Condensed Storage of Multi-Set Sequences

Marko Mäkelä[*]
Helsinki University of Technology,
Laboratory for Theoretical Computer Science,
P.O.Box 9700, 02015 HUT, Finland

June 27, 2000

Abstract

Tools for state space exploration, or reachability analysers, work by incrementally constructing a set of reachable states. The applicability of these tools is limited by the vast state space of real systems. One way to attack this problem are different reduction methods—another approach is to come up with techniques for representing the set of reachable states in a compact way.

The state—or marking—of a high-level Petri net can be viewed as a sequence of finite multi-sets. A method for encoding markings containing structured values is described, and a comparison to an earlier implementation is presented.

Keywords. Petri nets, reachability analysis, encoding multi-sets, ordered data types

1 Introduction

The limited amount of system memory is a major bottleneck in reachability analysis. Algorithms for reachability analysis and model checking need to keep track of the states that have been explored. In that way, they can detect cyclic behaviour and limit the investigation of successors to truly new states.

There are some techniques that only manage the set of reachable states and utilise similarities between the states. One of them, Binary Decision Diagrams [1, Chapter 5], has been successfully applied mainly in the verification of digital circuits. Techniques applied on the analysis of software systems include a state compaction method for product automata [5] and a method known as Graph Encoded Tuple Sets [6].

One problem with these so called symbolic techniques is that inserting a state may involve global changes, slowing down disk-based implementations. Another problem is that states have no identities: there is no way to retrieve a state from the structure by specifying an index number. Using such a structure for anything else than searching for states fulfilling a predicate or for determining whether a particular state has been explored is tricky.

[*]This research was financed by the National Technology Agency of Finland (TEKES), the Nokia Corporation, the Helsinki Telephone Corporation and the Finnish Rail Administration.

Explicit techniques, which store each state separately, make it possible to navigate in the generated reachability graph and to perform all sorts of queries on it afterwards. When the states are stored separately, they can be assigned index numbers, and it is easy to encode events, the edges of the reachability graph, as triples of two state numbers and a label identifying the action.

This work describes an explicit technique, a method of encoding sequences of multi-sets in a string of binary digits. Symbolic techniques appear promising, but we believe that explicit techniques have an advantage in some applications, such as in the analysis of general software systems, which cannot be characterised by simple laws and which make heavy use of structured data types. Our method, implemented in MARIA [11], has turned out to yield up to an order of magnitude smaller encodings than the method used in PROD [15], although MARIA allows the user to define data types just like in programming languages.

Since our techniques are not specific to any particular class of high-level Petri nets, we try to write in general terms. Even if all data types in MARIA have a finite domain, we shall see that our approach can also handle infinite-domain data types, such as lists.

2 The Reachability Graph

The reachable state space of a model can be represented as a reachability graph, a directed graph whose vertices correspond to reachable states and edges correspond to actions leading from one state to another.

In high-level Petri nets, the states are called *markings* and the actions are called *transition instances*. A transition instance consists of a high-level transition and an assignment for the variables that appear in the arcs and guards connected to the transition.

2.1 Managing the State Space in the File System

Applying explicit analysis techniques to models comprising tens or hundreds of millions of reachable states usually calls for the use of disk storage. Typical reachability analysis algorithms require random access to the set of states explored so far. A similar structure is not required for actions; for most purposes, they can be stored sequentially.

To optimise access to the stored states, one can calculate hash values of the states. When an analysis algorithm wants to determine whether a particular state has been explored, it computes a hash value of the state and searches for it in a memory-based data structure that maps hash values to state numbers. Only if a hash value match is found, the disk address of the encoded state is fetched from a directory file and the state is retrieved from a state file for comparison.

If the encoded states are very small, the memory-based map from hash values to state numbers may exceed the memory limit before the state file exceeds the size limit imposed by the file system. This problem can be addressed by maintaining the map in a disk-based B-tree [13, Ch. 18]. In that case, the system memory consumption remains bounded throughout the analysis, unless some data structures for on-the-fly model checking are kept in the main memory.

The edges of the reachability graph, consisting of source and target state numbers and of an encoded transition instance, are best stored in a separate file. Because the length of the encoded transition instance may vary, also the length is encoded in the file.

2.2 Encoding the Edges and Vertices

2.2.1 Mapping Items to Bit Strings

In order to represent the vertices and edges of the reachability graph as sequences of binary digits, we have to define how the entities they consist of are mapped to such sequences.

Places and Transitions If we denote the set of the places of a Petri net model with $\mathcal{P}$ and assume that there is a bijective mapping

$$o_{\mathcal{P}} : \mathcal{P} \to \{0, \ldots, |\mathcal{P}| - 1\}$$

we can uniquely represent each place with a string of $\lceil \log_2 |\mathcal{P}| \rceil$ binary digits. If $|\mathcal{P}| \leq 1$, no bits are required. The same applies for transitions, characterised by the set $\mathcal{T}$ and the order $o_{\mathcal{T}}$.

Data Items A value of a finite-domain data type $\mathcal{D}$ can be represented as a $\lceil \log_2 |\mathcal{D}| \rceil$-digit binary number, possibly spanning several machine words. This requires a *total order*

$$<_{\mathcal{D}} \subseteq \mathcal{D} \times \mathcal{D}$$

for each data type $\mathcal{D}$. For simple types, such as integers and enumerations, defining the order is straightforward. For structured types, such as tuples, tagged unions and fixed-length or variable-length vectors, the order can be defined lexicographically, e.g. so that variable-length vectors with less elements come first, and that the last component of a structure is the most significant one. This has been implemented in MARIA also for nested structured types.

