Next-state computation templates in state space exploration

Ruud Koolen
r.p.j.koolen@student.tue.nl

January 19, 2019

Abstract

In the context of the mCRL2 model checking toolkit, we investigate the transition system computation process of *state space exploration* in search of cases in which similar states lead to duplicate work being performed. We present two techniques to take advantage of this scenario to speed up state space exploration by memoizing the results of expensive computations among usable state patterns, resulting in a performance improvement of almost a factor 200 for real-world workloads.

1 Introduction

In model checking applications, the generation of an explicit state space from an algebraic system description is a common and usually time-consuming task. Tools for such computations generally work by implementing the primitive of computing the set of outgoing transitions from a given state, and running a generic graph exploration algorithm on top of that system.

Something such tools do not generally take advantage of is the phenomenon that different states may have (parts of) outgoing transition sets that are very similar, such that parts of the computational work is effectively duplicated when computing outgoing transition sets for multiple such "similar" states. By recognizing such cases of redundant computations, major performance gains could be possible.

In this paper, we present two techniques to take advantage of this phenomenon, in the context of the mCRL2 model checking toolset [**?**]. We do this by analyzing system descriptions in search of opportunities to share computational work between several states, implemented by memoizing certain results among the largest set of states to which it applies. We show how this can speed up the exploration process by almost a factor 200 for particular real-world models.

In Section 2 we formally introduce the different formalisms involved, as well as the base algorithms on which our techniques are based. Section 3 describes

a technique for sharing the result of the expensive set comprehension computations as widely as possible; in Section 4 we describe a technique for bulk dismissal of transition terms known not to apply to states sharing particular patterns. Finally, in Section **??** we summarize our results, and describe some future research that can be done on this topic.

2 Background

2.1 Labelled transition systems

Labelled transition systems are used in mCRL2 both as the formal semantics of a given process description, and as an explicit data structure on which various operations can be performed (in which case it must be finite). In the context of this document, a labelled transition system is an edge-labelled directed multigraph with a designated initial node; formally, it is a tuple $(S, A, \rightarrow, s_0)$ [**?**] where:

- S is a set of states;
- A is a set of action labels;
- $\rightarrow \subseteq S \times A \times S$ is a transition relation;
- $s_0 \in S$ is the initial state.

2.2 Linear process specifications

The notion of linear process specifications forms a formalism for specifying processes in the version of the Algebra of Communicating Processes used by the mCRL2 toolset [**?**]. Specifically, a linear process specification is an equation of the following form:

$$P(\vec{d}_0) \text{ where } P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{\vec{e}:\vec{E}_i} c_i(\vec{d}, \vec{e}) \rightarrow a_i(\vec{f}_i(\vec{d}, \vec{e})) \cdot P(\vec{g}_i(\vec{d}, \vec{e}))$$

in which the variables $\vec{d}$ are called the *process parameters*, the values i in the finite index set I are the *summands* of the process, the variables $\vec{e}$ are the *enumeration variables*, the boolean expression c_i in terms of $\vec{d}$ and $\vec{e}$ is the *condition* for summand i, the term a_i with arguments $\vec{f}_i$ in terms of $\vec{d}$ and $\vec{e}$ is the parameterized *action label* for summand i, the expressions $\vec{g}_i$ in terms of $\vec{d}$ and $\vec{e}$ are the *resulting state* for summand i, and the expressions $\vec{d}_0 : \vec{D}$ form the *initial state*. A linear process specification represents a (not necessarily finite) labelled transition system with one state for each $\vec{d} : \vec{D}$, with $\vec{d}_0$ being the initial state, in which each state $\vec{d}$ has an outgoing transition with action label $a_i(\vec{f}_i(\vec{d}, \vec{e}))$ and target state $\vec{g}_i(\vec{d}, \vec{e})$ for each $i \in I$, $\vec{e} \in \vec{E}_i$ such that $c_i(\vec{d}, \vec{e})$ holds.

