Portfolio Management in Explicit Model Checking

Karsten Wolf

Universität Rostock
Institut für Informatik
karsten.wolf@uni.rostock.de

Abstract. Thanks to a rich Petri net theory, there exists a broad range of verification techniques for Petri nets. Most of them have a performance that highly depends on the particular combination of net and property. That is why several verification tools for Petri nets use portfolio approaches where various verification algorithms are run concurrently.
In this note, we sketch the architecture of a portfolio manager, using the tool LoLA 2.0 as a running example. The portfolio of a verification problem is organized as a task tree. The leafs of the task tree are actual verification algorithms while the inner nodes represent the logical structure of the portfolio. The portfolio manager schedules verification algorithms and assigns resources to them (processor cores, memory, and time). Moreover, it evaluates the consequences of returned results with regard to the original verification problem.

1 Introduction

There exist several approaches for verification, ranging from explicit model checking [4] via BDD based model checking [3] to SAT based model checking [23]. For Petri nets, the variety of methods is larger than elsewhere since we can also use unfoldings [6] and the whole bandwidth of Petri net structure theory.

The verification methods have in common that their performance on a particular model is almost unpredictable. Most methods have an unpleasant worst-case complexity ranging somewhere between NP-completeness and EXPSPACE-completeness [12]. Reduction techniques such as the stubborn set method [19] and the symmetry method [8, 14] try to alleviate that complexity but their success again depends on the shape of the model and the property. For end users with limited expertise in Petri net theory, it is difficult to choose the most promising methods for their particular problem instance.

That is why several contemporary tools such as Tapaal [5], ITS-Tools [17], or LoLA [26] use portfolio approaches. That is, several promising algorithms are launched (sequentially or concurrently) until one of them delivers an answer to the original verification problem.

In this paper, we shall discuss the architecture of a portfolio manager, the component of a Petri net verification tool that takes care of organizing a portfolio of verification algorithms. We use the portfolio manager of LoLA as a reference. However, we expect the central statements of the paper to be universal.

A portfolio manager has two main duties. First, it has to record the results of the launched verification algorithms and to trigger consequences for the remaining algorithms. Second, it has to schedule the resources (available processor cores and memory as well as available run time) to the running algorithms.

We shall first discuss typical ingredients of a portfolio (Section 2). We then encapsulate verification algorithms in *tasks*, the main object to be treated by the portfolio manager (Section 3). Section 4 reveals the lifecycle of a task. Subsequently, we introduce *task trees* for representing the logical dependencies between the portfolio members (Section 5). Finally, we discuss the scheduling of tasks (Sections 6 and 7).

2 Constituents of a Portfolio

In this section, we give examples of verification algorithms that may serve as constituents of a verification portfolio.

Search algorithms Most verification problems can be solved by a traversal of the state space or the product of the state space and an automaton representing the verification problem. For reachability queries, a simple depth-first traversal of the state space is sufficient. For model checking CTL formulas [4, 22] or LTL formulas [21], search needs to be combined with a detection of strongly connected components [16].

In all cases, search is subject to the state explosion problem. For concurrent systems, the main application domain of Petri nets, stubborn set methods are among the most important state space reduction methods. Stubborn set is a whole family of approaches (see [20] for a recent overview). The particular approach to be used depends on the property under verification. However, even for one and the same property, more than one stubborn set method may be available. In [10], several stubborn set methods for reachability and home properties are discussed. One of the reachability preserving methods performs particularly well if the given state predicate is indeed reachable. In this case, it tends to find a very short path to a target state. As soon as a target state is reached, search may be stopped. That is, when searching for reachable states, we typically produce only a tiny portion of the state space (a phenomenon that is referred to as on-the-fly verification). If that stubborn set, however, is applied to a state predicate that is unreachable, the whole reduced state space needs to be explored and is typically much larger than the reduced state space obtained using an alternative stubborn set method proposed in [10]. This asymmetric behavior with respect to the prospective outcome of verification can be observed in virtually all verification problems used in the yearly model checking contests (MCC, [9]). Needless to mention that it is not possible to select the right method in advance unless the answer to the problem is known anyway.

Consequently, a portfolio for reachability may already include two different search algorithms, one speculating on reachability of the predicate, the other speculating on unreachability of the predicate.

Similar pairs of stubborn set methods exist for other classes of properties as well.

Symbolic methods Symbolic methods include BDD based model checkers [3] or similar approaches [18], SAT based methods [23], and unfolding approaches [6]. As we have no particular experience with these methods, we cannot elaborate much on details of these methods.

Petri net structure theory In [24], a method for verifying reachability queries has been presented that is based on the Petri net state equation. It can give negative answers (state equation has no solution) as well as positive answers (state equation has a solution that can be turned into a fireable transition sequence). If the state equation has a solution that cannot be arranged to a fireable transition sequence, it searches for alternative solutions of the state equation. The method is not guaranteed to terminate. However, since solving a linear system of equations and inequations is "only" NP-complete and hence requires only polynomial space, memory consumption of the state equation approach is rather moderate.

Another purely structural approach is the invocation of the siphon/trap-property [7]. It establishes a sufficient criterion for non-reachability of a deadlock. The property can be verified as a SAT problem [13]. Hence, it requires only polynomial space but has a rather unpredictable (NP-complete) runtime. If the siphon/trap property does not hold, deadlocks may or may not be reachable.

Underapproximation In some application domains (including biochemical reaction networks), Petri net models may have initial markings with a large number of tokens on some places. An extreme example is the GPPP benchmark [9] that is used in the MCC and has places with initially more than 2^{32} tokens. Consequently, moving these tokens just to the next place may include 2^{32} transition occurrences and induce more than 2^{32} states, too much for explicit model checking.

