All from One, One for All:
on Model Checking Using Representatives

Doron Peled
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Abstract

Checking that a given finite state program satisfies a linear temporal logic property is suffering in many cases from a severe space and time explosion. One way to cope with this is to reduce the state graph used for model checking. We define an equivalence relation between infinite sequences, based on infinite traces such that for each equivalence class, either *all* or *none* of the sequences satisfy the checked formula. We present an algorithm for constructing a state graph that contains at least one representative sequence for each equivalence class. This allows applying existing model checking algorithms to the reduced state graph rather than on the larger full state graph of the program. It also allows model checking under fairness assumptions, and exploits these assumptions to obtain smaller state graphs. A formula rewriting technique is presented to allow coarser equivalence relation among sequences, such that less representatives are needed.

1 Introduction

When a program allows concurrent or independent activities, their executions are interleaved in many possible orders. It is often the case that a formula φ is insensitive to reordering some of the concurrent activities of the program, i.e., that any two sequences that are obtained from each other by such reordering, either both satisfy φ or both satisfy $\neg\varphi$. This phenomenon allows reducing the state graph used for model checking by constructing a smaller state graph that represents only a subset of the interleaving sequences.

In this paper, we study model checking techniques based on constructing a reduced state graph that generates representative interleaving sequences. We base our state graph reduction on an equivalence relation on classes of *infinite* sequences (called infinite traces [7]), equivalent up to commuting independent operations (i.e., pairs of operations whose effect on the global states is commutative). This equivalence relation is refined in order to guarantee that the checked formula is equivalence robust [2] w.r.t. the equivalence relation. That is, in each equivalence class, either all the sequences satisfy the formula, or none of them satisfy it.

A family of algorithms is suggested which generate a state graph G for the modeled program such that at least one representative sequence for each equivalence class is present in G. Thus, model checking algorithms for linear temporal logic (LTL) [9] such as [8], that are otherwise applied to the (full) state graph of the program, can be applied to the smaller state graph G. The gain in time and space lies then in the fact that these algorithms are applied to a state graph which can be in some cases considerably smaller than the full state graph of the program.

In order to make the checked formula equivalence robust, the equivalence relation between sequences that is induced by the program may need to be refined. This is done by introducing new dependencies between program operations whose execution in different orders can alter the truth value of the checked property. Since the size of the reduced state graph grows with the number of such dependencies (the equivalence relation becomes finer and more representatives are required to appear in the state graph), we are motivated to add as little such dependencies as possible. We show how rewriting the checked formula using LTL equivalences can help in reducing the number of dependencies that are added in order to achieve equivalence robustness.

We also generalize the framework of model checking using representative sequences by allowing fairness conditions to assist in the generation of the reduce state graphs. When fairness is assumed, there are less interleaving sequences that correspond to executions of the program. Thus, in the presence of a fairness assumption, it is possible that *less* equivalence classes and thus less representatives are required. We show how fairness assumptions can be exploited when expanding the state graph to reduce it further.

2 Preliminaries

A *finite-state program* P is a triple $\langle T,\ Q,\ \iota\rangle$ where T is a finite set of operations, Q is a *finite* set of *states*, and $\iota \in Q$ is the *initial state*. The enabling condition $en_\alpha \subseteq Q$ of an operation α is the set of states from which α can be executed. This is denoted also as a boolean predicate on states. Each operation α can be seen as a partial transformation $\alpha : Q \mapsto Q$ which needs to be defined at least for each $q \in en_\alpha$.

Definition 2.1 *A* dependency relation *is a binary reflexive and symmetric relation* $D \subseteq T \times T$ *such that for each pair of operations* $(\alpha,\ \beta) \notin D$ *(called* independent operations*) it must hold that for each* $q \in Q$, *if* $q \in en_\alpha$, β *is enabled in* q *iff it is enabled in* $\alpha(q)$. *Moreover, if* $q \in en_\alpha \cap en_\beta$ *then* $\alpha(\beta(q)) = \beta(\alpha(q))$.

Programs written in programming languages, e.g., with shared variables or communication, can be translated into sets of operations [9]. A dependency relation can also be retrieved syntactically from the program (see for example [5, 13]). It will be evident in the sequel that the gain obtained from the model checking method suggested here can increase when a smaller dependency relation (i.e., one that contains less pairs) D is used, and vice versa. Notice that *any* reflexive and symmetric relation D' such that $D \subseteq D' \subseteq T \times T$ also satisfies the above conditions and thus can be considered as a dependency relation for P.

Definition 2.2 *An* interleaving sequence *of a program is a finite or infinite sequence of operations* $v = \alpha_1\alpha_2\ldots$ *that* generates *the sequence of states* $\xi = q_0\, q_1\, q_2 \ldots$ *from* Q, *of length* $|v+1|$ *(or* ω *when* v *is infinite), such that (1)* $q_0 = \iota$, *(2) for each* $0 \leq i < |v|$, $q_i \in en_{\alpha_{i+1}}$ *holds, and* $q_{i+1} = f_{\alpha_{i+1}}(q_i)$, *and (3) either* ξ *is infinite or its last state* q_n *satisfies* $q_n \notin \bigcup_{\tau \in T} en_\tau$.

The interleaving semantics of a program often involves a restricting condition on interleaving sequences called *fairness*. Only sequences satisfying the assumed fairness conditions are considered to be executions of the program. The temporal logic properties are interpreted accordingly only over fair sequences. Then, model checking algorithms must reflect this choice by checking that the *fair* sequences satisfy the temporal property.

An *admissible* sequence is an interleaving sequence or any (finite) prefix of such a sequence. We represent an admissible sequence either as a set of states from Q, or by the sequence of executed operations $\alpha_1 \alpha_2 \alpha_3 \ldots$. For a finite admissible sequence generated by the sequence of operations v, we denote its last state by fin_v. This is the state reached when executing the sequence of operations v from the initial state ι.