Once there is a total order among data items, we can define a mapping from the data items to integers:

$$o_{\mathcal{D}} : \mathcal{D} \to \{0, \ldots, |\mathcal{D}| - 1\} : d \mapsto |\{k \in \mathcal{D} \| \, k <_{\mathcal{D}} d\}| \, .$$

It is easy to see that the mapping is bijective and that it preserves the order of the mapped items. Because $<_{\mathcal{D}}$ is a total order, $\mathcal{D}$ can be written as

$$\mathcal{D} = \{d_0, \ldots, d_{n-1}\}$$

such that $d_{i-1} <_{\mathcal{D}} d_i$ for all $0 < i < n$. Now $o_{\mathcal{D}}$ maps each d_i, $0 \leq i < n$, to a unique value:

$$\begin{array}{rcl} o_{\mathcal{D}}(d_i) & = & |\{k \in \mathcal{D} \| \, k <_{\mathcal{D}} d_i\}| \\ & = & |\{d_0, \ldots, d_{i-1}\}| \\ & = & i. \end{array}$$

Since $o_{\mathcal{D}}(d_i) = i$, it holds that $o_{\mathcal{D}}(d_i) < o_{\mathcal{D}}(d_j)$ if and only if $i < j$, or $d_i <_{\mathcal{D}} d_j$. Thus, $o_{\mathcal{D}}$ is an order-preserving mapping.

MARIA allows the domains of data types to be restricted with type constraints, internally represented as an ordered list of closed ranges. Our implementation of $o_{\mathcal{D}}(d)$ for constrained types compares the value d to the endpoints of each range in the constraint and performs subtractions and additions.

Mappings for unconstrained structured values are constructed through multiplication and addition from mapped component values. This is similar to the technique represented in [2], but we manage also deeply structured values and constraints consisting of several disjoint ranges.

Structured types can easily have a bigger number of distinct values than one machine word can represent. Our implementation does not convert values of such types to a single binary number, but it handles them component by component. For example, let there be a variable-length vector type

$$\begin{aligned} \mathcal{D} &:= \bigcup_{i=0}^{k} \mathcal{D}_e^i \\ \mathcal{D}_e^i &:= \underbrace{\mathcal{D}_e \times \cdots \times \mathcal{D}_e}_{i \text{ times}} \end{aligned}$$

with $|\mathcal{D}|$ so big that it does not fit in a machine word. To convert a vector value $\langle d_1, \ldots, d_i \rangle \in \mathcal{D}$ to a sequence of binary digits, our implementation encodes i as a $\lceil \log_2 k \rceil$-bit number and converts each element $d_1, \ldots, d_i$ separately to a bit string. If also the element type $\mathcal{D}_e$ is a large structured type, the elements are handled in a similar way; otherwise, the mapping $o_{\mathcal{D}_e}$ can be applied.

Tagged unions are handled in an analogous way: First, the active component is identified with a binary number. Then the encoded representation of the active component is appended to the bit string. Tuples and fixed-length arrays are simpler, since the number and type of components remain constant.

All data types that can be defined in MARIA have a finite domain. Also the variable-length buffer data type is assigned a capacity, the maximum number of elements a buffer value can contain. If there were any infinite-domain data types,[1] they could be handled in a similar way with large structured types. For instance, an unbounded string or linked list of an item type $\mathcal{D}$ can be represented by encoding each item separately and by using a special value for signalling the end of the sequence. If $|\mathcal{D}|$ can be represented in a machine word, it can be used as the special value. Otherwise, it is easiest to use one extra bit per data item as the end marker.

2.2.2 Encoding Edges

An edge of the reachability graph consists of two numbers identifying the source and target states and of a transition instance consisting of a transition identifier and an assignment for the variables required for firing the transition.

If there is no statistical information available on the transition enablings, the transitions can be assumed to occur with equal probabilities. In that case, our representation of the transitions $t \in \mathcal{T}$ with $\lceil \log_2 |\mathcal{T}| \rceil$-digit binary numbers $o_{\mathcal{T}}(t)$ is close to the optimum defined by the entropy of the system [14, Ch. 6–7].

When the variables of the transition instance are processed in a systematic order, it suffices to encode only the values of the variables and to append them to the bit string representing the label of the edge. Similarly, if the analyser generates all successors of a state in one step, it suffices to store the source state number only once for a bunch of edges originating from the state. Keeping track of the number of states generated so far allows the encoder to use less bits for representing the state numbers.

[1]We restricted ourselves to finite types to avoid difficulties with verification algorithms that operate on unfolded nets.

If the formalism allows some of the variables of an enabled transition to be undefined—that is, if all arc expressions and gates can be evaluated without dereferencing a variable—the encoder must use one bit for signalling whether the variable has been assigned a value.

All this data can be encoded into one sequence of binary digits. When the binary digit string is written to a file, it is good to align it at a byte or machine word boundary.

In some applications, it is not necessary to store the labels of the edges, since they can be reconstructed by analysing all enabled transition instances in the source state, and by finding the instances that lead to the specified target state. This is computationally expensive, but if it only has to be done when displaying to the user a counterexample path of at most a few hundred or thousand steps, the cost of saving tens of megabytes of disk space might be only a few seconds of wasted processor time.