If a verification problem asks for the presence of just a few tokens on an initially empty place, it is unlikely that all of the 2^{32} tokens of an initially marked place are needed. Therefore, a portfolio for reachability and selected other problems may include a special search routine where most tokens on places with a large initial marking are "frozen". We obtain an underapproximation of the original state space which means that the property is indeed reachable in the original state space if it is reachable in the underapproximation while it may or may not be reachable in the original state space if it is unreachable in the underapproximation.

The dual approach, overapproximations, do not make sense in explicit verification since they require more space that the actual state space. For symbolic model checking, however, overapproximations are a valid tool for reducing the size of BDDs.

Skeleton net If the given net is a high-level net, we may consider the skeleton (the P/T net just obtained by ignoring colors). There is a net morphism between the high-level net and its skeleton, so some properties including reachability are preserved: if a marking is reachable in a high-level net, the corresponding marking is also reachable in the skeleton (the reverse is not true). Since it is very easy to obtain the skeleton from a high-level net, the approach may yield results for high-level nets that are too large to be unfolded to equivalent P/T nets. The MCC contains some high-level nets of this kind. For verifying the skeleton, we may again employ several algorithms such as search or the state equation.

If the original net is a P/T net, we may fold that into a high-level net for obtaining a skeleton, so the approach is applicable for both high-level and low-level input.

Strength reduction For some property ϕ, there may be a simpler property that implies ϕ or is implied by ϕ. Adding a verification algorithm for the simpler property may thus help for verifying ϕ. For instance, satisfaction of the CTL property $EF\ \psi$ is necessary for satisfaction of $E(\chi\ U\ \psi)$ while $AG\ \psi$ is sufficient for $EG\ \psi$. The pure reachability problems $EF\ \psi$ and $AG\ \psi$ are indeed simpler since they enable the use of additional verification algorithms such as the state equation approach mentioned above.

Random walks For properties where the witness or counterexample is just a single finite path, one can simply launch random walks through the state space. If such a walk hits a witness (counterexample), the property holds (does not holds) while otherwise the method does not terminate. The method is extremely memory-efficient since we do not need to store visited states. At the same time, it can fire transitions at an extremely high rate as we do not need to search nor store markings. Consequently, random walks are a poor verification method when applied standalone, but a very useful member of a portfolio. The likelihood of hitting a witness or counterexample path can be increased by applying suitable stubborn set methods in the selection of the next transition to be fired.

Boolean combinations If a formula to be verified is a Boolean combination, sub-formulas can be verified separately. That is, the verification algorithms for the individual sub-formulas establish separate portfolio members.

Conclusion Portfolios may be large (with more than 10 constituents) and diverse. We have complete and incomplete methods (incomplete in the sense that they do not always terminate, or may terminate without a definite answer to the problem). Methods may answer to the original problem or only to a sub-problem. Methods have a broad range of expected memory and runtime consumption. It is therefore necessary to establish a structured approach to portfolios, the *portfolio manager*.

3 Tasks of a Portfolio

With the concept of a *task*, we design an object that encapsulates a particular verification algorithm and augments it with all necessary information to execute it as a portfolio member. These data include the necessary inputs, the status of the task in its lifecycle (see Section 4), results and statistics (for completed tasks), assigned resources (discussed in Section 7), and data that are relevant for execution and scheduling (also discussed there).

Input Currently, LoLA is called with a single Petri net and a list of verification problems, given as CTL* formulas. This standard is established by the setup of the MCC. However, a verification tool needs to internally deal with more than one net and more than one formula, independent of the original input. Multiple formulas come into play since we may want to separately verify sub-formulas if the verification problem is a disjunction or conjunction. In addition, we may add distinct formulas using the strength reduction explained in Section 2. If we apply the skeleton approach explained in the same section, we have two distinct nets for every given verification problem. In addition, we may want to apply net reduction [1, 15] to the net before running an actual verification algorithm. Since the applicability of reduction rules depends on the verification problem, we may end up with several different nets. We conclude that we need to assign an individual net and an individual formula to every verification task.

Results and Statistics Since many portfolio members are only necessary or only sufficient, or do not terminate in all cases, it is reasonable to introduce a value *unknown* to the usual values *true* and *false*. In addition, we propose a fourth value *void* that is used if the verification algorithm has not yet run. The difference between *unknown* and *void* is that *unknown* may be propagated as final value to the original verification problem if no task in the portfolio delivers a *true* or *false*. In contrast, *void* is not propagated since a definite result may be found subsequently.

Beyond the plain yes/no answer to the verification problem approached by a task, results may include additional diagnostic information such as witness or counterexample paths as well as witness or counterexample states. If search algorithms use storage methods based on Bloom filtering, diagnostic information may include a likelihood of hash conflicts that may be used for judging about the risk that the state space is actually incomplete.

Statistical information includes the number of visited states, fired transitions, run time, consumed memory and other information that is useful for evaluating the performance of the algorithm or the difficulty of the problem instance with respect to the verification algorithm.

4 Lifecycle of a Task

Figure 1 depicts a state machine representing the lifecycle of a task. In the sequel, we shall describe its states and transitions.

