Optimising Enabling Tests and Unfoldings of Algebraic System Nets

Marko Mäkelä*

Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 9700, 02015 HUT, Finland
`msmakela@tcs.hut.fi`
`http://www.tcs.hut.fi/Personnel/marko.html`

Abstract. Reachability analysis and simulation tools for high-level nets spend a significant amount of the computing time in performing enabling tests, determining the assignments under which transitions are enabled. Unlike the majority of earlier work on computing enabled transition bindings, the techniques presented in this paper are highly independent of the algebraic operations supported by the high-level net formalism.
Performing enabling tests is viewed as a unification problem. A unification algorithm is presented and modifications to it are suggested. One variant of the algorithm constructs finite unfoldings for nets with unbounded domains. Some heuristics for optimising the enabling tests are discussed and their usefulness is evaluated based on experiments. The algorithms have been implemented in the reachability analyser Maria.

Keywords: high-level Petri nets, reachability analysis, unification, unfolding

1 Introduction

Constructing computer-readable models for systems resembles programming in many aspects. High-level languages make it easier to create models or programs, but analysing or executing them involves an overhead, since the operations in the high-level specification have to be transformed to simpler operations that the underlying computing machinery is able to perform. This can be done either in one big preprocessing step that translates the whole input to a simpler language, or in smaller steps that interpret the high-level operations one at a time, or by performing a mixture of preprocessing and interpreting.

There are several approaches to analysing high-level Petri nets. *Structural techniques*, such as determining invariants of a high-level net [10] and proving some properties based on them, are typically applied by humans and therefore

* This research was financed by the Helsinki Graduate School on Computer Science and Engineering, the National Technology Agency of Finland (TEKES), the Nokia Corporation, Elisa Communications and the Finnish Rail Administration.

only work on relatively small, highly abstracted models. Many computer-aided techniques are based on *exhaustive state space exploration* or *reachability analysis*, generating all states reachable from the initial state of the model.

Some reachability analyser tools work on low-level nets [15]. Such tools can analyse high-level nets if these are *unfolded*, translated to low-level nets in a preprocessing step. A straightforward unfolding, as the one defined for Algebraic system nets in [10, Section 5.1], may yield places not connected to any transition, or transitions whose input places will never become marked.

The unfolded net of a high-level net can be reduced by analysing the high-level net and overestimating the set of reachable markings. The unfolded net needs to contain only such places that can ever become marked according to the estimate. Similarly, only those transitions that are connected to these places need to be included in the unfolded net. Even when such reductions are applied, a high-level model whose variables have large domains may yield an unmanageably large unfolding, even if the full state space of the model is moderate.

Reachability analysis can also be performed on the high level. This is computationally more complex than analysing low-level nets, since the transitions may fire in different *modes*, depending on the values assigned to their variables. Compared to unfolding, analysing models on the high level usually trades execution time for memory space. When a net is unfolded, its high-level transitions are processed only once. When it is analysed on the high level, the transitions must be "unfolded", or interpreted in each state that is explored.

Our approach to the reachability analysis of high-level nets is a mixture of preprocessing and interpreting. We perform a series of translations on the model and set up auxiliary data structures that make it possible to use a simpler and more efficient algorithm for performing enabling tests. The idea is to find efficient static schedulings for input arc inscriptions and to apply computationally cheap heuristics for pruning transitions that are disabled in a marking.

The notations in this paper is based on Algebraic system nets, defined by Kindler and Völzer in [10]. They can be considered as a slightly more formal version of coloured Petri nets [9]. The class of nets we consider is more generic than the well-formed nets used by Chiola et al. [2], Gaeta [4] and Ilié et al. [7,8] and others at least in the following aspects:

– data types are not limited to enumerations and tuples
– algebraic operations may be irreversible
– arcs may have variable-dependent weights or be multiset-valued

The coloured nets used by Sanders [16] are more generic than well-formed nets but less generic than Algebraic system nets. His approach represents input arc expressions as variables with constant multiplicity. Since Sanders performs enabling tests by solving constraint satisfaction problems, it is nontrivial to allow the arcs in his formalism to have variable-dependent weights.

The formalism supported by our tool Maria is Algebraic system nets with some extensions and limitations. The main limitations are that variables on input arcs may not be multiset-valued, and all data types must have finite domains. These limitations ensure that every model in our formalism can be unfolded to a finite low-level net.

1.1 Example: Changing Money

Figure 1 illustrates a situation that could happen near a coin-operated machine. A customer comes to a cashier with a bank bill in his hand, asking "Could you break this for me?" The cashier then changes the money to an equivalent amount of money in smaller coins. In our algorithm, he always returns one type of coin, e.g. ten ① coins for 10 units of money, and not e.g. one ⑤ coin and five ① coins.

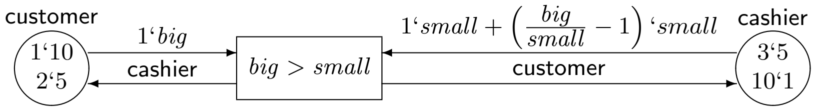

Fig. 1. An algorithm for breaking money.

The model contains two places, customer and cashier, which represent the money held by the two parties. The only transition of the model has two input variables, *big* and *small*, the monetary values. A transition guard specifies that the monetary value of *big* must be greater than that of the change coins *small*.

When the cashier receives a piece of money from the customer, he first chooses one of his coins and then picks enough of them so that the monetary values match. The output arcs of the transitions make use of special multiset-valued variables, which are short-hand notation of our tool for more complex arc inscriptions. These variables refer to the multisets that the input arcs connected to the corresponding places evaluate to. Thus, the cashier receives the coins the customer took from his purse and vice versa.

As we shall see later, the definition of Algebraic system nets allows arbitrary multiset-valued arc inscriptions. We could replace the complex inscription of the arc running from the place cashier to the transition in Figure 1 with a reference to a multiset-valued variable *change*, and replace the transition guard with $big = \sum_m change(m)m$, requiring that no money is made or lost. Alas, this kind of a definition could introduce a combinatorial explosion in the analysis. On input arcs, our approach does not allow multiset-valued *variables*, but it does support more complex multiset-valued terms, provided that they can be evaluated based on variable bindings obtained from other arcs.

2 Basic Concepts

Before presenting our algorithm, we must define some basic concepts. We refer the reader to [10, Section 3.1–3.2] for a more detailed introduction to Algebraic system nets and the underlying mathematical concepts. Our definition of algebras has some extensions to the original. *Evaluation errors* are helpful in tracking modelling errors. Our tool does not silently ignore transitions that cannot be fired due to errors such as arithmetic overflow. Models with variable-weight arcs may benefit from *undefined variables*. If a variable occurs only on arcs whose multiplicity evaluates to zero, it does not need to be defined in order for the transition to be fired. Space limitations prohibit us from formally defining another extension, *short-circuit evaluation* of if-then-else expressions.

