Faculty of Informatics
Masaryk University Brno

On-the-fly State Space Reductions

by

Radek Pelánek

FI MU Report Series **FIMU-RS-2005-03**

On-the-fly State Space Reductions[*]

Radek Pelánek
Masaryk University Brno, Czech Republic
xpelanek@fi.muni.cz

February 1, 2005

Abstract

We present an overview of equivalences of finite state structures and discuss methods for computing reduced structures on-the-fly. We evaluate merits of these reductions on a large set of model checking case studies. It turns out that the achieved reduction can be significant, but that it is not so “drastic” as sometimes claimed in the literature. We also propose some new reduction methods.

1 Introduction

Explicit finite state model checking is a successful technique for verification of concurrent systems and particularly communication protocols. It is based on an exhaustive exploration of the whole state space of a given model. Even sophisticated verification techniques like automatic abstraction refinement [13, 3] or compositional verification [28] are, at the end, based on the exhaustive exploration of the state space. The main limitation of the exhaustive exploration is the state explosion problem.

Specifications which are checked over the model are usually expressed in some temporal logic. Temporal logics cannot distinguish between structures which are ’equivalent’; the exact meaning of ’equivalent’ depends on the logic we are interested in. Instead of checking the specification over the (very big) state space we can check it over some (smaller) equivalent structure. One possibility is to generate the whole state space, reduce it by suitable equivalence, and finally check the specification over the reduced structure. This approach, however, do not reduce the peak memory requirements

[*]Supported by GA ČR grant no. 201/03/0509 and the research centre ITI, project no. 1M0021620808

which are the main practical limitation of model checking. Another possibility is to employ static analysis and use specific information about the model to compute a reduced structure on-the-fly.

There are many techniques for such on-the-fly reduction. The most well-known are symmetry reduction and partial order reduction. However, it is not clear what are the practical merits of these reductions. Researchers usually demonstrate the effectiveness of proposed reduction on just one or two (well selected) examples. Often we can find claims about exponential or at least “drastic” reduction. Are these claims appropriate?

We make the following contributions to the field of on-the-fly state space reductions:

- We present a systematic overview of equivalences, connections to logics, and methods for the on-the-fly computation of reduced structures.
- We present several new methods for on-the-fly reduction. First, we present reductions preserving bisimulation which are based on identification of equivalent values, equivalent states, and on linear transformations of variables. Second, we present reductions preserving reachability which are based on boring states and dominating values.
- We give a realistic evaluation of merits of discussed reductions. For the evaluation we use three model checking tools and many case studies previously studied in the literature. The results show that the effect of reductions can be significant but it is not so drastic as often claimed in the literature.

2 Models, equivalences, and logics

In this section we discuss the basic theory behind state space reductions. We introduce Kripke structures as a basic model of state spaces of concurrent systems and networks of extended finite state machines as a basic modeling language. Then we define equivalences between two Kripke structures.

2.1 Models

Let AP be a finite set of atomic propositions. *Kripke structure* over AP is a tuple $M = (S, s_0, \rightarrow, L)$ where S is a finite set of states, $s_0 \in S$ is an initial states, $\rightarrow \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function.

As a modeling language we use networks of extended finite state machines (EFSM). Let V be a finite set of variables (with a finite domain $D \subseteq \mathbb{Z}, 0 \in D$). A guard g over V is an expression over variables V (set of all guards is denoted $guard(V)$), effect f over V is a sequence of assignments $v := e$, where $v \in V$ and e is an expression over V (set of all effects is denoted $effect(V)$). An *EFSM over* V is a tuple $A = (Q, l_0, Act)$ where Q is a finite set of location, $l_0 \in Q$ is an initial location, and $Act \subseteq Q \times guard(V) \times effect(V) \times Q$ is a set of actions. A *network of EFSM over* V is a tuple $N = (A_1, \ldots, A_n)$ where each A_i is an EFSM over V.

A *valuation* is a function $\gamma : V \to D$. Let Γ be set of all valuations. The semantics of guard ($\gamma \models g$) and effect ($\gamma' = f(\gamma)$) is defined in standard way. We suppose that the set of atomic propositions AP is given as a set of expressions over V. The semantics of network N with respect to a set of propositions AP is a Kripke structure $M(N) = (S, s_0, \to, L)$ where

- $S = Q_1 \times \ldots \times Q_n \times \Gamma$
- $s_0 = (l_0^1, l_0^2, \ldots, l_0^n, \gamma_0)$ where $\gamma_0(v) = 0$ for each $v \in V$
- $(l_1, \ldots, l_i, \ldots l_n, \gamma) \to (l_1, \ldots, l_i', \ldots, l_n, \gamma')$ iff $(l_i, g, f, l_i') \in Act_i$ and $\gamma \models g$, $\gamma' = f(\gamma)$.
- $L(s) = \{a \in AP \mid s \models a\}$

In the following we use $\vec{l}$ to denote a vector $(l_1, \ldots, l_n)$ and $Act = \bigcup Act_i$. We use the standard substitution notation like $\gamma[x := a]$ and $\vec{l}[l_i'/l_i]$.