2.2.3 Encoding Vertices

In the case of high-level Petri nets, the vertices of the reachability graph are markings. A marking is a family of multi-sets, indexed by places. A multi-set over a set is a mapping from the items of the set to the set of natural numbers, $\mu : A \to \mathbb{N}$. Unlike normal sets, a multi-set may contain more than one instance of an item. The number of times an item $a \in A$ is contained in a multi-set μ is called the *multiplicity* $\mu(a)$. The union operation of normal sets can be extended to multi-sets as an operation that adds multiplicities.

When the places $p \in \mathcal{P}$ are mapped to numbers $o_{\mathcal{P}}(p)$, the marking can be viewed as a sequence of multi-sets. The multi-set at the position $o_{\mathcal{P}}(p)$ of the sequence corresponds to the local marking of the place p.

A straightforward implementation encodes each multi-set in the sequence separately and appends it to a bit string representing the marking. The details are shown in the following section.

3 Storing Markings

Storing sequences of multi-sets in finite space involves a fundamental problem: the range of a multi-set μ is the infinite set of natural numbers. An implementation in a finite-memory computer must restrict the choice of the multiplicities $\mu(a)$ to a finite set, typically $0 \leq \mu(a) < 2^n$ with $n = 16$ or $n = 32$.

Since the multi-sets in the reachable markings of practical models usually map most items to zero multiplicity, it makes sense to represent each multi-set as a sequence of pairs $\langle \mu(a), a \rangle$ having $\mu(a) > 0$.

An implementation that enforces a limit $0 \leq \mu(a) < 2^n$ could encode the multiplicity of each $\langle \mu(a), a \rangle$ pair in n binary digits and mark the end of the sequence with a string of n zero bits. Such a simple encoding requires $(|\mathcal{P}| + d)\, n$ bits for storing the multiplicities of a marking of a $|\mathcal{P}|$-place net containing d distinct tokens.

3.1 Representing Multiplicities

A multi-set μ over a set A can be characterised by two quantities: the cardinality, or the total number of items

$$t = \sum_{a \in A} \mu(a)$$

and the number of distinct items

$$d = |\{a \in A \| \mu(a) > 0\}| .$$

The cardinality can theoretically be any natural number, but a finite-memory implementation limits it, typically $0 \leq t < 2^n$ for some n.

An user-defined *capacity constraint*, a Boolean condition on t, can reduce the number of bits required for representing t. If there are m different possibilities for the total number of tokens in a place, the actual number t can be represented using $\lceil \log_2 m \rceil$ bits, since a k-digit binary number can represent 2^k different things.

Encoding the cardinality t before the number of distinct items d has one advantage: it is straightforward to see that $1 \leq d \leq t$ when t is nonzero. Therefore, d can be represented using $\lceil \log_2 t \rceil$ bits.

For the greatest multiplicity $\mu_{\max}$ in the multi-set it holds that

$$\left\lceil \frac{t}{d} \right\rceil \leq \mu_{\max} \leq 1 + t - d.$$

If $\mu_{\max}$ is at its upper bound $1 + t - d$, the other $d - 1$ distinct items must have a multiplicity of 1 in order for the total number of items to be t. Similarly, if $\mu_{\max} = \lceil \frac{t}{d} \rceil$, the multiplicities of the remaining items must be equal to $\mu_{\max}$ or $\mu_{\max} - 1$.

So, the greatest multiplicity $\mu_{\max}$ can always be represented with

$$\left\lceil \log_2 \left(2 + t - d - \left\lceil \frac{t}{d} \right\rceil \right) \right\rceil$$

binary digits. After decoding $\mu_{\max}$, the decoder knows the remaining cardinality $t' = t - \mu_{\max}$ and the number of remaining distinct items $d' = d - 1$. If the multiplicities are encoded in descending order, the encoder always selects the greatest of the remaining multiplicities and represents it using less and less bits.

This encoding of multiplicities appears to be quite compact even when capacity constraints are not used. For representing $d = 5$ multiplicities, the simple encoding described in Section 2.2 would use $6n$ bits. The optimised encoding needs n bits for representing the cardinality. Assuming that it is 8, the number of distinct tokens is encoded in 3 bits. The greatest multiplicity lies between $\lceil \frac{8}{5} \rceil = 2$ and $8 - 5 + 1 = 4$; therefore it can be represented with 2 bits. Clearly, the improved encoding requires less than $n + 3 + 5 \cdot 2 = n + 13$ bits. The difference between $6n$ and $n + 13$ is tangible already when $n = 16$.