A *state graph* for a program P is a graph $G = \langle \hat{s}, V, E \rangle$, where V is the set of nodes, $\hat{s} \in V$ is the *starting node*, and the edges E are labeled with operations from T. For each node $s \in V$, $val(s)$ is a state of Q, and in particular, $val(\hat{s}) = \iota$. If $s \xrightarrow{\alpha} t \in E$, then $val(s) \in en_\alpha$ (we also say that α is *enabled from* s), and $val(t) = \alpha(val(s))$. It is said that α is one of the *directions* taken from s. A state graph *generates* a sequence of operations $\alpha_1\alpha_2 \ldots$ (or their corresponding sequence of states), if there exists a (finite or infinite) path starting with $\hat{s}$ whose edges are labeled with $\alpha_1\alpha_2 \ldots$ in this order. The *full state graph* of a program P satisfies that for each node s, for each operation α such that α is enabled in s, there is an outgoing edge from s, labeled with α.

The dependency relation D is used to define an equivalence relation on interleaving sequences. First, an equivalence between finite strings of operations is defined [10]: two strings $v, w \in T^*$ are equivalent, denoted $v \equiv_D w$, iff there exists a sequence of strings $u_0, u_1, \ldots, u_n$, where $u_0 = v$, $u_n = w$, and for each $0 \leq i < n$, $u_i = \bar{u}\alpha\beta\hat{u}$ and $u_{i+1} = \bar{u}\beta\alpha\hat{u}$ for some $\bar{u}, \hat{u} \in T^*$, $\alpha, \beta \in T$, $(a, \beta) \notin D$. That is, w is equivalent to v if it can be obtained from it by repeatedly commuting adjacent independent operations. It can be easily seen that '$\equiv_D$' is indeed an equivalence relation among finite strings. Notice that if v is a finite admissible sequence of a program P, and $v \equiv_D w$, then w is also an admissible sequence of P.

The definition of equivalence between finite strings is now extended to interleaving sequences [7]. Denote by $Pref(w)$ the set of finite prefixes of the (finite or infinite) string w. A relation '$\preceq$' is defined between pairs of admissible sequences as follows: $v \preceq v'$ iff $\forall u \in Pref(v)\, \exists w \in Pref(v')\, \exists z \in T^*\, (w \equiv_D z \wedge u \in Pref(z))$. That is, each finite prefix of v is a prefix of a permutation (under commuting adjacent independent operations) of some prefix of v'.

Definition 2.3 *Define $v \approx v'$ iff $v \preceq v'$ and $v' \preceq v$.*

It is easy to see that '$\approx$' is an equivalence relation [7]. A *trace* is an equivalence class of admissible sequences. Notice that this allows traces of finite as well as infinite sequences. Denote a trace σ also by $[v]$, where v is any member of σ. The *length* of a trace is the length of any of its sequences (they are all of equal length). A finite trace is thus a set of finite admissible sequences equivalent to each other up to commuting adjacent independent operations. It can be easily shown that if $v \equiv_D v'$, then $fin_v = fin_{v'}$. Therefore, for a finite trace σ, we can write fin_σ for the last state reached by any sequence in σ.

Concatenation of two traces $\sigma = [\alpha_0\,\alpha_1 \ldots \alpha_n]$ and $\sigma' = [\beta_0\,\beta_1 \ldots \beta_m \ldots]$, where σ is finite and σ' is either finite or infinite, is defined as $\sigma\sigma' = [\alpha_0\,\alpha_1 \ldots \alpha_n\beta_0\,\beta_1 \ldots \beta_m \ldots]$, provided that $\alpha_0\alpha_1 \ldots \alpha_n\beta_0\beta_1 \ldots \beta_m \ldots$ is admissible. If $\rho = \sigma\,\sigma'$, denote $\sigma \sqsubseteq \rho$.

A *run* π of a program P is a trace that contains interleaving sequences of P. Thus, it is finite iff each one of its sequences cannot be extended by another operation (according to the third requirement in Definition 2.2). The set of runs of P under a dependency relation D is denoted by R_P^D. Notice that for any run π of P and finite trace $\sigma \sqsubseteq \pi$, if $\sigma[\alpha\beta] \in \pi$ and $(\alpha, \beta) \notin D$, then $\sigma[\alpha], \sigma[\beta] \sqsubseteq \pi$. Consider now two dependency relations D and D' such that $D \subseteq D'$. Then '$\equiv_{D'}$' is a refinement of the relation '$\equiv_D$'. That is, if $v \equiv_{D'} w$, then $v \equiv_D w$. Denote [12]:

$\pi \overset{\forall}{\models} \varphi$ For each interleaving sequence ξ of π, $\xi \models \varphi$. I.e., π satisfies φ *universally.*

$\pi \overset{\exists}{\models} \varphi$ There exists an interleaving sequence ξ of π, such that $\xi \models \varphi$. I.e., π satisfies φ *existentially.*

A property φ such that for each $\pi \in R_P^D$, $\pi \overset{\exists}{\models} \varphi$ iff $\pi \overset{\forall}{\models} \varphi$ is said to be *equivalence robust* with respect to R_P^D. We will omit R_P^D when clear from text. Equivalence robustness was first defined for fairness properties in [2].

It is possible to weaken the conditions for dependency given in Definition 2.1, achieving further reduced state graphs when using conditional dependency relation (see [6]).

3 Spawning Reduced State Graphs

3.1 Existing Partial Order Model-Checking Methods

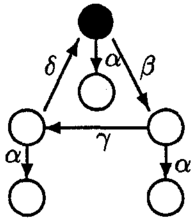
The algorithm suggested in this paper is related to previous algorithms of Valmari [15, 16] and of Godefroid and Wolper [3, 4]. It is also connected to a verification method by Katz and Peled [5].

