Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 80
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 80
Espoo 2003 HUT-TCS-A80

ON THE SYMMETRY REDUCTION METHOD FOR PETRI NETS AND SIMILAR FORMALISMS

Tommi Junttila

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 80
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 80
Espoo 2003 HUT-TCS-A80

ON THE SYMMETRY REDUCTION METHOD FOR PETRI NETS AND SIMILAR FORMALISMS

Tommi Junttila

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering, for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 31st of October, 2003, at 12 o'clock noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

© Tommi Junttila

ISBN 951-22-6744-6
ISSN 1457-7615

Multiprint Oy
Helsinki 2003

ABSTRACT: The symmetry reduction method is a technique for alleviating the combinatorial explosion problem arising in the state space analysis of concurrent systems. This thesis studies various issues involved in the method. The focus is on systems modeled with Petri nets and similar formalisms, such as the Murφ description language.

For place/transition nets, the computational complexity of the sub-tasks involved in the method is established. The problems of finding the symmetries of a net, comparing whether two markings are equivalent under the symmetries, producing canonical representatives for markings, and deciding whether a marking symmetrically covers another are classified to well-known complexity classes. New algorithms for the central task of producing canonical representatives for markings are presented. The algorithms apply and combine techniques from computational group theory and from the algorithms for the graph isomorphism problem. The experimental results show that the new algorithms are competitive against the previous ones described in the literature.

Data symmetries, i.e., state space symmetries produced by symmetric use of data values, of a class high-level Petri nets, algebraic system nets, are also studied. It is defined how the permutations of the data values produce corresponding permutations in the state space of the net. In addition, sufficient conditions for the annotations in the net are defined in order to ensure that the produced permutations are indeed symmetries. Because these conditions turn out to be computationally difficult to verify, an approximation rule is additionally given. The practical use of the developed theory is illustrated by defining a class of high-level Petri nets allowing the use of common data types such as lists, sets, and arrays. The data symmetries of such nets are produced in a way similar to well-formed nets and the Murφ system, i.e., by declaring some primitive data types to be permutable and restricting the set of applicable operations on such types.

New algorithms for checking whether two states are equivalent and for producing representatives for states under data symmetries are also described. The proposed algorithms either directly use the existing algorithms for the graph isomorphism problem, or use a partition refinement process modified from such algorithms. The algorithms are not limited to high-level Petri nets but are also applicable to the Murφ description language. The experimental results show that the new algorithms are competitive against the previous ones implemented in the Murφ tool.

KEYWORDS: State space analysis, symmetry, Petri nets, place/transition nets, algebraic system nets, the Murφ system.

CONTENTS

1 Introduction **1**
- 1.1 This Thesis . . . 2
- 1.2 Other Related Work . . . 6
- 1.3 Organization . . . 8

2 Preliminaries **9**
- 2.1 State Space Analysis . . . 9
- 2.2 The Symmetry Reduction Method . . . 12
  - 2.2.1 Finding State Space Symmetries . . . 13
  - 2.2.2 Reduced Reachability Graphs . . . 14
  - 2.2.3 Analysis of Reduced Reachability Graphs . . . 16
- 2.3 Other Preliminaries . . . 19

3 Place/Transition Nets: Computational Complexity **22**
- 3.1 Basic Definitions . . . 22
  - 3.1.1 Representing Symmetries . . . 25
- 3.2 Finding the Symmetries . . . 26
  - 3.2.1 Computational Complexity . . . 26
  - 3.2.2 Algorithms . . . 27
- 3.3 Computational Complexity of the Orbit Problems . . . 28
  - 3.3.1 The Marking Equivalence Problem . . . 28
  - 3.3.2 Finding the Lexicographical Leader Marking . . . 33
- 3.4 Symmetric Coverability . . . 38

4 Place/Transition Nets: New Canonical Marking Algorithms **40**
- 4.1 Preliminaries . . . 41
  - 4.1.1 The Schreier-Sims Representation . . . 42
  - 4.1.2 Compatible Permutations . . . 43
- 4.2 Using the Canonical Version of the Characteristic Graph . . . 49
- 4.3 Backtrack Search in the Schreier-Sims Representation . . . 53
- 4.4 Partition Guided Schreier-Sims Search . . . 56
  - 4.4.1 Partition Generators . . . 56
  - 4.4.2 Partition Refiners and Invariants . . . 60
- 4.5 Experimental Results . . . 63
  - 4.5.1 Net Classes . . . 63
  - 4.5.2 Results . . . 65

5 Data Symmetries of Algebraic System Nets **71**
- 5.1 Signatures and Algebras . . . 74
- 5.2 Algebraic System Nets . . . 76
- 5.3 Data Symmetries . . . 79
  - 5.3.1 Domain Permutations . . . 79
  - 5.3.2 Actions of Domain Permutations . . . 80
  - 5.3.3 Term Compatibility . . . 81
  - 5.3.4 Complexity of Deciding Term Compatibility . . . 84
  - 5.3.5 Approximating Term Compatibility . . . 87

6 Extended Well-Formed Nets **89**
6.1 Type System . . . 89
6.1.1 Types . . . 90
6.1.2 Permutable Primitive Types . . . 91
6.2 Operations . . . 92
6.3 Examples . . . 100

7 Algorithms for Data Symmetries **102**
7.1 An Abstract System Class . . . 103
7.1.1 Stabilizers and Storing Subgroups . . . 104
7.2 Value Trees and Characteristic Graphs . . . 106
7.3 A Basic Partition based Algorithm . . . 110
7.3.1 Partition Refiners and Invariants . . . 113
7.3.2 Some Useful Invariants . . . 117
7.4 Improvements based on Search Trees . . . 124
7.4.1 Properties of Search Trees . . . 127
7.4.2 Producing Canonical Representative States . . . 129
7.4.3 A Relative Hardness Measure for States . . . 130
7.4.4 A Sidetrack on Equivalence Testing of States . . . 132
7.5 Handling Large and Infinite Unordered Primitive Types . . . 134
7.6 Algorithms based on Characteristic Graphs . . . 136
7.7 Some Experimental Results . . . 139
7.8 Related Work . . . 142

8 Conclusions **145**
8.1 Future Work . . . 147

Bibliography **149**

List of Figures

2.1 A simple mutual exclusion program for P processes 10
2.2 The state space of the program in Figure 2.1 for $P = 2$ 11
2.3 Reduced reachability graphs for the program in Figure 2.1 when $P = 2$ 15
3.1 A net for a railroad system 24
3.2 The reachability graph of the net in Figure 3.1 24
3.3 Two reduced reachability graphs for the net in Figure 3.1 . . . 25
3.4 Mappings between graphs and nets for the automorphism problem 27
3.5 Reduction from a graph to a live and 1-safe net 30
3.6 A net with no suitable canonical representative function . . . 39
4.1 An example net 43
4.2 Schreier-Sims representation seen as a tree 44
4.3 Schreier-Sims representation tree augmented with the base element images 44
4.4 Schreier-Sims representation tree augmented with the base element images and their values 44
4.5 Pruned Schreier-Sims representation trees 45
4.6 A marked net and its characteristic graphs 50
4.7 A marked net, its characteristic graph, and the canonical version of the characteristic graph 52
4.8 A three dimensional grid with two agents per row 64
4.9 A net enumerating all directed graphs without self-loops over three vertices 65
4.10 A net enumerating all undirected graphs without self-loops over four vertices 65
5.1 A place/transition net and two corresponding high-level Petri nets 72
5.2 The reachability graph of the net in Figure 5.1(c) 73
6.1 The distributed database net 101
7.1 An EWF-net for a railroad system 104
7.2 A Murφ version of the mutual exclusion program in Example 2.1 105
7.3 A value tree 107
7.4 A characteristic graph 108
7.5 A modified characteristic graph 109
7.6 An ordered value tree 119
7.7 Mapping an unordered value tree to an ordered one 119
7.8 A search tree 127
7.9 Two isomorphic graphs 132
7.10 Two characteristic graphs and their common canonical version 138
7.11 A system enumerating undirected graphs 140

List of Algorithms

2.1 An algorithm for computing reachability graphs 10
2.2 An algorithm for computing reduced reachability graphs . . . 15
4.1 An algorithm enumerating all compatible permutations . . . 48
4.2 An algorithm finding the smallest marking in *posreps*(M) . . 54
7.1 A representative algorithm based on partitions 114
7.2 A representative algorithm based on search trees 125
7.3 A canonical representative algorithm based on characteristic graphs 137

LIST OF SYMBOLS AND NOTATIONS

$\emptyset$	the empty set or the empty multiset
$\cdot$	multiplication of a multiset by a natural number
$\times$	cartesian product or multiplication
$\circ$	the composition operator for functions
$*$	the composition operator for domain permutations
$\star$	the composition operator for partition refiners
$\leq_{\mathrm{m}}^{\mathrm{p}}$	the polynomial time many-one reducibility relation
$<_\beta$	the lexicographical order under the base β
$\preceq$	the cell order preserving partial order between ordered partitions
$\equiv_G$	the equivalence relation induced by the group G
$[q]_G$	the orbit of the state q under the group G
$[A \to B]$	the set of functions from the set A to the set B
$[A \to \mathbb{N}]$	the set of multiset over the set A
$[\rangle$	the symbol for firing and enabledness of Petri nets
$\{A_i\}_{i \in I}$	a family of sets indexed by the set I
$\|A\|$	the cardinality of the set A
$\mathcal{A}$	an algebra
α	an assignment to variables
$\mathrm{Aut}(A)$	the automorphism group of the object A
$\mathbb{B}$	the set of Booleans
β	a base
C_i	the ith cell in an ordered partition
C_i^T	the ith cell in an ordered partition for the type T
canrepr	a canonical representative function
$\mathscr{D}_T$	the domain of the type T
Δ	the transition relation of an LTS
ε	the empty string
E	the edge set of a graph
$\mathfrak{err}$	the error element
$eval_\alpha$	the term evaluation function under the assignment α
F	the set of arcs of a Petri net
f	an operation (symbol)
$\mathcal{F}$	an operation family
G	a group or a graph
$\mathcal{G}_x$	the characteristic graph of a marking or state x
g	a permutation
guard	the transition guard function of an ASN
$f[x \mapsto y]$	the function f except that x is mapped to y
I	an invariant
$\mathbf{I}$	the identity permutation
$incell(\mathfrak{p}, x)$	the function returning the cell number in which the element x belongs to in the ordered partition $\mathfrak{p}$
$\mathcal{K}$	a graph canonizer function
$\mathcal{L}$	a labeled transition system (LTS)

L	the label set of an LTS or the labeling function of a graph
M	a marking of a Petri net
$\mathbb{M}$	the set of all markings of a Petri net
N	a net
$\mathcal{N}$	an algebraic system net (ASN)
$\mathbb{N}$	the set of natural numbers
$\mathit{orbitnum}(x)$	the orbit number of the element x
$\wp(A)$	the set of all subsets of the set A
P	the set of places of a Petri net
p	a place of a Petri net
$\mathfrak{p}$	an ordered partition
$\mathfrak{p}^T$	an ordered partition for the type T
$\mathfrak{P}$	the set of all ordered partitions
pg	a partition generator function
π	an automorphism of an LTS
$\mathit{posreps}(x)$	the set of possible representatives for the object x
ψ	a domain permutation
ψ^T	a domain permutation for the type T
pval	a (place) valuation function
Q	the state set of an LTS
q	a state of an LTS
$\mathcal{R}$	a partition refiner function
repr	a representative function
$\mathcal{RG}(\mathcal{L})$	the reachability graph of an LTS $\mathcal{L}$
$\mathcal{RRG}(\mathcal{L})$	a reduced reachability graph of an LTS $\mathcal{L}$
$\mathfrak{s}$	a state
$\mathcal{S}$	the set of all states
select	a multiset selector function
Sig	a signature
$\mathcal{ST}(\mathfrak{s})$	the search tree for the state $\mathfrak{s}$
$\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$	the search tree for the state $\mathfrak{s}$ and partition $\mathfrak{p}$
$\mathrm{Stab}(A, a)$	the stabilizer subgroup of a under A
succ	a successor function
$\mathrm{Sym}(A)$	the group of all permutations of the set A
T	the set of transitions of a Petri net or a type
$\mathcal{T}$	a set of types
t	a transition of a Petri net
term	a term
$Terms_T^{Sig}(\mathcal{X})$	the set of terms of type T over $\mathcal{X}$
θ	an allowed domain permutation
θ^T	an allowed domain permutation for the type T
type	the place type function of an ASN
V	the vertex set of a graph
vars	the transition variable function of an ASN
$\mathcal{VT}(T, v)$	the value tree of an element v of type T
x	a variable
$\mathcal{X}$	a set or a family of variables
W	the arc annotation function of a Petri net

PREFACE

This thesis is the result of my postgraduate studies and research at the Laboratory for Theoretical Computer Science of Helsinki University of Technology from 1998 to 2003. I'm grateful to Professor Emeritus Leo Ojala and Professor Ilkka Niemelä for giving me this opportunity. I also would like to thank my colleagues in the laboratory for creating a pleasant working atmosphere. I'm especially grateful for the comments that I have got from Keijo Heljanko, Petteri Kaski, Ilkka Niemelä and Patric Östergård.

I would also like to thank David L. Dill (Stanford University), Brendan McKay (Australian National University), and Karsten Schmidt (Humboldt University Berlin) for putting the source codes of the Murφ, *nauty*, and *LoLA* tools, respectively, available in the web. This has helped me considerably in experimenting with the new algorithms.

This research has been funded by the Helsinki Graduate School in Computer Science and Engineering and by the Academy of Finland (projects number 47754, 43963, and 53695). The financial support from the Foundation of Technology (Tekniikan Edistämissäätiö) is gratefully acknowledged.

Finally, I would like to thank my parents and my sister for all their support during these years.

Otaniemi, September 2003

Tommi Junttila

1 INTRODUCTION

Concurrent and distributed hardware and software systems are increasingly used nowadays in many applications where failure is highly undesirable: telecommunication network switches, medical instruments, traffic control systems, and so on. Therefore, there is also a growing need for computer-aided analysis and verification techniques for such systems. The so-called state space based methods are one of the most important approaches for this task [Valmari 1998; Clarke et al. 1999]. They are based on enumerating all the possible states that (the model of) a system may reach during its execution. This set of states is known as the reachability graph of the system. Various properties of the system can be verified by using its reachability graph, such as absence of deadlocks, unreachability of undesirable "bad" states, or, more generally, that the system's behaviors fulfill a property specified by a temporal logic formula. The main advantages of state space methods are that they can be automated and, in the case the verified property does not hold, can usually give a counter-example execution violating the property. The latter property is very helpful when debugging systems. However, the state space based methods suffer from the state explosion problem, meaning that the number of possible states the system may have can be extremely large. Many techniques have been suggested for alleviating the state explosion problem, see e.g. [Valmari 1998; Clarke et al. 1999], including (i) representing state sets symbolically by using e.g. Binary Decision Diagrams (BDDs), (ii) exploiting the independence of concurrent transitions in the so-called partial order methods, (iii) using abstractions to simplify the system, and (iv) exploiting the symmetries of the system. This thesis studies the last technique, known as the symmetry reduction method, in the context of Petri nets [Reisig and Rozenberg 1998a; 1998b] and similar system description formalisms such as the Murφ system [Dill 1996].