Algebras and signatures. A signature $SIG = \langle S, OP \rangle$ consists of a finite set S of *sort symbols* and a pairwise disjoint family $OP = (OP_a)_{a \in S^+}$ of *operation symbols.* A *SIG-algebra* $\mathcal{A} = \langle A, f \rangle$ consists of a family $A = (A_s)_{s \in S}$ of sets and a family $f = (f_{op})_{op \in OP}$ of total functions. Let $\epsilon \notin A$ be an *error symbol* and $A'_s = A_s \cup \{\epsilon\}$. For $op \in OP_{s_1 \ldots s_n s_n+1}$, let $f_{op} : A'_{s_1} \times \cdots \times A'_{s_n} \to A'_{s_{n+1}}$ such that the image of the subset $(A'_{s_1} \times \cdots \times A'_{s_n}) \setminus (A_{s_1} \times \cdots \times A_{s_n})$ equals ϵ; that is, whenever an argument equals ϵ, so does the result. A set A_s of an algebra is called a *domain* and a function f_{op} is called an *operation* of the algebra.

In the following we assume that a signature SIG has the sort symbols $bool, nat \in S$ and in each SIG-algebra the corresponding domains are $A_{bool} = \mathbb{B} = \{\text{true}, \text{false}\}$ and $A_{nat} = \mathbb{N} = \{0, 1, \ldots\}$.

Variables and terms. For a signature $SIG = \langle S, OP \rangle$ we call a pairwise disjoint family $X = (X_s)_{s \in S}$ with $X \cap OP = \emptyset$ a *sorted SIG-variable set.* A *term*, associated with a particular sort, is built up from variables and operation symbols. The *set of SIG-terms over X of sort s* is denoted by $\mathbf{T}_s^{SIG}(X)$ and inductively defined by:

1. If $x \in X_s$, then $x \in \mathbf{T}_s^{SIG}(X)$.
2. If $u_k \in \mathbf{T}_{s_k}^{SIG}(X)$ for some $k \in \{1, \ldots, n\}$ and $op \in OP_{s_1 \ldots s_n s_{n+1}}$, then $op(u_1, \ldots, u_n) \in \mathbf{T}_{s_{n+1}}^{SIG}(X)$.

The set of all terms (of any sort) is denoted by $\mathbf{T}^{SIG}(X)$. A term without variables, a *ground term*, of sort s belongs to the set $\mathbf{T}_s^{SIG} = \mathbf{T}_s^{SIG}(\emptyset)$.

Evaluation of terms. For a signature $SIG = \langle S, OP \rangle$, a sorted SIG-variable set $X = (X_s)_{s \in S}$, and a SIG-algebra $\mathcal{A} = \langle (A_s)_{s \in S}, (f_{op})_{op \in OP} \rangle$, a mapping $\beta : X \to A \cup \{\epsilon\}$ is an *assignment* for X iff for each $s \in S$ and $x \in X_s$ holds $\beta(x) \in A_s \cup \{\epsilon\}$ where $\epsilon \notin A$ denotes an *undefined variable.* We canonically extend β to a mapping $\bar{\beta} : \mathbf{T}^{SIG}(X) \to A \cup \{\epsilon\}$ by:

1. $\bar{\beta}(x) = \beta(x)$ for $x \in X$.
2. $\bar{\beta}(op(u_1, \ldots, u_n)) = f_{op}(\bar{\beta}(u_1), \ldots, \bar{\beta}(u_n))$ for $op(u_1, \ldots, u_n) \in \mathbf{T}^{SIG}(X)$.

Let $\beta_\emptyset : \emptyset \to A \cup \{\epsilon\}$ be the unique assignment for the empty variable set.

2.1 Algebraic System Nets

Algebraic system nets are based on a special case of the algebras defined above. We distinguish some *ground-sorts* and assign a *bag-sort* (a finite nonnegative multiset sort) to each ground-sort. The domain associated with a bag-sort must be a multiset over the domain of the corresponding ground-sort.

Definition 1 (Bag-signature, BSIG-algebra). *Let $\mathit{SIG} = \langle S, \mathit{OP}\rangle$ be a signature and $\mathit{BS}, \mathit{GS} \subseteq S$. $\mathit{BSIG} = \langle S, \mathit{OP}, \mathit{bs}\rangle$ is a* bag-signature *iff $\mathit{bs} : \mathit{GS} \to \mathit{BS}$ is a bijective mapping. An element of GS is called a* ground-sort*, an element of BS is called a* bag-sort *of BSIG. A SIG-algebra $\mathcal{A} = \langle A, f\rangle$ is a BSIG-*algebra *iff for each $s \in \mathit{GS}$ holds $A_{\mathit{bs}(s)} = \mathrm{BAG}(A_s) = (A_s \to \mathbb{N})$.*

Definition 2 (Algebraic system net). *Let $\mathit{BSIG} = \langle S, \mathit{OP}, \mathit{bs}\rangle$ be a bag-signature with bag-sorts BS. An* algebraic system net *$\Sigma = \langle N, \mathcal{A}, X, i\rangle$ over BSIG consists of*

1. *a finite net $N = \langle P, T, F\rangle$ where $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ and P is sorted over BS, i.e., $P = (P_s)_{s\in \mathit{BS}}$ is a bag-valued BSIG-variable set,*
2. *a BSIG-Algebra $\mathcal{A}$,*
3. *a sorted BSIG-variable set X disjoint from P,*
4. *a net inscription $i : P \cup T \cup F \to \mathbf{T}^{\mathit{BSIG}}(X)$ such that*
   a) *for each $p \in P_s : i(p) \in \mathbf{T}_s^{\mathit{BSIG}}$,*
   b) *for each $t \in T : i(t) \in \mathbf{T}_{\mathit{bool}}^{\mathit{BSIG}}(X)$, and*
   c) *for each $t \in T$ and $p \in P_s$ and $f \in F$ with $f = \langle p, t\rangle$* (input arc) *or $f = \langle t, p\rangle$* (output arc) *holds $i(f) \in \mathbf{T}_s^{\mathit{BSIG}}(X)$.*

For a place $p \in P$, the inscription $i(p)$ is called the symbolic initial marking *of p; for a transition $t \in T$, the term $i(t)$ is called the* guard *of t.*

It is worth noting that Definition 2, replicated from [10, Definition 3] allows multiset-valued operations and variables in arc inscriptions (annotations). The Maria tool [11,12] makes use of both.[1] Arcs with variable weights are useful when modelling certain types of resource management.

The basic semantics of Algebraic system nets, including the firing rule, have been defined by Kindler and Völzer in [10, Definitions 4–6].

2.2 Unification Concepts

There are at least two ways to construct the set of assignments under which a transition is enabled. One way is to construct all possible assignments for the variables that occur in the arc inscriptions and in the guard of the transition, and to prune those assignments under which the arc inscriptions and the guard fail to fulfil the firing rule. This is the usual way when a net is unfolded; see e.g. [10, Definition 13]. This approach does not work very well if the transitions have a large (or infinite) number of possible assignments (firing modes), and the transitions are enabled in only a few firing modes in the reachable states.