Modeling languages used in practice are extended with additional features, e.g., with more complex data domains and data structures, communication, or synchronous execution. Most reductions can be straightforwardly modified to work with these extended modeling languages.

2.2 Equivalences

A *path* π in a Kripke structure is a (possibly infinite) sequence $\pi = s_0 s_1 s_2 \ldots$ such that $\forall i \geq 0 : s_i \to s_{i+1}$. Two paths $\pi = s_0 s_1 \ldots$ and $\pi' = s_0' s_1' \ldots$ are *equivalent* iff $\forall i \geq 0 : L(s_i) = L(s_i')$. Paths π, π' are *stutter equivalent* iff there exists a partition $B_1, B_2, \ldots$ of π and a partition $B_1', B_2', \ldots$ of π' such that $\forall j \geq 0$: blocks B_j, B_j' are nonempty and finite and $\forall s \in B_j, s' \in B_j' : L(s) = L(s')$. A state s is *reachable* in M iff there exists path from s_0

to s. State s is *deadlocked* iff $\forall s' \in S : s \not\to s'$. An atomic proposition $a \in AP$ (respectively deadlock) is reachable in M iff there exists a state s reachable in M such that $s \models a$ (respectively s is deadlocked).

For the rest of this section let $M = (S, s_0, \to, L)$ and $M' = (S', s_0', \to', L')$ be two Kripke structures with the same set of atomic propositions AP. We have several notions of an equivalence between Kripke structures:

- The structures M, M' are *reachability equivalent* iff for each $a \in AP$: a is reachable in $M \Leftrightarrow a$ is reachable in M'.
- The structures M, M' are *deadlock equivalent* iff deadlock is reachable in $M \Leftrightarrow$ deadlock is reachable in M'.
- The structures M, M' are *trace equivalent* iff for each path π in M there exists an equivalent path π' in M' and vice versa.
- The structures M, M' are *stutter trace equivalent* iff for each path π in M there exists a stutter equivalent path π' in M' and vice versa.
- A relation $R \subseteq S \times S'$ is a simulation relation iff for all $(s, s') \in R$ the following hold:
  - $L(s) = L'(s')$
  - For every $s' \to s_1'$ there exists $s \to s_1$ such that $R(s_1, s_1')$.

  A state s *simulates* s' (denoted $s \succeq s'$) iff there exists simulation R such that $(s, s') \in R$. The structure M *simulates* M' ($M \succeq M'$) iff $s_0 \succeq s_0'$. The structures M, M' are *simulation equivalent* iff $s_0 \succeq s_0'$ and $s_0' \succeq s_0$.
- A relation $R \subseteq S \times S'$ is a bisimulation relation iff R is symmetrical and R is a simulation relation. States s, s' are *bisimilar* (denoted $s \sim s'$) iff there exists a bisimulation relation R such that $(s, s') \in R$. The structures M, M' are *bisimilar* iff $s_0 \sim s_0'$.
- A relation $R \subseteq S \times S'$ is a stutter simulation relation iff for all $(s, s') \in R$ the following hold:
  - $L(s) = L'(s')$
  - For every path π' in M' that starts in s' there is a path π in M that starts in s, a partition $B_1 B_2 \ldots$ of π, and a partition $B_1' B_2' \ldots$ of π' such that $\forall j \in \mathbb{N}$: blocks B_j and B_j' are nonempty and finite, and $\forall t \in B_j, t' \in B_j' : (t, t') \in R$.

The structures M, M' are *stutter simulation equivalent* iff there exists stutter simulation relations R_1, R_2 such that $(s_0, s'_0) \in R_1$, $(s'_0, s_0) \in R_2$.

- *Stutter bisimulation equivalence* is defined analogically to bisimulation equivalence.

Figure 1 summarize relations among these equivalences and show which temporal logic is preserved by which equivalence.

$$\begin{tikzcd}
 & \begin{array}{c}\text{bisimulation}\\ CTL^*\end{array} \arrow[dl] \arrow[dr] & & \\
\begin{array}{c}\text{stutter bisimulation}\\ CTL^*_{-X}\end{array} \arrow[dr] \arrow[ddd] & & \begin{array}{c}\text{simulation}\\ ACTL^*\end{array} \arrow[dl] \arrow[dr] & \\
 & \begin{array}{c}\text{stutter simulation}\\ ACTL^*_{-X}\end{array} \arrow[dr] & & \begin{array}{c}\text{trace equiv.}\\ LTL\end{array} \arrow[dl] \\
 & & \begin{array}{c}\text{stutter equiv.}\\ LTL_{-X}\end{array} \arrow[d] & \\
\text{deadlock equiv.} & & \text{reachability equiv.} &
\end{tikzcd}$$

Figure 1: Relations among equivalences and temporal logics. For each equivalence we give temporal logic which is preserved by the equivalence. Proofs of can be found in [11, 21, 36].