Our encoding scheme for multiplicities is a variable-length code. In the best case, when $d = 1$ or $d = t$, our code only requires $\lceil \log_2 t \rceil$ bits for representing d—no further bits are required for representing the multiplicities. Figure 1 compares the performance of our code against a fixed-length code that maps multiplicity distributions to a zero-based index numbers. For instance, there are 7 different multiplicity distributions for multi-sets of cardinality 5, if the

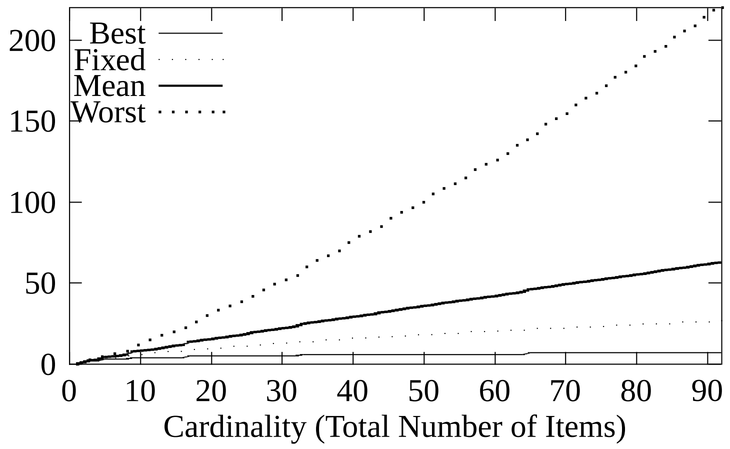

Figure 1: Number of Bits Required for Representing Multiplicities

multiplicities are sorted in descending order: 5, 41, 32, 311, 221, 2111, 11111. Each distribution can be represented by a $\lceil \log_2 7 \rceil$-bit number.

The average bit consumption of our code is slightly more than two times the size required by the fixed-length code. The result is not so bad, since the fixed-length code is computationally much more expensive than our method. Also, the reachable markings in typical Petri net models tend to consist of ordinary sets—an optimal case for our code.

3.2 Representing Empty Multi-Sets

In many practical models, there is a substantial number of empty places in most reachable markings. With our optimised multiplicity encoding, an empty place requires $\lceil \log_2 m \rceil$ bits of storage, if there are m different possibilities for the total number of tokens in the place.

As it is rather uncommon to define tight capacity constraints in models, representing the cardinalities typically requires one machine word per place. If the machine word length is n bits, we would still need $|\mathcal{P}|\, n$ bits for representing an empty marking. There ought to be a more compact encoding for empty places.

Our solution is to start the encoded marking with the number of empty places n_e, $0 \leq n_e \leq |\mathcal{P}|$. This requires $\lceil \log_2(|\mathcal{P}|+1) \rceil$ binary digits. There are

$$\binom{|\mathcal{P}|}{n_e} = \frac{|\mathcal{P}|!}{n_e!\,(|\mathcal{P}|-n_e)!}$$

ways to pick a subset of n_e empty places from the set of all $|\mathcal{P}|$ places. It is possible to enumerate these subsets and to represent each of them as a binary number with

$$\left\lceil \log_2 \binom{|\mathcal{P}|}{n_e} \right\rceil$$

digits. It is easy to see that this code occupies at most $|\mathcal{P}|$ bits, since the total amount of all subsets of the set $\mathcal{P}$

$$\sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|}{n_e}$$

evaluates to exactly $2^{|\mathcal{P}|}$. For $|\mathcal{P}| = 1$ we have $1+1 = 2^1$, and assuming that the claim holds for a set of magnitude $|\mathcal{P}|$, it follows that

$$\begin{aligned}
\sum_{n_e=0}^{|\mathcal{P}|+1} \binom{|\mathcal{P}|+1}{n_e} &= \binom{|\mathcal{P}|+1}{0} + \sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|+1}{n_e+1} \\
&= 1 + \sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|}{n_e} + \sum_{n_e=0}^{|\mathcal{P}|-1} \binom{|\mathcal{P}|}{n_e+1} \\
&= 1 + 2^{|\mathcal{P}|} + (2^{|\mathcal{P}|} - 1) \\
&= 2^{|\mathcal{P}|+1}.
\end{aligned}$$

Instead of constructing this kind of a fixed-length code, we developed and implemented in MARIA a simple variable-length encoding scheme, which we shall present below.

3.2.1 A Variable-Length Code

Clearly, if the number of empty places n_e happens to be 0 or $|\mathcal{P}|$, there is only one way to select the subset, and it can be identified by a zero-length code. In the following, we assume that $0 < n_e < |\mathcal{P}|$.

If $\frac{1}{2}|\mathcal{P}| < n_e < |\mathcal{P}|$—that is, there are more empty places than nonempty ones—then it makes sense to explicitly represent the identity of the nonempty places. The encoding we have defined so far identifies each place $p \in \mathcal{P}$ with an index number $0 \le o_{\mathcal{P}}(p) < |\mathcal{P}|$, and it encodes the multi-sets associated with the places in ascending order of index numbers.

The smallest index number i_1 of a nonempty place must be in the range

$$0 = l_1 \le i_1 \le h_1 = n_e$$

since there are at most n_e empty places in the beginning of the sequence. So, i_1, the index number of the first nonempty place, can be stored using $\lceil \log_2(h_1 - l_1 + 1) \rceil$ binary digits. What about the following nonempty places i_{k+1}? It holds that

$$i_{k+1} \ge l_{k+1} = i_k + 1,$$

since the indices are processed in ascending order. It is easy to see that there are $i_k - (k-1)$ empty places before i_k, since i_k is the kth smallest index of a nonempty place. Thus, of the places following i_k, $n_e - (i_k - (k-1))$ are empty, and for the upper limit $h_{k+1} \ge i_{k+1}$ we have

$$\begin{aligned}
h_{k+1} &= l_{k+1} + n_e - (i_k - (k-1)) \\
&= i_k + 1 + n_e - i_k + k - 1 \\
&= n_e + k.
\end{aligned}$$

Since $h_1 = n_e$, it is easy to see that $h_{k+1} = h_k + 1$.