Fortunately, there is a more efficient approach for the case when the input places of a transition are marked sparsely. The process of finding assignments or substitutions under which two algebraic terms are equivalent is often referred to as *unification*, e.g. [1, pp. 74–76]. In algebraic system nets, we can unify input arc

[1] Maria allows multiset-valued variables on output arcs, where they refer to the multisets removed from the input places; see Figure 1.

inscriptions with a marking of the net. In this case, a unifier is an assignment for the transition variables under which the evaluations of the input arc inscriptions are contained in the corresponding input place markings.

If the algebraic operations are not restricted, there might be prohibitively many unifiers. For instance, consider the constant $2 \in \mathbb{N}$ and the expression $x + y$. If the variables x and y are known to be sorted over *nat*, then three assignments are possible unifiers: $\{\langle x, 0\rangle, \langle y, 2\rangle\}$, $\{\langle x, 1\rangle, \langle y, 1\rangle\}$, and $\{\langle x, 2\rangle, \langle y, 0\rangle\}$. If the constant was n, there would be $n + 1$ different unifiers. If either variable was allowed to be negative, there would be infinitely many unifiers.

In order to avoid a combinatorial explosion, we have to restrict the set of algebraic terms that the unification algorithm examines to find values for variables. A natural way of making this restriction is to limit the set of operations the unification algorithm recognises in such a way that the choice of unifiers is always unique. This rules out the operation + in our previous example.

We distinguish two classes of operations that are recognised by our algorithm. Reversible unary operations, such as taking the successor of an element in a sequence, can be "neutralised" by applying a reverse operation, such as the predecessor operator. Other operations that the algorithm must know are constructors that tie terms together. For instance, we want to be able to unify the variables in the term $\langle x, y\rangle$ with the constants in the ground term $\langle 1, 2\rangle$.

Definition 3 (Unifier candidate, assignment compatibility). *Let $SIG = \langle S, OP\rangle$ be a signature with the variable set X, and let $\mathcal{A} = \langle A, f\rangle$ be a SIG-algebra with the error symbol $\epsilon \notin A$. Let $OP_c \subseteq OP$ be the* set of constructor operations*, and let $rop \subseteq (OP \to OP)$ be the* map of reversible unary operations *such that*

$$\forall op \in \mathbf{dom}\, rop : \exists s, s' \in S : op \in OP_{s\,s'} : \forall a \in A_s : f_{rop(op)}(f_{op}(a)) = a.$$

Furthermore, let $s, s' \in S$, $x \in X_s$ and $T \in \mathbf{T}^{SIG}_{s'}(X)$. The variable x is said to be unifiable from T*, denoted $x \lhd T$, if*

1. *$T = x$, or*
2. *for some $op \in OP_c$ and $k \in \{1, \ldots, n\}$, $T = op(T_1, \ldots, T_n)$ and $x \lhd T_k$, or*
3. *for some $op \in \mathbf{dom}\, rop$, $T = op(T')$ and $x \lhd T'$.*

Let $T_\emptyset \in \mathbf{T}^{SIG}_{s'}$ and $x \lhd T$. A unifier candidate *$x \lhd_{T_\emptyset} T$ is inductively defined as follows:*

1. *$T_\emptyset$, if $T = x$*
2. *$x \lhd_{T_{k\emptyset}} T_k$, if for some $op \in OP_c$, $T = op(T_1, \ldots, T_n)$, $T_\emptyset = op(T_{1\emptyset}, \ldots, T_{n\emptyset})$, $k \in \{1, \ldots, n\}$ and $x \lhd T_k$, and there is no $1 \le j < k$ such that $x \lhd T_j$,[2] or*
3. *$x \lhd_{T'_\emptyset} T'$, if for some $op \in \mathbf{dom}\, rop$, $T = op(T')$ and $x \lhd T'$ for $T'_\emptyset = rop(op)(T_\emptyset)$.*

Let $\beta : X \to A$. The terms T and $T_\emptyset$ are compatible under β*, denoted $T \sim_\beta T_\emptyset$, if either*

[2] Requiring the smallest k to be chosen ensures that unifier candidates are unique.

1. *$\bar{\beta}(T) = \bar{\beta}_\emptyset(T_\emptyset)$, or*
2. *for some $op \in OP_c$, $T = op(T_1, \ldots, T_n)$ and $T_\emptyset = op(T_{1\emptyset}, \ldots, T_{n\emptyset})$, and $T_k \sim_\beta T_{k\emptyset}$ for $k \in \{1, \ldots, n\}$, or*
3. *for some variable x in T, $\beta(x) = \epsilon$.*

Continuing our example, and assuming that the operation $+$ is a constructor, $+ \in OP_c$, the definition yields no unifier candidates for x and y, if T is $x + y$ and $T_\emptyset$ is 2. If $T_\emptyset$ was $1 + 2$, then we would have $x \lhd_{T_\emptyset} T$ equal to 1 and $y \lhd_{T_\emptyset} T$ equal to 2. The terms are compatible under the assignment $\beta = \{\langle x, 1\rangle, \langle y, 2\rangle\}$ constructed from these candidates, since $x + y \sim_\beta 1 + 2$.

Restrictions of Unification. An analyser implementation can considerably restrict the set of operations supported by unification and the set of reversible operations. MARIA only looks for variables inside so-called constructor terms which construct values of structured data types out of components. From the constructor term $\langle x, \langle y + 1, z\rangle\rangle$, it could find unifier candidates for x and z, but not for y, since $y \not\lhd y + 1$. It also performs *constant folding* by replacing ground terms with equivalent nullary operators. It would transform the $T_\emptyset = 1 + 2$ in our above example to $\bar{\beta}_\emptyset(T_\emptyset) = 3$.

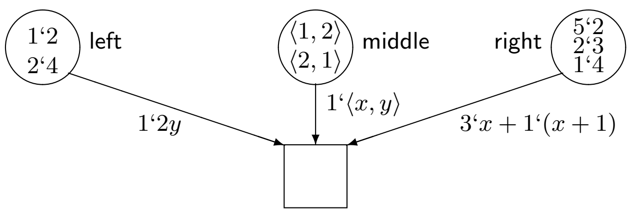

Fig. 2. A simple model for illustrating unification.

The operation collections OP_c and **dom** *rop* in Definition 3 strongly affect the set of unifiable variables. The more operations are contained in these sets, the more unifier candidates are possible. Consider the model shown in Figure 2. If multiplication by a constant belongs to the set of reversible operations, it is possible to unify y from the term $2y$ by unifying y with the given ground term (2 or 4) divided by 2.

We have not presented any algorithms yet, but we are about to face a somewhat philosophical question. Should a unification algorithm be able to find all possible assignments that enable the transition, or does it suffice for the algorithm to deal with real models, and report errors for cases it cannot handle? An implementation that restricts the sets of supported operations is likely to be more efficient and less prone to errors than one that tries to handle everything. For instance, when making basic arithmetic operations reversible, one must take care of arithmetic precision and exceptional situations.