2.2.1 Example

As an example linear process specification, one can consider the following equation:

$$\begin{aligned} P(\mathsf{true}, 0) \text{ where } P(b : \mathbb{B}, n : \mathbb{N}) \quad &= \quad \sum_{m:\mathbb{N}} b \to \mathit{red} \cdot P(\mathsf{false}, m) \\ &+ \quad \neg b \wedge n > 0 \to \mathit{blue} \cdot P(b, n-1) \\ &+ \quad \neg b \wedge n = 0 \to \mathit{red} \cdot P(\mathsf{true}, n) \end{aligned}$$

This describes a process that can perform *red* and *blue* actions, such that no infinite sequence of *blue* actions is possible.

2.3 State space exploration

The process of *state space exploration* is the problem of computing, for a given linear process specification, the reachable part of the labelled transition system it represents — assuming this reachable part is finite. Since this is a standard graph exploration problem, standard graph exploration algorithms like depth-first search or breadth-first search can be used to solve it, given an algorithm for computing the outgoing transitions for a given state.

To compute the set of outgoing transitions from a given state (also known as the *next-state set*), mCRL2 currently uses the following algorithm:

Algorithm outgoing-transitions($\vec{d}$)
1 $R := \emptyset$
2 **for each** $i \in I$
3 $\quad \mathcal{E} := \{\vec{e} : \vec{E}_i \mid c_i(\vec{d}, \vec{e})\}$
4 $\quad$ **for each** $\vec{e} \in \mathcal{E}$
5 $\quad\quad A := a_i(\vec{f}_i(\vec{d}, \vec{e}))$
6 $\quad\quad S := g_i(\vec{d}, \vec{e})$
7 $\quad\quad R := R \cup \{(A, S)\}$
8 **return** R

At line 3, the (hopefully finite) *enumeration* $\mathcal{E}$ of terms matching the condition c_i is computed; at lines 5 and 6, two vectors of expressions $\vec{f}_i$ and $\vec{g}_i$ (together called the *transition arguments*) are computed by substituting $\vec{d}$ and $\vec{e}$ in the given expressions.

3 Enumeration caching

In the algorithm described in Section 2.3, both the computation of the enumeration $\mathcal{E}$ and the transition arguments $\vec{f}_i$ and $\vec{g}_i$ are nontrivial computation steps

depending on the process parameters $\vec{d}$. For both types of computations, we can analyze, based on the structure of the expressions involved, to what degree the computation depends on the value of $\vec{d}$ and (for the transition arguments) $\vec{e}$. Given this analysis, we can proceed to cache the results of all computations performed (within the limits of memory constraints), and use the cached result whenever we would otherwise perform a computation that only differs from a cached entry in ways shown not to affect the outcome of the computation.

3.1 Enumeration caching

Since the enumeration computation at line 3 is typically the most expensive computation step in a complete state space exploration instance, it is an attractive optimization target.

For a given $i \in I$, the value of $\mathcal{E}$ depends only on $\vec{d}$, which is a vector of independent parameters $(d_1, \ldots, d_N)$. For each of those parameters d_j, we can show $\mathcal{E}$ does not depend on d_j whenever d_j simply does not *occur* in the (maximally simplified) condition c_i; clearly, when the condition c_i does not depend on d_j, then neither does the set of terms satisfying that condition. Hence, whenever two values $\vec{d}$ differ only in the parameters not occurring in c_i, we can conclude that they must have the same value of $\mathcal{E}$.

Using this notion, we can — for each $i \in I$ — define a function $\mathsf{key}_i(\vec{d})$ that, for a given parameter vector $\vec{d}$, selects those parameters that occur in c_i and ignores the others:

$$\mathsf{key}_i(\vec{d}) = \left[d_j \in \vec{d} \mid d_j \text{ occurs in } c_i \right]$$

For example, for $\vec{d} = [d_1, d_2, d_3, d_4]$ and $c_i = c(d_1, d_3, \vec{e})$, $\mathsf{key}_i(\vec{d}) = [d_1, d_3]$. It now holds that for any $\vec{d_1}$ and $\vec{d_2}$ such that $\mathsf{key}_i(\vec{d_1}) = \mathsf{key}_i(\vec{d_2})$, it is also true that $\{\vec{e} : \vec{E_i} \mid c_i(\vec{d_1}, \vec{e})\} = \{\vec{e} : \vec{E_i} \mid c_i(\vec{d_2}, \vec{e})\}$, for all $i \in I$. Moreover, since the "occurs in" relation can be precomputed before starting the state space exploration proper, the function value $\mathsf{key}_i(\vec{d})$ is very inexpensive to compute. This leads to the following algorithm that maintains a dictionary M_i for each $i \in I$ storing $(\mathsf{key}_i(\vec{d}), \mathcal{E})$ pairs:

Algorithm outgoing-transitions($\vec{d}$)
1 $R := \emptyset$
2 **for each** $i \in I$
3 $k := \mathsf{key}_i(\vec{d})$
4 **if** $M_i[k]$ **defined**
5 $\mathcal{E} := M_i[k]$
6 **else**
7 $\mathcal{E} := \{\vec{e} : \vec{E_i} \mid c_i(\vec{d}, \vec{e})\}$
8 $M_i[k] := \mathcal{E}$
9 **for each** $\vec{e} \in \mathcal{E}$
10 $A := a_i(\vec{f_i}(\vec{d}, \vec{e}))$
11 $S := g_i(\vec{d}, \vec{e})$

12 $R := R \cup \{(A, S)\}$
13 **return** R

Assuming the computation of simplifying the condition c_i is performed perfectly, and that therefore the only process parameters remaining in c_i can indeed (for suitable values of all other parameters) make the difference between the condition evaluating to true and it evaluating to false, this algorithm is optimal in the sense that no other process parameters can be unconditionally ignored in the computation of $\mathcal{E}$. It can, however, be possible to cache $\mathcal{E}$ across different values of a parameter occurring in the condition *for particular values of other variables*; for example, for $c_i \equiv (d_1 = 1 \vee d_2 = 2)$, the value of $\mathcal{E}$ can be used for different values of d_2 whenever d_1 happens to have the value 1. This area of techniques is outside the scope of this research, however, and is left as future research.

3.2 Transition argument caching

Building on the structure introduced in Section 3.1, we can apply a similar analysis to lines 5 and 6 of the original algorithm. Computing terms A and S involves substituting $\vec{d}$ and $\vec{e}$ in the expression vectors $\vec{f_i}$ and $\vec{g_i}$, and simplifying the result; for each expression involved, we can determine what variables occur in it. Occurring variables can be classified into three groups:

- process parameters that occur in the condition c_i;
- process parameters that do not occur in the condition c_i; and
- enumeration variables.

Based on which of those variables occur in a given expression, the resulting expression can be shared across a different range:

- expressions that do not contain any variables at all are constants, and can be trivially cached globally;
- expressions that only contain process parameters occurring in c_i are equal for all i-transitions for states sharing a value of $\mathcal{E}$, and can therefore be cached along with $\mathcal{E}$;
- expressions that only contain process parameters occurring in c_i and enumeration variables are different for different i-transitions within a given state, but equal among different states sharing a value of $\mathcal{E}$, and can therefore be cached along with each particular $\vec{e} \in \mathcal{E}$;
- expressions that contain process parameters not occurring in c_i are different for each state, and therefore cannot be cached at all.

This analysis leads to a fairly obvious extension of the algorithm from Section 3.1:

Algorithm outgoing-transitions($\vec{d}$)

1 $R := \emptyset$
2 **for each** $i \in I$
3 $\quad \vec{h} := \vec{f}_i +\!\!+ \vec{g}_i$
4 $\quad k := \mathsf{key}_i(\vec{d})$
5 $\quad$ **if** $M_i[k]$ **defined**
6 $\quad\quad (\mathcal{E}', \vec{h}_{\text{enum}}) := M_i[k]$
7 $\quad$ **else**
8 $\quad\quad \mathcal{E} := \{\vec{e} : \vec{E}_i \mid c_i(\vec{d}, \vec{e})\}$
9 $\quad\quad \mathcal{E}' := \{(\vec{e}, p_{\text{term}}(\vec{h}, \vec{d}, \vec{e})) \mid \vec{e} \in \mathcal{E}\}$
10 $\quad\quad \vec{h}_{\text{enum}} := p_{\text{enum}}(\vec{h}, \vec{d})$
11 $\quad\quad M_i[k] := (\mathcal{E}', \vec{h}_{\text{enum}})$
12 $\quad$ **for each** $(\vec{e}, \vec{h}_{\text{term}}) \in \mathcal{E}'$
13 $\quad\quad \vec{h}_{\text{none}} := p_{\text{none}}(\vec{h}, \vec{d}, \vec{e})$
14 $\quad\quad$ construct A and S out of $\vec{h}_{i,\text{global}}$, $\vec{h}_{\text{enum}}$, $\vec{h}_{\text{term}}$, $\vec{h}_{\text{none}}$
15 $\quad\quad R := R \cup \{(A, S)\}$
16 **return** R