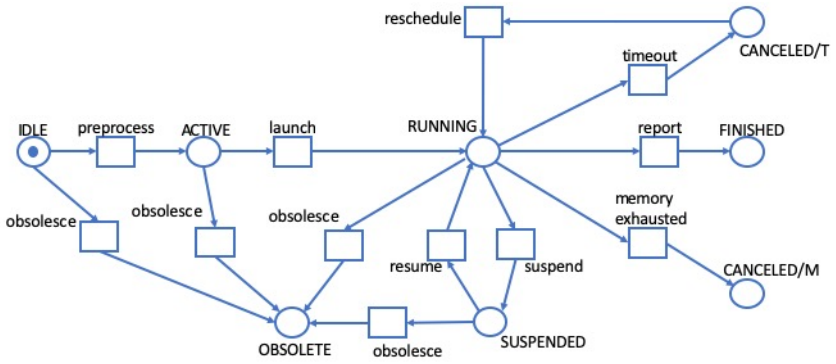

Fig. 1. The lifecycle of a task.

4.1 States of the Lifecycle

IDLE This is the initial state of the lifecycle. The task is created as a result of the planning phase of the verification tool. The planning phase of a verification tool is run subsequent to parsing the net(s) and verification problem(s). For temporal logic formulas, rewriting is applied [2] since that may simplify the verification problem, or may identify it as tautology or contradiction. After rewriting, the formula is categorized. The planning phase then puts together the portfolio, based on the category of the problem and, if given, command line options of the verification tool. In state IDLE, the task is not ready for execution. In particular, the net assigned to the task may require preprocessing such as net reduction or the calculation of auxiliary data. For the latter, an interesting example is a pre-calculated set of conflicting transitions (the set $(\bullet t)\bullet$ for a given transition t). This information is needed in every marking of a search algorithm, especially for stubborn set calculations. Since a typical search visits hundreds of millions of markings, a pre-computed list of conflicting transitions saves a substantial amount of runtime. Its calculation may consume several seconds or even minutes. Although a task is not ready for execution in state IDLE, it is beneficial for the portfolio manager to know idle tasks since this information enables the portfolio manager to plan resources for their future execution.

ACTIVE An active task is ready for execution, This means that all preprocessing of inputs is completed. The task is just lacking the assignment of resources for execution by the scheduler that is part of the portfolio manager.

RUNNING A running task is executing the actual verification algorithm. During this phase, there are two kinds of interaction between the algorithm and the

```
                                FINISHED FORMULA: CATEGORY    VALUE  PRODUCED BY
SharedMemory-PT-000050-ReachabilityCardinality-00:  INITIAL     true  preprocessing
SharedMemory-PT-000050-ReachabilityCardinality-01:       AG    false  state space
SharedMemory-PT-000050-ReachabilityCardinality-02:  INITIAL     true  preprocessing
SharedMemory-PT-000050-ReachabilityCardinality-03:       AG    false  state space
SharedMemory-PT-000050-ReachabilityCardinality-05:  INITIAL     true  preprocessing
SharedMemory-PT-000050-ReachabilityCardinality-08:       AG    false  state equation
SharedMemory-PT-000050-ReachabilityCardinality-09:       AG    false  state equation
SharedMemory-PT-000050-ReachabilityCardinality-10:       EF    false  state equation
SharedMemory-PT-000050-ReachabilityCardinality-11:       EF     true  state equation
SharedMemory-PT-000050-ReachabilityCardinality-12:       EF     true  state equation
SharedMemory-PT-000050-ReachabilityCardinality-13:       AG     true  state equation

                                PENDING FORMULAS: CATEGORY  IDL  ACT  RUN  SUS  FIN  C/T  C/M  OBS
SharedMemory-PT-000050-ReachabilityCardinality-04:       EF    0    1    1    0    2    0    0    0
SharedMemory-PT-000050-ReachabilityCardinality-06:       AG    0    1    1    0    2    0    0    0
SharedMemory-PT-000050-ReachabilityCardinality-07:       EF    0    0    2    0    2    0    0    0
SharedMemory-PT-000050-ReachabilityCardinality-14:       AG    0    0    0    0    1    0    0    0
SharedMemory-PT-000050-ReachabilityCardinality-15:       EF    0    0    0    0    1    0    0    0

 TASK CATEGORY  TYPE   TIME/TLIMIT  MEM PG/PGLIMIT FORMULA                                                  STATUS
   57       EF  STEQ     3/32000000      0/5       SharedMemory-PT-000050-ReachabilityCardinality-06  sara is running.
   60       EF  STEQ     3/32000000      0/5       SharedMemory-PT-000050-ReachabilityCardinality-07  sara is running.
   61       EF  EXCL     3/32000000      1/2048    SharedMemory-PT-000050-ReachabilityCardinality-07 152 m, 30 m/sec, 240 t fired, .
   63       EF  STEQ     3/32000000      0/5       SharedMemory-PT-000050-ReachabilityCardinality-04  sara is running.
```

Fig. 2. Displaying the status of tasks.

portfolio manager. First, the portfolio manager controls the use of resources assigned to the task. We discuss this issue in Section 6. Second, the portfolio manager collects status data of the algorithm (e.g. number of markings visited so far, number of fired transitions, etc.). Such status information is quite useful to give the user the impression that the tool is running properly. The status information needs to be aggregated for all running tasks in order to create a readable display.

Figure 2 shows an example of an aggregated status report in LoLA. The first block reports all verification problems (referred to by an identifier) which have already been solved, their category, their value, and the portfolio member that delivered the result for that particular problem. The second block displays, for every other formula, the number of portfolio members that are available for that problem, and the status of these tasks. The third block displays the running tasks with the type of underlying algorithm (in the example: three times state equation and one depth-first search), their assigned resources, and the algorithm-dependent status information collected by the portfolio manager.

FINISHED A finished task has completed the execution of its verification algorithm and delivered its result.