Variables that are not unifiable by Definition 3 could be handled by nondeterministically picking values for them from their domains and by checking the terms for compatibility, but doing so is computationally expensive if the domains are large or there are many such variables. It is easier to report "variables cannot be unified" even though it might be possible to unify them. According to our experience with practical models, this works pretty well. One can always gain expressive power by replacing problematic terms with new variables and guards.

Splitting the Arcs. In typical models, arc expressions consist of elementary multisets (single-item multiset constructor terms) combined with multiset summation. In order to improve the granularity of our algorithm, we write each input arc inscription as a such combination of terms. Sanders refers to this as *arc unfolding* [16, Section 3]. For instance, the rightmost arc of the model illustrated in Figure 2 is split into two arcs, with the inscriptions $3`x$ and $1`(x+1)$.

The claim "any arc with a non-elementary multi-set may be 'unfolded' into multiple arcs" by Sanders [16, Section 3] is difficult to fulfil if the arcs contain multiset-valued variables or other multiset operations than the two we defined above. Our approach does not restrict the set of multiset operations.

We distinguish three kinds of split arc inscriptions: ones that contain unifiable variables, ones that can be evaluated under a partial assignment incrementally constructed by our algorithm, and others. In Figure 2, the arcs $1`\langle x, y\rangle$ and $3`x$ contain variables that our implementation can unify. Other inscriptions can be arbitrary multiset-valued terms. What matters is that whenever the unification algorithm finds a complete assignment, all arc inscriptions are compatible under it with the ground terms corresponding to the given marking of the model.

3 The Unification Algorithm

Our unification algorithm performs a depth-first search on the input arc inscriptions of the transition, split as described earlier. The algorithm is remarkably simple, since it processes the arcs in a fixed order produced in static analysis. Static analysis also determines which variables will be unified from which arc inscriptions, and verifies that all variables can be unified.[3]

The input arc inscriptions are split into items $S_k = \langle T_k, X_k, p_k, m_k\rangle \in \mathbf{T}^{BSIG}(X) \times \mathcal{P}(X) \times P \times \mathrm{BAG}(A_s)$, $k \in \{1, \dots, n\}$ for some n, such that

- the variable sets X_k are pairwise disjoint: $X_j \cap X_k = \emptyset$ if $j \neq k$
- no T_k refers to a variable outside $\bigcup_{j=1}^{n} X_j$: $T_k \in \mathbf{T}^{BSIG}(\bigcup_{j=1}^{n} X_j)$
- if $X_k = \emptyset$: $T_k \in \mathbf{T}^{BSIG}(\bigcup_{j=1}^{k-1} X_j)$
- if $X_k \neq \emptyset$: $T_k = T'_k`T''_k$ and $T'_k \in \mathbf{T}^{BSIG}(\bigcup_{j=1}^{k-1} X_j)$ and $\forall x \in X_k : x \lhd T''_k$
- the input arcs of the transition and their inscriptions can be constructed from all places p_k and split inscriptions T_k via multiset summation

The last component, m_k, is a place-holder for the multiset the term is supposed to evaluate to. Our algorithm does not refer to it before initialising it; for convenience, we can assign it to the empty multiset here.

[3] A variable unified from a variable-multiplicity arc may remain undefined if the multiplicity of the arc evaluates to zero.

The input arcs of the only transition in Figure 2 can be split e.g. so that

$$\begin{aligned}
S_1 &= \langle T_1, X_1, p_1, m_1 \rangle = \langle 3`x, \{x\}, \mathsf{right}, \emptyset \rangle \\
S_2 &= \langle T_2, X_2, p_2, m_2 \rangle = \langle 1`\langle x, y \rangle, \{y\}, \mathsf{middle}, \emptyset \rangle \\
S_3 &= \langle T_3, X_3, p_3, m_3 \rangle = \langle 1`(x+1), \emptyset, \mathsf{right}, \emptyset \rangle \\
S_4 &= \langle T_4, X_4, p_4, m_4 \rangle = \langle 1`2y, \emptyset, \mathsf{left}, \emptyset \rangle .
\end{aligned}$$

The first components of the tuples are the split arc inscriptions. The second components are the "new" unifiable variables. Let us observe S_2 a bit more closely. We have $X_2 = \{y\}$, although also x could be unified: $x \triangleleft \langle x, y \rangle$. Including x in X_2 would violate the disjointness property, since $x \in X_1$. In a sense, the variable x will "already" be unified from S_1. Also, the terms T_3 and T_4 are "constant" since their variables can be unified from the earlier arcs S_1 and S_2.

3.1 The Basic Algorithm

```
Analyse arcs $S_1..S_n$ w.r.t. marking $M$
ANALYSE($S, n, M$):
$\beta \leftarrow (\bigcup_{k=1}^{n} X_k) \times \{\epsilon\}$
ANALYSE-ARCS($S, 1, n, M, \beta$)

Analyse arcs $S_k..S_n$ w.r.t. $M$ and $\beta$
ANALYSE-ARCS($S, k, n, M, \beta$):
if $k = n$ then print $\beta$
else $\triangleright$ $S_k = \langle T_k, X_k, p_k, m_k \rangle$
  if $X_k = \emptyset$ then
    ANALYSE-CONSTANT($S, k, n, M, \beta$)
  else
    ANALYSE-VARIABLE($S, k, n, M, \beta$)

Evaluate arc $S_k$
ANALYSE-CONSTANT($S, k, n, M, \beta$):
$\triangleright$ $S_k = \langle T_k, X_k, p_k, m_k \rangle$
$m_k \leftarrow \bar{\beta}(T_k)$
if $m_k = \epsilon$ then
  print "undefined arc", $\beta, T_k$
else
  if $M(p_k) \geq m_k$ then
    $M' \leftarrow M$
    $M'(p_k) \leftarrow M(p_k) - m_k$
    ANALYSE-ARCS($S, k+1, n, M, \beta$)

Analyse arc $S_k$, augment $\beta$
ANALYSE-VARIABLE($S, k, n, M, \beta$):
$\triangleright$ $S_k = \langle T_k, X_k, p_k, m_k \rangle$
$\triangleright$ $T_k = T'_k`T''_k$
$c \leftarrow \bar{\beta}(T'_k)$
if $c = \epsilon$ then
  print "undefined multiplicity", $\beta, T_k$
  return
if $c = 0$ then
  $m_k \leftarrow \emptyset$
  ANALYSE-ARCS($S, k+1, n, M, \beta$)
else
  for each $m : M(p_k) \geq c`m$ do
    $m_k \leftarrow c`m$
    $\beta' \leftarrow \beta$
    for each $x \in X_k$ do
      $\beta'(x) \leftarrow (x \triangleleft_m T''_k)$
    if $\bigwedge_{j=1}^{k} T_j \sim_{\beta'} m_j$ then
      $M' \leftarrow M$
      $M'(p_k) \leftarrow M(p_k) - m_k$
      ANALYSE-ARCS($S, k+1, n, M', \beta'$)
```

Fig. 3. The unification algorithm.