Similarly, if $0 < n_e \le \frac{1}{2}|\mathcal{P}|$, we represent the indices of empty places. This is analogous to the previous case; we just start with $h_1 = |\mathcal{P}| - n_e$.

This technique is illustrated in Figure 2, which demonstrates a case with 13 places, 6 of which are to be identified. The one with the smallest index i_1 must fulfill the condition $0 \le i_1 \le$

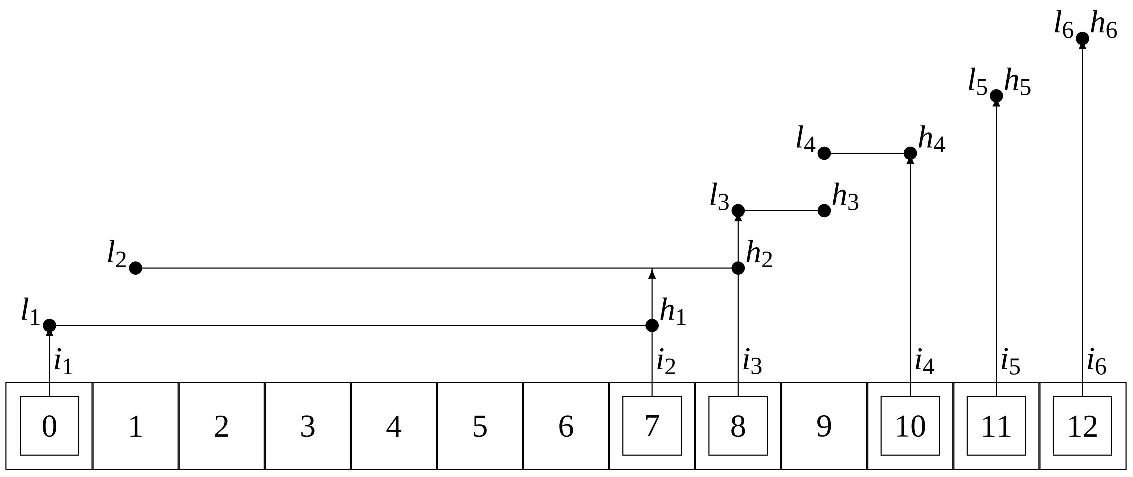

Figure 2: Representing a 6-Element Subset of a Set with 13 Elements

7. A fixed-length code representing i_1 takes 3 bits. Unfortunately for us, i_1 is at the smallest possible position, and the range for the next index is of the same size: $1 \leq i_2 \leq 8$. Since i_2 occurs almost at the end of its range, the uncertainty over the position of the remaining indices reduces. Our approach requires 1 bit for storing i_3 and i_4. After i_4 has been stored, no further bits are required. Our encoding uses a total of 8 bits for identifying the empty places. The fixed-length code would use $\left\lceil \log_2 \binom{13}{6} \right\rceil = 11$ bits for this case.

In the worst case, when all m places to be identified occur in the first m positions, our approach requires the same number of bits for representing each index, a total of $m \lceil \log_2(m+1) \rceil$ bits. In the best case where the first index occurs at the end of its range, the total requirement drops to $\lceil \log_2(m+1) \rceil$ bits.

3.2.2 Keep it Simple

Figure 3 compares the space consumption of our variable-length encoding scheme against the fixed-length code discussed in the beginning of this section. We have seen that the fixed-length code never uses more than $\lceil \log_2(|\mathcal{P}|+1) \rceil + |\mathcal{P}|$ binary digits. Its average bit consumption is

$$\log_2(|\mathcal{P}|+1) + \frac{1}{|\mathcal{P}|+1} \sum_{n_e=1}^{|\mathcal{P}|-1} \left\lceil \log_2 \binom{|\mathcal{P}|}{n_e} \right\rceil .$$

The average space consumption of our variable-length code appears to be more than one bit per place. Even if our implementation made use of fractional bits, the worst case for $|\mathcal{P}| = 20$ would require almost 39 bits, nearly two bits per place. This raises a thought: Why not use exactly one bit per place for marking empty places? The decoder would not even need to know the number of empty places in advance, which allows us to save further $\lceil \log_2(|\mathcal{P}|+1) \rceil$ bits.

This simple code can easily be optimised further for places having a capacity constraint. No signalling bit is required for places that are constrained to be nonempty. Also, if it is possible to represent the cardinality using no more than, say, 2 bits, the emptiness bit can be omitted.