OBSOLETE A task is obsolete if its result is not needed anymore to determine the answer to an original verification problem. If, for instance, a verification problem is a disjunction of two subproblems, and one of the subproblems is found to be true, all tasks supporting the other subproblem become obsolete. The main factual difference between a finished and an obsolete task is that statistical information and results such as counterexamples or witnesses are present and meaningful only for finished, but not for obsolete tasks.

CANCELED/T, CANCELED/M Tasks are canceled as soon as they exceed their assigned resources. We distinguish cancelation by exceeding the time limit from cancelation by exceeding the memory limit. This way, we may consider rescheduling the task if, later on, additional resources become available.

SUSPENDED A task is suspended if assigned resources are not available. Consider, for example, memory resources. A task, for instance a search algorithm, does not need all its memory resources immediately. So we may optimistically assign more memory resources than actually available at a certain point in time, speculating that another task will meanwhile release its resources. Suspending a task permits a seamless continuation of an already running task as soon as required resources are indeed available. The difference between a suspended task and a canceled task is that a canceled task releases all assigned resources while a suspended task keeps its resources and may resume execution. Resuming a canceled task amounts to re-execution from the beginning.

4.2 Transitions of the Lifecycle

preprocess (IDLE to ACTIVE) This transition is triggered by the completion of the preprocessing routines for the inputs to the task. In the LoLA implementation, we use the pthread condition mechanism to announce the completion of preprocessing routines.

The amount of necessary preprocessing depends on the verification algorithm. While search algorithms require intensive preprocessing (as already explained), the state equation approach just needs a simple net structure for deriving the incidence matrix. When skeleton nets are involved, we may skip net reduction since the resulting nets are already quite small. An individual transition from idle to active permits an early start of the actual verification. We may run tasks that require little preprocessing while other tasks are still in the preprocessing phase. If such early tasks deliver their result quickly, we may be able to skip some of the preprocessing thus saving resources for other verification problems.

launch (ACTIVE to RUNNING) We launch a task by spawning a new thread for the actual verification algorithm of the task. The transition is triggered by the scheduler which is discussed in Section 7.

report (RUNNING to FINISHED) When a verification algorithm terminates, it triggers the report activity. The results of the algorithm (value, witnesses or counterexamples, statistics) are recorded in the portfolio manager. Resources assigned to the task are released and a signal is sent to the portfolio manager that triggers the next scheduling activity (for launching another task).

timeout (RUNNING to CANCELED/T), memory exhausted (RUNNING to CANCELED/M) These activities are triggered by the resource management of the portfolio manager. We discuss this issue separately in Section 6.

suspend (RUNNING to SUSPENDED) This activity is triggered by the resource management of the portfolio manager. Suspension happens if a task is acquiring a granted resource tat is not immediately available. The thread executing the verification algorithm is frozen until the request can be satisfied.

resume (SUSPENDED to RUNNING) When some task releases its resources (through cancelation or termination), the resource manager may trigger the resume transition and grant fresh resources to a suspended task.

reschedule (CANCELED/T to RUNNING) The actual runtime of a verification algorithm is virtually unpredictable. It may this happen that a task needs much less runtime than originally scheduled. This way, additional runtime may become available. If, after having executed all tasks once, more runtime is available than originally scheduled for some canceled task, we may reschedule that task. Giving more runtime to it, we gain some opportunity to finish it successfully this time.

obsolesce (any to OBSOLETE) A task becomes obsolete if some other task for the same verification problem has answered the given problem. As soon as any task executes its report transition, all other tasks for the same verification problem are checked whether they can still contribute to the original problem. If not, they become obsolete. In case they are currently running or suspended, execution is stopped and resources are released. Obsolete tasks are detected by the evaluation of task trees to be introduced in Section 5.

Conclusion Tasks have a complex life cycle. The transitions of the lifecycle are triggered by various components of the verification tool. Consequently, portfolio management requires a clean assignment of responsibilities to the various components. We have been experimenting with the portfolio manager of LoLA for several months before the structure of the lifecycle converged to the one just reported.

5 Task Trees

The leafs of the task tree are the tasks as discussed so far. Whenever tasks complete, the result value is propagated bottom-up in the tree. Whenever an inner node gets a final truth value, the remaining branches The inner nodes reflect the logical dependencies between the portfolio members. We identified several useful types of inner nodes and discuss them one by one.

Conjunction This binary (or n-ary) node reflects the fact that the original problem is a conjunction of subproblems. The children of a conjunction node are the roots of the portfolios of the subproblems. The following table reflects the propagation of values of a conjunction node.

	true	false	unknown	void
true	true	false	unknown	void
false	false	false	false	false
unknown	unknown	false	unknown	void
void	void	false	void	void

Disjunction This binary (or n-ary) node reflects the fact that the original problem is a disjunction of subproblems. The following table reflects the propagation of values.

	true	false	unknown	void
true	true	true	true	true
false	true	false	unknown	void
unknown	true	unknown	unknown	void
void	true	void	void	void

Aggregation This binary (or n-ary) node represents a portfolio where two (or more) algorithms are available for exactly the same (sub-)problem. It propagation behavior can be reflected in the following table.

	true	false	unknown	void
true	true	(error)	true	true
false	(error)	false	false	false
unknown	true	false	unknown	void
void	true	false	void	void

There are two entries marked with *error*. If any of these situations would ever occur, one of the involved verification algorithms is wrong, or has an incorrect implementation.