3 Computing Reductions On-the-fly

The basic algorithm GENERATE(N) which generates the reachable part of the structure $M(N)$ is given in Fig. 2. The model generated by this algorithm is bisimilar to the full structure $M(N)$. Our aim is to modify this basic algorithm in such a way that it will produce structure that is smaller and yet equivalent (up to one of the equivalences discussed above). In some cases we are able to perform the reduction by static transformation of the model prior to the exploration of the state space.

All on-the-fly reductions depend on static analysis to compute some information that is used for the reduction. We employ both data flow analysis techniques (e.g., live variable analysis, reaching definitions, constant propagation) and control flow analysis techniques. Most of the information can be computed by standard static analysis algorithms. In some cases, it is quite challenging, particularly for extended modeling languages with communication, synchronization, and more complex data types. We may even employ model checker itself to compute the necessary information on some abstract version of the model (this correspond to static analysis with abstract interpretations). We can also employ theorem prover [6] or iterate between static analysis and model checking [10].

```
proc GENERATE($N$)
  $W = \{s_0\}$
  while $W \neq \emptyset$ do
        get $s$ from $W$
        add $s$ to $S$
        $L(s) = \{a \in AP \mid s \models a\}$
        foreach $s \to s'$ do
          add $(s, s')$ to $R$
          if $s' \notin S \cup W$ then add $s'$ to $W$ fi
        od
  od
  return $(S, s_0, R, L)$
end
```

Figure 2: The basic algorithm

3.1 Reductions Preserving Bisimulation

Reductions preserving bisimulation are based on the notion of canonization function. Let us add the line

s' = *canonize*(s')

after the line 7 of the basic algorithm. If the *canonize* function give unique representant from each class then we obtain the bisimulation collapse (minimal bisimulation equivalent structure). However, it is not feasible to compute unique representant on-the-fly. For the correctness of the algorithm it is sufficient that the canonization function returns *some* bisimilar state, i.e., that $\forall s \in S : \mathit{canonize}(s) \sim s$. As long as the canonization function can be performed efficiently, the modified algorithm involves no significant overhead over the basic algorithm.

In the following we discuss several ways of computing the canonization function. Note that different canonization functions can be straightforwardly combined.

3.1.1 Dead variables

A variable $x \in V$ is *dead* in a state $(\vec{l}, \gamma) \in S$ iff for all $a \in D : (\vec{l}, \gamma) \sim (\vec{l}, \gamma[x := a])$. Dead variables can be computed (respectively safely approximated) by standard static analysis algorithms. We can get better results by incorporating a dependency analysis: variables which cannot influence neither the control flow nor the value of variables which occur in atomic propositions are dead — this idea is also called faith variable analysis or cone of influence reduction. The definition of dead variables lends itself to the canonization function: $C((\vec{l}, \gamma)) = (\vec{l}, \gamma')$ where $\gamma'(x) = 0$ if x is a dead variable and $\gamma'(x) = \gamma(x)$ otherwise.

For extensions of the basic model the idea of dead variables can be extended:

- For models with arrays it is useful to consider that each individual position in array can be dead. The static analysis is more complicated — it is useful to perform "constant propagation" (in order to find constant array indexes) and "live range analysis".
- For models with message queues it is possible to generalize the notion of deadness to the content of a queue [18].
- For timed automata it is customary to talk about "active clock reduction" [15].

3.1.2 Equivalent values

Values $a, b \in D$ are *equivalent for a state* $(\vec{l}, \gamma) \in S$ *and a variable* $x \in V$ iff $(\vec{l}, \gamma[x := a]) \sim (\vec{l}, \gamma[x := b])$. This is an extension of the idea of dead variables. Note that the region constructions for timed automata [1] is in fact based on equivalent values. In the case of timed automata, special symbolic representation is used for the whole set of equivalent values.

Static detection of equivalent values is more complicated than dead variables detection. It is possible for example in the following cases:

- Variable x is (locally) used only in expression $x = k$ and for several values the same action is performed, e.g., process is waiting for an reset signal, all other signals are discarded.
- Monotone increasing variable x is used only in guards $x \leq k$, e.g., in (discrete) time models and scheduling problems. In this case all values larger than the maximal constant to which x is compared are equivalent.