Godefroid [3] suggested an algorithm called the *sleep set method* for spanning a reduced state graph for finite-state programs. Such a reduced state graph G contains for each *finite* trace a representative linearization. More formally, for each finite trace σ of P, G generates an admissible sequence of operations $\gamma_1\,\gamma_2 \ldots \gamma_n$ such that $\sigma \sqsubseteq [\gamma_1\,\gamma_2 \ldots \gamma_n]$. This algorithm can be used for checking deadlocks and other safety properties. The intuition behind the algorithm is as follows: to each node s expanded by the algorithm, a set $sleep(s)$ of directions (operations), called its *sleep set*, is associated. These (enabled) operations do not need to be explored from s.

If from some node s, some direction α is already explored, then when any edge $s \xrightarrow{\beta} t$, with β independent of α is explored, there is no need to expand the direction α from t and α is added to $sleep(t)$. This follows form the fact that when the expansion of α is finished, enough representatives for successor operations of α (including β) are explored, and exploring α after β is the same as expanding β after α, i.e., $\sigma[\alpha\beta] = \sigma[\beta\alpha]$. Moreover, if expanding α is not needed from some node s (i.e., $\alpha \in sleep(s)$) and an edge $s \xrightarrow{\gamma} r$ with γ independent of α is explored, then α is added to $sleep(r)$, since the occurrences of α immediately following γ can be commuted to represent an already unnecessary sequence.

When a node is reached again during expansion, a new sleep set is calculated for it, and is compared with the one it had before. If the old sleep set contained some operations that are not included in the new sleep set, the node is expanded again with a sleep set which is the intersection of the new and the old sleep set. This guarantees that if the node is reached from two or more directions, it will provide enough successors for all of them[1]. This algorithm does not guarantee that the state graph it spawns generates at least one representative interleaving sequence for each run.

As an example, consider a program P_1 with two sets of operations $O_1 = \{\alpha\}$ and $O_2 = \{\beta, \gamma, \delta\}$, initialized with a state in which both α and β are enabled. The operations O_2 are interdependent and constitute an infinite loop such that $\beta\,\gamma\,\delta$ is repeatedly executed. The operation α is independent of the operations in O_1, and can be executed only once. Each interleaving sequence of P_1 is infinite and contains at most one occurrence of α, interleaved after an arbitrary prefix of $(\beta\,\gamma\,\delta)^\omega$. The reduced state graph constructed according to the above principles is depicted on the right (the starting node is black). For each node s reached by an edge labeled with α, the only enabled operation is in $sleep(s)$. Thus, in each sequence generated by the graph, once α is taken, it is the last operation. The only infinite sequence generated is $(\beta\,\gamma\,\delta)^\omega$.



Valmari [15] suggested an algorithm for spawning reduced state graphs for programs, extending a previous method by Overman [11]. His method is based on avoiding the expansion of all the enabled directions from a given node s. This is done by finding a sufficient subset of the enabled directions that need to be explored from s called a *stubborn set*. Then, only these directions are explored. The reduced state graph obtained in this way is guaranteed to contain all of the deadlocks (i.e., states that do not have successors) that appear in the full state graph.

The idea of reducing the set of successors from a global state was also developed independently by Katz and Peled [5]. There, proof rules were given, based on a construction of *faithful decompositions* that are similar to stubborn sets. This allows verifying equivalence robust liveness properties of programs by using well founded induction that considers only a subset of representatives of the interleavings sequences. Then, it is possible to deduce that these properties hold for all the execution sequences of the program.

Valmari has extended his methods [16] to verify properties of next-time free (i.e., without '○') LTL formulas using the reduced state graph. There, an operation is defined to be *visible* if it can change the truth value of a proposition that appears in the checked formula. The gain from commutativity strongly depends upon the number of visible operations. When all the operations are visible, there is no gain at all, as the full state graph will be constructed.

The stubborn sets are generated with respect to no particular fairness assumption. If some fairness φ is assumed, it is possible to check that the property ψ holds under the assumption φ by model checking that $\varphi \to \psi$ holds. In addition to the inefficiency

[1] The algorithm described in [3] includes also some additional heuristics to reduce the subset of edges expanded from a given node.

caused by model checking being PSPACE-complete in the size of the formula, adding fairness as a part of the formula also causes *all* the operations of the program to be visible.

3.2 The Suggested Algorithm

We present an algorithm that improves the above techniques in order to achieve the following new goals:

- Representative sequences for all the *infinite* runs (rather than the *finite* traces, as in [3, 16]) of the program are generated by the reduced state graph G. That is, for every run $\pi \in R_P^D$, at least one interleaving sequence $\xi \in \pi$ is generated by G.
- Algorithms that treat various fairness assumptions efficiently, such as [8] can directly use the reduced state graph to check that a property holds under a fairness assumption. Moreover, fairness can be used *during the state graph expansion*, to further reduce the state graphs.
- Heuristics that were used by the previous methods are generalized in such a way that (1) a generic framework is presented for predicting the directions needed to be explored from a given node, such that suggested prediction heuristics can be validated against it, and (2) it is shown that making an optimal prediction is NP-hard.
- Any next-time free temporal property can be model-checked, and (as will be shown in the next section) a reduced state graph can be constructed even in the extreme case when all operations are visible (i.e., can change the truth value of some proposition that appear in the checked formula).
- Additional reduction of the state graph is achieved by deciding whether to construct a new edge only upon *backtracking* from the node to which it is directed. Moreover, after completing the expansion of the state graph, it is possible to identify some states as redundant, and free the space that was used to store them.

The algorithm presented herein is a modification of the algorithm of [3] so that the spawned state graph contains representatives for *infinite* traces. It does so by guaranteeing that for each run π, for each expanded node s and trace $\sigma \sqsubseteq \pi$ with $fin_\sigma = val(s)$, if $\sigma[\alpha] \sqsubseteq \pi$ and α is not in *sleep*(s) (if $\alpha \in$ *sleep*(s) the direction α was already taken care of from some other node), then a path $\zeta = s \xrightarrow{\beta_0} t_1 \xrightarrow{\beta_1} t_2 \ldots t_n \xrightarrow{\beta_n} t_{n+1} \xrightarrow{\alpha} r$ will be generated, with $\sigma[\beta_0 \ldots \beta_n \alpha] \sqsubseteq \pi$.