A further optimisation can be made regarding places with no capacity constraints. In practice, places in Petri nets are likely to contain a small number of tokens. Using a shorter repre-

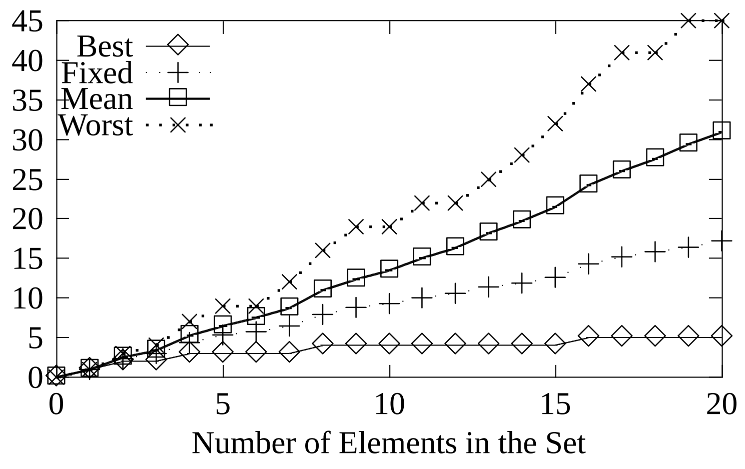

Figure 3: Number of Bits Required for Representing Subsets

sentation for small cardinalities seems to make sense. Above we have suggested a code of at most 1 bit for representing the cardinality $t = 0$. This code can only tell whether $t = 0$ or $t > 0$. In the latter case, more bits are required for encoding the exact value of t. Our implementation in MARIA uses 4 more bits for representing the values $1 \leq t \leq 8$, 10 for $9 \leq t \leq 264$, 19 for $265 \leq t \leq 65800$, and $4 + n$ bits for representing the values $65801 \leq t < 2^n$.

3.3 Redundant Places

Certain commonly applied modelling practices introduce redundancy in the markings of Petri net models. Some of it can be removed by transforming the net to an equivalent one, but not everything. For instance, if there are no inhibitor arcs in the formalism, it is difficult to remove complement places.

All practical models are likely to contain redundant places. The state encoder would perform better if it could somehow omit all redundant places from the encoded marking. The only problem is that there must be a mechanism for computing the contents of redundant places when decoding the marking.

MARIA solves the problem by allowing the initialisation expressions of places to refer to the markings of other places. When a marking is about to be added to the reachability graph, the encoder ensures that there is no controversy in the initialisation expressions of redundant places, and issues an error message if there is. Thus, these user-supplied “invariants” can be viewed as an additional safety check supplied by the analyser, just like capacity constraints and checks in the expression evaluator.

4 An Example

Figure 4 illustrates a high-level Petri net model of a distributed data base management system, originally presented by Genrich, Lautenbach and Jensen [3, 9]. In the initial marking of the model, all places except **exclusion** and **inactive** are empty. The latter place is initialised with

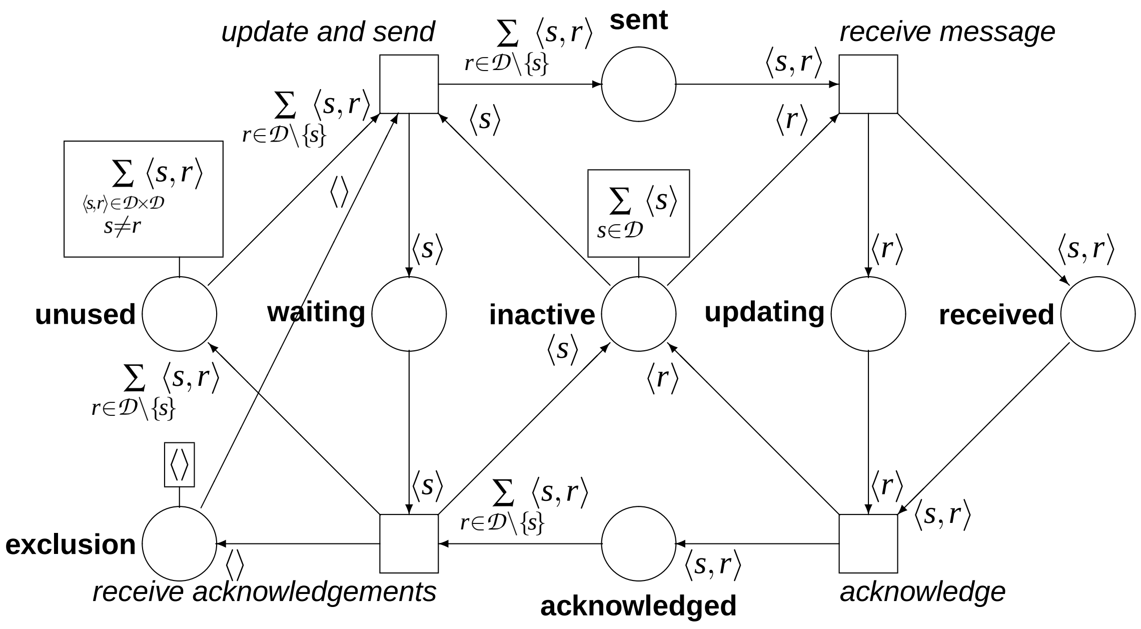

Figure 4: Model of a Distributed Data Base Management System

a multi-set sum of the items in the set $\mathcal{D}$ representing the data base servers. In other words, all data base servers are inactive in the initial state of the model.

It is fairly easy to see that when $\mathcal{D}$ is finite, the reachable state space of the model is finite. The model is also bounded: the places **waiting** and **exclusion** contain at most one token, the place **inactive** contains at most $|\mathcal{D}|$ tokens, and the other places may contain at most $|\mathcal{D}|-1$ tokens.

4.1 Encoding the Initial Marking

The initial marking has 3 nonempty places: **unused**, **inactive** and **exclusion**. These places are actually redundant: The place **unused** and the arcs attached to it could be removed from the model without affecting its behaviour. The place **exclusion** is kind of complementary to the place **waiting**, and **inactive** contains all those items of $\mathcal{D}$ not contained in the places **waiting** and **updating**. Utilising this information, our scheme would encode the initial marking in 5 bits, one for each non-redundant place, signalling that the places are empty.