We can use the following canonization function: $C(\vec{l}, \gamma) = (\vec{l}, \gamma')$ where $\gamma'(x) = \min\{a \mid \text{values } a \text{ and } \gamma(x) \text{ are equivalent values for } (\vec{l}, \gamma) \text{ and } x\}$.

3.1.3 Equivalent states

We can also detect equivalent states. A *local bisimulation relation* on a machine A_i is a relation $R \subseteq Q_i \times Q_i$ such that $\forall (l, l') \in R$:

- for each $(l, g, f, l_1) \in Act_i$ there exists $(l', g, f, l'_1) \in Act_i$ such that $(l_1, l'_1) \in R$
- for each $(l', g, f, l'_1) \in Act_i$ there exists $(l, g, f, l_1) \in Act_i$ such that $(l_1, l'_1) \in R$

It is, clearly, redundant to have two locally bisimilar locations in a model. More formally, if $l_i, l'_i \in Q_i$ and there exists a local bisimulation relation $R \subseteq Q_i \times Q_i$, $(l_i, l'_i) \in R$ then $(\vec{l}, \gamma) \sim (\vec{l}[l'_i/l_i], \gamma)$. Equivalent states can be merged by static transformation prior to the exploration of the state space.

3.1.4 Symmetry

Bisimulation relation can also be identified by exploiting symmetries in the model. Symmetries are formalized by the notion of an automorphism. An *automorphism* is a bijection $h : S \to S$ such that $\forall s \in S : L(s) = L(h(s))$ and $s \to s' \Leftrightarrow h(s) \to h(s')$. If h is an automorphism then $\forall s \in S : s \sim h(s)$. Automorphisms can be detected by static analysis if the model exhibit some kind of symmetry. As an elementary example consider network $N = (A_1, \dots, A_n, A_{n+1}, \dots, A_m)$ such that $A_1 = \dots = A_n$ are identical machines. Let π be permutation on $\{1, \dots, n\}$. Then $h_\pi((\vec{l}, \gamma)) = (\vec{l}[l_{\pi(1)}/l_1, \dots, l_{\pi(n)}/l_n], \gamma)$ is an automorphism. In this case we can use as a canonization function permutation which sorts first n locations in a state vector. This basic idea can be extended to a more general class of models with the use of special data type scalarset [27] (only 'symmetrical operations' are allowed over scalarset).

We propose another approach for detecting automorphisms which is based on linear transformations. For the moment, let us suppose that the data domain $D = \mathbb{Z}_n$ or $D = \mathbb{Z}$. We say that a set of variables $V' \subseteq V$ is *linearly transformable* iff all uses of these variables are either in guards $x \bowtie y + k$ or in effects $x := y + k$ ($x, y \in V', k \in \mathbb{N}$). A typical example of model with linearly transformable variables is an "alternating bit protocol" (where $D = \mathbb{Z}_2$). Many other protocols use modular arithmetic as well.

Lemma 3.1 *Let* $V' \subseteq V$ *be a set of linearly transformable variables. Then the function* $h_k((\vec{l}, \gamma)) = (\vec{l}, \gamma[V' := V' + k])$ *is an automorphism for each* $k \in D$.

We can use following canonization function: we select one fixed variable $v \in V'$ and then use *canonize*$(\vec{l}, \gamma) = h_{-\gamma(v)}((\vec{l}, \gamma))$, i.e., the canonization function always sets value of v to zero.

3.2 Reductions Preserving Stutter Equivalences

Action $\alpha \in$ *Act* is *invisible* iff $\forall s \in S : L(s) = L(\alpha(s))$. On intuitive level, invisible actions are not important for stutter equivalences. Thus we can do reductions which change the order and the execution of invisible actions. Here we discuss the main idea of several reductions based on this observation. First two reductions are 'static' (based on transformations of the model), the other three reductions are 'dynamic' (based on modification of the algorithm).

3.2.1 Slicing

Slicing identifies parts of the model that are relevant to the verified property [22]. Actions which cannot influence visible actions are removed by a static transformation of the model prior to the state space exploration. The resulting model is stutter trace equivalent to the original model.

3.2.2 Transition Merging

If there are two consecutive local invisible actions in the model then we can statically merge them into one atomic action. This basic idea have been formalized in different ways and under different names. Each formalization preserves some stutter equivalence: transition merging, transition compression (stutter trace equivalence) [30, 17, 24], "next" heuristics (stutter simulation) [2], path reduction (stutter bisimulation) [38].

A special case of transition merging is loop acceleration. Loop acceleration aims at reducing the 'fragmentation' of state space caused by repeated execution of some simple cycle. The cycle is statically substituted by meta-transition, which captures repeated executions of the cycle. This techniques is usually used with some kind of symbolic representation [8, 31, 23, 7].