Our unification algorithm is presented in Figure 3. The computation is initiated by invoking ANALYSE with the split input arc inscriptions S and their amount n and a marking $M : P \to \mathrm{BAG}(A)$ of the net. The computation step of the depth-first search is divided into two alternatives: processing a "constant" arc (arc with no new bindable variables), and obtaining new variable bindings from an arc.

An Example. Continuing our running example from Figure 2, the call to ANALYSE on the initial marking of the model proceeds as follows. The assignment is initialised to $\beta = \{\langle x, \epsilon\rangle, \langle y, \epsilon\rangle\}$, and control is passed to ANALYSE-ARCS and further to ANALYSE-VARIABLE. The multiplicity of $T_1 = 3\text{`}x$ evaluates to $c = 3$. Now ANALYSE-VARIABLE loops over all items in the marking of right whose multiplicity is at least 3. It turns out that $m = 2$ is the only choice.

A new assignment with $\beta'(x) = 2$ is computed. Since all terms unified so far are compatible under this assignment, the multiset m_k is reserved from the marking and the control is transferred to ANALYSE-ARCS, which passes it again to ANALYSE-VARIABLE to handle the next term, $1\text{`}\langle x, y\rangle$. Both tokens in the place middle are tried, but only $\langle 2, 1\rangle$ passes the compatibility check with x. Therefore, the assignment is transformed to $\beta' = \{\langle x, 2\rangle, \langle y, 1\rangle\}$.

The remaining two arcs are handled by ANALYSE-CONSTANT, which ensures that there are enough tokens for them. Finally, ANALYSE-ARCS prints out the assignment. At this point, the marking passed to it equals the original marking minus the evaluations of the input arcs under the assignment. The algorithm starts to backtrack. Since there were no other feasible choices in either active instance of ANALYSE-VARIABLE, the algorithm terminates.

Some Remarks. For the sake of simplicity, the illustrated procedures do not cover guards. In our implementation, guards are split to terms combined via logical conjunction. Whenever all the variables of a guard term become defined (due to assignments to $\beta'(x)$ in ANALYSE-VARIABLE), the term is evaluated. If the guard evaluates to false, the algorithm backtracks, just like it does in case a term T_k becomes incompatible. If an evaluation error occurs, the algorithm displays the assignment for diagnostics and backtracks.

Procedure ANALYSE-VARIABLE evaluates the multiplicity of a term T_k. When the multiplicity evaluates to zero, the variables in X_k remain undefined. As a result of this, the completed valuations displayed by ANALYSE-ARCS may contain undefined variables. This is not a problem if these variables are never evaluated due to short-circuit evaluation. Otherwise an error may occur when the transition is fired and its output arcs are evaluated. Also, before firing a transition, our implementation ensures that the guard evaluates to true.[4]

Correctness. The calling hierarchy of the algorithm is straightforward. The main procedure ANALYSE invokes ANALYSE-ARCS, which passes control to ei-

[4] Traditionally, "don't care" variables are assigned a nondeterministic choice of values from their domains. This generates unnecessary transition instances with identical behaviour. To avoid this, we assign these variables the special value ϵ.

ther ANALYSE-CONSTANT or ANALYSE-VARIABLE, which in turn call ANALYSE-ARCS. Each recursive call to ANALYSE-ARCS increments k, and the recursion terminates at $k = n$.

The assignment passed to ANALYSE-ARCS initially maps each variable to the undefined value. The only place where the assignment is modified is in ANALYSE-VARIABLE, where only previously undefined variables can be assigned.[5]

When ANALYSE-ARCS is invoked with $k = n$, the evaluation of the arc inscriptions under the gathered assignment is a subset of the marking passed to ANALYSE. This follows from two facts. Firstly, whenever the algorithm unifies an inscription and a multiset, it ensures that the marking contains the multiset and removes the multiset from the marking used for unifying further inscriptions.

Secondly, all split arc inscriptions are evaluated and ensured to match the multiset assigned to them. The procedure ANALYSE-CONSTANT evaluates the split arc inscription under the assignment gathered so far. In the procedure ANALYSE-VARIABLE, the relationship between arc inscriptions and markings is restricted by the compatibility check $\bigwedge_{j=1}^{k} T_j \sim_{\beta'} m_j$. At the deepest call to ANALYSE-VARIABLE, all variables have been assigned, and the compatibility check is equivalent to $\bigwedge_{j=1}^{k} \bar{\beta}'(T_j) = \bar{\beta}'(m_j)$.

To be sure that the algorithm finds all assignments or reports errors, we must investigate the conditions under which it backtracks without reporting anything. The procedures ANALYSE and ANALYSE-ARCS do not backtrack. ANALYSE-CONSTANT does backtrack when an input place would have an insufficient marking, when the test $M(p_k) \geq m_k$ fails. ANALYSE-VARIABLE silently backtracks when the arc inscriptions unified so far would be incompatible under the assignment, causing the test $\bigwedge_{j=1}^{k} T_j \sim_{\beta'} m_j$ to fail. Clearly, all assignments under which transitions are enabled must pass these tests. Therefore, the algorithm finds all relevant assignments.

3.2 Firing Transitions

At the moment when ANALYSE-ARCS displays a completed valuation β, the marking M passed to it is exactly the original marking passed to ANALYSE, minus the evaluations of the input arc inscriptions under β. Transition firing can be integrated to our enabling test algorithm by just replacing the **print** statement with something that binds the rest of the variables[6] and adds the evaluations of the output arcs to M.

Our current implementation of the algorithm combines enabling tests with firing. This is useful when all immediate successors of a state are to be generated, since there is no need to explicitly store the assignments. Also, if input and output arcs have similar inscriptions, some output arc inscriptions may have

[5] The variables are previously undefined, since the variable sets X_k are required to be pairwise disjoint.

[6] In our implementation, output arc inscriptions may make use of nondeterministically bound variables and multiset-valued variables that represent the tokens removed from the input places.

already been evaluated on the input side, and applying an optimisation technique called common subexpression elimination can save computations.

3.3 Unfolding

With slight modifications, the enabling test algorithm can also be used for unfolding Algebraic system nets to compact Place/Transition nets. Doing so has at least the following advantages:

- simple modifications: easy implementation, small chance of errors
- smaller unfolded net:
  - no unconnected places
  - sometimes finite unfoldings for nets with infinite domains

There are two unfolding options in Maria: reduced and traditional. The latter option essentially implements the traditional definition of unfolding, e.g. [10, Definition 13], generating all possible assignments for all transitions. It doesn't generate all low-level places, though; only places that are connected to a low-level transition or are initially marked are generated.

The reduced unfolding option works by maintaining a set of low-level places that can ever be marked. This set is represented as a marking M of the high-level net. For all places $p \in P$ that contain tokens m in the marking, $M(p)(m) \geq 0$, there exists a low-level place $\langle p, m \rangle$. This marking is constructed incrementally, starting from the initial marking of the net.