The following changes are made to the sleep set algorithm: after an edge $s \xrightarrow{\gamma} t$ with $(\alpha, \gamma) \notin D$ that *closes a cycle* is explored (this is detected by testing if the expansion of t is not completed, i.e., is still active, when the edge $s \xrightarrow{\gamma} t$ is explored), the operation γ is *not* added to the sleep set of any node s' such that $s \xrightarrow{\alpha} s'$. We say that the direction γ is *unreliable* from s. The reason is that since the expansion of t is not completed and thus the existence of a path ζ (with $\beta_0 = \gamma$ and $t_1 = t$) for the runs π such that $\sigma[\alpha]$, $\sigma[\gamma] \sqsubseteq \pi$ is not yet guaranteed.

Another change is that an edge is only created upon backtracking it, when it leads to a node from which other edges that were already created, or to a node that has no enabled operations (i.e., a deadlock or termination state). This prevents creating edges that only lead to redundant nodes where all their enabled operations are in their sleep set.

The algorithm that appears in Figure 1 uses the following notation. We identify a node s with $val(s)$. Thus, the α successor of a node s has the value $\alpha(s)$. The operations that are enabled from s are denoted by $en(s)$. A state that is guaranteed to remain in the reduced state graph is labeled with the flag $fixed(s)$ (states that are not fixed will be eventually removed), while $open(s)$ is a boolean flag that holds when s is not fully expanded (is still active) at this point of the execution. The set of directions (i.e., operations) already explored from the node s is denoted by $explored(s)$. The directions which, when explored from s, reach a node that is open (and therefore, close a cycle) are denoted by $unreliable(s)$. The set of directions yet to be explored from node s is $working\_set(s)$. Finally, let $dep(\alpha)$ be the set of operations dependent on α, i.e., $\{\beta \mid (\beta, \alpha) \in D\}$. For the moment, let $ample(t)$ be $en(t)$.

Consider now the complexity of the spawning algorithm when a perfect hashing function to locate old nodes in $\mathcal{O}(1)$ is given. Each node s can be expanded at most as many times as it is assigned a new sleep set, since in each time, $sleep(s)$ is diminished. Thus, each edge can be explored no more than $|T|$ times. Each time an edge $s \xrightarrow{\alpha} t$ is explored, a new sleep set is calculated for t and compared against the old one. This can be done in time $\mathcal{O}(|T|)$. Thus, the time complexity is therefore $\mathcal{O}(|T|^2|E|)$ (notice that the graph is connected), where $|E|$ is the *actual* number of edges explored, rather than the number of edges in the full state graph. The space complexity is $\mathcal{O}((|T||V|)+|E|)$, where V is the actual number of nodes. Notice that space is further saved in this modified version of the sleep set algorithm by not creating edges that lead to non-fixed nodes.

The reduction of the state graph was achieved so far by collecting information about the explored directions. We allow a further reduction of the state graph by allowing $ample(s) \subseteq en(s)$. This is related to [5, 15, 3] and others, where a subset of the enabled directions to be explored from a node is calculated *before* starting its expansion, based on a static analysis of the modeled program. Our adaptation to this principle will again insure representations for all the runs rather than only for finite traces. The following definition, adapted from [13], characterizes all the sets of operations that guarantee at least a single successor for a state q for each run in a set of runs $\mathcal{R}$.

Definition 3.1 *A set* $\mathcal{T} \subseteq T$ *is called an* ample set for q *with respect to a set of runs* $\mathcal{R}$ *if for each* σ *such that* $fin_\sigma = q$, *for each run* $\pi \in \mathcal{R}$ *such that* $\sigma \sqsubseteq \pi$, $\{\tau \mid \tau \in T \wedge \sigma[\tau] \sqsubseteq \pi\} \cap \mathcal{T} \neq \phi$.

Before expanding a node s whose value is q in the reduced state graph, an ample set $\mathcal{T}_s$ for q w.r.t. R_P^D is calculated. We say then that $\mathcal{T}_s$ is an *ample set for* s. Again, we need to prevent cycles in which an enabled operation is not taken as each node relies on its successor to take this direction.

Ample sets are implemented by using a procedure $ample(s)$ that returns an ample set $\mathcal{T}_s \subseteq en(s)$ of directions that do not lead to an open node. If no such subset exist, then $ample(s)$ simply returns $en(s)$. The directions $ample(s) \setminus sleep(s)$ are explored from s. It is also possible to use ample sets without sleep sets. Then one does not have to keep the sleep set of each expanded node in memory, or ever re-expand a node.

```
proc $expand\_node(s)$;
  if $en(s) = \phi$ then $set(fixed(s))$ fi;
  $working\_set(s) := ample(s) \setminus sleep(s)$ ;
  while $working\_set(s) \neq \phi$ do
    $\alpha :=$ some operation of $working\_set(s)$;
    $working\_set(s) := working\_set(s) \setminus \{\alpha\}$;
    $explored(s) := explored(s) \cup \{\alpha\}$;
    $s' := \alpha(s)$;
    $new\_sleep := (sleep(s) \cup explored(s)) \setminus$
        $(dep(\alpha) \cup unreliable(s))$;
    if not exists node $s'$ then
      $create\_node(s')$; $set(open(s'))$;
      $explored(s'), unreliable(s') := \phi$;
      $sleep(s') := new\_sleep$; $expand\_node(s')$ fi
    else if $sleep(s') \not\supseteq new\_sleep$ then   /* re-expand an old node ... */
        $explored(s'), unreliable(s') := \phi$;
        $sleep(s') := sleep(s') \cap new\_sleep$;   / * ... with a smaller sleep set */
        if $\neg open(s')$ then $set(open(s'))$; $expand\_node(s')$
        else $working\_set(s') := ample(s') \setminus sleep(s')$ fi
      fi
    fi;
    if $fixed(s')$ or $open(s')$ then   /* permanent node */
      $set(fixed(s))$;
      if not exists edge $(s, \alpha, s')$ then $create\_edge(s, \alpha, s')$ fi;
      if $open(s')$ then $unreliable(s) := unreliable(s) \cup \{\alpha\}$ fi /* $\alpha$ closes cycle */
    fi
  elihw;
  $unset(open(s))$;
end $expand\_node$.
```

```
program $reduced\_graph$;
  $s := \iota$;  /* starting node */
  $explored(s) := \phi$;
  $unreliable(s) := \phi$;
  $sleep(s) := \phi$;
  $set(open(s))$;
  $expand\_node(s)$;
  foreach $s$ do /*remove nonfixed*/
    if $\neg fixed(s)$ then $remove(s)$ fi
  hcaerof
end $reduced\_graph$.
```