3.2.3 Partial Order Reductions

Partial order reduction traverse only a subset of all enabled actions in a given state. We use the basic algorithm with the line 7 changed into:

foreach $s' \in \{\alpha(s) \mid \alpha \in \mathit{ample}(s)\}$ **do**

According to the exact requirements on the function *ample* we obtain structure that can be deadlock equivalent, reachability equivalent for local properties [20], stutter trace equivalent [35], stutter simulation equivalent [36], or stutter bisimulation equivalent [19]. The basic requirement is that actions in *ample* 'commute' with other actions (this is formalized by the notion of independence).

3.2.4 Confluence

Similar strategy is based on the identification of τ-confluent actions [6, 33] — invisible actions which satisfy some additional confluence requirements (which are similar to the independence requirements of partial order reduction). On-the-fly reduction algorithm work either in the same way as the modified algorithm for partial order reduction, i.e., by choosing only subset of enabled actions (τ-prioritization [33]) or by using τ-confluent actions to compute canonization function [6]. In both cases the reduced structure is stutter bisimilar to the original one.

3.2.5 Simultaneous Reachability Analysis

When faced with several possible interleavings of independent and invisible actions, partial order reduction tries to traverse only one of these interleavings. Simultaneous reachability analysis rather tries to perform *all* these actions at once [32, 37], i.e., instead of executing individual actions it executes combined actions. To preserve correctness it is necessary that there is no dependence among combined actions and that there is at most one visible action in the combined action.

3.3 Reductions Preserving Reachability and Deadlock Equivalences

If we are interested only in the reachability of atomic propositions or deadlock states we can even stop the search in some appropriate states. Here we discuss several ways how to detect such appropriate states. In the following let a be an atomic proposition.

3.3.1 Doomed States

We say that a state s is a-*doomed* (*deadlock-doomed*) if we can guarantee that some state satisfying a (deadlock) is reachable from s.

De Alfaro et al. [16] detect doomed states with the use of the notion of uncontrollability. A location of a machine is uncontrollable if no environment can prevent the machine from reaching a state satisfying a.

For analysis of deadlock-doomed states we can employ an analysis of local cycles. A *covering set* [29, 5] is a set of actions such that each cycle in the structure $M(N)$ contains at least one of these actions. A covering set can be computed by static analysis of local cycles of individual machines and it is often very small [5]. Let $(\vec{l}, \gamma)$ be a state such that from no l_i it is possible to reach in a machine A_i an action in a covering set. Then this state is deadlock-doomed.

3.3.2 Boring States

We say that a state s is a-*boring* (*deadlock-boring*) if we can guarantee that no state reachable from s satisfies a (is deadlocked).

For the detection of boring states we can employ the notion of progress function which was proposed for the sweep line method of state space exploration [12]. Function f is a progress function if $q \rightarrow q' \Rightarrow f(q) \leq f(q')$. If we can show that $\forall q \models a : f(q) \leq k$ then each state q such that $f(q) > k$ is a-boring. The progress function usually needs to be provided by the user.

Boring states can also be detected by analysis of abstract models. Let α be a sound abstraction function, i.e., $M \preceq \alpha(M)$. If no state satisfying a is reachable from $\alpha(s)$ in an abstract structure $\alpha(M)$ then the state s is boring in the structure M.

3.3.3 Dominating Values and States

If $s \succeq s'$ then it is sufficient to visit successors of the state s in the reachability analysis. Thus we can modify the basic algorithm by changing the line 9 into:

<u>if</u> not$(\exists s'' : s'' \in S \cup W \wedge s' \preceq s'')$ **<u>then</u>** add s' to W **<u>fi</u>**

In this case the number of visited states during the search depends on the order in which states are visited. We demonstrate in section 4 that in practice there are quite significant differences between breadth-first and depth-first search order. Nevertheless, the correctness is ensured for each order of visits.

In order to apply this method we need to be able to safely approximate the simulation relation $s \succeq s'$. This can be done by analysis of dominating values. Let $(\vec{l}, \gamma) \in S$ and $x \in V$. We say that value $a \in D$ *dominates* $b \in D$ *for* $(\vec{l}, \gamma)$ *and* x if $(\vec{l}, \gamma[x := a]) \succeq (\vec{l}, \gamma[x := b])$. This is an extension of the notion of equivalent values. The fact that one value dominates another can be detected by analyzing monotone variables. If x is a monotone increasing variable which is used only in guards $x \leq k$ then smaller values of x dominate larger values. This situation occurs in models with discrete time (for dense time we have described reduction based on similar observation in [4]), in scheduling problems, in models with restricted number of occurrences of certain event (e.g., bounded retransmission protocol), or in cryptographic protocols (the intruder knowledge represented by boolean variables is monotone). Similar reasoning works for other combinations of increasing / decreasing variables and lower / upper bounds.

In a similar way as we can identify equivalent states by local bisimulation relation, we can define local simulation relation on machines and use it for identification of simulation between states.