The multiset containment comparisons $M(p_k) \geq m_k$ in Analyse-Constant and Analyse-Variable are modified so that they ignore the exact multiplicities: $M(p_k) \succeq m_k$ if and only if for each d such that $m_k(d) > 0$, it holds that $M(p_k)(d) > 0$.

Once the modified algorithm completes an assignment β of a transition t, it must unfold the input and output arcs of the transition. Our implementation accomplishes this by constructing two collections of multisets for the high-level input and output arcs: $M_-(p) := \bar{\beta}(i(\langle p, t \rangle))$ and $M_+(p) := \bar{\beta}(i(\langle t, p \rangle))$. For each value d such that $M_-(p)(d) > 0$, it constructs a low-level input arc of weight $M_-(p)(d)$ from the low-level place $\langle p, d \rangle$ to the low-level transition $\langle t, \beta \rangle$. The output arcs are constructed in similar way. The marking M is augmented with M_+. The algorithm keeps unfolding the high-level transitions in different modes until no new items are introduced in M.

When applied to the net illustrated in Figure 1, this algorithm yields the place/transition system illustrated in Figure 4, no matter how big domains the high-level places have. It can be easily seen that if the initial marking of this net contains n different tokens, the unfolded net can have at most $2n$ places and $\frac{1}{2}(n^2 + n)$ transitions.

It should be noted that also the reduced unfolding may be unmanageably large even if the high-level system has a small state space. A minimal unfolding (with no dead places or transitions) could be extracted from the full reachability graph of the high-level system by constructing only those low-level places that ever become marked and those transitions that ever fire. In the case of

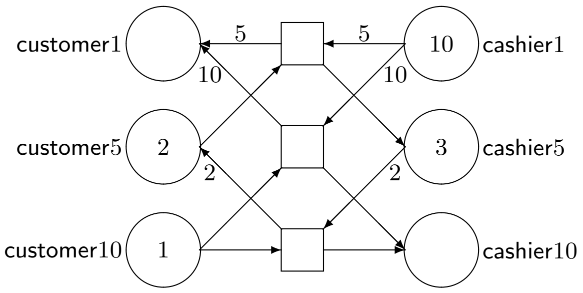

Fig. 4. A reduced unfolding of the net presented in Figure 1.

Figure 4, the reduced unfolding is also a minimal unfolding, since all transitions are enabled in the initial state and there cannot be dead places or transitions.

4 Optimisation Techniques

The first unification algorithm implemented in Maria was very dynamic. It even rewrote input arc expressions on the fly, expanding multiset summations whose limits may depend on other variables. The first optimisation step was to expand quantifications in the parsing stage, transforming dynamic limits to variable-dependent multiplicities, so that the arc expressions remained static during the analysis. This improved the overall performance of the tool by 15–20 percent. At the same time, we started to make experiments with executable code generation [14]. Using the C program code generated by our current implementation usually shortens the analysis times to less than a third of the times consumed by the interpreter written in C++.

Previously presented algorithms, such as [7,8,16], dynamically schedule the input arcs. When we replaced dynamic scheduling with static scheduling in our implementation, we noticed a significant performance boost, 20–30 percent. This is mainly due to less bookkeeping, as the variables are always unified in the same order.[7] Dynamic scheduling may provide shortcuts for simulators that randomly pick one assignment for firing a transition without generating all assignments.

4.1 Representing Multisets

It is important to choose the right data structures for representing multisets in the enabling test algorithm. It appears that we are not the first ones to come up with binary search trees. Haagh and Hansen [6, Chapter 3] suggest using a form of balanced binary trees.

Our implementation uses two representations for markings: encoded (used for storing states) and expanded (for computations). The expanded representation is

[7] Variables that are unifiable from several variable-multiplicity arcs and no constant-multiplicity arcs form an exception. Our implementation attempts to unify them from each arc having nonzero multiplicity.

a binary search tree whose keys are multiset items and values are multiplicities. When a multiset is decoded from the state storage, only items with nonzero multiplicity are added. When items are removed from the tree, their multiplicities are set to zero. Since single items are never removed from the multiset, there is no need for costly balancing operations.

According to our experiments, using unbalanced trees is faster than using red-black trees if the places contain a small number of distinct tokens in the reachable markings. Even though unbalanced trees easily degenerate to linked lists, and searches may have to process n nodes instead of $\lceil \log_2 n \rceil$, the savings in insertions dominate for small values of n. Our executable code generator contains an option for enabling or disabling red-black trees.

4.2 Static Heuristics: Sorting the Arcs

We use multisets in two remarkably different ways. Analyse-Constant (Figure 3) performs one containment comparison on a multiset and calls Analyse-Arcs zero or one times. Analyse-Variable may iterate through all items in a multiset and invoke Analyse-Arcs for any number of them.

Let us assume that our enabling test algorithm is invoked on a sequence of n arcs, k of which are constant. Furthermore, let us assume that all multisets contain m distinct items. If the constant arcs are processed first, the search tree will consist of a linear sequence of k calls to Analyse-Constant followed by a tree of Analyse-Variable invocations. There will be at most $k + m^{n-k}$ recursive calls to Analyse-Arcs. The other extreme, analysing constant arcs as late as possible, yields at most $(k+1)m^{n-k}$ recursive calls.

The proportion of these numbers of iteration steps is

$$\frac{k + m^{n-k}}{(k+1)m^{n-k}} = \frac{k}{k+1}\frac{1}{m^{n-k}} + \frac{1}{k+1} \approx \frac{1}{k+1},$$

and the approximation is pretty good already for $m = 2$. Thus, if there are k constant arcs, it is about k times slower to analyse them at the leaves of the search tree than at the root. The difference becomes even more significant if the transition only has a few enabled instances in each marking and most instances of Analyse-Constant backtrack. The earlier this can happen, the better.

The problem, finding an optimal static scheduling that minimises the number of Analyse-Arcs calls, becomes more complicated when we consider the fact that Analyse-Constant can handle non-ground terms that can be evaluated under the assignment generated so far. Intuitively, an optimal scheduling should

– minimise the number of arcs from which variables are unified, and
– minimise the worst-case number of multiset iterations, and
– schedule the remaining arcs as early as possible in such an order that the algorithm is most likely to backtrack early.

Especially the last requirement is difficult to fulfil in static analysis. We apply Gaeta's "Less Different Tokens First" policy [4, Section 5.1] and compute the maximum numbers of distinct items in the input places. A multiset associated with a place whose domain is $BAG(A_s)$ can have at most $|A_s|$ distinct items.[8]

Let us shortly return to our example from Figure 2. If we assume that the domain sizes of the places left, middle and right are d, d^2 and d for some $d > 1$, then the scheduling we presented in the beginning of Section 3 is not very optimal. In the worst case, it iterates through d items in right and d^2 items in middle, at most d of which can pass the compatibility requirements. Analyse-Arcs can be invoked $1 + d(1 + d(1 + 2)) = 3d^2 + d + 1$ times, and Analyse-Variable may scan up to $d + d^3$ multiset items. Scheduling the term $1`(x + 1)$ before $1`\langle x, y\rangle$ would reduce the maximum number of invocations to $1 + d(1 + 1 + d(1 + 1)) = 2d^2 + 2d + 1$. The same number of multiset items need to be scanned in the worst case, but if analysing the term $1`(x + 1)$ fails every time, the $d^3$ scans for $1`\langle x, y\rangle$ can be avoided.

