

# 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  
<http://www.tcs.hut.fi/maria/>

28<sup>th</sup> June 2001

\*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.

## Outline

- Strengths and Weaknesses of High-Level Net Modelling
- The Problem: Finding Enabled Transition Instances
- The Solution: A Unification Algorithm
  - Optimisation Techniques
  - Applications
- Conclusion

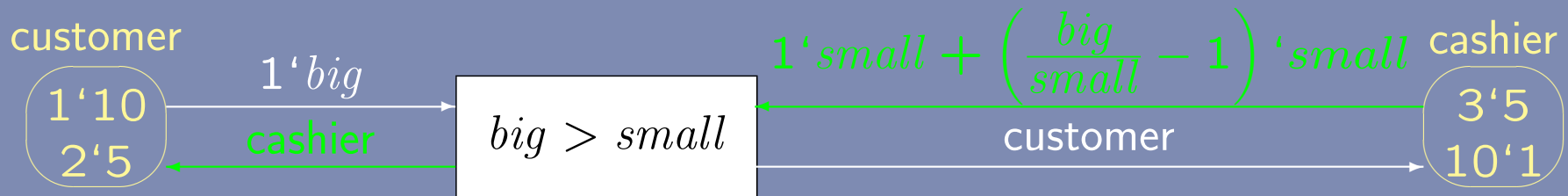
## Strengths and Weaknesses of High-Level Net Modelling: Strengths

Using a high-level language when modelling complex systems has numerous proposed advantages:

- 😊 are more compact than their low-level counterparts
- 😊 models may be easier to understand and to modify
- 😊 modelling in the high level may save time considerably
- 😊 the states of the model can be compressed better

Alas, high-level languages introduce an overhead, since the operations must be transformed into something the underlying computing machinery can perform:

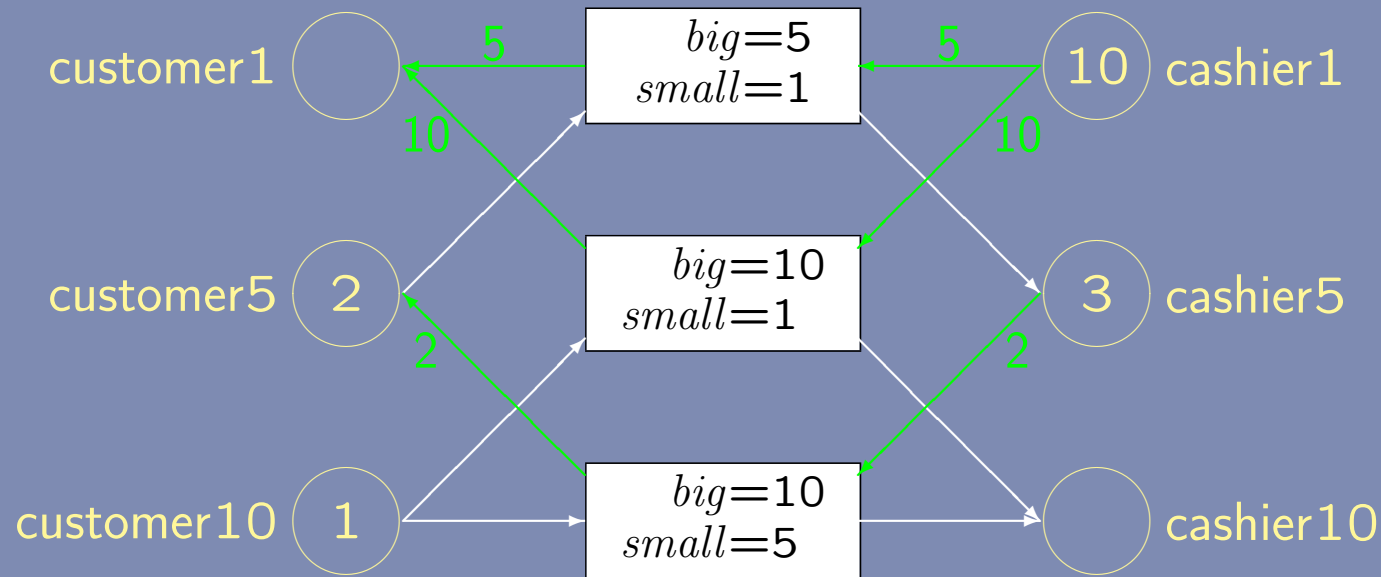
- ☹️ compiling and precomputing: the result may be infinite or prohibitively large  
(Petri nets: unfolding)
- ☹️ interpreting on high level: this may increase time complexity  
(Petri nets: structural techniques, model checking, reachability analysis)