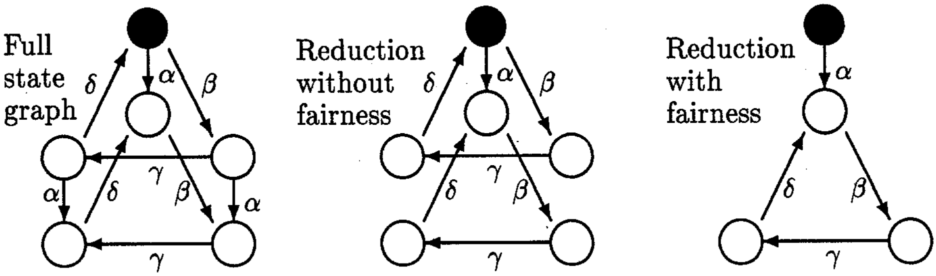
Figure 1: A modified sleep set algorithm

Fairness assumptions play an important rôle in calculating $\mathcal{T}_s$: an equivalence robust fairness assumption can be conceived also as a condition on runs. Then, an ample set can be sought with respect to the set of fair runs. Thus, dealing with a smaller set of runs can be used to diminish the size of the ample sets, resulting in smaller state graphs. Let us consider a specific weak fairness (justice) assumption: an interleaving sequence is fair in this sense iff the following holds:

($*$) if an operation α is enabled from some state of the sequence, then some operation that is dependent on α (possibly α itself) will appear eventually in the sequence.

Notice that from Definition 2.1, once α is enbaled, if no operation dependent on α is executed, it remains enabled.

Example. Consider again the program P_1 in Section 3.1. The sequence $(\beta\gamma\delta)^\omega$ in which α does not appear is unfair with respect to (*). Notice that in the state graph depicted in Section 3.1, no fair interleaving sequence is generated, and thus model checking algorithms such as [8] cannot be applied to it. Three state graphs are depicted below:



The reduction in the rightmost state graph, spawned when assuming the fairness condition (*), is due to the fact that the state graph does not have to generate a representative for the unfair sequence in which α is never taken. Notice in the above example that if no fairness is assumed, this state graph does not contain all the representatives. That is, the sequence $(\beta\gamma\delta)^\omega$ is not represented. ⌟

Theorem 3.2 *Calculating ample sets that can be used to generate minimal sized state graphs, with representative for each equivalence class of R_P^D, is NP-hard.*

The above theorem justifies seeking heuristics, rather than an optimal algorithm to find ample sets. The reduction used in the proof of Theorem 3.2 generates a program that has only finite interleaving sequences, and thus the theorem does not depend on any fairness condition. There is a clear tradeoff between execution time and making a good prediction: the better prediction is more time consuming to make. In Section 4, it will be shown that the choice of the heuristics should depend on the fairness assumption used. Some calculations of subsets of enabled operations in [5, 3, 15] are also particular cases of ample sets.

In some cases, it is also possible to reduce the ample sets when relaxing the requirement that there must be at least one representative sequence for each equivalence class. In these cases, several disjoint equivalence classes can be combined into a new single equivalence class, requiring a single representative, and hence resulting in a smaller state graph.

4 Model Checking Using Reduced State Graphs

4.1 Achieving Equivalence Robustness

It will now be shown that for *any* checked next-time free LTL formula φ we can extend D in such a way that φ becomes equivalence robust. Then it is possible to spawn the reduced state graph (w.r.t. the extended dependency relation) for P and use it to model check if φ holds.

The size of the reduced state graph grows when adding more dependencies. This is because with a larger relation D (that contains more dependent pairs), the equivalence relation becomes finer, and more representatives are needed in the generated state graph.

Thus, our aim is to find a small as possible extension of D that will guarantee equivalence robustness.

An operation $\alpha \in T$ *affects* a proposition Υ, if there exists a finite admissible sequence $v\alpha$ such that the truth value of Υ is different in fin_v than in $fin_{v\alpha}$. We assume that for each proposition Υ that appears in the checked formula φ it is possible to calculate efficiently (e.g., syntactically, during translation of the checked program to a set of operations) a set $a(\Upsilon) \subseteq T$ that *includes at least* the operations that affects Υ. (It will be evident that it is beneficial to keep $a(\Upsilon)$ as small as possible. However, calculating a *minimal* $a(\Upsilon)$ can be expensive.) The set $a(\varphi)$ is defined to be the union of $a(\Upsilon)$ for all propositions Υ in φ. This can be defined inductively on the structure of the formula, e.g., $a(\eta\mathcal{U}\mu) = a(\eta) \cup a(\mu)$, $a(\eta \vee \mu) = a(\eta) \cup a(\mu)$, $a(\Box\eta) = a(\eta)$.