If our encoding scheme is told nothing about the redundancy, it uses a total of $|\mathcal{P}| = 8$ bits for identifying the three nonempty places.

The cardinality of the multi-set associated with the place **inactive** is $t = d = |\mathcal{D}|$. If there is a capacity constraint $0 \leq t \leq |\mathcal{D}|$, these two quantities can be encoded in $\lceil \log_2(|\mathcal{D}|+1)\rceil + \lceil \log_2 |\mathcal{D}|\rceil$ bits; otherwise, $n_c(|\mathcal{D}|) + \lceil \log_2 |\mathcal{D}|\rceil$ bits are required where n_c tells how many bits our variable-length code for cardinalities takes:

$$n_c(t) = \begin{cases} 4 & \text{if } 1 \leq t \leq 8 \\ 10 & \text{if } 9 \leq t \leq 264 \\ 19 & \text{if } 265 \leq t \leq 65800 \\ 4+n & \text{if } 65801 \leq t < 2^n \end{cases}$$

All items in the multi-set for **inactive** have the multiplicity 1. Although the items in the multi-set represent the whole set $\mathcal{D}$, our encoder represents each value separately, using $|\mathcal{D}|\lceil\log_2|\mathcal{D}|\rceil$ bits.

For the place **exclusion**, we have $t = 1$. This leaves no choice for the number of distinct tokens, which is also 1. Encoding the item takes no bits, since the data type associated with the place has only one value—the empty tuple $\langle\rangle$. If there is a capacity constraint that dictates $0 \le t \le 1$, one bit is enough for encoding the multi-set. Otherwise t is represented with 4 binary digits, since it is in the range $1 \le t \le 8$.

The place **unused** is initially marked with a multi-set of the cardinality $t = |\mathcal{D}|^2 - |\mathcal{D}|$. The reachable markings of the model appear to fulfill the capacity constraint $t = 1 + |\mathcal{D}|^2 - 2|\mathcal{D}| \vee t = |\mathcal{D}|^2 - |\mathcal{D}|$ for this place. This capacity constraint allows t to be represented in one bit. The encoder does not know that the number of distinct items is $d = t$; it assumes $1 \le d \le t$ and therefore represents d as a $\lceil\log_2 t\rceil$-digit binary number. Each item in the multi-set requires $\lceil 2\log_2|\mathcal{D}|\rceil$ bits of storage, assuming that the model uses an unconstrained data type for storing the pairs.[2] Later we shall see that representing this redundant multi-set substantially increases the space requirements. In the following summary, we consider a model where this place has been removed.

To summarise, the initial marking—excluding the place **unused**—fits in $11 + n_c(|\mathcal{D}|) + (|\mathcal{D}|+1)\lceil\log_2|\mathcal{D}|\rceil$ bits when no capacity constraints or redundancy information are exploited. With tight capacity constraints, only $8 + \lceil\log_2(|\mathcal{D}|+1)\rceil + (|\mathcal{D}|+1)\lceil\log_2|\mathcal{D}|\rceil$ bits are required. The difference $3 + n_c(|\mathcal{D}|) - \lceil\log_2(|\mathcal{D}|+1)\rceil$ is always positive in our implementation, since $n_c(t) > \lceil\log_2 t\rceil$. For $|\mathcal{D}| = 10$, utilising the capacity constraints saves 9 bits. When the redundancy information is utilised, the initial marking (where all non-redundant places are empty) can be encoded in only 5 bits, independent of $|\mathcal{D}|$ and n.

4.2 Encoding All Reachable Markings

Our scheme for encoding markings has been implemented in MARIA [11, 12], a reachability analyser for Algebraic System Nets [10] with user-definable finite-domain structured data types. We compare the performance of MARIA with PROD [15], a reachability analyser for a kind of Predicate/Transition Nets [4].

Tables 1 and 2 illustrate the performance of our state encoding scheme. We analysed the model `dbm.net` distributed with PROD without and with unfolding, and three variants of a corresponding model with MARIA: without and with capacity constraints, and with redundant places indicated.

The figures in Table 2 are for models where the redundant place **unused** has been removed. The space consumption drops to less than a fifth when this place is omitted. A natural explanation is that this place has a complementary character: it contains a large number of tokens in all reachable markings. If PROD or MARIA used the initial marking as a reference when encoding other markings, the differences between the two tables would be considerably smaller.

The figures do not include the space required for the graph directory. In PROD, it consists of a fixed header and of a record of 8 machine words—typically 32 bytes—per state. In MARIA, the directory is a table of hash values and file offsets. On a 32-bit system with 32-bit file offsets,

[2]This number would drop to $\lceil\log_2(|\mathcal{D}|^2 - |\mathcal{D}|)\rceil$ if we defined the domain of the place to be a multi-set over $(\mathcal{D}\times\mathcal{D}) \setminus \bigcup_{s\in\mathcal{D}}\{\langle s,s\rangle\}$ instead of $\mathcal{D}\times\mathcal{D}$.