Algebraic system nets are a formalistic version of coloured nets whose inscriptions are terms of many-sorted algebras. Above is an algorithm for changing money.\*

- A customer gives a coin to a cashier who changes it to smaller coins.
- (Another transition can be added for the symmetric case.)
- What about changing  $(2)+(2)+(1)$  for a  $(5)$  coin?

\*The domains of the places and variables and the algebraic operations are not illustrated.



This picture illustrates the unfolding of the net presented on the previous slide, assuming that the domains of the places are implied by the initial marking. If the domain included e.g.  $\textcircled{50}$ , the unfolding would have two more places and three more transitions, all of which would be unmarked or dead in the illustrated marking.

- Earlier work: mainly Well-formed nets (a proper subset of Algebraic system nets)
- Our approach: Algebraic system nets
  - dynamic weights
  - arbitrary operations and data types
  - extensions for safety checking: short-circuit evaluation and evaluation errors
  - some restrictions against combinatorial explosion; can be statically detected; can be removed with an exponential-time algorithm

## The Problem: Finding Enabled Transition Instances

There are two ways to construct the set of assignments under which a transition is enabled:

1. construct all possible assignments for the variables that occur in the arc inscriptions and in the guard, and prune those that do not fulfil the firing rule, or
2. construct the assignments by finding unifiers for the input arc inscriptions and a given marking so that the firing rule is fulfilled

The former approach is common when the net is unfolded. The latter one is superior when the domains of the variables are large and the net is sparsely marked—which is usually the case when modelling protocols or high-level programs having complex data types. This work concentrates on the latter approach.

## The Unification Approach

A model of the UMTS Radio Link Control protocol we have been analysing has complex data types. An arbitrary queue of at most  $n$  items having a domain of  $1,363,531,857,395,729 > 2^{50}$  possible values takes more than  $50n$  bits of storage. In the model, there are a handful of such queues with  $n \geq 4$ .

The process of finding assignments or substitutions under which two algebraic terms are equivalent is often referred to as *unification*. In algebraic system nets, 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. Unification works in two ways.

1. If all variables of a term are known, the evaluation can be compared with the marking.
2. Otherwise some variable bindings are derived from the term and the marking.

## The Unification Approach: Basic Definitions

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. For instance, we cannot match a constant  $n$  and the term  $x + y$  if the values of  $x$  and  $y$  are unknown. Even if the variables are prohibited to be negative, there would be  $n + 1$  different unifiers:  $\{x \mapsto 0, y \mapsto n\}$ ,  $\{x \mapsto 1, y \mapsto n - 1\}$ ,  $\dots$ ,  $\{x \mapsto n, y \mapsto 0\}$ . If the variables are allowed to have opposite signs, there are infinitely many unifiers.

We distinguish two classes of operations that are recognised by our unification algorithm.

**reversible unary operations** that can be “neutralised”:  $f^{-1}(f(x)) = x$ , and

**constructors** that tie terms together, e.g.  $\langle x, \langle y, z \rangle \rangle$ .

The set of reversible operations can be extended with operator applications having one free variable. For instance, the term  $x + y$  can be matched with  $n$  if one of the variables is known:  $y = n - x$ . For simplicity and efficiency, our implementation does not support any reversible operations.

For a sort  $s \in S$  and a term  $T \in \mathbf{T}_s^{SIG}(X)$ , we say that a variable  $x \in X$  is *unifiable* from  $T$ , denoted  $x \triangleleft T$ , if either  $x = T$ , or  $T = o(T')$  and  $x \triangleleft T'$  where  $o$  is a reversible operation, or  $T = c(T_1, \dots, T_n)$  and  $x$  is unifiable from some  $T_k$  where  $c$  is a constructor operation.