Lemma 4.1 *If $D \supseteq a(\varphi) \times a(\varphi)$, then φ is equivalence robust w.r.t. R_P^D.*

Thus, if D is a dependency relation of P, then $D' = D \cup (a(\varphi) \times a(\varphi))$ is a dependency relation that guarantees that φ can be model checked using a reduced state graph (constructed w.r.t. D'). However, the size of this relation D' can still prevent any practical gain using this method. Obtaining a smaller relation D' is based on the following lemma, which stems directly from the semantic definition of temporal logic formulas.

Lemma 4.2 *If φ and ψ are equivalence robust, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$.*

It is easy to check that equivalence robustness is not preserved under the temporal modalities. Observe that if $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$, then $a(\varphi) = a(\varphi_1) \cup a(\varphi_2)$ and thus $a(\varphi) \times a(\varphi) \supseteq (a(\varphi_1) \times a(\varphi_1)) \cup (a(\varphi_2) \times a(\varphi_2))$.

This provides a strategy of obtaining a smaller dependency relation: given a formula φ, rewrite φ in an equivalent form in which as many as possible boolean operators appear at the outermost levels (i.e., not within the scope of any temporal modal). That is, φ is written in an equivalent form as a boolean combination of some formulas $\varphi_1, \ldots, \varphi_n$. Then, extend the dependency relation D given for a program P to $D' = D \cup (a(\varphi_1) \times a(\varphi_1)) \cup \ldots \cup (a(\varphi_n) \times a(\varphi_n))$. Rewriting the formula can be based on the following (and other) equivalences among temporal formulas: $\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$, $\Box\Diamond(\varphi \vee \psi) \equiv \Box\Diamond\varphi \vee \Box\Diamond\psi$, $\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$, $\Diamond\Box(\varphi \wedge \psi) \equiv \Diamond\Box\varphi \wedge \Diamond\Box\psi$, $(\varphi \wedge \psi)\mathcal{U}\eta \equiv (\varphi\mathcal{U}\eta) \wedge (\psi\mathcal{U}\eta)$, $\eta\mathcal{U}(\varphi \vee \psi) \equiv (\eta\mathcal{U}\varphi) \vee (\eta\mathcal{U}\psi)$. Additional equivalences that do not directly separate the formula into components can also be used Such as $\Box\Box\varphi \equiv \Box\varphi$ and $\Diamond\Diamond\varphi \equiv \Diamond\varphi$.

Rewriting the formula can also be automated: whenever the left hand sides of one of the above temporal equivalences is found, it is replaced by its corresponding right hand side. However, notice that when an until operator ('$\mathcal{U}$') is encountered in a subformula, either the left or the right side of the subformula needs to be duplicated in the rewriting. Thus, the rewriting can increase its length exponentially. But observe that we do not actually need the rewritten formula for model checking, as we can model check the original, equivalent formula. The rewriting is needed only for the purpose of adding less dependencies. Thus, it is only important to identify the boolean components of the rewritten formula, and then add dependencies separately for each one of them. Therefore, instead of rewriting the formula, we can obtain its components, which can be done in time linear in the size of the formula. (But *optimal* rewriting can be easily shown to be NP-hard in the size of the formula.)

The following algorithm pushes components of the formula η that contribute separately to the dependence relation into its stack. As a preparatory step for applying the algorithm to η, it is beneficial to use de-Morgan equivalences (i.e., $\neg(\eta \vee \psi) \equiv \neg\eta \wedge \neg\psi$ and $\neg(\eta \wedge \psi) \equiv \neg\eta \vee \neg\psi$) in order to push negations that precede conjunctions or disjunctions inwards (since, for example, $\Box\neg(\varphi \vee \psi)$ will not be separated, while the equivalent $\Box(\neg\varphi \wedge \neg\psi)$ will).

```
proc separate($\eta$)
    if $\eta$ match formula in table then
      foreach $\mu$ in $sub(\eta)$ do
        separate ($\mu$)
    else push $\eta$ fi
end separate;
```

Type	**Formula** η	$sub(\eta)$	Type	**Formula** η	$sub(\eta)$
1	$\Box(\varphi \wedge \psi)$	φ, ψ	4	$\Diamond\Box(\varphi \wedge \psi)$	φ, ψ
2	$\Box\Diamond(\varphi \vee \psi)$	φ, ψ	5	$(\varphi \wedge \psi)\mathcal{U}\eta$	η, φ, ψ
3	$\Diamond(\varphi \vee \psi)$	φ, ψ	6	$\eta\mathcal{U}(\varphi \vee \psi)$	η, φ, ψ

4.2 Treating Fairness

It is important to consider fairness carefully: it might be the case that some run π contains fair as well as unfair sequences. If the only interleaving sequence representing π in the reduced state graph is unfair, then this sequence is not taken into account during model checking (see [8]), regardless of other sequences of π that are fair. This can lead to an incorrect conclusion about the satisfaction of the checked formula. For this reason, it is important that the fairness assumption is also equivalence robust.

In order to force the assumed fairness condition to be equivalence robust, additional dependencies need to be added. It is important to express the fairness in such a way that the minimal number of dependencies are added. (Alternatively, it is sometimes possible to add dependencies directly for a fairness requirement without expressing it as a temporal formula.)

Consider the following fairness assumptions: a sequence is *operation just* if whenever an operation is enabled from some point onwards, it will be executed infinitely often, i.e., $\bigwedge_{\alpha \in T}(\Diamond\Box en_\alpha \rightarrow \Box\Diamond exec_\alpha)$. A sequence is *operation fair* if whenever an operation is enabled infinitely often from some point, it will be executed infinitely often, i.e., $\bigwedge_{\alpha \in T}(\Box\Diamond en_\alpha \rightarrow \Box\Diamond exec_\alpha)$.

In order to express these assumptions, a special predicate $exec_\alpha$ is used, which is interpreted over *occurrences of operations* in a sequence. It is easy to see that $\Box\Diamond exec_\alpha$ is equivalence robust. In order to make the subformulas $\Box\Diamond en_\alpha$ and $\Diamond\Box en_\alpha$ equivalence robust as well, we can add to the dependency relation for each operation α dependencies between all pairs of operations that can make it enabled or disabled.