Dual This is a unary node in the task tree. It maps a verification problem to the corresponding dual problem. Using the dual node, we may have algorithms for reachability in a portfolio of an invariance problem, to name just one example. Whenever a dual node occurs, the verification problem of the child node refers to the negation of the verification problem for the current node. In the mentioned example, if we want to verify $AG\ \phi$, the child node is attached to formula $EF\ \neg\phi$. The behavior of this node is defined by the following table.

true	false	unknown	void
false	true	unknown	void

Sufficient This unary node is used if the result of the child node only establishes a sufficient condition for the original verification problem. If, for instance, the original problem is $EG\ \phi$, a portfolio for $AG\ \phi$ can be wrapped with this node to reflect that, if the answer to $AG\ \phi$ is false, this does not means that $EG\ \phi$ is false, while a true answer to $AG\ \phi$ means that $EG\ \phi$ is indeed true. The following table can be used.

true	false	unknown	void
true	unknown	unknown	void

Necessary This unary node is the dual counterpart of a *Sufficient* node for necessary conditions (such as $EF\ \psi$ for $E(\phi\ U\ \psi)$). Its table looks as follows.

true	false	unknown	void
unknown	false	unknown	void

Conclusion With the help of the tables defining each inner node, every distribution of values for the actual tasks defines unique values for the inner nodes of a task tree. The value of the root node is the actual outcome of verification for the original verification problem. With the help of task trees, we obtain an easy criterion for obsolete tasks: Whenever a node has a value different from void, all void children are obsolete.

Figure 3 exhibits an example of a portfolio, organized as task tree. For every node, the corresponding net and formula are attached. N stands for the original net, $Skel(N)$ for the corresponding net obtained by the skeleton approach. In the figure, we abstract from potential differences between nets caused by net reduction.

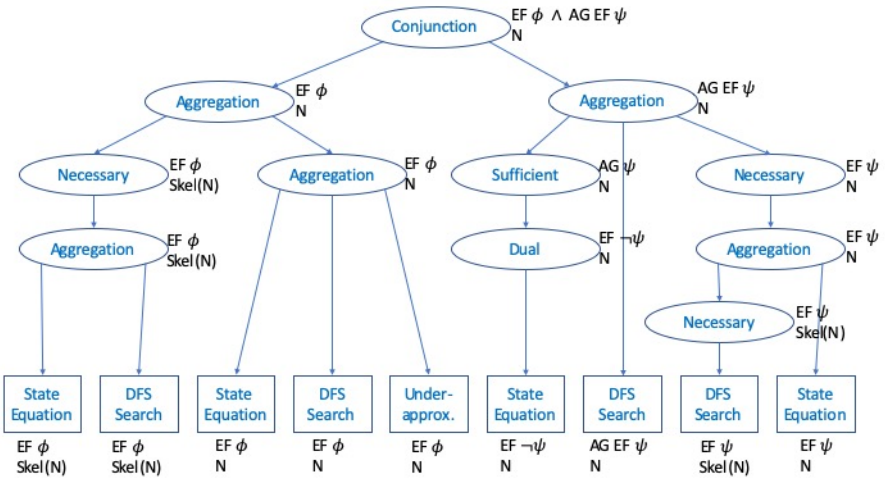

Fig. 3. An example of a task tree.

The original problem is the CTL formula $EF\ \phi \wedge AG\ EF\ \psi$, for any state predicates ϕ and ψ. This original problem forms the root of the task tree. The *conjunction* node signals that the two sub-formulas are verified separately. The first sub-formula, the reachability problem $EF\phi$, can be approached using the original net or the corresponding skeleton net (obtained by folding if N is a P/T net). These alternatives are modeled as an *aggregate* node. Reachability in the skeleton is only a necessary condition for reachability in the original net, so the results of the left branch are filtered by a *necessary* node. This way, despite possible true and false answers of the underlying algorithms, only a false result is propagated towards the root. The *aggregate* node below the *necessary* node reflects the fact that reachability in the skeleton can be investigated by two algorithms: evaluation of the state equation or depth-first search. For the original net, the task tree offers three alternative approaches: state equation,

depth-first search, and an underapproximation that tries to freeze tokens in the initial marking. Reachability in the underapproximation is a sufficient condition for actual reachability. However, we do not need a *sufficient* node here since the algorithm itself would only answer yes or unknown.

For the sub-formula $AGEF\phi$, the task tree offers three alternative approaches. The one displayed in the middle is conventional depth-first search that, through investigation of strongly connected components, is able to answer the task. This search is flanked by a sufficient and a necessary condition that both have been obtained by strength reduction. Indeed, a net satisfying $AG\ \psi$ also satisfies $AG\ EF\ \psi$, and a net satisfying $AG\ EF\ \psi$ must also satisfy $EF\ \psi$. The verification of $AG\ \psi$ is traced back to the verification $EF\ \neg\psi$ using the tautology $AG\ \psi \iff \neg EF\ \neg\psi$. This way, actual verification algorithms only need to care about reachability and not about invariance. The *dual* node in the task tree is responsible for negating the result of the underlying algorithm. For checking the necessary criterion $EF\ \psi$, we employ the state equation and depth-first search in the skeleton.

The example shows the potential complexity of a portfolio and demonstrates the necessity of a systematic approach.

6 Controlling Resources

Cores, memory, and runtime are the critical resources when running a portfolio. If the number of algorithms executed in parallel exceeds the number of available cores, the operating system is forced into frequent context switches, with negative impact on overall runtime and on caches. Since depth-first search is extremely space-consuming, we have to take care that algorithms running in parallel have a well-designed way of competing for memory. Last but not least, we need to make sure that all portfolio members get a fair portion of available runtime, if that is limited. The actual scheduling of resources is discussed in the next section. We can schedule resources only if we can control the access of tasks to these resources. That is why we included this section.