We extend the definition of unifiable variables to *unifier candidates* that map terms and constants (ground terms) to feasible value candidates,  $x \triangleleft_{T_\emptyset} T$ .

The idea is to extend the unifier candidates to assignments under which the transition is enabled. To prune illegible assignments as early as possible, we introduce the concept of *assignment compatibility*.

A term  $T$  and a ground term  $T_\emptyset$  are compatible under an assignment  $\beta$ ,  $T \sim_\beta T_\emptyset$ , if  $T$  evaluates to  $T_\emptyset$ , or both  $T$  and  $T_\emptyset$  are constructor terms, and all components are mutually compatible, or  $T$  depends on an undefined variable.

For instance, if  $T = \langle x, y \rangle$ ,  $T_\emptyset = \langle 1, 2 \rangle$ ,  $\beta(x) = 3$  and  $\beta(y) = \epsilon$ , we would have  $\bar{\beta}(T) = \epsilon$ , but since we “look inside the constructor”, the incompatibility can be detected:  $\langle x, y \rangle \not\sim_\beta \langle 1, 2 \rangle$  since  $x \not\sim_\beta 1$  since  $\bar{\beta}(x) = \beta(x) = 3 \neq 1$ .

## The Unification Approach: Splitting the Arcs

In order to improve the granularity of our algorithm, we split each input arc inscription to a list of multiset constructors combined with multiset summation.

Our algorithm derives unifier candidates from *elementary multiset constructors*: terms that have a multiplicity and a basic (scalar) term. In our **money-changing example**, the split input arcs are  $1 'big$ ,  $1 'small$  and  $\left(\frac{big}{small} - 1\right) 'small$ . With each split arc, we associate

- the arc inscription term, e.g.  $1 'big$ ,  $1 'small$  or  $\left(\frac{big}{small} - 1\right) 'small$
- a set of variables whose values are derived from the arc, e.g.  $\{big\}$ ,  $\{small\}$ , or  $\{\}$
- the input place connected to the arc, e.g. customer, cashier or cashier
- a multiset that is being matched with the term (work storage), initially  $\emptyset$ .

## The Unification Approach: A Depth-First Algorithm

The algorithm builds the assignments in a depth-first traversal on the split input arcs. Arcs are divided into four categories, listed from the easiest to the hardest:

**closed arcs** whose all variables are defined in the assignment gathered so far

**constant-multiplicity arcs** that contain unifiable variables

**variable-multiplicity arcs** with closed multiplicity that contain unifiable variables

**other arcs** that the algorithm can do nothing about

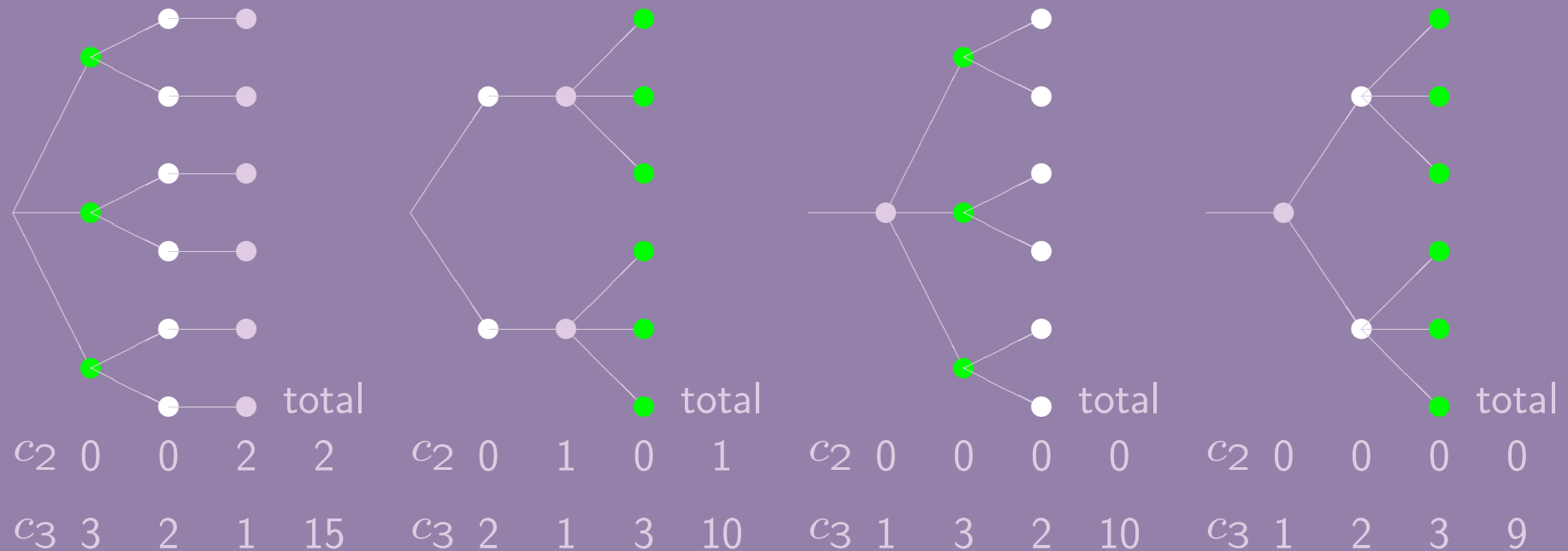
## The Unification Approach: Sorting the Arcs