The fairness assumption $(*)$ is equivalence robust (see [7, 13]), regardless of the actual modeled program. Thus, it is not necessary to consider extending the dependency relation because of the assumed fairness property. It is weaker than both operation justice and operation fairness. This stems from the reflexivity of D, independently of the choice of D.

Definition 4.3 *A* faithful decomposition [5] *for (a node) s is a partition $\langle \mathcal{T}_s, \bar{\mathcal{T}}_s \rangle$ of the operations (i.e., $\bar{\mathcal{T}}_s = T \setminus \mathcal{T}_s$), satisfying the following conditions:*

1. *All the operations in $\mathcal{T}_s$ are enabled from $val(s)$.*
2. *If an operation $\alpha \in \bar{\mathcal{T}}_s$ is dependent on some operation $\beta \in \mathcal{T}_s$, then α is disabled from $val(s)$, and cannot become enabled unless at least one operation from $\mathcal{T}_s$ is executed.*

Lemma 4.4 *Under the fairness assumption $(*)$, if $\langle \mathcal{T}_s, \bar{\mathcal{T}}_s \rangle$ is a faithful decomposition in s, then $\mathcal{T}_s$ is an ample set for s.*

Notice that the notion of ample sets is more general than faithful decompositions.

In order to calculate faithful decompositions, define a symmetric relation $E \subseteq T \times T$ such that if $(\alpha, \beta) \in E$ then α and β cannot be enabled both from a mutual state (i.e., $en_\alpha \cap en_\beta = \phi$). It is preferable, for achieving smaller ample sets, to have as big an E relation as possible. Evaluating a relation E for a program P that will have *exactly* all the pairs of operations that cannot become enabled simultaneously is expensive. However, a practical evaluation of E can be done when translating a program written in some programming language to a set of operations: e.g., operations that correspond to a mutual process can be in E, unless they belong to the same non-deterministic choice.

Let s be a node of the expanded state graph that we wish to expand. Then $\langle \mathcal{T}_s, \bar{\mathcal{T}}_s \rangle$ is a faithful decomposition if $\mathcal{T}_s \subseteq en(s)$ and $\{\tau \mid \exists \alpha \in \mathcal{T}_s \, (\alpha, \tau) \in D \setminus E\} \subseteq \mathcal{T}_s$. That is, all the operations that are dependent on operations in $\mathcal{T}_s$, and are not necessarily disabled when an operation of $\mathcal{T}_s$ is enabled, are already in $\mathcal{T}_s$.

Based on the above description, calculating an ample set U for a node s can start with choosing U as a singleton operation from $en(s)$. Then U is repeatedly expanded by adding all the operations that are dependent on operations in U, and are not disabled when one of the operations selected before is enabled. If during this expansion of U, it contains operations that are not included in $en(s)$, this set U is abandoned, and a new search is started from a different operation of $en(s)$ that was not used before. Selecting the first operation α can be prioritized, as in some cases it is possible to estimate on the size of the ample sets in which they may participate.

```
proc ample($s$);
    $V := en(s)$;
    while $V \neq 0$ do
        choose some $\alpha \in V$;
        $X, U := \{\alpha\}; DIS = \phi$;
        repeat $DIS := DIS \cup \{\beta \mid \exists\gamma \in X \, (\beta, \gamma) \in E\}$;
            $X := \{\beta \mid \exists\gamma \in X \, (\gamma, \beta) \in D \wedge \beta \notin U \cup DIS\}$;
            $U := U \cup X$;
        until $X = \phi$ or $X \not\subseteq V$ ;
        if $X = \phi$ and not
          $\exists\tau \in U \, \exists s' \, (s \xrightarrow{\tau} s' \wedge open(s')) \; / * \; \tau$ closes a cycle*/
```

```
            then return($U$) fi;
         $V := V \setminus U$
    elihw;
    return($en(s)$);  /* cannot find a smaller ample set */
end ample;
```

4.3 Model Checking without Fairness Assumption

Consider now the case where no fairness is assumed. In this case, constructing ample sets from faithful decompositions is not possible. As a counter example consider the program P_1 in section 3.1. Then, $\{\alpha\}$ can be considered as a faithful decomposition for the starting node, but not as an ample set, since when no fairness is assumed $(\beta\gamma\delta)^\omega$ is a legal execution sequence, in which α is never executed. The definition of a faithful decomposition needs to be strengthened in order to give enough representatives:

Definition 4.5 A strongly faithful decomposition *in (a node) s is a faithful decomposition $\langle \mathcal{T}_s, \bar{\mathcal{T}}_s \rangle$ satisfying that no operation from $\bar{\mathcal{T}}_s$ can be executed infinitely many times without any operation of $\mathcal{T}_s$ being executed.*

Previously, Definition 4.3 allowed that operations that are independent of the operations in $\bar{\mathcal{T}}_s$ are unconditionally repeated, as the fairness assumption $(*)$ does not allow them to occur exclusively. In practice, one might need to further strengthen strongly faithful decomposition, e.g., replace 'infinitely many times' by 'once' in the above definition. This is still expensive to check, but an estimating procedure can be given, based on a syntactic analysis of the program.

4.4 Checking Safety Properties

As seen from the above examples, fairness assumptions, such as $(*)$ can be taken into account when calculating ample sets, resulting in smaller ample sets. The following theorem guarantees that when model checking *safety* properties, it is possible to gain from incorporating fairness conditions, even if the executions are *not* assumed to be fair. Thus in the case of safety properties, we can choose the most beneficial fairness assumption that can be used for generating ample sets (while adding as few dependencies as possible), even if no particular fairness is assumed.