For controlling access to cores, we believe that it is sufficient to take care that we launch at most as many parallel threads (i.e. concurrent verification algorithms) as there are cores in the underlying machine. In case of LoLA, the user specifies that number using a command-line option. Runtime is controlled by an additional timer thread that is running as long as the portfolio manager is active. It is executing an infinite loop where, after sleeping for five seconds, it checks for tasks that have exceeded their time limit and, at the same time, collects status information from running tasks and displays them. According to our experience, the granularity of five seconds seems to be sufficient for controlling time limits of tasks. By sleeping a long time between activities, the timer thread does not severely interfere with the threads that execute the actual verification algorithms.

The most interesting resource with respect to resource control is memory. As a preliminary step, we discuss access to memory in general. Based on this

discussion, we can then propose a strategy for controlling the amount of memory assigned to a task.

Depth first search, the main method for explicit verification, is an extremely memory-consuming task. Profiling with LoLA revealed that, during depth-first search, about 90% of the runtime is spent for checking whether or not the currently visited marking has been visited before, and to insert it into the data structure if it has not visited yet. To our own surprise, about 40% of that time is spent in *malloc*, the C routine for allocating memory on the heap of the main memory (including calls to the corresponding *new* operator in C++). The explanation of that large percentage has two aspects. First, it shows that the remaining parts of a depth-first search algorithm indeed have an extremely lightweight implementation. Second, it shows that allocating memory on the heap is a rather involved task. First, *malloc* needs to find a suitable portion of memory in the list of free memory segments, and to launch a call to the operating system if there is none. Second, it needs to take precautions for the eventual return of the memory to be allocated. And finally, malloc must be thread-safe, i.e. it must provide a mechanism for mutual exclusion between concurrent calls to the routine.

Based on this analysis, we conclude that depth-first search should include a tailored memory management. In LoLA, we allocate memory in large pages of, say, 100 Megabytes. Then, our own memory manager allocates portions of such a page to individual requests of the search algorithm. We obtain the following runtime benefits:

- Since every concurrent depth first search uses its own memory management, we do not need to provide any mechanism for mutual exclusion during memory management;
- Since memory allocated for depth-first search is never returned before that search as such is finished, we do not need to provide any mechanism for returning memory to the list of free portions;
- Since memory is never returned, we have no fragmentation issues and memory management is mostly trivial;
- When depth-first search terminates, or is canceled, we only need to return the allocated pages instead of freeing hundreds of millions of individual data objects.

The first three items cause a speedup of 20% for our own memory allocation requests, compared to traditional calls to malloc. The last item reduces the time for terminating a depth-first search from several minutes to a fraction of a second. Hence, the page mechanism is a prerequisite for running nontrivial portfolios.

Once we have a page based memory management for depth-first search, we can gain complete control of the memory usage. We simply need to count (and limit) the number of pages that the memory manager of a depth-first search is allowed to allocate. A task is suspended by simply not answering to its request for memory. A task can be canceled by not answering to its memory request while releasing its resources (otherwise, canceling a pthread is a nontrivial endeavor).

Other verification algorithms involve depth-first search as well, like our underapproximation approach, or require only marginal memory resources, like

random walks. For two classes of algorithms, we do not provide any control of their memory usage in LoLA: state equation and siphon/trap property. Here, we use off-the-shelf libraries for solving linear problems resp. SAT problems and so far we did not dare to intervene in their memory management. However, both problems are in NP and thus in PSPACE, so we hope that their memory requirements are less severe than that of depth-first search. With this sloppy way of controlling memory resources, we at least managed to stay within the harsh memory limit of 16 Gigabytes in recent issues of the MCC.

7 Scheduling Tasks

Scheduling refers to two kinds of decisions to be repeatedly taken in a running portfolio manager. First, we need to select the order in which we execute tasks. Second, we need to decide how many resources we assign to the selected tasks.

7.1 Selection Strategy

As long as we do not have a strict limit for runtime, the order of executing tasks is less important. That is why we discuss this issue under the assumption that there is in fact a time limit. Using such a strategy in a scenario without time limit, helps us to reduce the overall run time in many cases but has no severe impact if it fails.

So, assuming a strict time limit, our duty is to get as many as possible firm results (true or false) for the given verification problems. In [25], we observed that problems can be divided into simple, challenging, and impossible. Simple means that virtually all methods are able to solve the problem in little time. Impossible means that all known methods fail. Only for the challenging problems, selection and configuration of verification algorithms matter. Most problems in the MCC are either simple or impossible. This is, of course, an a posteriori classification. However, the size and shape of the net and the structure of a temporal logic formula may give us valuable a priori hints for guessing the category a problem.

Concerning the net, large nets tend to be more challenging than small nets. Nets with large conflict clusters tend to be more challenging than nets with small conflict clusters (more concurrency means better applicability of stubborn sets).

Regarding a temporal logic formula, the number of occurring temporal operators seems to be a good basis for judging its difficulty. The simplest case are without doubt formulas without temporal operators. They correspond to constants true or false, or to a state predicate that can be evaluated by just inspecting the initial marking. Such formulas frequently appear in the MCC as the result of linear programming approaches to the atomic propositions and application of temporal logic tautologies [2]. Of course, a portfolio manager will launch tasks for such formulas with highest priority since they do not require any measurable resources.

Of the remaining formulas, the ones with one or two temporal operators are the simplest. Most of these formulas are supported by specialized algorithms

[11]. Their presence increases the success rate. In addition, also pointed out in [11], they occur more frequently than other formulas. For differentiating between formulas with one or two temporal operators, we employ the past issues of the MCC. Based on these data, we can calculate the success rate of LoLA for the various types of formulas. This success rate is an excellent measure for difficulty.