The input arcs are sorted so that as many arcs as possible remain in the easiest categories. The hardest category (“other arcs”) could be managed by enumerating through the domains of the unbound variables, but our implementation reports an error instead.

The search tree is minimised by minimising the following three cost functions:

1. number of variables that will be unified from variable-multiplicity arcs
2. number of preceding non-closed arcs, or 0 if the arc is not closed
3. maximum number of possible iterations in the search

## The Unification Approach: Sorting the Arcs



This figure illustrates how the secondary and ternary cost functions work when one arc is inscribed with a constant and the other arcs contain variables bound to places that contain at most 2 or 3 distinct tokens in any marking.

## Applications of the Unification Algorithm

In its basic form, the algorithm can be used for obtaining the set of assignments under which a given transition is enabled in the given marking.

While the algorithm executes, it keeps track of the available tokens in the input places of the transition. Whenever the algorithm completes an assignment, it has implicitly computed the input effect of the transition and subtracted it from the marking.

Thus, we can replace the **print** statement with some code that computes the output effect and adds it to the intermediate marking. In that way, the enabled transition instances need not be explicitly stored anywhere, and the input arcs need not be evaluated twice.

Another possible modification is unfolding. The multiset containment checks that guide the search can be replaced with something that makes the algorithm “think” there are always enough tokens in the input places. In traditional unfolding, the whole domains of all places are considered.

A *reduced unfolding* is constructed by growing a coverable marking in an unbounded loop over all high-level transitions. The loop terminates once a fixpoint is reached.

The places of the reduced unfolding correspond to the  $\langle \text{place}, \text{value} \rangle$  pairs in the coverable marking, and the transitions correspond to each  $\langle t, \beta \rangle$  pair completed during the search.

In our **money-changing example**, the coverable marking, or the set of necessary low-level places, is initialized as  $\{\text{customer}10, \text{customer}5, \text{cashier}5, \text{cashier}1\}$ . The only transition can fire in three modes in the initial marking:  $\{\text{big} \mapsto 5, \text{small} \mapsto 1\}$ ,  $\{\text{big} \mapsto 10, \text{small} \mapsto 1\}$ , and  $\{\text{big} \mapsto 10, \text{small} \mapsto 5\}$ . Evaluating the outputs with  $\text{big} \mapsto 10$  and  $\text{small} \mapsto 1$  introduce the low-level places  $\text{cashier}10$  and  $\text{customer}1$ . The coverable marking does not grow further.

## Conclusion

Earlier work on performing enabling tests for high-level nets appears to be limited to nets whose arc inscriptions have constant weights.

At ICATPN 1998, Kindler and Völzer presented Algebraic system nets as a modelling formalism for distributed algorithms, but did not define any high-level analysis algorithms.

We approached the problem of finding enabled transition instances as a unification problem. Our approach avoids the combinatorial explosion inherent in this problem by introducing some restrictions that can be checked in static analysis.

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.

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.