4 Evaluation

For most of the discussed methods it is easy to come up with an (artificial) example on which the reduction gives exponential or at least very significant improvement. Unfortunately, many authors evaluate their techniques on such examples. Moreover, reduction techniques are often evaluated on toy models with high values of parameters — this makes state spaces big and reductions significant. However, in real usage of model checkers models are complex and parameter values are low; very rarely it happens that the model is correct for low values of parameters and erroneous for high values. In [34] we argue that experiments on such toy models can lead to misleading conclusions. In this section we provide an evaluation of merits of individual reductions on realistic models. It turns out that the reduction is more temperate than often claimed.

Most of the models that we use for the evaluation are well-known model checking case studies: alternating bit protocol, Peterson's mutual exclusion protocol, bounded retransmission protocol, I-protocol, firewire link protocol, leader election protocol, real-time Ethernet protocol, cache coherence protocol, firewire tree identification protocol, file transfer protocol, X.509 authentication protocol, Needham-Schroeder protocol, production cell case study, etc.

The evaluation was done with three explicit model checkers: Spin (version 4.0.6), Murphi (version 3.1), and DiVinE (a prototype of a model checker developed in our laboratory[1]).

We discuss to what kind of models each reduction is suitable, we summarize experimental results in other papers and report about the effect of the reduction on our models. We also discuss the run-time overhead of the reduction and the 'complexity' of its implementation. We give results only on those models on which the reduction had some effect. We present only the number of states in full and reduced structure. Reduction with respect to the number of transitions is usually very similar.

4.1 Dead variables

Dead variable reduction can reduce the size of the state space up to 10% of the size of the full state space (see Table 1). Yorav [38] gives similar results on four software models. Bozga et al. [18] reports more impressive reduction but only on one parametric model. This reduction strategy is applicable to a wide class of models and it brings nearly no run-time overhead. For local variables, which are responsible for most of the reduction, we can perform the canonization by static transformation. Dead variables are easy to detect statically; it becomes more challenging only for arrays and more complex data types.

We have not implemented the equivalent values reduction. The manual inspection of models suggests that this method improves over dead variable reduction only in few cases and that the improvement is not very significant. Moreover, the static analysis needed for this reduction is quite complicated.

[1]`http://anna.fi.muni.cz/divine/`

Table 1: Results for dead variables reduction and POR

Dead variables (DiVinE)				POR (Spin)			
Model	Full	Reduced		Model	Full	Reduced	
rether	10,462	1,192	11.3%	cambridge	146,471	6,298	4.2%
synapse	13,973	1,981	14.1%	erathostenes	25,295	2,093	8.2%
peterson	12,498	2,376	19.0%	snoopy	61,619	9,707	15.7%
brp	4,792	1,571	32.7%	smcs	4,634	1,196	25.8%
production_cell	77,416	32,854	42.4%	mobile	30,652	9,971	32.5%
iprotocol_good	29,994	12,770	42.5%	pftp	144,813	47,356	32.7%
firewire	55,887	24,323	43.5%	i-protocol	2,207,190	919,978	41.7%
abp	11,286	5,652	50.0%	relay	876	442	50.4%
bridge	3,186	1,676	52.6%	peterson	30,432	16,720	54.9%
tip	86,556	49,082	56.7%	brp	290,174	169,208	58.3%
elevator	1,139	723	63.4%	X.509	9,028	6,094	67.5%
bakery	109,144	84,517	77.4%	sgc	299,270	293,126	97.9%
cambridge	8,592	6,962	81.0%	sliding	16,441	14,645	89.0%
resistance	151,587	129,177	85.2%	giop	638,525	638,520	99.9%

4.2 Partial order reduction

Notwithstanding the large body of theoretical work about partial order reduction methods, the number of studies concerning practical results of partial order reduction is rather small. Godefroid [20] gives evaluation on four realistic models. The sizes of reduced structures are between 3% and 55% of the size of original structure. Clarke at el. [14] reports similar results on three realistic models. Table 1 presents results of our experiments. The size of reduced structure is between 4% and 99%. Partial order reduction is applicable mainly to models with loosely coupled processes. For many of our models, which use either lot of rendezvous communication or shared variables, the method is not applicable.

The run-time overhead and complexity of static analysis depends on the quality of the reduction which we want to achieve. In our experiments we use the tool Spin. Spin uses a rather conservative approach which sacrifices some possible reduction for low overhead [25].

Confluence and simultaneous reachability analysis, other two techniques based on similar ideas as partial order reduction, have comparable results [33, 6, 32].