Formulas with more than two temporal operators occur less frequently in practice. That is why the number of occurring operators appears to be a sufficient criterion for judging their complexity. Alternatively, LTL formulas could be judged according to the number of states of the Büchi automata that represent the formulas and which are used in LTL model checking [21].

For formulas with the same temporal structure, the number of places mentioned in the formula is another criterion for differentiating the difficulty. A large number of mentioned places leads to a large number of visible transitions.

The overall difficulty of a problem would be some Pareto style aggregation of the difficulty of the formula and the difficulty of the net. In LoLA, we currently consider only the difficulty of the formula.

For search algorithms, we schedule the easiest problems first. If any of the problems is solvable then the easiest ones have the greatest likelihood to do so. For algorithms that require only few resources, such as random walks, we schedule the most difficult problems first. If a problem is indeed difficult, it appears to be easier to land a lucky punch with an unusual algorithm than with a state space exploration. In any case, we schedule tasks in a way that, if possible, tasks running in parallel concern different problems. In the case where some problems indeed turn out to be simple, we solve them without running too many algorithms in parallel on them, so more time remains for the challenging problems.

7.2 Assigning Resources

Cores For sequential algorithms, a single core is assigned. This is the case for the whole LoLA portfolio manager. For parallel algorithms, it could be reasonable to assign more than one core. Here we have a conflict between assigning cores to that algorithm versus assigning cores to other portfolio members. To date, most parallel algorithms have a saturating behavior with respect to the number of cores. That is, there is a number of cores where additional cores do not lead to an improvement of runtime. If that number is known (by experimental evidence), it would be reasonable to assign not more than that number of cores to the parallel algorithm, and cores might be left for other portfolio members.

Memory The memory-critical tasks are those that execute depth-first search. We pointed out above that, depending on the stubborn sets used, they are optimized for true (e.g. reachable) or false (e.g. unreachable) cases. One of the cases (reachable) benefits from the on-the-fly effect while in the other case the whole reduced state space needs to be computed. For memory assignment, we may derive two conclusions. First, if the task is set up for the case where we benefit

from the on-the-fly effect, the task either delivers its result after having consumed only few memory pages, or it is unlikely to deliver a result at all. Second, if the task is setup for the case where the whole reduced state space needs to be computed, we either assign sufficient resources for finishing the search, or any assigned resource is wasted. In consequence, taking reachability as an example, we assign only few pages for tasks where search is optimized for reachability and we assign a lot of memory to tasks that are designed for unreachability. In the portfolio manager, we distinguish *search* (optimized for reachability) from *exclusive memory* (optimized for unreachability) tasks. The scheduler takes care that, at all times, only one exclusive memory task is running. All the other tasks get a fixed small number of memory pages assigned, and the single exclusive memory task basically is permitted to use all remaining memory.

Runtime For runtime, we identified two major scenarios: limited versus unlimited availability. For the use of verification in practice, we expect that an unlimited amount of runtime is available. For search tasks, the monotonic request for memory establishes some bound for runtime anyway. For constant-memory tasks such as random walks, a scenario with unlimited runtime should implement some criterion for interrupting the method (number of tries or an algorithm-specific local time limit). Otherwise, we can schedule tasks as soon as cores and memory are available.

Use cases with a strict time limit include the MCC conditions as well as practical situations where we have a firm deadline for reporting results. Taking into consideration that we can schedule at most one *exclusive memory* task at a time, we propose the following strategy. First we separately schedule all remaining *exclusive memory* tasks. We compute the available time as the difference of deadline and current time. If an exclusive memory task is currently running, we replace the current time with the start time of the running task. This amount of time is divided by the number of remaining (and running) exclusive memory tasks. Every task is given that amount of time. The end time of the running task is replaced accordingly.

For all other tasks, we can use all available cores if no exclusive memory task is present, and we can use all but one core if there are remaining exclusive memory tasks. Depending on that number of tasks, we compute the largest number n of seconds such that all nonexclusive tasks can be scheduled for n seconds before the deadline expires. This is a bit tricky since we need to take care about already running tasks, and we need to reflect the fact that the runtime for a single algorithm cannot be parallelized. That is why the calculation requires more efforts than just dividing the available time by the number of remaining tasks.

We compute the remaining time per core which is the difference between the deadline and the start time of an already running task (there should be as many such tasks as we have available cores). If less tasks are running, we use the current time instead. Let k be the number of available cores, and n be the number of remaining nonexclusive tasks. Then we schedule $n \; div \; k$ tasks for every core. The remaining $n \; mod \; k$ tasks are schedule for the cores with largest

remaining time. Then we can compute, for every core, the remaining time on that core and distribute it equally for the tasks planned for this core. That is, tasks do not necessarily get exactly the same time, but processing resources are used exhaustively.

We repeat the scheduling procedure every time a new task is checked in, or a running task finishes. This way, we continuously adapt our scheduling to the current situation.