Gaeta divides input arc expressions to three categories: simple, complex and guarded. We use four categories: closed arcs (arcs that may only depend on already unified variables), constant-multiplicity arcs with unifiable variables, variable-multiplicity arcs with unifiable variables, and other arcs.

We have implemented a depth-first search algorithm for splitting the input arc inscriptions as described in the beginning of Section 3. Since the algorithm has exponential complexity with regard to the number of split arcs containing unifiable variables, we programmed a special condition that terminates the search when a solution is found with more than five arcs containing unifiable variables.

Our algorithm uses three cost functions. The primary cost function is the number of variables that will be unified from variable-multiplicity arcs. The secondary cost function is a sum of costs $c_2(S_k)$ for each arc $S_k = \langle T_k, X_k, p_k, m_k\rangle$ defined as

$$c_2(S_k) = \begin{cases} 0 & \text{if } X_k \neq \emptyset \\ \sum_{j=1}^{k-1} [X_k \neq \emptyset] & \text{if } X_k = \emptyset \end{cases}$$

where the square brackets map truth values to 0 and 1. Thus, for closed arcs, the secondary cost is the number of preceding non-closed arcs. Minimising this cost ensures that all closed arcs will be scheduled as early as possible.

As a shortcut, our algorithm prioritises closed arcs over arcs with unifiable variables. Every time the algorithm picks an arc with unifiable variables, some of the remaining arcs may become closed. Only after the closed arcs run out, the algorithm picks the next arc with unifiable variables. If the only arcs left are in the "other" category, the search backtracks. If no complete schedulings are found, a unification error will be reported.

The third and last cost function is the maximum number of iterations possible with the scheduling. For each split arc $S_k = \langle T_k, X_k, p_k, m_k\rangle$, we define

$$c_3(S_k) = \begin{cases} m(p_k) & \text{if } X_k \neq \emptyset \\ 1 & \text{if } X_k = \emptyset \end{cases}$$

[8] If the multiset is associated with a maximum cardinality, then it is another limit.

where $m(p_k)$ denotes the maximum possible number of distinct tokens in the place p_k. The total cost is defined as

$$c_3(S_1) \cdot (1 + c_3(S_2) \cdot (1 + c_3(S_3) \cdot (1 + \cdots)))$$

where the term $(1 + \cdots)$ after the last cost $c_3(S_n)$ is replaced with 1.

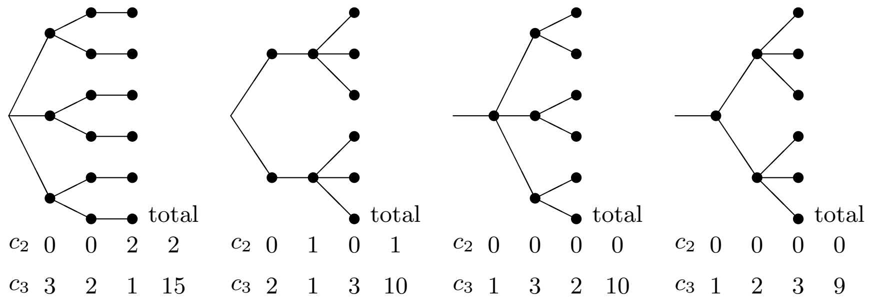

Fig. 5. Minimising Search Trees with Cost Functions

Figure 5 demonstrates the relation between the secondary and ternary cost functions. It illustrates the maximal search trees imposed by four different orderings for three split arcs, one of which is constant. The constant arc is the only contributor to the secondary cost, and scheduling it first minimises the secondary cost. The total ternary cost is the maximum number of recursive invocations to Analyse-Arcs, denoted in the figure with opaque circles. The scheduling presented on the right implies the smallest tree, 9 nodes.

The algorithm selects a scheduling that has the minimum primary cost, number of variables unified from variable-multiplicity arcs. If there are several such schedulings, then the one with the least secondary cost is selected. If that selection is not unique, then one with the smallest ternary cost is chosen.

As a finishing touch for the found static scheduling, our algorithm sorts contiguous sequences of closed terms in such a way that arcs whose places have the smallest number of distinct tokens are scheduled first. This enforces Gaeta's "Less Different Tokens First" policy.

4.3 Dynamic Heuristics

Caching. Ilié and Rojas suggest in [8, Section 3.4] that to speed up a simulator, one could build up a cache that maps input place markings of transitions to sets of enabling assignments. In our experiments, it turned out that in exhaustive reachability analysis, this kind of cache is only useful when there are a large number of states in which the input places of a transition are marked in exactly the same way.

In simulations of timed nets, where exactly the same states are visited over and over again, using such caches may pay off if the bookkeeping overhead (comparing and duplicating input place markings and copying associated sets of enabling assignments) is smaller than the cost of computing the enabling assignments from the scratch. This depends on the implementation, on the size and the scheduling policy of the cache and on the model.

We experimented with an artificial model, one of whose transitions had n input arcs from a place holding n tokens, with a total of $n!$ enabled assignments. For $n = 7$, generating the 5040 assignments took several thousands of times longer than a cache look-up. In more realistic models, we witnessed differences of at most a few percent. In many models, such as the dining philosophers [3] or the distributed data base management system [5], all cache look-ups failed. Due to this experience, we decided to eliminate the cache altogether and to integrate transition firings with enabling tests. This improved the execution times by about ten percent.

Cardinality Tests. Gaeta [4, Section 4.3] has implemented heuristics for detecting when a transition is disabled. He keeps track of the number of tokens in each input place. If a place contains less tokens than a transition would consume, the transition cannot be enabled and the search for enabling assignments can be avoided.

Our implementation of the cardinality test needs to consider arcs with variable multiplicity. Their multiplicities are assumed to be zero. Since the heuristics is implemented in generated code, the comparisons can be omitted if they are known to hold. In Maria models, it is possible to speed up analysis by specifying conditions on the amounts of tokens places may hold in reachable markings.

4.4 Some Experimental Results

Maria uses an explicit technique for maintaining the set of reachable states and the transition instances leading from one state to another. Everything related to the reachability graph is kept in disk files [13]. Because of this, the analyser spends most of its execution time checking whether an encoded state exists in the reachability graph. In our tests, Maria has generated full state spaces of converted Prod [18] models in 0.5 to 1 times the Prod speed. One explanation for the slowness is that Maria detects evaluation errors and supports much more powerful algebraic operations than Prod, which makes optimisations in the C code generation difficult. Also, it is possible to use probabilistic verification with Maria. When no arcs are stored and a reachability set is maintained in memory, the tool performs an order of magnitude faster.