As its name implies, the symmetry reduction method exploits the symmetries (that is, automorphisms) of the state space. Such state space symmetries are present in many systems, and are usually induced by a symmetric system structure, use of replicated components, or symmetric use of data values. For instance, the behavior of a distributed database system composed of a server process and several identical client processes is usually symmetric with respect to the clients. Thus the situation in which client 1 is accessing data while the others are idle can be considered equivalent to the one in which client 3 is accessing data while the others are idle. The state space symmetries partition the states into equivalence classes of states called orbits. The main idea of the symmetry reduction method is that, for many verification tasks, it is sufficient to consider only one representative state in each reachable orbit. In short, the symmetry reduction method can be seen as a process of three phases.

1. Finding some information in the system description level that produces state space symmetries. Since the whole purpose of the symmetry reduction method is to avoid enumerating the entire reachability graph, state space symmetries must be found in the system description level. Obviously, the nature of the state space symmetry producing informa-

tion depends on the system description formalisms. In place/transition nets, for instance, the structural symmetries of the net produce corresponding symmetries in the state space. On the other hand, in the Murφ system description formalism, as well as in many classes of high-level Petri nets, symmetric use of data values produces state space symmetries.

2. Building a reduced reachability graph. This step is similar to the usual reachability graph construction except that the states that are equivalent under the symmetries are identified. The goal is to visit only one representative state from each reachable orbit of states induced by the symmetries. There are two ways to achieve this during the iterative reachability graph construction process:
   (a) Compare each newly visited state with all the already visited states for equivalence. In this approach, one must be able to answer the *orbit problem* asking whether two states are equivalent.
   (b) Transform each newly visited state into an equivalent, canonical representative state. Only these representative states are stored in the reduced reachability graph. The task of transforming a state into its canonical representative is called the *constructive orbit problem*.

   As the orbit problems above are, in general, computationally difficult, they can be approximated (i) by using a sound but incomplete state equivalence check, or (ii) by producing representative states that are not necessarily canonical. Of course, this kind of approximation may result in reduced reachability graphs containing more than one representative from certain orbits.
3. Analysis of properties based on the reduced reachability graph. The complexity of this step depends on the analyzed property and its relationship with the applied symmetries. For instance, a reduced reachability graph contains a deadlock state if and only if the original reachability graph does. Thus, checking deadlock freedom under symmetries is straightforward. The same applies to temporal logic model checking of formulae that are preserved by the symmetries. On the other hand, temporal logic model checking of formulae that are not preserved by the applied symmetries requires the use of more involved algorithms.

1.1 THIS THESIS

The aim of this thesis is to study and improve various aspects of the symmetry reduction method in the context of Petri nets and similar formalisms. In short, the following is achieved.

- For place/transition nets, the computational complexity of the subtasks appearing in the symmetry reduction method is established and new algorithms for producing canonical representatives for markings are developed.
- For a class of high-level Petri nets, it is shown how symmetries induced by symmetric use of data can be defined and detected by using an

approach similar to that used in well-formed nets [Chiola et al. 1991] and in the Murφ system [Ip and Dill 1996].
– Finally, new algorithms for the orbit problems under data symmetries are developed, covering both classes of high-level Petri nets and the Murφ system.

The contributions and their relationship to related work are discussed in more detail below.

Place/Transition Nets. The symmetry reduction method for place/transition nets is introduced in [Starke 1991]. It is shown that the symmetries of a net produce symmetries in its state space. A preliminary algorithm for computing the symmetries of a net is additionally given. A considerably improved algorithm for computing the symmetries of a net is described in [Schmidt 2000a], while [Schmidt 2000b] gives algorithms for integrating the symmetries into the reachability graph generation process. This thesis extends these results in two ways.

– The computational complexity of the sub-tasks appearing in the symmetry reduction method for place/transition nets is established. First, it is shown that finding a generating set for the symmetries of a net is as hard as the graph automorphisms problem. It is then shown that the problem of deciding whether two markings are equivalent under the symmetries is as hard as the graph isomorphism problem. In addition, it is shown that finding the lexicographically greatest marking equivalent to a given marking is an $\mathbf{FP}^{\mathbf{NP}}$-complete problem, and thus as hard as many well-known optimization problems such as the traveling salesperson problem. These latter results hold even for 1-safe and live nets when a generating set for the net symmetry group in question is given and the markings in question are actually reachable. Finally, it is shown that the problem of deciding whether a marking symmetrically covers another marking is an **NP**-complete problem. Furthermore, it is proven that the symmetric coverability problem cannot be combined with the canonical representative approach in a straightforward way.
– New algorithms for producing canonical representatives for markings are developed. The algorithms use and combine techniques from computational group theory and from algorithms for producing canonical versions of graphs. They require that the symmetry group of the net is computed prior to the reachability analysis and is stored in a standard representation form for permutation groups. The first algorithm maps the marking to a corresponding graph and then utilizes an algorithm such as the *nauty* tool [McKay 1990] to obtain a canonical version of the graph. The canonical representative for the marking is then computed from an isomorphism between the graph and its canonical version. The second algorithm finds a canonical representative for the marking by performing a backtracking search in the symmetry group representation, pruning the search (i) by considering only symmetries that are "compatible" with the marking, (ii) by using the best candidate marking found so far, and (iii) by using the symmetries that stabilize the marking. The third algorithm first computes an ordered partition of

the elements in the net for the marking in a symmetry-respecting way, and then uses the partition to further prune the backtracking search in the symmetry group representation. The ordered partition is computed by applying partition refinement techniques developed in the context of graph isomorphism algorithms [McKay 1981; Kreher and Stinson 1999]. Some experimental results are also given, showing that the proposed algorithms are competitive against those implemented in the *LoLA* tool [Schmidt 2000b; 2000c].

These results have been published previously in [Junttila 2000; 2001; 2002a].

Data Symmetries of High-Level Petri Nets. Symmetries of high-level Petri nets (i.e., nets in which the tokens can contain data values) are not normally produced by a symmetric structure but by symmetric use of data values. One of the earliest studies of the symmetry reduction method is in the context of colored Petri nets [Huber et al. 1985a; Jensen 1995]. However, as colored Petri nets do not define the syntax for the annotations appearing in the net, automatic detection of data symmetries is difficult. For instance, in the current version of the Design/CPN tool, the user must (i) provide the symmetries in some form, (ii) check that they actually are symmetries, and (iii) write functions that check whether two markings are equivalent [Jørgensen and Kristensen 1998]. This approach requires a considerable amount of expertise of the user.

Another kind of approach is taken in well-formed nets [Chiola et al. 1991] and in the Murφ verification system [Ip and Dill 1996], in which the type system and the data manipulation operations are defined. In these formalisms, some primitive data types can be declared to be "permutable". The permutations of the domains of such data types produce corresponding permutations in the state space. In order to ensure that the produced permutations are symmetries, the set of data manipulation operations applicable on the permutable primitive types is restricted. For instance, in the Murφ system, a data type can be declared to be a "scalar set" whose values can be freely permuted but it is not allowed to compare whether an element is smaller than another. Therefore, the symmetry exploitation process in these formalisms is much simpler: (i) the user declares some primitive data types to be permutable, and (ii) the state space analyzer tool checks that only allowed operations are used on these types, and employs general purpose algorithms for the orbit problems during the reduced reachability graph generation.

In this thesis, defining and detecting data symmetries of high-level Petri nets are studied in the context of algebraic system nets (ASNs) [Kindler and Völzer 1998; 2001], which have the advantage of offering a framework for defining both the syntax and the semantics of the type system and the data manipulation operations appearing as annotations in the nets. In this sense, they are a special case of colored Petri nets. On the other hand, as the syntax and semantics are not permanently fixed, ASNs are more flexible than well-formed nets. The contributions are:

– A general, abstract framework is developed for defining how the domains of the data types appearing in an ASN can be permuted, and how these domain permutations act on the markings and transition in-

stances, i.e., on the state space of the net. A sufficient compatibility condition is defined between the domain permutations and the annotations in the net, ensuring that the induced state space permutations are indeed symmetries. The computational complexity of checking this condition is analyzed. Since it turns out to be **co-NP**-complete even for very simple cases, an approximation rule for the compatibility condition is additionally given.

– The use of the abstract framework is illustrated by defining a high-level Petri net class called *extended well-formed nets* (in short, EWF-nets). The EWF-nets have a very rich data type system, including many common structured data types such as lists, association arrays, and sets, making them quite practical for modeling systems employing data manipulation. The data symmetries of EWF-nets are defined in a way similar to well-formed nets and the Murφ system described above, i.e., by declaring some primitive data types to be permutable. The compatibility of the operations that can be applied to the data types is analyzed by using the approximation rule defined in the abstract framework.

These results have been reported previously in [Junttila 1998; 1999b; 1999a].

Algorithms for Data Symmetries. Finally, several algorithms for deciding state equivalence and for building representative states under data symmetries are proposed. The class of systems studied is so abstract that it covers the Murφ system formalism, well-formed nets (both the standard and extended ones), and the most commonly used instances of colored Petri nets. Basically, the states of the systems in this class are vectors of typed state variables, the type system being the same as in the extended well-formed nets described above.

One of the proposed algorithms exploits a mapping that transforms states into corresponding characteristic graphs. The mapping is originally introduced in [Junttila 1999a] for determining the computational complexity of deciding whether two states are equivalent. Given a state, a canonical representative for it can be obtained by first transforming it into the corresponding graph. The canonical version of the graph is then obtained by applying a tool such as *nauty* [McKay 1990]. Finally, the canonical representative for the state is obtained by using an isomorphism mapping the graph to its canonical version.

In the second algorithm family, ordered partitions of the permutable primitive type elements are built for states in a symmetry-respecting way. The partitions can then be utilized to limit the set of symmetries that have to be considered when comparing whether two states are equivalent or when building a representative for a state. Building and exploiting partitions of this kind is not a new idea but already used in [Huber et al. 1985b; Jensen 1995; Ip 1996; Sistla et al. 2000]. However, in this work (i) the partition building process is formally defined, (ii) both freely and cyclically permutable primitive data types are handled in a uniform way, and (iii) some very expressive invariants, needed in the partition building process, are proposed. For instance, an invariant that can handle all the considered data types is developed, and other, highly efficient, invariants are proposed for data types of special forms. Furthermore, a novel improvement based on considering a

partition refinement tree, adapted from the algorithms for obtaining canonical forms of graphs [McKay 1981; Kreher and Stinson 1999], is described.

Some of the proposed algorithms have been implemented in the Murφ tool and the experimental results show that they are competitive against the previous ones described in [Ip 1996].

These results have been reported previously in [Junttila 2002b].

1.2 OTHER RELATED WORK

In addition to the related work already mentioned (which will be further discussed during the thesis), there is a lot of other related work in the literature concerning the symmetry reduction method.

Most of the work mentioned above concentrates on verifying only simple properties such as deadlock freedom, non-reachability of (symmetric) bad states and some Petri net related properties such as the home state property. Verification of more complex properties by combining temporal logic model checking and symmetries is introduced in [Clarke et al. 1993; Clarke et al. 1996; Emerson and Sistla 1993; 1996]. In [Clarke et al. 1993; Clarke et al. 1996] it is shown that reduced reachability graphs can be used in model checking CTL* formulae, provided that the atomic propositions appearing in the formula are invariant under the applied symmetry group. In [Emerson and Sistla 1993; 1996] a stronger result is obtained, requiring only that certain subformulae are invariant under the symmetries. In addition, [Emerson and Sistla 1993; 1996] describe an automata theoretic approach for model checking asymmetric properties, see also [Sistla and Godefroid 2001]. In this approach, the reduced reachability graph is partially unwound by adding some additional information making it possible to track how the atomic propositions are permuted. The approach is extended to handle fairness in [Emerson and Sistla 1995; 1997], and a further improved on-the-fly version of it is described in [Gyuris and Sistla 1997; 1999]. Furthermore, [Ajami et al. 1998] describes an approach exploiting the symmetries of the Büchi automaton corresponding to (the negation of) the verified property. Using the nested depth-first search algorithm in model checking under symmetries is discussed in [Bošnački 2002a].

In [Bošnački et al. 2000; 2001; 2002], the Spin model checker [Holzmann 1997] is extended to handle symmetries produced by symmetric use of data and replicated processes. Algorithms for producing representative states are proposed. The algorithms are, in a sense, simple modified versions of the partition based algorithms in the Murφ tool [Ip 1996]. A similar approach is presented in [Derepas and Gastin 2001], where the input language of Spin is also extended with new keywords in order to automatically detect symmetries produced by replicated processes. The symmetries produced by dynamic creation of objects in object-based programs are discussed in [Iosif 2001; 2002]. [Iosif 2001] gives an efficient algorithm for producing canonical representative states in the presence of such symmetries, while [Iosif 2002] considers combining such symmetries with the symmetries produced by replicated processes. The symmetries produced by class loading and object allocation in the Java language are also described in [Lerda and Visser 2001]. A heuris-

tic is proposed for producing representative states, trying to always allocate the same object (class) in the same position independently of the execution order of concurrent threads.

In the context of hardware verification, symmetries and the use of Binary Decision Diagrams (BDDs) is combined in [Clarke et al. 1996]. The complexity and algorithms for solving the orbit problems by using BDDs is discussed. It is shown that the BDD for the orbit relation can be exponentially large, and the use of multiple representative states is suggested for avoiding this problem. See also [Clarke et al. 1998] for further complexity analysis and discussion on the use of multiple representatives. [Barner and Grumberg 2002] proposes another approach for avoiding the BDD explosion problem. [Wang and Schmidt 2002] describes an approach for verifying concurrent software with pointer data structures. The approach uses BDD-like data structures for storing sets of states and also exploits symmetries produced by replicated processes. In [Manku et al. 1998] it is described how structural symmetries of hardware designs and temporal logic formulae can be found and utilized. Another approach for exploiting symmetries during hardware verification is presented in [Pandey and Bryant 1999], where the symmetries are used to prune the number of properties that have to be checked.