Here, the three functions p_{enum}, p_{term}, and p_{none} respectively compute the subsets of transition arguments in the second, third, and fourth category described above, substituting the required values for $\vec{d}$ and $\vec{e}$. The vector $\vec{h}_{i,\text{global}}$ is the precomputed analogous set representing the first category of constant transition arguments for summand i.

3.3 Memory usage

Both algorithms described in this Section require $O((|I|\cdot|V|+|E|)\cdot\alpha)$, where $|V|$ and $|E|$ are respectively the number of vertices (states) and edges (transitions) in the generated labelled transition system, and α is the *cache miss ratio*, i.e. the number of times line 8 is executed divided by the number of times line 4 is executed. This means that if the cache is functioning properly, i.e. when α is low, memory usage is quite modest compared to the memory required to store the generated labelled transition system. It also means that even when the cache is *not* functioning properly — when α is close to 1, and most of the cache bookkeeping is performed in vain because cache hits are rare — then the memory used by the cache is comparable to the memory used to store the resulting labelled transition system, which is significant but not unacceptably large. Nonetheless, in situations where memory is tight, it might be useful to implement some system to keep the memory usage from growing out of control. The details of such a scheme are outside the scope of this paper, however.

3.4 Results

In order to test how well the algorithm described in Section 3.1 works in practice, we implemented it as an extension to a version of the mCRL2 toolset and compared its performance to the base version using the unmodified algorithm from

Section 2.3. For a number of selected systems, we performed a full state space exploration, measuring the executing time of the exploration proper (excluding the start-up time, as it is fairly constant and not influenced by the algorithm used) for both implementations. We also measured the cache miss ratio for these systems. Table 1 shows the ratio of the two execution times (which should be more-or-less independent from machine-specific details) and the value $1/\alpha$.

The systems we used to test are the firewire protocol specification, available as part of the official mCRL2 example set ("1394"); a description of a fault-tolerant egg incubator device we have used in past projects ("incubator-2"); and a version of incubator-2 that is structurally almost identical, but has a much larger state space ("incubator-5"). These systems were chosen due to having a practical size for testing, and being fairly representative.

System	*states*	*transitions*	*speedup factor*	$1/\alpha$
incubator-2	18803	46018	2.85	49.26
1394	197197	354155	29.6	212.5
incubator-5	11732672	53042671	6.19	117.5

Table 1: Performance measurements of the enumeration-caching algorithm relative to the base algorithm.

Additionally, as documented in Table 2 we measured the gains of the transition-argument-caching algorithm relative to the enumeration-caching algorithm, using the same procedure as above.

System	*speedup factor*
incubator-2	1.12
1394	0.90
incubator-5	1.17

Table 2: Performance measurements of the transition-argument-caching algorithm relative to the enumeration-caching algorithm.

A quick glance at Table 1 shows that enumeration caching does indeed work, and that performance improvements of more than one order of magnitude are possible for real-world systems. Unfortunately, the results for transition argument caching aren't quite as straightforward; while Table 2 shows that modest gains are possible, it also shows that the extra caching may actually by counterproductive for less susceptible systems.