Analysing the biggest state space so far with Maria, a translation of a radio link control protocol specified in SDL consisting of 15,866,988 states and 61,156,129 events, took 5 megabytes of memory and 1.55 gigabytes of disk space, most of which was consumed by the arc inscriptions and double links stored with the reachability graph. The analysis was completed in less than nine hours on a 266 MHz Pentium II system.

Table 1 lists some models we have analysed and unfolded in Maria. All models except "rlc" are distributed with the tool. There are three figures for

Table 1. Unfoldings and State Spaces of Selected Models

	Folded		Unfolded		Reduced		Minimal		State Space	
Model	$\lvert P\rvert$	$\lvert T\rvert$	$\lvert P\rvert$	$\lvert T\rvert$	$\lvert P\rvert$	$\lvert T\rvert$	$\lvert P\rvert$	$\lvert T\rvert$	states	arcs
dining(10)	2	3	40	30	40	30	40	30	6,726	43,480
dbm(5)	8	4	111	60	96	50	96	50	406	1,090
dbm(10)	8	4	421	220	391	200	391	200	196,831	1,181,000
sw(1,1)	12	9	41	422	35	288	35	54	164	352
sw(2,2)	12	9	2,048	1,087,382	729	239,478	129	688	2,640	7,716
sw(6,6)	12	9	–	–	–	–	9,805	145,464	1,774,716	7,127,688
resource	4	3	336	8,158	193	4,610	111	45	538,318	4,136,459
rlc	18	104	708	14,736	114	1,429	–	–	15,866,988	61,156,129

unfolded net sizes. The first column is for "traditional" unfoldings, excluding unconnected places; the second is for unfoldings reduced with our method, and the third is for minimal unfoldings obtained from the reachability graph.

The model named "resource" solves a resource allocation problem. On our system, Maria generates its full reachability graph in 26 minutes, using 3 megabytes of memory and 85.7 megabytes of disk space. With the default capacity limit, LoLA consumes 4 minutes less time but about 530 megabytes more memory on the reduced unfolding of this model. We tried to tighten the capacity limit to save memory but failed, because the limit is global for all places.

Reduced unfoldings work best for models with a sparse state space, i.e. only a fraction of the possible states are actually reachable. For some theoretically pleasing symmetric models, our reduced unfolding does not gain much.

We have the feeling that models of communication protocols, especially those translated from a high-level programming language, have sparse state spaces. The sliding window protocol model we experimented with ("sw" in Table 1) is a good example of this. Already with very small window sizes its traditional unfoldings become unmanageably large. Even the reduced and minimal unfoldings are not very helpful for larger window sizes. This is because all buffer reads and writes in the model are atomic, which reduces the reachability graph but makes the unfolding explode.

5 Conclusion and Future Work

Earlier work on performing enabling tests for high-level nets appears to be limited to nets whose arc inscriptions have constant weights. According to Kindler and Völzer [10], it is difficult to model distributed network algorithms under such limitations. They present Algebraic system nets as a solution, but do not define any algorithms for analysing these nets on the high level.

We viewed enabling tests for high-level nets—constructing the set of assignments under which a transition is enabled in a given marking—as a unification problem, matching multiset-valued terms and subsets of constant multisets. Our approach avoids the combinatorial explosion inherent in this problem by disallowing multiset-valued variables on input arc inscriptions, by restricting the set of operations recognised by the unification algorithm, and by requiring that variable-dependent weights and arbitrary multiset-valued terms can be evaluated based on variable bindings gathered from other terms.

Our reachability analyser MARIA supports queues and stacks on the data type level. Powerful operations, such as removing multiple items from the middle of a queue, make it easy to construct compact high-level models. Experiments show that it is often infeasible to unfold such models in the traditional way. Special constructs for translating large blocks of atomic operations in high-level nets into behaviour-equivalent compact low-level nets are subject to further research.

The presented unification algorithm processes terms in a fixed order. A method for statically ordering the terms in a close to optimal way was presented, and some optimisations to the algorithm were discussed. Some of the presented techniques may be best suited for exhaustive analysis tools; their applicability in simulators was not tested.

A new method for unfolding high-level nets based on a kind of "coverable marking" was presented. The method often produces considerably smaller unfoldings than the common approach of iterating over all domains.

Acknowledgements. The author would like to thank Dr. Kimmo Varpaaniemi, who implemented the skeleton of the first unification algorithm in MARIA, for fruitful discussions and ideas for improving the algorithm, and the referees for their constructive comments.

References

1. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, NY, USA, 1973.
2. Gianfranco Chiola, Giuliana Franceschinis and Rossano Gaeta. A symbolic simulation mechanism for well-formed coloured Petri nets. In Philip Wilsey, editor, *Proceedings, 25th Annual Simulation Symposium*, pages 192–201, Orlando, FL, USA, April 1992. IEEE Computer Society Press, Los Alamitos, CA, USA.
3. Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
4. Rossano Gaeta. Efficient discrete-event simulation of colored Petri nets. *IEEE Transactions on Software Engineering*, 22(9):629–639, September 1996.
5. 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.
6. Torben Bisgaard Haagh and Tommy Rudmose Hansen. Optimising a Coloured Petri Net Simulator. Master's thesis, University of Århus, Denmark, December 1994. `http://www.daimi.au.dk/CPnets/publ/thesis/HanHaa1994.pdf`.
7. Jean-Michel Ilié, Yasmina Maîzi and Denis Poitrenaud. Towards an efficient simulation based on well-formed Petri nets, extended with test and inhibitor arcs. In Tuncer I. Oren and Louis G. Birta, editors, *1995 Summer Computer Simulation Conference*, pages 70–75, Ottawa, Canada, July 1995. Society for Computer Simulation International.

8. Jean-Michel Ilié and Omar Rojas. On well-formed nets and optimizations in enabling tests. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 300–318, Chicago, IL, USA, June 1993. Springer-Verlag, Berlin, Germany.
9. Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1992.
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 Analyser for Algebraic System Nets.* On-line 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. Marko Mäkelä. Condensed storage of multi-set sequences. In Kurt Jensen, editor, *Practical Use of High-Level Petri Nets*, DAIMI report PB-547, pages 111–125. University of Århus, Denmark, June 2000.
14. Marko Mäkelä. Applying compiler techniques to reachability analysis of high-level models. In Hans-Dieter Burkhard, Ludwik Czaja, Andrzej Skowron and Peter Starke, editors, *Workshop on Concurrency, Specification & Programming 2000*, Informatik-Bericht 140, pages 129–142. Humboldt-Universität zu Berlin, Germany, October 2000.
15. Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag, Berlin, Germany, 1985.
16. Michael J. Sanders. Efficient computation of enabled transition bindings in high-level Petri nets. In *2000 IEEE International Conference on Systems, Man and Cybernetics*, pages 3153–3158, Nashville, TN, USA, October 2000.
17. Karsten Schmidt. LoLA: a low level analyser. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474, Århus, Denmark, June 2000. Springer-Verlag, Berlin, Germany.
18. 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.