The symmetry reduction and partial order methods can be combined. In [Valmari 1991] it is shown that a combination preserves the existence of dead markings and infinite paths (see also [Tiusanen 1994]). It seems that the proof of this does not require the use of canonical representatives, meaning that approximation by using non-canonical representatives is allowed. [Emerson et al. 1997] describes a combination that preserves next-operator free temporal logic formulae whose atomic propositions are invariant under the applied symmetry group. However, the proof requires canonical representative states, i.e., approximation by using non-canonical representatives is not considered (the same seems to apply to the combination presented in [Iosif 2002]). [Bošnački 2002b] shows that it is actually possible to use non-canonical representatives.

The symmetry reduction method has also been applied to certain systems that are only "almost", or partially, symmetric. For instance, a system of n communicating processes may be otherwise symmetric except that the processes have different priorities when entering in the mutual exclusion section. [Haddad et al. 1995] studies reachability analysis of partially symmetric well-formed nets by adding information in the symbolic markings in order to handle asymmetric transitions. In [Haddad et al. 2000], asymmetric systems are verified by "moving" the asymmetries in the system into the automata to be model checked. In [Emerson and Trefler 1999; Emerson et al. 2000], the asymmetry problem is handled by defining weaker symmetry conditions for the transition relation of the system. These conditions still ensure that the symmetry reduced state space preserves temporal logic formulae with symmetry invariant atomic propositions. [Sistla and Godefroid 2001] presents an approach that allows symmetry reductions while model checking asymmetric temporal logic formulae in asymmetric systems. It this approach, the symmetry reduced reachability graph is unwound with respect to both the sub-formulae of the verified property and some transition predicates that capture the asymmetries of the system.

Symmetries can also be exploited in the performance analysis of systems. A stochastic version of well-formed nets [Chiola et al. 1991] is introduced in [Chiola et al. 1993]. Furthermore, [Emerson and Trefler 1998] gives a model checking procedure for a real-time Mu-calculus.

Finally, a quite different approach is taken in [Godefroid 1999]. In the approach, software systems written in full-fledged programming languages, such as C,C++ or Java, are model checked without having an explicit encoding for states by using the state-less search method [Godefroid 1997]. Because the encoding for states is not available in the approach, the equivalence of states is not directly exploited but the corresponding equivalence of transition sequences is used instead.

1.3 ORGANIZATION

This thesis is organized as follows.

Chapter 2 gives the necessary preliminaries. The symmetry reduction method is explained to the extent needed in this work and some other preliminary definitions are also given.

Chapters 3 and 4 discuss the symmetry reduction method for place/transition nets. Chapter 3 gives the basic definitions and studies the computational complexity of the sub-tasks in the symmetry reduction method, while Chapter 4 describes new algorithms for the orbit problems.

Chapters 5 and 6 discuss the data symmetries of algebraic system nets. First, Chapter 5 gives an abstract framework for defining data symmetries. Chapter 6 then illustrates the use of the framework by defining the class of high-level Petri nets called extended well-formed nets.

Chapter 7 proposes several algorithms for deciding state equivalence and for building representative states under data symmetries.

Finally, Chapter 8 concludes the thesis with some possible future research topics.

2 PRELIMINARIES

This chapter introduces the state space analysis and the symmetry reduction method to the extent needed in this thesis. The presentation is based on the papers discussed in Sections 1.1 and 1.2. At the end of the chapter, some other preliminaries relevant to the thesis are also defined.

2.1 STATE SPACE ANALYSIS

Consider a system given in a system description formalism. The semantics of the formalism describe the state space of the system, consisting of the set of all possible states the system may have, the transitions transforming the states to others, and the initial state of the system. Formally, the *state space* of the system is a labeled transition system (LTS)

$$\mathcal{L} = \langle Q, L, \Delta, q_{init} \rangle,$$

where

1. Q is a non-empty set of *states*,
2. L is a non-empty set of *transition names* (transition labels) such that $Q \cap L = \emptyset$,
3. $\Delta \subseteq Q \times L \times Q$ is the *transition relation*, and
4. $q_{init} \in Q$ is the *initial state*.

One may use $q_1 \xrightarrow{l} q_2$ to abbreviate $\langle q_1, l, q_2 \rangle \in \Delta$, i.e., the fact that executing the transition l in the state q_1 leads to the state q_2. A transition $l \in L$ is *enabled* in a state $q_1 \in Q$, denoted by $q_1 \xrightarrow{l}$, if there is a state $q_2 \in Q$ such that $q_1 \xrightarrow{l} q_2$. A state q_1 is a *deadlock* state if no transition is enabled in it. A *path* is a (finite of infinite) sequence $q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \ldots$ of states and transitions such that $q_i \xrightarrow{l_i} q_{i+1}$ holds for each i. A state q' is *reachable from a state* q if there is a finite path starting in q and ending in q'. A state is *reachable* if it is reachable from the initial state. The *reachability graph* of an LTS $\mathcal{L} = \langle Q, L, \Delta, q_{init} \rangle$ is the LTS

$$\mathcal{RG}(\mathcal{L}) = \langle \vec{Q}, L, \vec{\Delta}, q_{init} \rangle,$$

where $\vec{Q} \subseteq Q$ and $\vec{\Delta}$ are inductively defined by the following rules.

1. $q_{init} \in \vec{Q}$,
2. if $q_1 \in \vec{Q}$ and $\langle q_1, l, q_2 \rangle \in \Delta$, then $q_2 \in \vec{Q}$ and $\langle q_1, l, q_2 \rangle \in \vec{\Delta}$, and
3. nothing else is in $\vec{Q}$ or in $\vec{\Delta}$.

That is, the reachability graph is the subgraph of the state space containing exactly all the reachable states and the transitions between them. In other words, it describes all the possible behaviors the system may have when started in the initial state. The standard algorithm for computing reachability graphs is shown in Algorithm 2.1.

Algorithm 2.1 An algorithm for computing reachability graphs

1: Set $unprocessed = \{q_{init}\}$
2: Set $\vec{Q} = \{q_{init}\}$
3: Set $\vec{\Delta} = \emptyset$
4: **while** $unprocessed \neq \emptyset$ **do**
5: Take any $q \in unprocessed$ and set $unprocessed = unprocessed \setminus \{q\}$
6: **for all** $q \xrightarrow{l} q'$ **do**
7: Set $\vec{\Delta} = \vec{\Delta} \cup \langle q, l, q' \rangle$
8: **if** $q' \notin \vec{Q}$ **then**
9: Set $unprocessed = unprocessed \cup \{q'\}$
10: Set $\vec{Q} = \vec{Q} \cup \{q'\}$
11: **return** $\mathcal{RG}(\mathcal{L}) = \langle \vec{Q}, L, \vec{\Delta}, q_{init} \rangle$

Example 2.1 Consider the very simple program for the mutual exclusion problem with P processes shown in Figure 2.1, described in an informal guarded command style programming language. The program consists of a shared state variable s_i with the domain $\{N, T, C\}$ for each process i in the index set $I = \{1, \ldots, P\}$. The value N denotes the non-critical section, T the trying section, and C the critical section. The variables are manipulated by the P asynchronous processes whose transitions are given by imposing pre- and postconditions on variables. For instance, the transition t_1 of process 1 is enabled if the state variable s_1 is set to N. When t_1 is enabled and executed, the state variable s_1 is assigned to the value T in the next state.

The set of states of the program is $Q = [I \to \{N, T, C\}]$, i.e., the set of all functions from the index set I to $\{N, T, C\}$. A state $s : I \to \{N, T, C\}$ can also be denoted by the vector $\langle s(1), \ldots, s(N) \rangle$, i.e., the first element describes the value of the variable s_1, the second element describes the value of s_2, and so on. Transition labels are the transitions names for each process, $L = \bigcup_{i \in I} \{t_i, e_i, l_i\}$, e.g., $L = \{t_1, e_1, l_1, t_2, e_2, l_2\}$ for $P = 2$, and the transition relation is defined by the program. The state space $\mathcal{L}$ of the program for $P = 2$ is shown in Figure 2.2. The state $\langle N, N \rangle$ is the initial state, pointed out by an arrow originating nowhere. The reachability graph of $\mathcal{L}$ is the $\mathcal{L}$ itself except the state $\langle C, C \rangle$ and the arcs originating from it. Thus the "bad" state $\langle C, C \rangle$ in which both of the processes are in the critical section is not reachable. ♣

Let $I = \{1, \ldots, P\}$ be the *index set*
State variables:
s_i with domain $\{N, T, C\}$ for each $i \in I$, initialized to N
Transitions for each $i \in I$:
t_i: $s_i = N \to s_i' = T$
e_i: $(s_i = T \wedge \forall_{j \in I} s_j \neq C) \to s_i' = C$
l_i: $s_i = C \to s_i' = N$

Figure 2.1: A simple mutual exclusion program for P processes

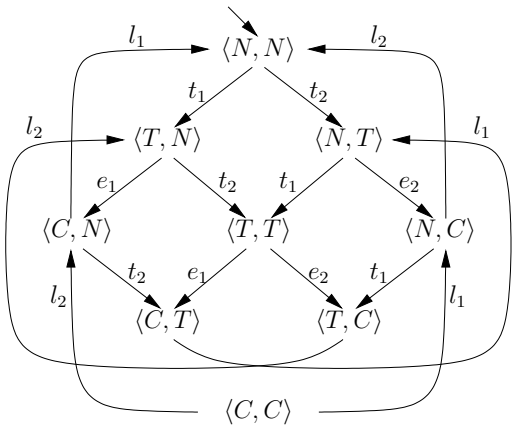


Figure 2.2: The state space of the program in Figure 2.1 for $P = 2$

Obviously, checking whether a system may enter a deadlock state or a state fulfilling some property can be easily performed on its reachability graph. In addition to these kind of properties, more general ones can be specified and verified by applying temporal logic model checking. In the following, the very expressive branching time temporal logic CTL* is briefly reviewed. For more details on temporal logic model checking, refer to [Clarke et al. 1999]. First, a set AP of *atomic propositions* is assumed. The truth of the atomic propositions in the states of the system is given by a truth valuation function $\mu : Q \to \wp(AP)$, where $\wp(AP)$ denotes the set of all subsets of AP (the power set of AP). An atomic proposition $p \in AP$ is said to hold in a state q if $p \in \mu(q)$. The formulae in CTL* are defined by the following grammar when started in the nonterminal f:

$$\begin{aligned} f &::= p \mid \mathbf{E}g \mid \mathbf{A}g \mid f \vee f \mid f \wedge f \mid \neg f \\ g &::= f \mid g \vee g \mid g \wedge g \mid \neg g \mid \mathbf{X}g \mid \mathbf{F}g \mid \mathbf{G}g \mid g\ \mathbf{U}\ g \mid g\ \mathbf{V}\ g, \end{aligned}$$

where p ranges over AP. The f formulae are called state formulae and the g are path formulae. The operators **E** and **A** are the existential and universal path quantifiers, while **X**, **F**, **G**, **U**, and **V** are the temporal operators "next", "finally", "globally", "until" and "release", respectively. One writes $\mathcal{L}, q \vDash f$ to denote that the (state) formula f holds in the state q (for the definition of $\vDash$, see [Clarke et al. 1999]). The model checking problem is: given a CTL* formula f, does it hold in the initial state of the system? For algorithms solving this problem, see [Clarke et al. 1999]. The linear time temporal logic LTL is the sublogic of CTL* in which each formula is of form $\mathbf{A}g$, where g is a path formula not involving any **E** or **A** operators.

Example 2.2 Recall the system discussed in Example 2.1 and its state space shown in Figure 2.2. Assume the atomic propositions N_i, T_i and C_i for each process i in $\{1, \ldots, P\}$. Define that the atomic proposition N_i holds in a state s if and only if $s(i) = N$, and similarly for T and C. Now the LTL property $\mathbf{AG}\neg(C_1 \wedge C_2)$ states that for all the paths starting in the initial state it always holds that the processes 1 and 2 are not simultaneously in their critical sections, i.e., it is the mutual exclusion property for the two process case $P = 2$. The property holds in the system because the state $\langle C, C\rangle$ is

not reachable. Similarly, the property $\mathbf{AG}(T_1 \Rightarrow \mathbf{F}C_1)$ states that for all the paths starting in the initial state it always holds that if the process 1 is in its trying section, then it will finally get in its critical section ($a \Rightarrow b$ is the standard abbreviation for $\neg a \vee b$). It does not hold in the system because in the infinite path $\langle N, N\rangle \xrightarrow{t_1} \langle T, N\rangle \xrightarrow{t_2} \langle T, T\rangle \xrightarrow{e_2} \langle T, C\rangle \xrightarrow{l_2} \langle T, N\rangle \xrightarrow{t_2} \cdots$ the process 1 always stays in the trying section but never enters the critical section. ♣

2.2 THE SYMMETRY REDUCTION METHOD

As its name implies, the symmetry reduction method exploits the symmetries in the state space of a system. If two states are equivalent under the symmetries, then the behaviors starting from them are also equivalent. For many verification tasks, such equivalent states can be identified, meaning that only one or few states from each, possibly very large, set of mutually equivalent states need to be considered during the state space analysis. This section describes the symmetry reduction method to the extend needed in this thesis.

Formally, a *symmetry* (or an *automorphism*) of a state space LTS $\mathcal{L} = \langle Q, L, \Delta, q_{init}\rangle$ is a permutation π of $Q \cup L$ that respects

- the sets of states and transition names: $\pi(Q) = Q$ and $\pi(L) = L$, and
- the transition relation: $\langle q_1, l, q_2\rangle \in \Delta \Leftrightarrow \langle \pi(q_1), \pi(l), \pi(q_2)\rangle \in \Delta$.

It is straightforward to see that the composition $\pi_1 \circ \pi_2$ of two state space symmetries as well as the inverse π^{-1} of a state space symmetry are also state space symmetries.[1] Furthermore, the set of *all state space symmetries* of $\mathcal{L}$ (the *automorphism group of* $\mathcal{L}$) is denoted by $\mathrm{Aut}(\mathcal{L})$ and forms a group under the function composition operation $\circ$.