Example. Suppose that we have four cores, and initially two exclusive memory tasks and ten other tasks. Let 1000 seconds be available. Then we would launch one exclusive memory task with a time limit of 500 seconds. For the remaining cores, we would assume a distribution of four versus three versus three tasks, so we would launch one task with 250 seconds, and two tasks with 333 seconds time limit. If, after 100 seconds, a non-exclusive task returns a result, we have nine remaining tasks (including the running tasks). Two cores have an available run-time of 1000 seconds (since the two running tasks have been launched initially) and the third core has 900 seconds remaining time. We have a distribution three versus three versus three, so we would launch a new task with a time limit of 300 seconds. If, after 200 seconds (counted from beginning), the exclusive memory task returns, we launch the remaining exclusive memory task with a time limit of 800 seconds. If it returns after 250 seconds (again counted from beginning) an additional core becomes available for the non-exclusive tasks. The individual available time is 1000 versus 1000 versus 900 versus 750 seconds and there are nine remaining tasks. So we would distribute them by the pattern three versus two versus two versus two. In effect, one of the running tasks would keep its time limit of 333 (= 1000 / 3) seconds. The second task that was started in the very beginning would get a new time limit of 500 (= 1000 / 2) seconds. The task started after 100 seconds would get a new time limit of 450 (= 900 / 2) seconds, and we would launch a fresh task with a time limit of 375 (= 750 / 2) seconds.

Using this scheduling policy, we benefit from low hanging fruits earned early in the scheduling sequence. If tasks finish early, remaining tasks immediately get more run-time assigned.

Sometimes it may happen that remaining tasks do not exhaust the scheduled time. In this case we check whether we have tasks that were canceled before due to lack of time. If they consumed less time than still available in the end, we reschedule such a task with a more generous time limit.

8 Conclusion

We have discussed the main design decisions to be made for the implementation of a portfolio manager for Petri net based verification. Due to the size and diversity of verification portfolios, a systematic approach is necessary. We also discussed possible deviations from the decision that we made in the LoLA tool. Since the first implementation of the task manager, we added several new verification algorithms to the portfolio and found that the design of the portfolio

manager was robust with respect to the changes. This way, the integration of the new methods validated our major design decisions.

Our next step for improving the manager will be a prognostic feature for memory consumption. We observed that search algorithms request new memory at a roughly constant rate. We can use the past rate of memory requests for estimating the future memory requirement of an exclusive memory task. This way, we may be able to assign additional memory to nonexclusive tasks.

References

1. G. Berthelot. Transformations and decompositions of nets. In *Advances in Petri Nets, LNCS 254*, pages 359–376, 1986.
2. F. Bønneland, J. Dyhr, P. G. Jensen, M. Johannsen, and J. Srba. Simplification of CTL formulae for efficient model checking of petri nets. In *Processings Application and Theory of Petri Nets and Concurrency, LNCS 10877*, pages 143–163, 2018.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
4. E.M. Clarke, E.A. Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
5. A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TAPAAL 2.0: Integrated development environment for timed-arc petri nets. In *Proceedings Tools and Algorithms for the Construction and Analysis of Systems, LNCS 7214*, pages 492–497, 2012.
6. J. Esparza and K. Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008.
7. M.H.T. Hack. Analysis of Production Schemata by Petri Nets. Master's thesis, MIT, Dept. Electrical Engineering,, Cambridge, Mass, 1972.
8. K. Jensen. Condensed state spaces for symmetrical coloured petri nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.
9. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, E. Amparore, M. Beccuti, B. Berthomieu, G. Ciardo, S. Dal Zilio, T. Liebke, S. Li, J. Meijer, A. Miner, J. Srba, Y. Thierry-Mieg, J. van de Pol, T. van Dirk, and K. Wolf. Complete Results for the 2019 Edition of the Model Checking Contest. http://mcc.lip6.fr/2019/results.php, April 2019.
10. L.M. Kristensen, K. Schmidt, and A. Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design*, 29(3):215–251, 2006.
11. T. Liebke and K. Wolf. Taking some burden off an explicit CTL model checker. In *Proceedings Application and Theory of Petri Nets and Concurrency, LNCS 11522*, pages 321–341, 2019.
12. R. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.
13. O. Oanea, H. Wimmel, and K. Wolf. New algorithms for deciding the siphon-trap property. In *Proceedings Applications and Theory of Petri Nets, LNCS 6128*, pages 267–286, 2010.
14. K. Schmidt. How to calculate symmetries of petri nets. *Acta Inf.*, 36(7):545–590, 2000.

15. S.M. Shatz, S. Tu, T. Murata, and S. Duri. An application of petri net reduction for ada tasking deadlock analysis. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1307–1322, 1996.
16. R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
17. Y. Thierry-Mieg. Symbolic model-checking using its-tools. In *Proceedings Tools and Algorithms for the Construction and Analysis of Systems, LNCS 9035*, pages 231–237, 2015.
18. A.A. Tovchigrechko. *Efficient symbolic analysis of bounded Petri nets using interval decision diagrams*. PhD thesis, Brandenburg University of Technology, Cottbus - Senftenberg, Germany, 2008.
19. A. Valmari. Stubborn sets for reduced state space generation. In *Proc. International Conference on Applications and Theory of Petri Nets, LNCS 483*, pages 491–515, 1989.
20. A. Valmari and H. Hansen. Stubborn set intuition explained. *Trans. Petri Nets and Other Models of Concurrency, LNCS 10470*, 12:140–165, 2017.
21. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS, IEEE*, pages 332–344, 1986.
22. B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In *Proceedings International Conference on Concurrency Theory, LNCS 715*, pages 447–461, 1993.
23. P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. Computer Aided Verification, LNCS 1855*, pages 124–138, 2000.
24. H. Wimmel and K. Wolf. Applying CEGAR to the petri net state equation. *Logical Methods in Computer Science*, 8(3), 2012.
25. K. Wolf. Running lola 2.0 in a model checking competition. *Trans. Petri Nets and Other Models of Concurrency, LNCS 9930*, 11:274–285, 2016.
26. K. Wolf. Petri net model checking with lola 2. In *Proceedings Application and Theory of Petri Nets and Concurrency, LNCS 10877*, pages 351–362, 2018.