4.3 Symmetry reduction

Symmetry reductions based on permutations can, in theory, achieve reduction up to $n!$ where n is the number of symmetrical entities. Table 2 presents practical results. Experiments were done in the tool Murphi on the set of model which was used by Dill and Ip [27]. We have just parametrized protocols by smaller values. This gives more realistic evaluation: the size of reduced structure is between 8% and 50%. Simillar results were reported by Bosnacki et al. [9] in Symmetric Spin and by Iosif [26] for object oriented programs.

The method is, of course, applicable only to models with symmetrical entities. Typical applications are cache coherence protocols, protocols over bus with several symmetrical parties, models with several symmetrical agents. The run-time overhead is non-trivial due to the computation of canonization function. Some reduction can be sacrificed for lower overhead. The detection of symmetries can be done fully automatically only in special cases. For practical purposes it is necessary to extend the modeling language with special 'symmetric constructs' (e.g., scalarset [27]).

Table 2: Results for symmetry reduction and linear transformations

Symmetry reduction (Murphi)

Model	Full	Reduced	
cache	67,418	5,629	8.3%
list4	8,893	1,489	16.7%
peterson3	882	172	19.5%
eadash	1,694	425	25.0%
sci	18,059	4,525	25.0%
ldash	740	372	50.2%

Linear transformations (DiVinE)

Model	Full	Reduced	
cambridge	827	222	26.8%
abp	11,286	5,958	55.1%
brp	14,720	8,121	55.2%

4.4 Linear transformations

This reduction is usable only for restricted set of models. The effect of the reduction is proportional to the size of domain of linearly transformable variables. Table 2 presents results for three protocols. The run-time overhead is neglible, but the static detection of a set of lineary transformable variables is not easy. It is profitable to introduce to the modeling language a new data type for domain $\mathbb{Z}_n$.

4.5 Static transformations

Our experience suggests that the effect of static transformations (merging of equivalent states, slicing, transition merging, loop acceleration) is not very dependent on the type of an application, but rather on the experience and the modeling style of a user and on possibilities of a modeling language. An experienced user can perform lot of the static transformations manually (even unconsciously). Most of our models were crafted by experienced users in rather low-level modeling formalisms and therefore these reductions are not very efficient for them. Table 3. presents results for models on which the transition merging technique was applicable. Dong and Ramakrishnan [17] report much better effect of these reduction. We suspect that this is because their modeling language do not contain atomic constructs (as opposed to Spin and DiVinE). Kurshan et al. [30], Yorav [38], and Holzmann [24] report results slightly better than we have obtained. Holzmann supports our claims about the dependence on modeling style of the user as he shows that by manual re-modeling he can achieve very significant reduction.

Table 3: Results for transition merging

Transition merging (Spin)				Transition merging (DiVinE)			
Model	Full	Reduced		Model	Full	Reduced	
sgc	607,750	299,270	49.2%	iprotocol	29,994	6,445	21.5%
erathostenes	47,669	25,295	53.0%	rether	10,462	6,970	66.6%
pftp	207,481	144,813	69.7%	resistance	151,587	108,095	71.3%
cambridge	166,510	146,471	87.9%	krebs	7,869	6,027	76.6%
snoopy	67,656	61,619	91.0%	firewire	55,887	45,155	80.8%
peterson	33,434	30,432	91.0%				
smcs	5,066	4,634	91.4%				
X.509	9,760	9,028	92.5%				
brp	309,676	290,174	93.7%				
mobile	32,668	30,652	93.8%				

We suppose that static transformations will be very important for models automatically generated from high level description languages and models created by 'naive' users who are not familiar with the underlying model checking algorithms. This type of application is becoming more and more important. In order to convincingly evaluate these techniques, it will be necessary to perform experiments on a large set of models created by non-expert users. At this moment, it is difficult to obtain such a set.

Static transformations do not bring any run-time overhead nor any changes to the model checker itself. This makes them very plausible.

4.6 Dominating values

Table 4 presents results of the reduction based on dominating values. This reduction is applicable only to models which contain some one-way bounded monotone variable — for our models it was either discrete time variable or counter for number of lost messages. The automatic detection of suitable monotone variable is not easy. It is better to let user give us some hints, e.g., by introducing special data type.

The technique involves non-trivial run-time overhead because we need to check whether the current state is simulated by some previously visited state. This check can be implemented by the following way. The identification of simulation relation is usually based only on some small part of the state vector (e.g., one monotone variable).

Table 4: Results for reduction based on dominating values

Model	Full	Reduced (BFS)		Reduced (DFS)	
jobshop	322,330	13,802	4.2%	116,769	36.2%
naive protocol	3,726	648	17.3%	923	24.7%
bridge	3,186	707	22.2%	1579	49.6%
fischer	1,670	704	42.1%	867	51.9%
abp	65,358	27,792	42.6%	30,647	47.0%
logistics	330,636	149,969	45.3%	306,776	92.7%

For computation of a hash function we use only the part of the state vector which do not influence the identification of simulation relation. In this way all possible 'candidate' states end up in a same collision list. We search this collision list exhaustively and make a check for simulation.