Take any subgroup G of $\mathrm{Aut}(\mathcal{L})$. Two states, q_1 and q_2, are *equivalent* under G if there is a state space symmetry $\pi \in G$ such that $\pi(q_1) = q_2$. This is denoted by $q_1 \equiv_G q_2$. Since G is a permutation group, $\equiv_G$ is an equivalence relation on $Q \cup L$ and the equivalence class of a state q, defined by $[q]_G = \{\pi(q) \mid \pi \in G\}$, is called the *$G$-orbit* of q. It is easy to see directly from the definition of state space symmetries that, for each $\pi \in G$, $q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \cdots$ is a path in $\mathcal{L}$ if and only if $\pi(q_1) \xrightarrow{\pi(l_1)} \pi(q_2) \xrightarrow{\pi(l_2)} \cdots$ is. That is, equivalent states have equivalent future behaviors. Furthermore, a state q is a deadlock state if and only if $\pi(q)$ is.

A state space symmetry π *stabilizes* (fixes) a state q if $\pi(q) = q$. The set of all symmetries in a subgroup G of $\mathrm{Aut}(\mathcal{L})$ stabilizing a state q, $\mathrm{Stab}(G, q) = \{\pi \in G \mid \pi(q) = q\}$, is the *stabilizer subgroup* of q in G. Assume that a symmetry π stabilizes a state q. Now $q \xrightarrow{l_1} q_1 \cdots \xrightarrow{l_n} q_n$ implies $q \xrightarrow{\pi(l_1)} \pi(q_1) \cdots \xrightarrow{\pi(l_n)} \pi(q_n)$ and thus the state $\pi(q_n)$ is reachable from q if the state q_n is. Furthermore, since $\pi(q) = q$ implies $\pi^{-1}(q) = q$, the state $\pi(q_n)$ is reachable from q if and only if the state q_n is. This is why it is sometimes required that each applied symmetry stabilizes the initial state, i.e., the stabi-

[1] The composition $f \circ g$ of two functions (including permutations) is evaluated from right to left in this work, i.e., $(f \circ g)(x) = f(g(x))$.

lizer subgroup $\mathrm{Stab}(\mathrm{Aut}(\mathcal{L}), q_{init})$ of the initial state (or any subgroup of it) is considered instead of $\mathrm{Aut}(\mathcal{L})$.

Example 2.3 Recall the program in Figure 2.1 and its state space $\mathcal{L}$ (for $P = 2$) in Figure 2.2, discussed in Example 2.1. The set of automorphisms of $\mathcal{L}$, $\mathrm{Aut}(\mathcal{L})$, consists of two permutations of $Q \cup L$: the identity permutation **I** and the permutation

$$\pi = \begin{pmatrix} \langle N,N\rangle & \langle T,N\rangle & \langle N,T\rangle & \langle C,N\rangle & \langle T,T\rangle & \langle N,C\rangle & \langle C,T\rangle & \langle T,C\rangle & \langle C,C\rangle & t_1 & e_1 & l_1 & t_2 & e_2 & l_2 \\ \langle N,N\rangle & \langle N,T\rangle & \langle T,N\rangle & \langle N,C\rangle & \langle T,T\rangle & \langle C,N\rangle & \langle T,C\rangle & \langle C,T\rangle & \langle C,C\rangle & t_2 & e_2 & l_2 & t_1 & e_1 & l_1 \end{pmatrix}.$$

Intuitively, the latter permutation corresponds to the swapping of process identities. The states $\langle T, N\rangle$ and $\langle N, T\rangle$ are equivalent under $\mathrm{Aut}(\mathcal{L})$ and the orbit of $\langle T, N\rangle$ is $[\langle T, N\rangle]_{\mathrm{Aut}(\mathcal{L})} = \{\langle T, N\rangle, \langle N, T\rangle\}$. Note that both of the symmetries stabilize the initial state $\langle N, N\rangle$. Now $\langle N, N\rangle \xrightarrow{t_2} \langle N, T\rangle \xrightarrow{t_1} \langle T, T\rangle$ is a path in the state space and so is its π-equivalent $\pi(\langle N, N\rangle) \xrightarrow{\pi(t_2)} \pi(\langle N, T\rangle) \xrightarrow{\pi(t_1)} \pi(\langle T, T\rangle)$, i.e., $\langle N, N\rangle \xrightarrow{t_1} \langle T, N\rangle \xrightarrow{t_2} \langle T, T\rangle$. ♣

2.2.1 Finding State Space Symmetries

Since the goal of the symmetry reduction method is to avoid enumerating the entire state space or the whole reachability graph, the state space symmetries must be found without explicitly using the state space itself. This is achieved by defining a group G on the system description level that then acts on the state space level in a way that produces state space symmetries. In symmetry reduction algorithms, one never explicitly uses a state space symmetry group but a system description level group producing it.

Formally, an *action* of a group G (under a binary operation $*$) on a set X is a function $h : G \times X \to X$ such that for all $x \in X$ it holds that

1. $h(\iota, x) = x$, where ι is the identity element of G, and
2. $h(g_1 * g_2, x) = h(g_1, h(g_2, x))$ for all $g_1, g_2 \in G$.

For each $g \in G$, define the function $g_h : X \to X$ by $g_h(x) = h(g, x)$. By standard group theory, the set $G_h = \{g_h \mid g \in G\}$ is a subgroup of $\mathrm{Sym}(X)$ and thus each g_h is a permutation of X.[2] When the action h is understood from the context, one may simply write $g(x)$ instead of $g_h(x)$ and G instead of G_h, whenever no confusion can arise.

Now the problem of finding state space symmetries consists of finding a group G on the system description level whose action on the set $Q \cup L$ of states and transition labels is a subgroup of $\mathrm{Aut}(\mathcal{L})$. Of course, the form of the group G and the action depend on the applied system description formalism. For instance, in place/transition nets discussed in Chapters 3 and 4, the symmetries of the net itself produce corresponding symmetries to its state space (see Section 3.1). On the other hand, in the classes of high-level nets studied in Chapters 5 and 6, as well as in the Murφ system description formalism [Ip and Dill 1996], state space symmetries are produced by permuting the values of state variables.

[2] $\mathrm{Sym}(X)$ denotes the group of all permutations of the set X under the function composition operator $\circ$.

Example 2.4 Recall the program in Figure 2.1 discussed in Examples 2.1 and 2.3. Let $\mathrm{Sym}(I)$ denote the group of all permutations of the index set $I = \{1, \ldots, P\}$. For instance,

$$\mathrm{Sym}(\{1,2\}) = \{g_{2,1} = \left(\begin{smallmatrix}1&2\\1&2\end{smallmatrix}\right), g_{2,2} = \left(\begin{smallmatrix}1&2\\2&1\end{smallmatrix}\right)\}$$

and

$$\begin{aligned}\mathrm{Sym}(\{1,2,3\}) = \{&g_{3,1} = \left(\begin{smallmatrix}1&2&3\\1&2&3\end{smallmatrix}\right), g_{3,2} = \left(\begin{smallmatrix}1&2&3\\1&3&2\end{smallmatrix}\right), g_{3,3} = \left(\begin{smallmatrix}1&2&3\\2&1&3\end{smallmatrix}\right),\\ &g_{3,4} = \left(\begin{smallmatrix}1&2&3\\2&3&1\end{smallmatrix}\right), g_{3,5} = \left(\begin{smallmatrix}1&2&3\\3&1&2\end{smallmatrix}\right), g_{3,6} = \left(\begin{smallmatrix}1&2&3\\3&2&1\end{smallmatrix}\right)\}.\end{aligned}$$

Define the action h of the group $G = \mathrm{Sym}(\{1, \ldots, P\})$ on the set $Q = [\{1, \ldots, P\} \to \{N, T, C\}]$ of states by $h(g, s) = s \circ g^{-1}$ and on the transition labels $L = \bigcup_{i \in I}\{t_i, e_i, l_i\}$ by $h(g, y_i) = y_{g(i)}$, where $y \in \{t, e, l\}$ and $i \in I$. For instance, when $P = 2$, $g_{2,2}(\{1 \mapsto T, 2 \mapsto N\}) = \{1 \mapsto N, 2 \mapsto T\}$ and $g_{2,2}(e_1) = e_2$. In fact, (under the action h) $g_{2,2}$ corresponds to the automorphism π described in Example 2.3. Similarly, if $P = 3$, then $g_{3,5}(\{1 \mapsto N, 2 \mapsto T, 3 \mapsto C\}) = \{1 \mapsto T, 2 \mapsto C, 3 \mapsto N\}$ and $g_{3,5}(e_1) = e_3$.

To see that h is a group action on $Q \cup L$, notice that $h(\mathbf{I}, s) = s \circ \mathbf{I}^{-1} = s$ and $h(\mathbf{I}, y_i) = y_{\mathbf{I}(i)} = y_i$ for the identity permutation $\mathbf{I}$, and

$$\begin{aligned} h(g \circ g', s) &= s \circ (g \circ g')^{-1} = s \circ (g'^{-1} \circ g^{-1}) = (s \circ g'^{-1}) \circ g^{-1} \\ &= h(g, s \circ g'^{-1}) = h(g, h(g', s)), \text{ and} \\ h(g \circ g', y_i) &= y_{(g \circ g')(i)} = y_{g(g'(i))} = h(g, y_{g'(i)}) = h(g, h(g', y_i)) \end{aligned}$$

for all $g, g' \in G$. Similarly, the fact that the action of $G = \mathrm{Sym}(I)$ on $Q \cup L$ is an automorphism group, i.e., that G_h is a subgroup of $\mathrm{Aut}(\mathcal{L})$, can be verified by examining the transitions of the program. ♣

2.2.2 Reduced Reachability Graphs

After finding a group of state space symmetries, the next step is to exploit them during the reachability graph generation. This is done by identifying the states in each orbit, to goal being to examine only one (or few) state(s) in each orbit.

Let G be a subgroup of $\mathrm{Aut}(\mathcal{L})$ for a state space LTS $\mathcal{L} = \langle Q, L, \Delta, q_{init}\rangle$. A *reduced reachability graph* (an RRG) of $\mathcal{L}$ under G is an LTS

$$\mathcal{RRG}(\mathcal{L}) = \langle \tilde{Q}, L, \tilde{\Delta}, q'_{init}\rangle,$$

such that (i) $q_{init} \equiv_G q'_{init}$ and (ii) $\tilde{Q} \subseteq Q$ and $\tilde{\Delta} \subseteq \tilde{Q} \times L \times \tilde{Q}$ fulfill the following rules.

1. $q'_{init} \in \tilde{Q}$,
2. if $q \in \tilde{Q}$ and $\langle q, l, q_1\rangle \in \Delta$, then $q'_1 \in \tilde{Q}$ and $\langle q, l, q'_1\rangle \in \tilde{\Delta}$ for a q'_1 such that $q_1 \equiv_G q'_1$, and
3. if $\langle q, l, q_1\rangle \in \tilde{\Delta}$, then (i) $\langle q, l, q'_1\rangle \in \Delta$ for a q'_1 such that $q_1 \equiv_G q'_1$, and (ii) q is reachable from q'_{init} in $\mathcal{RRG}(\mathcal{L})$.

That is, the initial state q'_{init} of the RRG is equivalent to the initial state q_{init} of the LTS $\mathcal{L}$, and each transition $q \xrightarrow{l} q_1$ originating from a state q in the

RRG is "redirected" to an equivalent successor state by having the transition $q \xrightarrow{l} q_1'$ for a $q_1' \equiv_G q_1$ in the RRG. The third rule in the definition ensures that every edge in the RRG is a result of applying the second rule, i.e., that there are no unjustified edges in the RRG. Note the indefinite article in the second rule of the definition. This implies that there may be several RRGs for $\mathcal{L}$ under G. The obvious algorithm for generating RRGs, derived from Algorithm 2.1, is shown in Algorithm 2.2.

Algorithm 2.2 An algorithm for computing reduced reachability graphs

```
1:  Choose any q'_init such that q_init ≡_G q'_init
2:  Set unprocessed = {q'_init}
3:  Set Q̃ = {q'_init}
4:  Set Δ̃ = ∅
5:  while unprocessed ≠ ∅ do
6:     Take any q ∈ unprocessed and set unprocessed = unprocessed \ {q}
7:     for all q -l-> q' do
8:        Choose any q'' such that q' ≡_G q''
9:        Set Δ̃ = Δ̃ ∪ ⟨q, l, q''⟩
10:       if q'' ∉ Q̃ then
11:          Set unprocessed = unprocessed ∪ {q''}
12:          Set Q̃ = Q̃ ∪ {q''}
13: return RRG(L) = ⟨Q̃, L, Δ̃, q'_init⟩
```

Example 2.5 Recall the program in Figure 2.1 discussed in Examples 2.1, 2.3, and 2.4. Figure 2.3 shows two RRGs for the program when $P = 2$. The one on the left hand side is minimal in the sense that it contains only one state from each reachable orbit. ♣

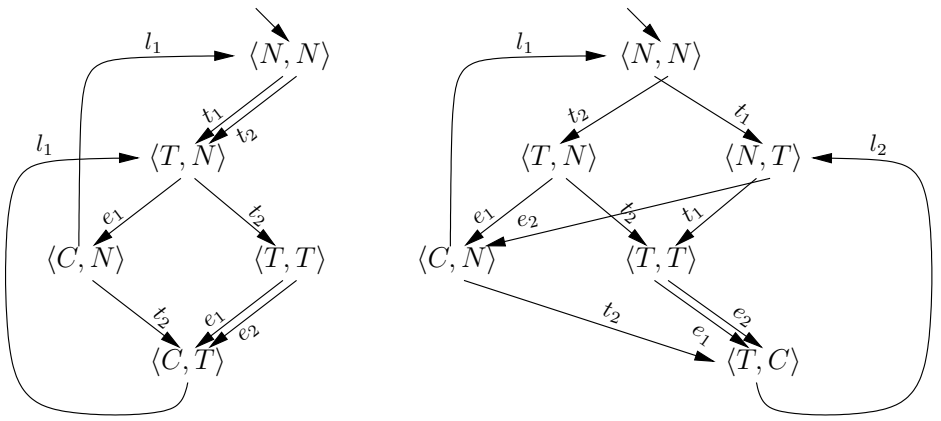


Figure 2.3: Reduced reachability graphs for the program in Figure 2.1 when $P = 2$

The crucial part in Algorithm 2.2 computing RRGs is the line 8, where an equivalent successor state q'' is selected. To obtain as small RRGs as possible, there should be exactly one representative state from each reachable orbit present in the state set of the reduced reachability graph. This goal can be achieved by the following two ways.

