 5% of the LTS (probably less depending on the amount of redundant work which could now be avoided), while the number of ISSs could be decreased by an order of magnitude. This makes ISV applicable in clusters and grids with large numbers of processors, each having access to e.g. 2 GB memory, even if exploring the whole LTS at once would require much more memory.

8 Conclusions

In this paper, we have proposed a new approach to parallel MC, aimed at large scale grid computing. Part of the system behaviour is analysed in isolation, yielding a set of possible traces, each representing a search through the full LTS. These searches are embarrassingly parallel, since only a trace, the set of actions of the subsystem, and the specification are needed for input. Once a search is completed, feedback is sent to the manager, giving him information on the validity of the remaining traces, which is invaluable, since the set of traces is an over-approximation of the possible traces of the subsystem in the bigger context of the full system. We believe that our method is fully compatible with existing techniques. E.g. one can imagine having multiple multi-core machines available; the ISV method can then be used to distribute the work over these machines, but each machine individually can perform the work using a multi-core search. Also reduction techniques like partial order reduction should be compatible with ISV. For good results, we expect it to be important that both during the analysis of the subsystem and the full system, the same reduction techniques are applied.

For future work, we plan to test ISV more thoroughly to see if it scales to real-life problems, and make the tools more mature. As the size of the subsystem has an effect on the work distribution, it is interesting to investigate what an ideal subsystem relative to a system would be. We also wish to generalise ISV, such that subsystems yielding infinite behaviour can be analysed, and to improve the trace set approximation with e.g. static analysis. One could also construct the trace set according to the MC task, e.g. taking the property to check into account. We plan to investigate different strategies for trace selection. A good strategy sends the workers into very different directions. Finally, [16,36] provide good pointers to develop a state-based ISV.

References

1. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core LTL model-checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
2. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core – A parallel LTL model-checker. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
3. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
4. Behrmann, G., Hune, T., Vaandrager, F.: Distributing Timed Model Checking - How the Search Order Matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
5. Blom, S.C.C., Calamé, J.R., Lisser, B., Orzan, S., Pang, J., van de Pol, J.C., Torabi Dashti, M., Wijs, A.J.: Distributed Analysis with μCRL: A Compendium of Case Studies. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 683–689. Springer, Heidelberg (2007)
6. Blom, S.C.C., Lisser, B., van de Pol, J.C., Weber, M.: A Database Approach to Distributed State Space Generation. In: Haverkort, B., Černá, I. (eds.) PDMC 2007. ENTCS, vol. 198, pp. 17–32. Elsevier, Amsterdam (2007)
7. Blom, S.C.C., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
8. BOINC: Visited on 18 February (2011), `http://boinc.berkeley.edu`
9. Dijkstra, E.W.: A note on two problems in connection with graphs. Numerische Mathematik 1, 269–271 (1959)
10. Dill, D.: The Murphi Verification System. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
11. Dwyer, M.B., Elbaum, S.G., Person, S., Purandare, R.: Parallel Randomized State-space Search. In: 29th Int. Conference on Software Engineering, pp. 3–12. IEEE Press, New York (2007)
12. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT 5(2), 247–267 (2004)
13. Engels, T.A.N., Groote, J.F., van Weerdenburg, M.J., Willemse, T.A.C.: Search Algorithms for Automated Validation. JLAP 78(4), 274–287 (2009)
14. Foster, I.: Designing and Building Parallel Programs. Addison-Wesley, Reading (1995)
15. Garavel, H., Mateescu, R., Bergamini, D., Curic, A., Descoubes, N., Joubert, C., Smarandache-Sturm, I., Stragier, G.: DISTRIBUTOR and BCG_MERGE: Tools for distributed explicit state space generation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 445–449. Springer, Heidelberg (2006)
16. Groce, A., Joshi, R.: Exploiting traces in static program analysis: better model checking through printfs. STTT 10(2), 131–144 (2008)
17. Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y.S., Weerdenburg, M.J.: The mCRL2 Toolset. In: 1st Int. Workshop on Academic Software Development Tools and Techniques 2008, pp. 5-1/10 (2008)
18. Groote, J.F., Ponse, A.: The Syntax and Semantics of μCRL. In: Algebra of Communicating Processes 1994. Workshops in Computing, pp. 26–62. Springer, Heidelberg (1995)

19. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. on Systems, Science and Cybernetics 2, 100–107 (1968)
20. Holzmann, G.J.: A Stack-Slicing Algorithm for Multi-Core Model Checking. In: Haverkort, B., Černá, I. (eds.) PDMC 2007. ENTCS, vol. 198, pp. 3–16. Elsevier, Amsterdam (2007)
21. Holzmann, G.J., Bošnački, D.: The Design of a Multicore Extension of the SPIN Model Checker. IEEE Trans. On Software Engineering 33(10), 659–674 (2007)
22. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification. In: 23rd IEEE/ACM Int. Conference on Automated Software Engineering, pp. 1–6. IEEE Press, New York (2008)
23. Holzmann, G.J., Joshi, R., Groce, A.: Tackling large verification problems with the swarm tool. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 134–143. Springer, Heidelberg (2008)
24. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. IEEE Trans. On Software Engineering (2010) (to appear)
25. Korf, R.E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence 27(1), 97–109 (1985)
26. Laarman, A., van de Pol, J.C., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: Int. Conference on Formal Methods in Computer-Aided Design (2010)
27. Lerda, F., Sisto, R.: Distributed-Memory Model Checking with SPIN. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 22–39. Springer, Heidelberg (1999)
28. Lowerre, B.T.: The HARPY speech recognition system. PhD thesis, Carnegie-Mellon University (1976)
29. Luttik, S.P.: Description and Formal Specification of the Link Layer of P1394. Technical Report SEN-R 9706, CWI (1997)
30. Pelánek, R., Hanžl, T., Černá, I., Brim, L.: Enhancing Random Walk State Space Exploration. In: 10th Int. Workshop on Formal Methods for Industrial Critical Systems. ACM SIGSOFT, pp. 98–105 (2005)
31. Peled, D., Pratt, V., Holzmann, G.J. (eds.): Partial Order Methods in Verification. Series in Discrete Mathematics and Theoretical Computer Science 29 (1996)
32. Romein, J.W., Plaat, A., Bal, H.E., Schaeffer, J.: Transposition Table Driven Work Scheduling in Distributed Search. In: 16th National Conference on Artificial Intelligence, pp. 725–731. AAAI Press, Menlo Park (1999)
33. Russell, S., Norvig, P.: Artificial intelligence: A modern approach. Prentice-Hall, New Jersey (1995)
34. SETI@home, `http://setiathome.berkeley.edu` (Visited on 18 February 2011)
35. Sivaraj, H., Gopalakrishnan, G.: Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking. ENTCS, vol. 89, pp. 51–67 (2003)
36. Staats, M., Păsăreanu, C.: Parallel Symbolic Execution for Structural Test Generation. In: 19th Int. Conference on Software Testing and Analysis, pp. 183–194. ACM, New York (2010)
37. Torabi Dashti, M., Krishnan Nair, S., Jonker, H.L.: Nuovo DRM Paradiso: Towards a Verified Fair DRM Scheme. Fundamenta Informaticae 89(4), 393–417 (2008)
38. Torabi Dashti, M., Wijs, A.J.: Pruning state spaces with extended beam search. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 543–552. Springer, Heidelberg (2007)

39. Valente, J.M.S., Alves, R.A.F.S.: Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. Computers & Industrial Engineering 48(2), 363–375 (2005)
40. West, C.H.: Protocol Validation by Random State Exploration. In: 8th Int. Conference on Protocol Specification, Testing and Verification, pp. 233–242. North-Holland, Amsterdam (1986)
41. Wijs, A.J.: What to Do Next?: Analysing and Optimising System Behaviour in Time. PhD thesis, Vrije Universiteit Amsterdam (2007)