As we have already noted in section 3.2, the reduction obtained by this method depends on the order in which states are visited. Table 4 shows that breadth-first order is better than depth-first order.

5 Summary

In this paper we provide a catalog of on-the-fly state space reduction techniques, including some novel ones, and a realistic evaluation of merits of these techniques. Our results can be summarized as follows:

- Each technique is applicable to some class of models. Nothing works really universally. More specialized techniques yield better reduction.
- On real models, no single reduction is able to reduce the size of the state space significantly under 5%. Claims about drastic reduction, which occur in some papers, are not really appropriate.
- Since there are many techniques and many of them are orthogonal, most models can be reduced quite significantly. Certainly, it is useful to implement these reductions.
- The reduction obtained depends not only on application domain, but also on the user's modeling style. Some reductions, particularly those based on static trans-

formation of the model, do not bring nearly no improvement when applied to models crafted by expert users but can be very efficient on models created by non-expert users.

We have also proposed some new reductions. The equivalent states and equivalent values reductions are not very useful for our current set of models. However, we suppose that these reductions will be useful for models created by non-expert users. The linear transformation reduction is applicable mainly to protocols with modular arithmetics. The boring states reduction and dominating values reduction are applicable mainly to models of scheduling problems.

Acknowledgment

I thank Ivana Černá for many valuable discussions and for comments on previous drafts of this paper.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R. Alur and B.-Y. Wang. “Next” heuristic for on-the-fly model checking. In *International Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 98–113, 1999.

[3] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of Programming Language Design and Implementation (PLDI 2001)*, pages 203–213, 2001.

[4] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 312–326, 2004.

[5] G. Behrmann, K.G. Larsen, and R. Pelánek. To store or not to store. In *Proc. of Computer Aided Verification (CAV’03)*, volume 2725 of *LNCS*, 2003.

[6] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proc. of Computer Aided Verification (CAV 2002)*, number 2404 in LNCS, pages 596–609, 2002.

[7] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of Computer Aided Verification (CAV 1996)*, volume 1102 of *LNCS*, pages 1–12, 1996.

[8] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. of Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 55–67, 1994.

[9] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *Proc. of SPIN Workshop*, volume 1885 of *LNCS*, pages 1–19, 2000.

[10] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proc. of Automated Software Engineering (ASE 2001)*, pages 262–272. IEEE Computer Society, 2001.

[11] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor. Comput. Sci.*, 59(1-2):115–131, 1988.

[12] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 450–464, 2001.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169, 2000.

[14] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, November 1999.

[15] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. of Real-Time Systems Symposium (RTSS '96)*, page 73. IEEE Computer Society, 1996.

[16] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Detecting errors before reaching them. In *Proc. of Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 186–201, 2000.

[17] Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proc. of Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 241–256. Kluwer, B.V., 1999.

[18] J.-C. Fernandez, M. Bozga, and L. Ghirvu. State space reduction based on live variables analysis. *Sci. Comput. Program.*, 47(2-3):203–220, 2003.

[19] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time model checking. *Information and Computation*, 150(2):132–152, 1999.

[20] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.

[21] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[22] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order Symbol. Comput.*, 13(4):315–353, 2000.

[23] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[24] G. J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Proc. of SPIN Workshop*, volume 1680 of *LNCS*, 1999.

[25] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. of Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.

[26] R. Iosif. Symmetric model checking for object-based programs. Technical Report TR 2001-5, Kansas State University, 2001.

[27] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[28] J.P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1997)*, volume 1217 of *LNCS*, pages 239–258, 1997.

[29] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS 1998)*, volume 1384 of *LNCS*, pages 345 – 357, 1998.

[30] R. P. Kurshan, V. Levin, and Hüsnü Yenigün. Compressing transitions for model checking. In *Proc. of Computer Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, pages 569–581, 2002.

[31] M. O. Möller. Parking can get you there faster - model augmentation to speed up real-time model-checking. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[32] K. Ozdemir and H. Ural. Protocol validation by simultaneous reachability analysis. *Computer Communications*, 20:772–788, 1997.

[33] G. J. Pace, F. Lang, and R. Mateescu. Calculating tau-confluence compositionally. In *Proc. Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 446 – 459, 2003.

[34] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22, 2004.

[35] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. of Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 377–390, 1994.

[36] W. Penczek, M. Szreter, R. Gerth, and R. Kuiper. Improving partial order reductions for universal branching time properties. *Fundamenta Informaticae*, 43(1-4):245–267, 2000.

[37] H. van der Schoot. Partial-order verification in spin can be more efficient. In *Proc. of SPIN Workshop*. Twente University, 1997.

[38] K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion – Israel Institute of Technology, 2002.