1. The new successor state q' is pairwisely compared with each state in the set $\tilde{Q}$ of already visited states. If a state equivalent to q' is found in $\tilde{Q}$, then the new successor state q'' is defined to be that state, otherwise q'' is defined to be q'. In this approach one has to be able to answer the *orbit problem*: given two states, are they equivalent? Symmetry-respecting hash functions can sometimes be used to prune the set of states in $\tilde{Q}$ that have to be checked for equivalence with q'. Nevertheless, it may be the case that the orbit problem has to be answered several times for each new successor state.
2. The successor state q' is transformed into a representative state $repr(q')$ and q'' is defined to be that state. Formally, a *representative function* is a function $repr : Q \to Q$ such that $repr(q) \equiv_G q$ holds for each $q \in Q$. The representative function $repr$ is *canonical* if $q_1 \equiv_G q_2$ implies $repr(q_1) = repr(q_2)$. In this case, $repr(q)$ is the *canonical representative* of q (under $repr$). In this approach, the initial state should be transformed into its representative as well, i.e., the line 1 in the algorithm is modified to "Let $q'_{init} = repr(q_{init})$". The problem of computing a canonical representative for a state is called the *constructive orbit problem*. Note that in this approach, as opposed to the first one, the constructive orbit problem is solved only once for each new successor marking. However, the constructive orbit problem can be computationally harder than the orbit problem.

Since the problems of deciding whether a state is equivalent to another and building canonical representative states are in general at least as hard as the graph isomorphism problem, both of the approaches above contain tasks for which no polynomial time algorithms are currently known. Luckily, the approaches can be approximated by (i) using a sound but incomplete state equivalence test in the first one, and (ii) by using a non-canonical representative function in the second one. Using such an approximation may result in that more than one state in a reachable orbit is visited during the search and thus the reduced reachability graph may not be of minimal size. Hence the space consumption (and sometimes the time consumption, too) may grow compared to the complete approach.

2.2.3 Analysis of Reduced Reachability Graphs

In the following, some basic properties of reduced reachability graphs are given.

Assume a state space LTS $\mathcal{L} = \langle Q, L, \Delta, q_{init} \rangle$, its reachability graph $\mathcal{RG}(\mathcal{L}) = \langle \vec{Q}, L, \vec{\Delta}, q_{init} \rangle$, a subgroup G of $\mathrm{Aut}(\mathcal{L})$, and a reduced reachability graph $\mathcal{RRG}(\mathcal{L}) = \langle \tilde{Q}, L, \tilde{\Delta}, q'_{init} \rangle$ of $\mathcal{L}$ under G. Let $q_1 \in \vec{Q}$ be a state in the reachability graph and $q'_1 \in \tilde{Q}$ be a state in the reduced reachability graph such that $q_1 \equiv_G q'_1$. For instance, q_1 could be the initial state q_{init} and q'_1 its representative q'_{init} in the reduced reachability graph. Let π be a symmetry in G that maps q_1 to q'_1. The following two lemmas and their corollaries show that the behaviors starting from q_1 and q'_1 are equivalent.

Lemma 2.6 *If $\langle q_1, l_1, q_2 \rangle \in \vec{\Delta}$ is a transition in the reachability graph, then there is a transition $\langle q'_1, l'_1, q'_2 \rangle \in \tilde{\Delta}$ in the RRG such that $l_1 \equiv_G l'_1$ and*

$q_2 \equiv_G q_2'$.

Proof. Because $\langle q_1, l_1, q_2 \rangle \in \vec{\Delta}$, it also holds that $\langle q_1, l_1, q_2 \rangle \in \Delta$. As π is a state space symmetry, $\langle \pi(q_1), \pi(l_1), \pi(q_2) \rangle \in \Delta$ holds, too. Because $\pi(q_1) = q_1' \in \tilde{Q}$, $\langle \pi(q_1), \pi(l_1), \pi(q_2) \rangle \in \Delta$ implies that $\langle \pi(q_1), \pi(l_1), \pi_1(\pi(q_2)) \rangle \in \tilde{\Delta}$ and $\pi_1(\pi(q_2)) \in \tilde{Q}$ for a $\pi_1 \in G$ by the rule 2 in the definition of RRGs. □

Lemma 2.7 *If $\langle q_1', l_1', q_2' \rangle \in \tilde{\Delta}$ is a transition in the RRG, then there is a transition $\langle q_1, l_1, q_2 \rangle \in \vec{\Delta}$ in the reachability graph such that $l_1 \equiv_G l_1'$ and $q_2 \equiv_G q_2'$.*

Proof. As $\langle q_1', l_1', q_2' \rangle \in \tilde{\Delta}$, $\langle q_1', l_1', \pi_1(q_2') \rangle \in \Delta$ for a $\pi_1 \in G$ by the rule 3 in the definition of RRGs. This implies that $\langle \pi^{-1}(q_1'), \pi^{-1}(l_1'), \pi^{-1}(\pi_1(q_2')) \rangle \in \Delta$ because π^{-1} is a state space symmetry. Because $\pi^{-1}(q_1') = q_1 \in \vec{Q}$, $\langle q_1, \pi^{-1}(l_1'), \pi^{-1}(\pi_1(q_2')) \rangle \in \vec{\Delta}$ and $\pi^{-1}(\pi_1(q_2')) \in \vec{Q}$. □

Corollary 2.8 *If $q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} q_3 \cdots$ is a path in the reachability graph, then there is a path $q_1' \xrightarrow{l_1'} q_2' \xrightarrow{l_2'} q_3' \cdots$ in the RRG such that $l_i \equiv_G l_i'$ and $q_i \equiv_G q_i'$ for each i.*

Corollary 2.9 *If $q_1' \xrightarrow{l_1'} q_2' \xrightarrow{l_2'} q_3' \cdots$ is a path in the RRG, then there is a path $q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} q_3 \cdots$ in the reachability graph such that $l_i \equiv_G l_i'$ and $q_i \equiv_G q_i'$ for each i.*

Because the initial states q_{init} and q_{init}' of the reachability graph and the RRG, respectively, are equivalent, the corollaries above imply the following correspondence between the reachable states.

Corollary 2.10 *If a state q is in the reachability graph, then there is a state q' in the RRG such that $q \equiv_G q'$.*

Corollary 2.11 *If a state q' is in the RRG, then there is a state q in the reachability graph such that $q \equiv_G q'$.*

Corollary 2.12 *There is a deadlock state in the reachability graph if and only if there is a deadlock state in the reduced reachability graph.*

Corollary 2.13 *Assume that G stabilizes the initial state q_{init} meaning that $\pi(q_{init}) = q_{init}$ for each $\pi \in G$. Then a state q is in the reachability graph if and only if there is a state q' in the RRG such that $q \equiv_G q'$.*

Proof. The "only if" direction is the same as Corollary 2.10. Assume that a state q' such that $q \equiv_G q'$ is in the RRG. By Corollary 2.11, a state q'' such that $q'' \equiv_G q'$ is in the reachability graph. Recalling the discussion in the beginning of Section 2.2, the state q'' is reachable in the state space (i.e., in the reachability graph) if and only if $\pi(q'')$ is for each $\pi \in G$, provided that G stabilizes the initial state q_{init}. Therefore, q is in the reachability graph as $q \equiv_G q'' \equiv_G q'$. □

Example 2.14 Recall the program in Figure 2.1, its reachability graph in Figure 2.2, and the one of its RRGs shown in the right hand side of Figure 2.3. Neither the reachability graph nor the RRG contains a deadlock state. For the path $\langle N, N\rangle \xrightarrow{t_2} \langle N, T\rangle \xrightarrow{e_2} \langle N, C\rangle$ in the reachability graph, there is an equivalent path $\langle N, N\rangle \xrightarrow{t_2} \langle T, N\rangle \xrightarrow{e_1} \langle C, N\rangle$ in the RRG. Similarly, for the path $\langle N, N\rangle \xrightarrow{t_1} \langle N, T\rangle \xrightarrow{t_1} \langle T, T\rangle$ in the RRG, there is an equivalent path $\langle N, N\rangle \xrightarrow{t_1} \langle T, N\rangle \xrightarrow{t_2} \langle T, T\rangle$ in the reachability graph. ♣

Recall the model checking concepts defined in Section 2.1. An atomic proposition p is said to be *symmetry invariant* (under the applied state space symmetry group G) if $p \in \mu(q) \Leftrightarrow p \in \mu(\pi(q))$ holds for all $\pi \in G$ and for all states $q \in Q$. That is, p holds in a state if and only if it holds in all the equivalent states. Let f be a CTL* formula containing only symmetry invariant atomic propositions. Then f holds in the reachability graph if and only if it holds in the reduced reachability graph [Clarke et al. 1996; Emerson and Sistla 1996] (the Lemmas 2.6 and 2.7 above establish the necessary bisimulation condition between the reachability graph and the RRG). Thus such CTL* formulae can be model checked by using the reduced reachability graph instead of the original one. In the case the CTL* formula to be verified contains atomic propositions that are not symmetry invariant, some more advanced algorithms have to applied instead [Emerson and Sistla 1996; 1997; Gyuris and Sistla 1999; Sistla and Godefroid 2001]. Typically, these advanced algorithms augment the edges of the reduced reachability graph with the permutations that were used to obtain the representative successor states. This enables the algorithms to unwind the reduced reachability graph in a necessary amount to deduce whether the formula holds. Model checking under fairness constraints can also be handled by advanced algorithms [Gyuris and Sistla 1999].

Example 2.15 Recall the system discussed in Examples 2.1 and 2.2. Consider an atomic proposition C_1 that holds in a state s if and only if $s(1) = C$, i.e., the process 1 is in the critical section. It is not symmetry invariant because it holds in the state $\langle C, N\rangle$ but not in the equivalent state $\langle N, C\rangle$. On the other hand, an atomic proposition $\vee_i N_i$ defined to hold in a state s if and only if $\exists i \in I : s(i) = N$, is symmetry invariant (and similarly for $\vee_i T_i$ and $\vee_i C_i$). Therefore, one can use reduced reachability graphs to verify the property $\mathbf{AG}\,((\vee_i T_i) \Rightarrow \mathbf{F}(\vee_i C_i))$ stating that, during all executions of the system, it holds that if there is at some point a process in the trying section, then at some future point there is a process (not necessarily the same one) in the critical section. However, the property $\mathbf{AG}\,(T_1 \Rightarrow \mathbf{F}C_1)$ stating the same for process 1 cannot be verified by using reduced reachability graphs. In fact, the property does not hold in the original reachability graph but holds in the reduced reachability graph in the left hand side of Figure 2.3. With the advanced algorithms mentioned above, the property can be verified on a reduced reachability graph by partially unwinding it. Similarly, one can use the advanced algorithms to verify the property $\wedge_i \mathbf{AG}\,(T_i \Rightarrow \mathbf{F}C_i)$ stating the same property for all processes. Finally, the mutual exclusion property can be expressed as $\mathbf{AG}\neg(\vee_{i\neq j}(C_i \wedge C_j))$, where $\vee_{i\neq j}(C_i \wedge C_j)$ is a symmetry invariant atomic proposition defined to hold in a state s if and only if

$\exists i, j \in I : i \neq j \wedge s(i) = C \wedge s(j) = C.$ ♣

2.3 OTHER PRELIMINARIES

The following text describes some basic definitions needed in this work.

Functions. Let X and Y be two sets. The set of all functions from X to Y is denoted by $[X \to Y]$. Let $f \in [X \to Y]$. $f[x \mapsto y]$ is the function defined by $f[x \mapsto y](x') = f(x')$ for each $x' \neq x$ and $f[x \mapsto y](x) = y$. For an $X' \subseteq X$, the *restriction of f to X'* if the function $f' \in [X' \to Y]$ such that $f'(x) = f(x)$ for each $x \in X'$.

Families. Let I be a set. A *family* A with I as the *index set* is a function that assigns each $i \in I$ a set A_i (or a set A^i if subscripts are already used for something else). A family A can also be denoted by $\{A_i\}_{i \in I}$. A family $\{A_i\}_{i \in I}$ is *pairwise disjoint* if $i \neq j$ implies $A_i \cap A_j = \emptyset$, and *finite* if $\bigcup_{i \in I} A_i$ is finite. If no confusion can arise, one may overload A to also denote the set $\bigcup_{i \in I} A_i$.

Multisets. A *multiset* over a set A is a function $m : A \to \mathbb{N}$ and the set of all multisets over A is denoted by $[A \to \mathbb{N}]$. For an element $a \in A$, the value $m(a)$ is called the *multiplicity* of a in m. A multiset m can also be represented by using the formal sum notation $\Sigma_{a \in A} m(a)'a$. For instance, for a set $A = \{a_1, a_2, a_3\}$, the multiset $m = \{a_1 \mapsto 1, a_2 \mapsto 3, a_3 \mapsto 0\}$ can be denoted by the formal sum $1'a_1 + 3'a_2 + 0'a_3$. Dropping the elements with multiplicity 0 and omitting unit multiplicities, m can also be written as $a_1 + 3'a_2$. The empty multiset mapping each $a \in A$ to 0 is denoted by $\emptyset$. Let m_1, m_2 be two multisets over A and n a natural number. Then

1. $m_1 \leq m_2$ if and only if $m_1(a) \leq m_2(a)$ for each $a \in A$,
2. $m_1 + m_2$ is the multiset fulfilling $(m_1 + m_2)(a) = m_1(a) + m_2(a)$ for each $a \in A$,
3. if $m_2 \leq m_1$, then $m_1 - m_2$ is the multiset fulfilling $(m_1 - m_2)(a) = m_1(a) - m_2(a)$ for each $a \in A$, and
4. $n \cdot m_1$ is the multiset fulfilling $(n \cdot m_1)(a) = n \times m_1(a)$ for each $a \in A$.

Ordered Partitions. An *ordered partition* of a non-empty set A is a list $[C_1, \ldots, C_n]$ such that the set $\{C_1, \ldots, C_n\}$ is a partition of A, i.e., (i) $\emptyset \neq C_i \subseteq A$ for all $1 \leq i \leq n$, (ii) $\bigcup_{i=1}^{n} C_i = A$, and (iii) $C_i \cap C_j = \emptyset$ for all $i \neq j$. The sets C_i are called the *cells* of the partition. An ordered partition is *discrete* if all its cells are singleton sets and *unit* if it contains only one cell (namely the set A). Define the function *incell* from the ordered partitions of A and the elements of A to natural numbers by $\mathit{incell}([C_1, \ldots, C_n], x) = i \Leftrightarrow x \in C_i$.