We restrict ourselves to deal with fairness conditions that satisfy that every finite admissible sequence can be extended to a full fair execution sequence. This requirement is called *feasibility* in [2]. It means that, according to the definitions in [1], these fairness assumptions are required to be liveness properties. This requirement is satisfied by many fairness assumptions, such as operation justice, operation fairness and many others (e.g., the ones in [9]).

Theorem 4.6 *Given a feasible fairness assumption ψ, a safety property φ, and a program P, φ holds in P under the assumption ψ iff it holds without any fairness assumption.*

The weak fairness assumption $(*)$ is equivalence robust (see [13]), and thus adds no new dependencies. Combined with the fact that an algorithm for constructing ample sets was given in Section 4.3, it turns to be beneficial to use it when checking safety properties.

5 Conclusions

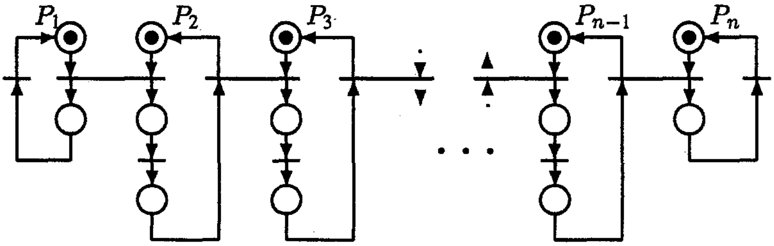
An algorithm for spawning reduced state graphs for finite state programs was suggested. The algorithm guarantees that the reduced state graph generates at least one representative interleaving sequence for each run of the program. Then, model checking algorithms such as [8] can be applied to it for checking any equivalence robust property using the reduced state graph rather than the full state graph.

It was also shown how to force any next-time free temporal formula φ to become equivalence robust by adding additional dependencies, based upon the checked formula φ (and the fairness assumption). By rewriting the checked formula as a boolean combination of temporal properties, less dependencies are added.

An additional feature of the framework presented herein is the ability to efficiently model-check properties with respect to fairness assumptions. It was shown that a fairness assumption can be exploited in the spawning stage to obtain smaller state graphs. The fairness assumption $(*)$ was found to be in particular beneficial for model checking using representatives: it is itself equivalence robust and thus contributes no additional dependencies. It is interesting to observe that $(*)$ is also a natural assumption for trace semantics, as it is equivalent to considering only maximal (under '$\sqsubseteq$') runs [7, 13].

An implementation of the algorithm was written in PROLOG. The table below compares the number of states, edges and run time in seconds of generating state graphs for a generic pipeline paradigm. This paradigm corresponds to various distributed algorithms such as finding prime numbers and sorting: the leftmost process is a generator of elements, while the other processes receive a value from the left, do some internal calculations, and (except the rightmost process) send the value to the right. The Petri-Net [14] that corresponds to the checked program appears bellow the table.

No. of	Full State Graph			Reduced State Graph		
Proc.	Nodes	Edges	Time	Nodes	Edges	Time
3	12	20	0.84	5	5	0.83
4	36	76	0.267	7	7	0.116
5	108	276	1.317	9	9	0.167
6	324	972	7.6	11	11	0.2
7	972	3348	59.016	13	13	0.216



Acknowledgements. I would like to express my gratitude to Patrice Godefroid, Gerard

Holzmann, Antti Valmari and Pierre Wolper for careful reading of earlier drafts, and their helpful comments on this subject.

References

[1] B. Alpern, F.B. Schneider, Defining liveness, Information Processing Letters 21 (1985), 181–185.

[2] K. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Distributed Computing, Vol 2 (1988), 226–241.

[3] P. Godefroid, Using partial orders to improve automatic verification methods, CAV'90, DIMACS Series, Vol 3, 1991, 321–339.

[4] P. Godefroid, P. Wolper, Using partial orders for the efficient verification of deadlock freedom and safety properties, CAV'91, Aalborg, Denmark, 1991, LNCS 575, Springer–Verlag, 332–342.

[5] S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, Distributed Computing 6 (1992), 107–120, A preliminary version, titled An efficient verification method for parallel and distributed programs, appeared in: Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, The Netherlands, 1988, LNCS 354, Springer–Verlag, 489–507.

[6] S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101 (1992), 337-359, a preliminary version appeared in BCS–FACS Workshop on Semantics for Concurrency, Leicester, England, July 1990, Springer–Verlag, 262–280.

[7] M. Z. Kwiatkowska, Fairness for non–interleaving concurrency, Phd. Thesis, Faculty of Science, University of Leicester, 1989.

[8] O. Lichtenstein, A. Pnueli, Checking that finite-state concurrent programs satisfy their linear specification, 11^{th} ACM POPL, 1984, 97–107.

[9] Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language. 9^{th} ACM POPL, Texas, 1983, 141–151.

[10] A. Mazurkiewicz, Trace semantics, in: W. Brauer, W. Reisig, G. Rozenberg (eds.) Advances in Petri Nets 1968, Bad Honnef, LNCS 255, Springer–Verlag, 1987, 279–324.

[11] W.T. Overman, Verification of concurrent systems: function and timing, Ph.D. dissertation, University of California at Los Angeles 1981, 174p.

[12] D. Peled, 'Sometimes' sometimes is as good as 'always', CONCUR'92, Stony Brook, NY, USA, August 1992, LNCS 630, Springer–Verlag 1992, 192–206.

[13] D. Peled, A. Pnueli, Proving partial order liveness properties, 17^{th} ICALP, LNCS 443, Springer–Verlag, 1990, 553-571.

[14] W. Reisig, Petri Nets: An Introduction, EATCS Monographs on Theoretical Computer Science, Springer–Verlag 1985.

[15] A. Valmari, Stubborn sets for reduced state space generation, 10^{th} International Conference on Application and Theory of Petri Nets, Vol. 2, 1–22, Bonn, 1989.

[16] A. Valmari, A Stubborn attack on state explosion, CAV'90, DIMACS Series, Vol 3, 1991, 25–42.