4 Summand pruning

4.1 Introduction

In Section 3, we detailed a technique to minimize the amount of work performed to compute the outgoing transitions for a state as pertaining to a given sum-

mand. Even if the amount of time required for a single summand is optimized to the point of being negligible, however, a complete state space exploration still takes $O(|V| \cdot |I|)$ time in the best case merely to check each $i \in I$ for each reachable state — even if the great majority of summands ends up not contributing any transitions for each given state. This presents a problem for linear process specifications containing large amounts of summands. To decrease this best-case lower bound, some technique must be used to avoid visiting the bulk of the available summands for most states. If we could show, for a given state $\vec{d}$, that a set of summands $J \subseteq I$ cannot contribute any transitions — by showing that substitution of the state vector in the condition c_i and simplifying it reduces the condition to false without substituting anything for $\vec{e}$, for all $i \in J$ — then the entire set J could be disregarded without further notice for the given state.

To accomplish this, we can compute the subset of summands whose condition does *not* necessarily reduce to false when substituting each prefix of the state vector $\vec{d}$, caching the results along the way. In other words, we iteratively compute (for given $\vec{d}$) $\{i \in I \mid c_i(d_1) \neq \mathsf{false}\}$, $\{i \in I \mid c_i(d_1, d_2) \neq \mathsf{false}\}$, $\{i \in I \mid c_i(d_1, d_2, d_3) \neq \mathsf{false}\}$, …, $\{i \in I \mid c_i(\vec{d}) \neq \mathsf{false}\}$, consulting the cache for each computation; the final term represents the set of summands for which enumeration proper is necessary.

To implement this, we can maintain a decision tree T in which each node represents a prefix $[d_1, \ldots, d_n]$ of a visited state vector, storing for this prefix the set of not-necessarily-empty summands, and having as children those visited prefixes obtained by adding a single value to the end, i.e. $\{[d_1, \ldots, d_n, d_{n+1}] \mid d_{n+1} : D_{n+1}\}$; the root of this tree would represent the empty prefix, corresponding to the full set of summands. With a tree node representing $[d_1, \ldots, d_n]$ implemented as a tuple (W, J), where W is a dictionary mapping values from D_{n+1} to children and J is the subset of not-necessarily-empty summands, this leads to the following algorithm:

Algorithm outgoing-transitions($\vec{d}$)
1 $(W, J) := T$
2 **for each** $d_n \in \vec{d}$
3 **if** $W[d_n]$ **not defined**
4 $J' := \{j \in J \mid c_j(d_1, \ldots, d_n) \neq \mathsf{false}\}$
5 $W[d_n] := (\emptyset, J')$
6 $(W, J) := W[d_n]$
7 $R := \emptyset$
8 **for each** $i \in J$
9 $\mathcal{E} := \{\vec{e} : \vec{E}_i \mid c_i(\vec{d}, \vec{e})\}$
10 **for each** $\vec{e} \in \mathcal{E}$
11 $A := a_i(\vec{f}_i(\vec{d}, \vec{e}))$
12 $S := g_i(\vec{d}, \vec{e})$
13 $R := R \cup \{(A, S)\}$
14 **return** R

Of course, the algorithm described in Section 3 can be substituted for the lower half of the above algorithm to combine the two caching techniques; the two techniques are independent.

4.2 Variable ordering

The performance of the above algorithm depends heavily on the order of the process parameters; a process parameter near the beginning of the process parameter vector whose value doesn't significantly help in disregarding summands duplicates effort for no gain, whereas having a very influential parameter near the end means its optimization potential is effectively wasted. Therefore, to make the above algorithm useful, the process parameters should be ordered in such a way that parameters that tend to reduce many conditions to false are near the start, and parameters that don't tend to reduce many conditions are near the end (or are even disregarded entirely for the purpose of the summand pruning algorithm). Intuitively, the parameters should be ordered by "influence" on the summand conditions; unfortunately, no obvious formal interpretations of this notion are evident.

This problem is reminiscent of the problem of *variable ordering* in the context of Binary Decision Diagrams, and indeed many of the same concerns apply. Since finding the optimal variable ordering for a binary decision diagram is generally infeasible [**?**], applications using binary decision diagrams usually use heuristic algorithms to find reasonably efficient variable orderings. We took the same approach, experimenting with several simple heuristics, the details of which are explained in Section 4.3.