An ordered partition p_1 of A is *finer than* (or a *refinement of*) an ordered partition p_2, denoted by $p_1 \leq p_2$, if each cell in p_1 is a subset of a cell in p_2. An ordered partition p_1 of A is a *cell order preserving refinement* of an ordered partition p_2, denoted by $p_1 \preceq p_2$, if $p_1 \leq p_2$ and for all

$x, y \in A$, $incell(p_1, x) < incell(p_1, y)$ implies $incell(p_2, x) \leq incell(p_2, y)$. That is, if $p_2 = [C_{2,1}, \ldots, C_{2,n}]$, then any p_1 such that $p_1 \preceq p_2$ is of form $[C_{1,1,1}, \ldots, C_{1,1,d_1}, \ldots C_{1,n,1}, \ldots, C_{1,n,d_n}]$ such that $\bigcup_{1 \leq j \leq d_i} C_{1,i,j} = C_{2,i}$ for each $1 \leq i \leq n$. For instance, it holds that $[\{b\}, \{a\}, \{c\}] \leq [\{a\}, \{b, c\}]$, $[\{b\}, \{a\}, \{c\}] \not\preceq [\{a\}, \{b, c\}]$, and $[\{a\}, \{c\}, \{b\}] \preceq [\{a\}, \{b, c\}]$. The relation $\preceq$ is reflexive, transitive and antisymmetric, i.e., a partial order on the set of all ordered partitions of A.

A permutation γ of A acts on ordered partitions of A by $\gamma([C_1, \ldots, C_n]) = [\gamma(C_1), \ldots, \gamma(C_n)]$. Clearly, $incell(p, x) = incell(\gamma(p), \gamma(x))$ for all ordered partitions p of A and for all $x \in A$. Furthermore, if $\gamma(p_1) = p_2$, $p_1 \preceq p_3$, and $p_2 \preceq p_3$, then $\gamma(p_3) = p_3$.

Computational Complexity. For computational complexity in general, refer, e.g., to [Garey and Johnson 1979; Papadimitriou 1995]. The class **P** (**NP**) consists of all decision problems decided by deterministic (non-deterministic) Turing machines in polynomial time. **co-NP** denotes the class of decision problems whose complements are in **NP**. For decision problems, polynomial time many-one reductions are used in this work. The fact that a decision problem A reduces to a decision problem B is denoted by $A \leq_{\mathrm{m}}^{\mathrm{p}} B$.

For search problems the notion of reducibility is not so well standardized as for decision problems. In this work, the following definitions are used. A search problem can be defined through a relation $A \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite, fixed alphabet. The relation is assumed to be polynomially balanced, meaning that there is a fixed polynomial p such that $\langle x, y \rangle \in A$ implies $|y| \leq p(|x|)$. The *search problem* associated with A is: given an input string $x \in \Sigma^*$, output a y such that $\langle x, y \rangle \in A$ or "no" if there is no such y. A search problem A *polynomial time many-one reduces* to a search problem B, denoted by $A \leq_{\mathrm{m}}^{\mathrm{p}} B$, if there are functions R and S computable in deterministic polynomial time such that for all instances $x \in \Sigma^*$ it holds that

- there is a w such that $\langle x, w \rangle \in A$ if and only if there is a z such that $\langle R(x), z \rangle \in B$, i.e., the reduced instance $R(x)$ has a solution in B if and only if the original instance x has a solution in A, and
- if $\langle R(x), z \rangle \in B$, then $\langle x, S(x, z) \rangle \in A$, i.e., from a solution z to the reduced instance $R(x)$ in B, a solution $S(x, z)$ to the original instance x in A can be computed.

Polynomial time many-one hardness and equivalence is defined as for decision problems. A search problem is in $\mathbf{FP}^{\mathbf{NP}}$ if there is a deterministic polynomial time Turing machine with an access to an **NP**-oracle that solves the problem. The reduction defined above is very similar to those used in [Krentel 1988; Papadimitriou 1995]. It is also a bit stronger, meaning that all the problems that are, for instance, $\mathbf{FP}^{\mathbf{NP}}$-complete under the reductions in [Krentel 1988; Papadimitriou 1995], are also $\mathbf{FP}^{\mathbf{NP}}$-complete under the proposed reduction.

Graph Iso- and Automorphisms. A directed *graph* is a pair $G = \langle V, E \rangle$, where V is the finite set of *vertices* (*nodes*) and $E \subseteq V \times V$ is the set of *edges*. A graph is *undirected* if its edge set is symmetric. An *isomorphism*

from a graph $G_1 = \langle V_1, E_1 \rangle$ to a graph $G_2 = \langle V_2, E_2 \rangle$ is a bijection γ from V_1 to V_2 such that $\langle v, v' \rangle \in E_1 \Leftrightarrow \langle \gamma(v), \gamma(v') \rangle \in E_2$. If there is an isomorphism from G_1 to G_2, then G_1 and G_2 are said to be *isomorphic*. An isomorphism from a graph $G = \langle V, E \rangle$ to itself is called an *automorphism* of G. The *set of all automorphisms*, denoted by $\mathrm{Aut}(G)$, forms a group under the function composition operator $\circ$, i.e., is a permutation group on V.

The computational complexity of deciding whether two graphs are isomorphic, or the GRAPH ISOMORPHISM problem, is an interesting topic in itself. It is one of the main candidates for a problem in **NP** that is neither in **P** nor **NP**-complete (such problems must exist if $\mathbf{P} \neq \mathbf{NP}$ as is widely believed). See, e.g., [Köbler et al. 1993] for further discussion on the complexity of the GRAPH ISOMORPHISM problem. Based on the results in [Miller 1979], it is easy to see that the complexity of the GRAPH ISOMORPHISM problem stays the same for different variants of graphs. Especially, the following two such graph classes will be used in this work.

- A directed, vertex and edge labeled graph is a triple $G = \langle V, E, L \rangle$, where V and E are as above, and L assigns each vertex and edge a label. An isomorphism from $G_1 = \langle V_1, E_1, L_1 \rangle$ to $G_2 = \langle V_2, E_2, L_2 \rangle$ is a bijection γ from V_1 to V_2 such that (i) $\langle v, v' \rangle \in E_1$ if and only if $\langle \gamma(v), \gamma(v') \rangle \in E_2$, (ii) $L_1(v) = L_2(\gamma(v))$ for each $v \in V_1$, and (iii) $L_1(\langle v, v' \rangle) = L_2(\langle \gamma(v), \gamma(v') \rangle)$ for each $\langle v, v' \rangle \in E_1$.
- A directed, vertex labeled and edge weighted graph is a triple $G = \langle V, E, L \rangle$, where V is as before, $E \subseteq V \times \mathbb{N} \times V$ the finite set of weighted edges (a triple $\langle v, w, v' \rangle$ in E denotes an edge from v to v' having weight w), and L assigns each vertex in V a label. Note that there may be multiple edges from a vertex to another, each having a different weight. An isomorphism from $G_1 = \langle V_1, E_1, L_1 \rangle$ to $G_2 = \langle V_2, E_2, L_2 \rangle$ is a bijection γ from V_1 to V_2 such that (i) $\langle v, w, v' \rangle \in E_1$ if and only if $\langle \gamma(v), w, \gamma(v') \rangle \in E_2$, and (ii) $L_1(v) = L_2(\gamma(v))$ for each $v \in V_1$.

3 PLACE/TRANSITION NETS: COMPUTATIONAL COMPLEXITY

Place/transition nets, see e.g. [Desel and Reisig 1998], are a popular formalism for modeling concurrent systems. Their major advantages are that they are easy to define and to understand, and also have a fairly standard graphical representation form. The symmetry reduction method for place/transition nets is introduced in [Starke 1991], showing that the structural symmetries of a net produce symmetries in its state space and also giving a preliminary algorithm for computing the symmetries of a net. A considerably improved algorithm for computing the symmetries of a net is given in [Schmidt 2000a]. The algorithm can also be used for checking whether two markings (i.e., states) are equivalent under the symmetries, and (when extended) also for checking whether a marking symmetrically covers another (see Section 3.4 for the definition of symmetric coverability). Algorithms for the orbit problems, needed in the reduced reachability graph construction, are described in [Schmidt 2000b].

This chapter studies the computational complexity of the sub-tasks involved in the symmetry reduction method for place/transition nets. Some of the results have been reported previously in [Junttila 2000; 2001]. For a survey of other complexity results concerning Petri nets, see [Esparza 1998].

First, some standard basic definitions of place/transition nets and their symmetries are given with examples. It is then shown that finding the symmetries of a net is a task equivalent to finding the automorphism group of a graph. The same applies to the task of finding all the symmetries of a net that stabilize a marking. The algorithms for the task are briefly discussed.

The computational complexity of the orbit problems is studied next. It is shown that deciding whether two markings are equivalent under the symmetries is from the computational complexity point of view equivalent to the graph isomorphism problem. Interestingly, it turns out that this result holds independently of whether the symmetry group of the net is given as input or not. It is also shown that finding the lexicographically greatest (or smallest) marking in the orbit of a given marking is a problem as hard as many well-known optimization problems such as the traveling salesperson problem, i.e., $\mathbf{FP}^{\mathbf{NP}}$-complete. Algorithms for the orbit problems, including some new ones for producing canonical representative markings, are discussed in the next chapter.

Finally, the computational complexity of the symmetry reduction method combined with the coverability graph approach is studied. It turns out that the problem of deciding whether there is a symmetry mapping a given marking to one covering another given marking is **NP**-complete. Furthermore, it is shown that the symmetric coverability problem cannot be combined in a straightforward way with the canonical representative marking approach.

3.1 BASIC DEFINITIONS

First, some basic definitions of place/transition nets and their symmetries are given. The representation is based on [Starke 1991; Schmidt 2000a; Desel

and Reisig 1998].

A *place/transition net* (or a P/T-net) is a tuple

$$N = \langle P, T, F, W, M_0 \rangle,$$

where

1. P is a finite, non-empty set of *places*,
2. T is a finite set of *transitions* such that $P \cap T = \emptyset$,
3. $F \subseteq (P \times T) \cup (T \times P)$ is the *flow-relation* (or the *set of arcs*),
4. $W : F \to \mathbb{N} \setminus \{0\}$ associates each arc in F with a positive *multiplicity* (or *weight*), and
5. $M_0 : P \to \mathbb{N}$ is the *initial marking*.

A *marking* of N is a multiset over P, i.e., a function $M : P \to \mathbb{N}$. The *set of all markings* is denoted by $\mathbb{M}$ and the *empty marking* is the one mapping each place to 0. One may also say that a place p has n *tokens* in a marking M if $M(p) = n$. The places and transitions are commonly called the *nodes* of the net. The arc weight function W is implicitly extended to $(P \times T) \cup (T \times P) \to \mathbb{N}$ by defining that $W(\langle x, y \rangle) = 0$ if $\langle x, y \rangle \notin F$. A transition $t \in T$ is *enabled* in a marking M, denoted by $M[t\rangle$, if $W(\langle p, t \rangle) \leq M(p)$ for each $p \in P$. If t is enabled in M, it may *fire* and transform M into the marking M' defined by $M'(p) = M(p) - W(\langle p, t \rangle) + W(\langle t, p \rangle)$ for each $p \in P$. This is denoted by $M\ [t\rangle\ M'$. The *state space* of N is the LTS $\langle \mathbb{M}, T, [\rangle, M_0 \rangle$, where $[\rangle = \{\langle M, t, M' \rangle \mid M\ [t\rangle\ M'\}$. The term marking is used as a synonym for state in this and the next chapter. The net N is *k-safe* if $M(p) \leq k$ holds for each place $p \in P$ in each reachable marking M. The net is *bounded* if it is k-safe for some fixed k. A transition $t \in T$ is *dead* at a marking M if it is not enabled in any marking reachable from M. The net is *live* if there are no dead transitions at any reachable marking.

Example 3.1 Consider the variant of Genrich's railroad system net [Genrich 1991] shown in Figure 3.1. It is a model of a railroad system in which two trains, call them a and b, drive in a cyclic railroad with six segments $0, \ldots, 5$. The semaphores V_i, $0 \leq i \leq 5$, are used to signal when it is allowed for a train to enter a segment i. The places of the net are drawn as circles, transitions as rectangles, and the arcs between them as directed edges. All the arc multiplicities in the net equal to 1 and are not drawn here or in any subsequent figures. The black filled circles, *tokens*, in the figure describe the initial marking $U_{a0} + U_{b3} + V_1 + V_4$ of the net. A token in a place U_{xi} denotes that the train x is in the segment i and a token in a place V_i denotes that a train can enter the segment i. The reachability graph of the net is shown in Figure 3.2. Based on it, it is easy to check that the net is 1-safe and live. Furthermore, it can be verified from the reachability graph that the trains cannot be in the same or adjacent segments at the same time. ♣

Symmetries of a net are automorphisms of the net when seen as a labeled directed graph. That is, they are permutations of the nodes of the net that respect (i) node type, (ii) the flow relation, and (iii) the arc multiplicities. Formally:

Definition 3.2 *A* symmetry *(or* automorphism*) of N is a permutation σ of $P \cup T$ that*

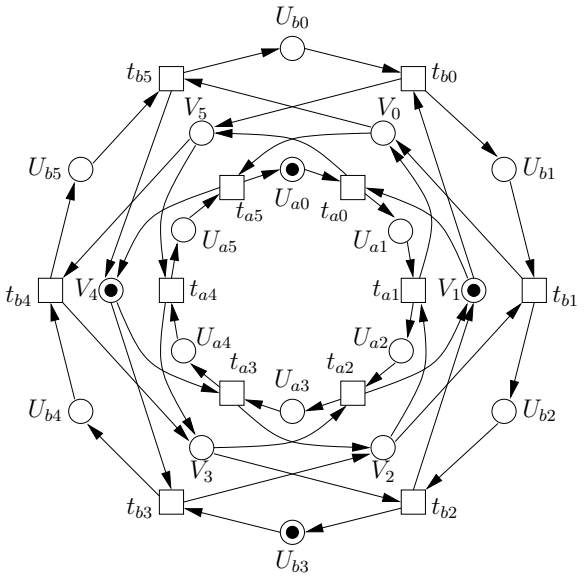


Figure 3.1: A net for a railroad system

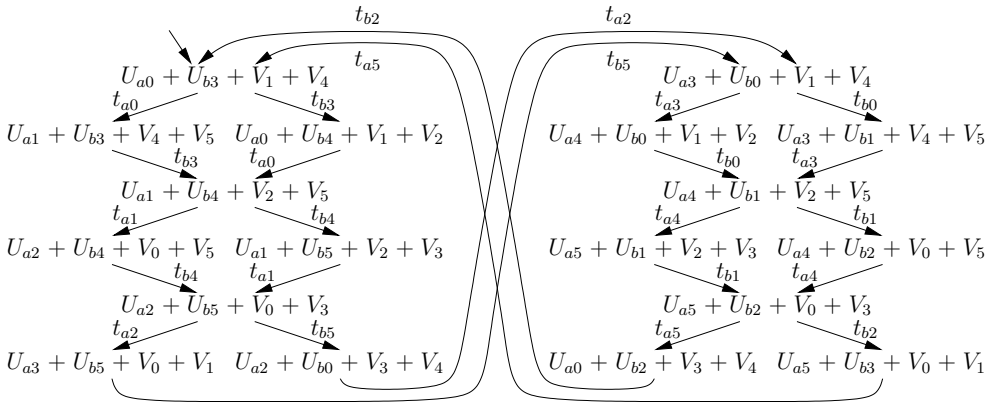


Figure 3.2: The reachability graph of the net in Figure 3.1

1. *respects node type:* $\sigma(P) = P$ *and* $\sigma(T) = T$*;*
2. *respects the flow relation:* $\langle x, y\rangle \in F \Leftrightarrow \langle \sigma(x), \sigma(y)\rangle \in F$*; and*
3. *respects the arc multiplicities:* $W(\langle x, y\rangle) = W(\langle \sigma(x), \sigma(y)\rangle)$ *for each* $\langle x, y\rangle \in F$*.*