Table 1: Reachability Graph Sizes for the Distributed Data Base Model

Model Size		Encoded State Space in Bytes				
$\lvert\mathcal{D}\rvert$	States	PROD	(unfolded)	MARIA	(cap.)	(red.)
1	2	19	3	4	2	2
2	7	99	29	28	15	9
3	28	645	844	223	168	86
4	109	3,925	1,745	1,403	1,090	414
5	406	21,519	10,151	8,439	7,487	2,396
6	1,459	107,967	54,307	46,109	42,292	10,807
7	5,104	505,297	280,229	208,807	198,599	43,569
8	17,497	2,239,617	1,296,227	908,810	873,816	168,089
9	59,050	9,507,051	5,605,363	4,423,852	4,308,020	738,397
10	196,831	38,972,539	23,134,187	18,006,540	17,685,747	2,683,381

Table 2: Reachability Graph Sizes for the Model Excluding the Place **unused**

Model Size		Encoded State Space in Bytes				
$\lvert\mathcal{D}\rvert$	States	PROD	(unfolded)	MARIA	(cap.)	(red.)
1	2	17	3	4	2	2
2	7	76	21	21	14	9
3	28	389	139	150	111	86
4	109	1,848	761	724	543	414
5	406	8,113	3,651	3,565	3,159	2,396
6	1,459	33,548	16,045	15,725	14,266	10,807
7	5,104	132,693	76,067	60,786	55,683	43,569
8	17,497	507,400	357,219	233,702	217,213	168,089
9	59,050	1,889,585	1,511,227	998,945	952,558	738,397
10	196,831	6,889,068	6,009,887	3,559,389	3,526,147	2,683,381

the bookkeeping overhead is 8 bytes per encoded state.

When there is no capacity constraint, our implementation represents the cardinality of a multi-set using a variable-length code, which occupies $1+4$ bits more than one machine word in the worst case. The other extreme is a capacity constraint that allows only one value for the total number of tokens. Defining a capacity constraint can thus save more than one machine word for each non-empty place in the marking.

We also translated a variant of the ISDN-DSS1 protocol model [8] from PROD to MARIA format. The encoded representation of the 20,084 states takes 37.2 bytes per state in PROD (38.3 for an unfolded model) and 9 in MARIA, or 13.7 if no capacity constraints are defined.

The run-time overhead of our encoding method is negligible, as our implementation makes heavy use of automatically generated, dynamically linked C code. When analysing the above mentioned ISDN-DSS1 model, the analyzer spends less than 4 percent of its total time in the encoder. This can partially be explained by the relatively large number of places and transitions in the model, which shifts the bottleneck to transition instance analysis. When analysing different variations of the data base model, we experienced that encoding states takes 9–22 percent of the total time. The worst figure was obtained for a model that included the redundant place **unused** and supplied a marking-dependent initialisation expression for it.

5 Conclusion and Future Work

We have presented a scheme for condensed explicit storage of markings of high-level Petri nets, representing the multiplicities of multi-set items in a compact way. Even though our scheme does not utilise any similarities between or inside markings in any way, an implementation of it performs up to an order of magnitude better than a previously implemented scheme even for simple models.

Our idea of using $\lceil \log_2 |\mathcal{D}| \rceil$ binary digits for representing multi-set items belonging to a finite set $\mathcal{D}$ assumes that each item occurs with equal probability. This is often not the case with practical models, and it would be worth investigating how well Holzmann's ideas on recursive indexing and compression training runs [7] could be combined with our approach.

References

[1] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[2] Jaco Geldenhuys and Pieter de Villiers. Runtime efficient state compaction in SPIN. In Dennis Dams, Rob Gerth, Stefan Leue and Mieke Massink, editors, *Theoretical Aspects of Model Checking, 5th and 6th International SPIN Workshops*, pages 12–21, Trento, Italy, July 1999. Springer-Verlag, Berlin, Germany, 1999.

[3] Hartmann J. Genrich and Kurt Lautenbach. The analysis of distributed systems by means of Predicate/Transition-Nets. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146, Evian, France, July 1979. Springer-Verlag, Berlin, Germany, 1979.

[4] Hartmann J. Genrich. Predicate/Transition Nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and their Properties—Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247, Bad Honnef, Germany, September 1986. Springer-Verlag, Berlin, Germany, 1987.

[5] Patrice Godefroid and Gerard J. Holzmann. On the verification of temporal properties. In *13th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 109–124, Liège, Belgium, May 1993.

[6] Jean-Charles Grégoire. State space compression in SPIN with GETSs. In *2nd International SPIN Verification Workshop*, New Brunswick, NJ, USA, August 1996.

[7] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *3rd International SPIN Verification Workshop*, Enschede, The Netherlands, April 1997.

[8] Nisse Husberg, Teemu Tynjälä and Kimmo Varpaaniemi. Modelling and analysing the SDL description of the ISDN-DSS1 protocol. To appear in *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN'00*, Århus, Denmark, June 2000.

[9] Kurt Jensen. Coloured Petri Nets and the invariant method. *Theoretical Computer Science*, 14(3):317–336, June 1981.

[10] Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–364, Lisbon, Portugal, June 1998. Springer-Verlag, Berlin, Germany.

[11] Marko Mäkelä. *Maria: Modular Reachability Analyzer for Algebraic System Nets*. Online documentation, `http://www.tcs.hut.fi/maria/`.

[12] Marko Mäkelä. *A Reachability Analyser for Algebraic System Nets*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, March 2000.

[13] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, USA, 1984.

[14] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, IL, USA, 1949.

[15] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995.