4.3 Results

To test whether the summand pruning algorithm works in principle, we implemented a version of it that does not perform any parameter reordering at all, and tested its performance on an artificially constructed system known to be susceptible to summand pruning. Specifically, we designed a system consisting of 10^6 states and 11^6 summands, of which no more than 2^6 can generate any transitions for each given state. Moreover, this system has the property that any parameter ordering should produce equivalent results. Comparing the execution time of the state space exploration using the summand pruning algorithm with the execution time using the unmodified algorithm described in Section 2.3, we found that the summand pruning algorithm explores this particular system over 1500 times faster than the base algorithm. This shows beyond doubt that summand pruning can indeed produce great performance improvements for well-chosen variable orderings.

In order to apply summand pruning to more realistic systems for which the process parameter ordering isn't as fortune, we implemented several different parameter ordering heuristics:

- an algorithm that simply counts, for each parameter, the number of summand conditions in which the variable occurs, sorting the summands by this number in decreasing order and disregarding parameters for which this value does not exceed a certain threshold ("occurrence count");

- an algorithm that tries to predict the size of the reachable part of each process parameter's domain by analyzing each summand's target state expression vector, and applies the above algorithm scaled by this number ("weighted occurrence count");
- an algorithm that, for each each parameter-summand pair, structurally analyzes the summand condition structure in search for subexpressions that are likely to reduce the condition to false for particular values of the parameter, such as $d_n = K$ for some constant K ("condition structure");
- an algorithm that (like "weighted occurrence count") enumerates reachable parts of parameter domains, substituting all resulting values in each summand condition, in an attempt to directly measure the parameter's selectivity in isolation ("selectivity measures").

For each of these algorithms, we measured its performance by measuring the execution time (and comparing this to the execution time of the Section 2.3 algorithm) when applied to four different systems: the three systems used in Section 3.4 (modified in such a way as to contain a relatively large amount of summands and relatively few enumeration variables), as well as a similarly processed model of the USB protocol ("usb"). Again, we calculated the ratio of the execution time of the base algorithm and the execution time of the summand-pruning algorithm.

Performance improvements relative to Section 2.3 algorithm

	incubator-2	1394	incubator-5	usb
occurrence count	2.40	10.51	2.51	184.55
weighted occurrence count	2.40	44.04	2.51	57.30
condition structure	2.40	40.10	2.51	168.67
selectivity measures	2.40	29.17	2.39	0.89

Table 3: Performance measurements of the summand pruning algorithm using different variable ordering heuristics relative to the base algorithm.

As Table 3 shows, summand pruning can indeed produce great performance improvements for real-world systems. It also demonstrates that a proper parameter ordering greatly affects the efficiency of the algorithm, with a factor 200 performance difference between the most effective ordering and the least effective ordering for the usb system. For this reason, future research on the topic of how to improve parameter ordering heuristics appears to be warranted.

5 Conclusions and future work

In this paper, we have introduced the technique of enumeration caching to take advantage of state similarity by minimizing the number of enumeration operations performed; we have seen that this technique can work well in practice, observing improvements of almost a factor 30 in real-world state space exploration problems. We have also introduced an extension to this technique that

aims to similarly minimize the computation of transition arguments, but for this technique we have only seen modest improvements and even decreases in performance.

Additionally, we have introduced the technique of summand pruning that aims to minimize the amount of work performed on enumerating summands for which it can be shown, based on state similarity, that the summand cannot produce any transitions. We have seen that this can work very well for particular systems, with an observed improvement of a factor 1500 in computation time for a constructed example. We have also seen that the effectivity of this scheme depends greatly on the order of process parameters; moreover, we have seen that relatively simple heuristics can already produce good results, with observed performance improvements of a factor 184.

As shown in Section 4.3, the process parameter ordering strongly influences the performance of the summand pruning algorithm. For that reason, it could be worthwhile to investigate whether better ordering heuristics can be found. In particular, the question of whether the techniques used in existing binary decision diagram implementations can be applied in this setting, and how well they work. Furthermore, to support a practical implementation of the techniques presented in this paper, the question of when to stop caching to avoid excessive memory usage needs to be addressed.