The set of *all symmetries* of N (the *automorphism group* of N) is denoted by $\mathrm{Aut}(N)$ and is a subgroup of $\mathrm{Sym}(P \cup T)$. A symmetry σ of N acts on the markings of N by $\sigma(M) = M \circ \sigma^{-1}$, or equivalently, $(\sigma(M))\,(\sigma(p)) = M(p)$ for each $p \in P$. That is, the place $\sigma(p)$ has multiplicity n in the marking $\sigma(M)$ if and only if the place p has multiplicity n in the marking M. Because $\mathbf{I}(M) = M$ and $(\sigma_1 \circ \sigma_2)(M) = M \circ (\sigma_1 \circ \sigma_2)^{-1} = M \circ \sigma_2^{-1} \circ \sigma_1^{-1} = \sigma_2(M) \circ \sigma_1^{-1} = \sigma_1(\sigma_2(M))$, the definition is a group action on the set $\mathbb{M}$ of markings. A symmetry of the net produces a corresponding state space symmetry:

Lemma 3.3 ([Starke 1991]) *If* σ *is a symmetry of* N*, then*

$$M\ [t\rangle\ M' \Leftrightarrow \sigma(M)\ [\sigma(t)\rangle\ \sigma(M').$$

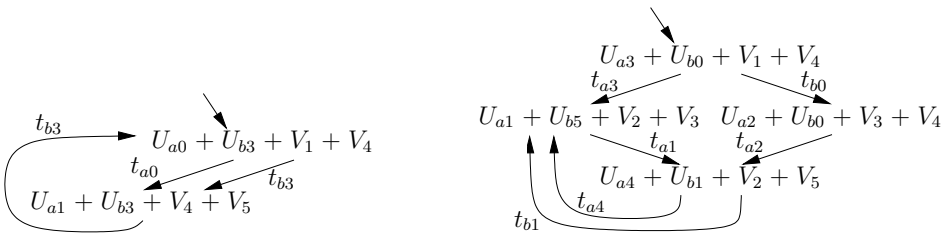


Figure 3.3: Two reduced reachability graphs for the net in Figure 3.1

Example 3.4 Recall the net N in Figure 3.1. The group $\mathrm{Aut}(N)$ is generated by the rotation

$$\sigma_{\text{rot}} = \begin{pmatrix} U_{a0} & U_{a1} & U_{a2} & U_{a3} & U_{a4} & U_{a5} & U_{b0} & \cdots & U_{b5} & V_0 & \cdots & V_5 & t_{a0} & \cdots & t_{a5} & t_{b0} & \cdots & t_{b5} \\ U_{a1} & U_{a2} & U_{a3} & U_{a4} & U_{a5} & U_{a0} & U_{b1} & \cdots & U_{b0} & V_1 & \cdots & V_0 & t_{a1} & \cdots & t_{a0} & t_{b1} & \cdots & t_{b0} \end{pmatrix}$$

and the swapping of train identities

$$\sigma_{\text{swap}} = \begin{pmatrix} U_{a0} & \cdots & U_{a5} & U_{b0} & \cdots & U_{b5} & V_0 & \cdots & V_5 & t_{a0} & \cdots & t_{a5} & t_{b0} & \cdots & t_{b5} \\ U_{b0} & \cdots & U_{b5} & U_{a0} & \cdots & U_{a5} & V_0 & \cdots & V_5 & t_{b0} & \cdots & t_{b5} & t_{a0} & \cdots & t_{a5} \end{pmatrix},$$

meaning that all the elements in $\mathrm{Aut}(N)$ (and only those) can be expressed as a finite composition of σ_{rot} and σ_{swap}. The group $\mathrm{Aut}(N)$ has 12 elements. The initial marking

$$M_0 = U_{a0} + U_{b3} + V_1 + V_4$$

is equivalent (under $\mathrm{Aut}(N)$) to the marking

$$M = U_{a4} + U_{b1} + V_2 + V_5$$

as $(\sigma_{\text{swap}} \circ \sigma_{\text{rot}})(M_0) = \sigma_{\text{swap}}(\sigma_{\text{rot}}(M_0)) = \sigma_{\text{swap}}(U_{a1} + U_{b4} + V_2 + V_5) = M$. The orbit of M_0 consists of the markings

$$\begin{array}{ll} M_0, & U_{a1} + U_{b4} + V_2 + V_5, \\ U_{a2} + U_{b5} + V_0 + V_3, & U_{a3} + U_{b0} + V_1 + V_4, \\ U_{a4} + U_{b1} + V_2 + V_5, \text{ and} & U_{a5} + U_{b2} + V_0 + V_3. \end{array}$$

Figure 3.3 shows two reduced reachability graphs for the net, the left one being minimal in the sense that it contains only one marking from each reachable orbit. ♣

Finally, assume a subgroup G of $\mathrm{Aut}(N)$. A symmetry $\sigma \in G$ stabilizes a marking M if $\sigma(M) = M$. The set of all stabilizers of M in G, denoted by $\mathrm{Stab}(G, M)$, is a subgroup of G. In the case $G = \mathrm{Aut}(N)$, one may write $\mathrm{Stab}(N, M)$ instead of $\mathrm{Stab}(G, M)$. Obviously, $\mathrm{Aut}(N) = \mathrm{Stab}(N, \tilde{M})$ for any marking $\tilde{M}$ for which $\tilde{M}(p) = \tilde{M}(p')$ for all $p, p' \in P$, e.g., for the empty marking. Furthermore, let $\mathrm{Stab}(G, M_1, \ldots, M_k)$ to denote the maximal subgroup of G stabilizing each of the markings $M_1, \ldots, M_k$, i.e., $\mathrm{Stab}(G, M_1, \ldots, M_k) = \mathrm{Stab}(G, M_1) \cap \cdots \cap \mathrm{Stab}(G, M_k)$.

3.1.1 Representing Symmetries

Since the automorphism group $\mathrm{Aut}(N)$ may have up to $|P|! \cdot |T|!$ permutations, its subgroups (including the group itself) cannot be efficiently represented by explicitly listing all the constituent permutations. Instead, a group

is represented by a set of generators, i.e., by giving a set of permutations belonging to the group such that any permutation in the group can be expressed as a finite composition of the permutations in the set. In fact, for any permutation group on a set with n elements there is a generating set consisting only of $n-1$ permutations [Jerrum 1986]. Furthermore, there are deterministic polynomial time algorithms that, given a generating set for a permutation group, compute a standard representation for the group. Using the presentation, testing whether a permutation belongs to the group can be done in polynomial time. One such standard representation, called Schreier-Sims representation, will be described and applied in the next chapter.

From now on, permutation groups are always represented by means of generating sets. This means that *"given the group G" should be read as "given a generating set for the group G"* and *"find the group G" should be read as "find a generating set for the group G"*.

3.2 FINDING THE SYMMETRIES

The first task in the symmetry reduction method is to find the symmetries. In the context of P/T-nets, this equals to finding the automorphism group of the net in question. In addition, it may be required that the group stabilizes a marking, for instance, the initial marking. This section discusses the computational complexity and algorithms for the task.

3.2.1 Computational Complexity

The symmetry finding task is formulated in the following problems.

Problem 3.5 NET AUTOMORPHISMS. *Given a net N, find* $\mathrm{Aut}(N)$.

Problem 3.6 MARKING STABILIZERS. *Given a net N and a set of markings $\{M_1, \ldots, M_k\}$ of N, find the group* $\mathrm{Stab}(\mathrm{Aut}(N), M_1, \ldots, M_k)$.

Note that the latter problem covers the problem of computing the subgroup of $\mathrm{Aut}(N)$ stabilizing the initial marking. Recall that the group $\mathrm{Aut}(N)$ is the same as the group $\mathrm{Stab}(\mathrm{Aut}(N), \tilde{M})$, where $\tilde{M}$ is the empty marking, and thus the latter problem definition also covers the first one. Therefore,

$$\text{NET AUTOMORPHISMS} \leq_{\mathrm{m}}^{\mathrm{p}} \text{MARKING STABILIZERS}.$$

As the following arguments show, both of these problems are equivalent to the GRAPH AUTOMORPHISMS problem.[1]

Theorem 3.7 MARKING STABILIZERS $\leq_{\mathrm{m}}^{\mathrm{p}}$ GRAPH AUTOMORPHISMS.

Proof. The net $N = \langle P, T, F, W, M_0 \rangle$ in question together with the given markings $M_1, \ldots, M_k$ is just interpreted as a directed, vertex and edge labeled graph $G = \langle V, E, L \rangle$, where

1. $V = P \cup T$,

[1] For a third version of the problem, see Problem 3.15.

2. $E = F$,
3. $L(p) = \langle M_1(p), \ldots, M_k(p) \rangle$ for each place $p \in P$, i.e., the number of tokens in the given markings,
4. $L(t) =$ "T" for each transition $t \in T$, and
5. $L(f) = W(f)$ for each arc $f \in F$.

See the left hand side of Figure 3.4 for a simple example (arc multiplicities are omitted for simplicity). Clearly the automorphism group of the constructed graph is exactly the automorphism group of the net stabilizing the given markings. □

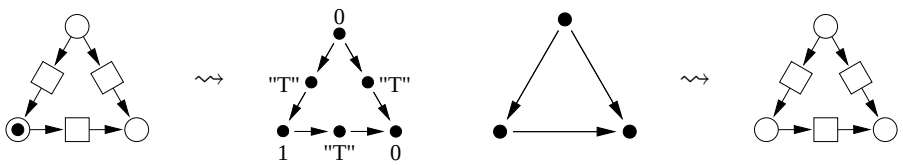
Theorem 3.8 GRAPH AUTOMORPHISMS $\leq_{\mathrm{m}}^{\mathrm{p}}$ NET AUTOMORPHISMS.

Proof. For a directed graph $G = \langle V, E \rangle$, construct the net $\langle P, T, F, W, M_0 \rangle$ where

1. $P = V$,
2. $T = E$,
3. $F = \{\langle v, \langle v, v' \rangle\rangle \mid \langle v, v' \rangle \in E\} \cup \{\langle \langle v, v' \rangle, v' \rangle \mid \langle v, v' \rangle \in E\}$,
4. $W(f) = 1$ for each $f \in F$, and
5. $M_0(p) = 0$ for each place $p \in P$.

See the right hand side of Figure 3.4 for a simple example (arc multiplicities and edge labels are omitted for simplicity). It follows directly from the definition that the group $\mathrm{Aut}(N)$ restricted to the set of places is exactly the group $\mathrm{Aut}(G)$. □

Corollary 3.9 *Both* NET AUTOMORPHISMS *and* MARKING STABILIZERS *are polynomial time many-one equivalent to* GRAPH AUTOMORPHISMS.



(a) From nets to graphs. (b) From graphs to nets.

Figure 3.4: Mappings between graphs and nets for the automorphism problem

3.2.2 Algorithms

A backtracking search algorithm for solving the NET AUTOMORPHISMS and MARKING STABILIZERS problems is described in [Schmidt 2000a]. The algorithm is based on representing the sets of possible net automorphisms as constraints. The constraints are then refined and split during the backtracking search until the constraints represent a single automorphism. Although the algorithm is somewhat different from the "standard" graph automorphism algorithms such as [McKay 1981; Kreher and Stinson 1999], it also has many common features: it is based on similar "split and refine" idea and it can

prune the search tree by using the already found automorphisms. The algorithm also produces a Schreier-Sims representation of the requested group (see Section 4.1.1 for the definition of the Schreier-Sims representation).

By the reduction from MARKING STABILIZERS to GRAPH AUTOMORPHISMS described above in the proof of Theorem 3.7, it is obvious that one can also employ standard tools for the GRAPH AUTOMORPHISMS problem to find the symmetries of a P/T-net. For instance, one may use the *nauty* tool [McKay 1981; 1990]. In the case the selected graph automorphism tool does not support directed, vertex and edge labeled graphs (for instance, *nauty* does not support edge labels), some extra vertices and edges have to be included in the graph corresponding to a net (cf. Section 4.2). Also note that some design choices made in graph automorphism tools may adversely affect their efficiency when applied to finding the symmetries of P/T-nets. For example, the *nauty* tool is optimized for undirected and dense graphs (the graph is internally represented as an adjacency matrix in *nauty*), while P/T-nets are usually sparse and always directed. For experimental evidence of this, see Section 4.5.2.

3.3 COMPUTATIONAL COMPLEXITY OF THE ORBIT PROBLEMS

After finding the symmetries of a net, the next task is to exploit them during the reduced reachability graph generation. That is, one has to be able to (i) decide whether two markings are equivalent under the symmetries, or (ii) build a canonical representative for a marking. This section studies the computational complexity of these two problems in the context of P/T-nets. Algorithms for the orbit problems are discussed in Chapter 4.

3.3.1 The Marking Equivalence Problem

The problem of deciding whether two markings are equivalent under the symmetries is studied by considering two versions of the problem. In the "hard", or general, version the symmetries of the net are not given as input:

Problem 3.10 MARKING EQUIVALENCE (ME). *Given a net N and markings M, M_1, and M_2 of N, are the markings M_1 and M_2 equivalent under the stabilizer group* $\mathrm{Stab}(N, M)$*?*

As marked nets can be seen as directed, vertex and edge labeled graphs, the following is a quite straightforward result.

Theorem 3.11 ME $\leq_{\mathrm{m}}^{\mathrm{p}}$ GRAPH ISOMORPHISM.

Proof. Let $N = \langle P, T, F, W, M_0 \rangle$. For the marking M_i of N, $i \in \{1, 2\}$, interpret the net marked with M_i as the directed, vertex and edge labeled graph $G_{M_i} = \langle V_{M_i}, E_{M_i}, L_{M_i} \rangle$, where

1. $V_{M_i} = P \cup T$,
2. $E_{M_i} = F$,
3. $L_{M_i}(p) = \langle M(p), M_i(p) \rangle$ for each $p \in P$,
4. $L_{M_i}(t) =$ "T" for each $t \in T$, and

5. $L_{M_i}(f) = W(f)$ for each $f \in F$.

This construction is essentially the same as the one in the proof of Theorem 3.7. It is easy to see directly from the definition of G_{M_i} that M_1 and M_2 are equivalent under $\mathrm{Stab}(N, M)$ if and only if G_{M_1} and G_{M_2} are isomorphic. □

In the "easy" version of the marking equivalence problem, the input contains more information about the net. Especially, (a generating set for) the automorphism group of the net is given.

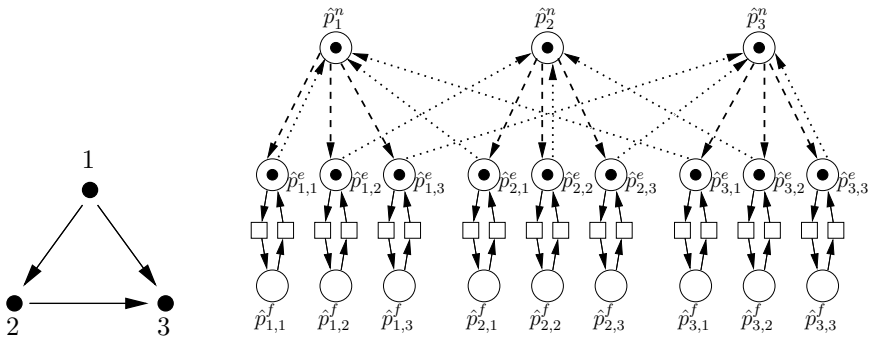
Problem 3.12 MARKING EQUIVALENCE*, version 2 (*ME2*). Given a 1-safe and live net N, the group* $\mathrm{Aut}(N)$ *and two reachable markings of N, are the markings equivalent under* $\mathrm{Aut}(N)$*?*

Obviously, ME2 $\leq^{\mathrm{p}}_{\mathrm{m}}$ ME. Interestingly, the upper computational complexity bound of the "hard" version ME is the same as the lower bound of the "easy" version ME2.

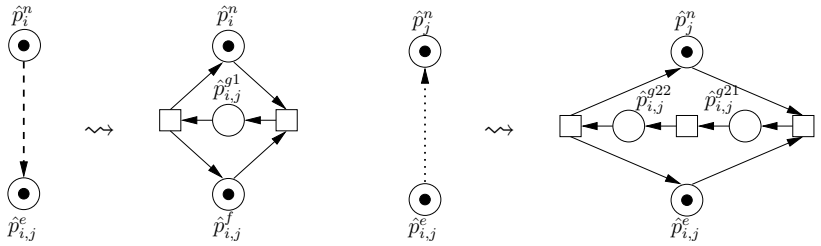
Theorem 3.13 GRAPH ISOMORPHISM $\leq^{\mathrm{p}}_{\mathrm{m}}$ ME2.

Proof. Suppose that two directed graphs, $G = \langle V, E\rangle$ and $G' = \langle V, E'\rangle$, with the same set of vertices are given. (If they have a different number of vertices, they cannot be isomorphic and one can output a simple 1-safe and live net having no non-trivial symmetries and two different reachable markings for it; if they have different sets of vertices, any renaming of the vertices will do.) The net $\hat{N} = \langle \hat{P}, \hat{T}, \hat{F}, \hat{W}, \hat{M}_0\rangle$ is defined as follows:

$$
\begin{aligned}
\hat{P} &= \{\hat{p}^n_v \mid v \in V\} \cup \left\{\hat{p}^e_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{p}^f_{v,v'} \mid v, v' \in V\right\} \cup \\
&\quad \left\{\hat{p}^{g1}_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{p}^{g21}_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{p}^{g22}_{v,v'} \mid v, v' \in V\right\} \\
\hat{T} &= \left\{\hat{t}^{\mathrm{del}}_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{t}^{\mathrm{add}}_{v,v'} \mid v, v' \in V\right\} \cup \\
&\quad \left\{\hat{t}^{g11}_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{t}^{g12}_{v,v'} \mid v, v' \in V\right\} \cup \\
&\quad \left\{\hat{t}^{g21}_{v,v'} \mid v, v' \in V\right\} \cup \left\{\hat{t}^{g22}_{v,v'} \mid v, v' \in V\right\} \cup \\
&\quad \left\{\hat{t}^{g23}_{v,v'} \mid v, v' \in V\right\} \\
\hat{F} &= \left\{\langle \hat{p}^e_{v,v'}, \hat{t}^{\mathrm{del}}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{t}^{\mathrm{del}}_{v,v'}, \hat{p}^f_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{p}^f_{v,v'}, \hat{t}^{\mathrm{add}}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{t}^{\mathrm{add}}_{v,v'}, \hat{p}^e_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{p}^e_{v,v'}, \hat{t}^{g11}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{p}^n_v, \hat{t}^{g11}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{t}^{g11}_{v,v'}, \hat{p}^{g1}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{p}^{g1}_{v,v'}, \hat{t}^{g12}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{t}^{g12}_{v,v'}, \hat{p}^e_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{t}^{g12}_{v,v'}, \hat{p}^n_v\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{p}^e_{v,v'}, \hat{t}^{g21}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{p}^n_{v'}, \hat{t}^{g21}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{t}^{g21}_{v,v'}, \hat{p}^{g21}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{p}^{g21}_{v,v'}, \hat{t}^{g22}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{t}^{g22}_{v,v'}, \hat{p}^{g22}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{p}^{g22}_{v,v'}, \hat{t}^{g23}_{v,v'}\rangle \mid v, v' \in V\right\} \cup \\
&\quad \left\{\langle \hat{t}^{g23}_{v,v'}, \hat{p}^e_{v,v'}\rangle \mid v, v' \in V\right\} \cup \left\{\langle \hat{t}^{g23}_{v,v'}, \hat{p}^n_{v'}\rangle \mid v, v' \in V\right\} \\
\hat{W}(\hat{f}) &= 1 \text{ for each } \hat{f} \in \hat{F} \\
\hat{M}_0 &= \sum_{v \in V} 1'\hat{p}^n_v + \sum_{v,v' \in V} 1'\hat{p}^e_{v,v'}
\end{aligned}
$$



(a) A graph. (b) The net for graphs over the vertex set $\{1, 2, 3\}$.



(c) Substitution rules for the dashed and dotted lines in (b).

Figure 3.5: Reduction from a graph to a live and 1-safe net

Figure 3.5(b) and (c) illustrate the construction by showing the net $\hat{N}$ (transition names are omitted for the sake of readability) for graphs over the vertex set $V = \{1, 2, 3\}$. (Figure 3.5(a) shows one such a graph.) It is not hard to see that the net $\hat{N}$ is live and 1-safe.

The main idea of the construction is that the places of the form $\hat{p}^e_{v,v'}$ are used to represent the adjacency matrix of the graph under consideration. For the graph G, the corresponding marking $\hat{M}_G$ of $\hat{N}$ is defined by

$$\hat{M}_G = \sum_{v \in V} 1'\hat{p}^n_v + \sum_{\langle v,v' \rangle \in E} 1'\hat{p}^e_{v,v'} + \sum_{\langle v,v' \rangle \notin E} 1'\hat{p}^f_{v,v'}.$$

The marking $\hat{M}_{G'}$ for the graph G' is constructed similarly. Obviously, both of these markings are reachable.

The automorphisms of $\hat{N}$ are exactly those that are produced by the homomorphism $h : \mathrm{Sym}(V) \to \mathrm{Sym}(\hat{P} \cup \hat{T})$ defined as follows. For each $\pi \in \mathrm{Sym}(V)$, $h(\pi)$ maps (i) each $\hat{p}^n_v$ to $\hat{p}^n_{\pi(v)}$, (ii) each $\hat{p}^x_{v,v'}$ to $\hat{p}^x_{\pi(v),\pi(v')}$, where $x \in \{e, f, g1, g21, g22\}$, and (iii) each $\hat{t}^x_{v,v'}$ to $\hat{t}^x_{\pi(v),\pi(v')}$, where $x \in \{g11, g12, g21, g22, g23\}$. That is, $\mathrm{Aut}(\hat{N}) = h(\mathrm{Sym}(V))$. Since the group $\mathrm{Sym}(V)$ can be represented by two generators, namely the rotation $\pi_1 = \begin{pmatrix} v_1 & v_2 & v_3 & \cdots & v_{|V|-1} & v_{|V|} \\ v_2 & v_3 & v_4 & \cdots & v_{|V|} & v_1 \end{pmatrix}$ and the permutation swapping of the first two elements $\pi_2 = \begin{pmatrix} v_1 & v_2 & v_3 & \cdots & v_{|V|} \\ v_2 & v_1 & v_3 & \cdots & v_{|V|} \end{pmatrix}$, the generators for $\mathrm{Aut}(\hat{N})$ are $h(\pi_1)$ and $h(\pi_2)$. Now it is reasonably easy to see that $\hat{M}_G$ and $\hat{M}_{G'}$ are equivalent under $\mathrm{Aut}(\hat{N})$ if and only if G and G' are isomorphic because $\mathrm{Aut}(\hat{N})$ corresponds to the group of all permutations on the vertex set V naturally extended to the adjacency matrix of a graph with the vertex set V. That is, if the vertices of G can be permuted in a way that the adjacency matrix of G becomes equal to

the adjacency matrix of G', then (and only then) can the marking $\hat{M}_G$ be permuted by $\mathrm{Aut}(\hat{N})$ to become equal to $\hat{M}_{G'}$. For instance, consider the marking

$$\sum_{v \in V} 1' \hat{p}_v^n + \hat{p}_{1,1}^f + \hat{p}_{1,2}^e + \hat{p}_{1,3}^e + \hat{p}_{2,1}^f + \hat{p}_{2,2}^f + \hat{p}_{2,3}^e + \hat{p}_{3,1}^f + \hat{p}_{3,2}^f + \hat{p}_{3,3}^f$$

corresponding to the graph in Figure 3.5(a). Applying the generator $h(\pi_1)$ to the marking, the marking

$$\sum_{v \in V} 1' \hat{p}_v^n + \hat{p}_{1,1}^f + \hat{p}_{1,2}^f + \hat{p}_{1,3}^f + \hat{p}_{2,1}^e + \hat{p}_{2,2}^f + \hat{p}_{2,3}^e + \hat{p}_{3,1}^e + \hat{p}_{3,2}^f + \hat{p}_{3,3}^f$$

is obtained. This marking corresponds to the graph obtained from that in Figure 3.5(a) by replacing the vertex "1" with "2", "2" with "3", and "3" with "1". By definition, this graph is isomorphic to the one in Figure 3.5(a).

Also notice that $\mathrm{Aut}(\hat{N})$ stabilizes the initial marking $\hat{M}_0$. □

Corollary 3.14 *Both* ME *and* ME2 *are polynomial time many-one equivalent to* GRAPH ISOMORPHISM.

This result implies that, from the computational complexity point of view, pre-calculation of the automorphism group of a net does not provide any help for solving the problem of whether two markings of the net are equivalent (not even for 1-safe and live nets). However, in practice it is probably reasonable to compute the automorphism group of the net since it yields useful information. For instance, it may reveal that the net has no non-trivial automorphisms and thus the symmetry reduction method is of no use for the net. Furthermore, knowing the automorphism group can assist in the choice of the algorithm for the orbit problem since the performances of different algorithms may depend on the order of the automorphism group, see [Schmidt 2000b] and Section 4.5.2.

As noted in [Jensen 1995; 1996], the stabilizers of markings can sometimes be exploited during the generation of reduced reachability graphs. That is, if $M\ [t\rangle\ M'$, then $M\ [\sigma(t)\rangle\ \sigma(M')$ for each $\sigma \in \mathrm{Stab}(G, M)$, where G is the group under which the reduced reachability graph is generated (usually, G is $\mathrm{Aut}(N)$ or $\mathrm{Stab}(N, M_0)$). Thus the transition $\sigma(t)$ is enabled in M if and only if t is, and the successor markings M' and $\sigma(M')$ are equivalent. Note that, given (a generating set for) the group $G' = \mathrm{Stab}(G, M)$, it is easy (i) to check whether there is a symmetry in G' mapping a transition t to another transition t', and (ii) to compute the G'-orbit of each transition t [Butler 1991]. Based on Theorem 3.7, finding the group $\mathrm{Stab}(G, M)$, where G is the stabilizer group of a set of markings, can be solved with an algorithm for the GRAPH AUTOMORPHISMS problem. Also consider the following "easy" version of the MARKING STABILIZERS problem.

Problem 3.15 MARKING STABILIZERS 2. *Given a 1-safe and live net N, the group* $\mathrm{Aut}(N)$*, and a reachable marking M of N, find the stabilizer group* $\mathrm{Stab}(N, M)$.

Considering the net $\hat{N}$ and the marking $\hat{M}_G$ corresponding to a graph G constructed in the proof of Theorem 3.13, it is easy to see that the stabilizer group

$\mathrm{Stab}(\hat{N}, \hat{M}_G)$ restricted to the places of form $\hat{p}_v^n$ is exactly the automorphism group of G. Thus

$$\text{GRAPH AUTOMORPHISMS} \leq_{\mathrm{m}}^{\mathrm{p}} \text{MARKING STABILIZERS 2}.$$

In addition, since MARKING STABILIZERS 2 $\leq_{\mathrm{m}}^{\mathrm{p}}$ MARKING STABILIZERS and MARKING STABILIZERS $\leq_{\mathrm{m}}^{\mathrm{p}}$ GRAPH AUTOMORPHISMS, MARKING STABILIZERS 2 is equivalent to the GRAPH AUTOMORPHISMS problem.

Relationship to a string orbit problem. The marking equivalence problem studied above is quite similar to a string orbit problem considered in [Babai and Luks 1983; Clarke et al. 1998]. Let Σ be a finite alphabet and I a finite index set. A Σ-string on I is a function $s : I \to \Sigma$. A permutation g of I acts on Σ-strings on I by $g(s) = s \circ g^{-1}$. Given a permutation group G on I, two Σ-strings on I, s and s', are said to be *G-equivalent* if there is a permutation $g \in G$ mapping s to s'. The STRING ORBIT problem is: Given two Σ-strings on an index set I and a permutation group G on I, are the strings G-equivalent? In [Clarke et al. 1998] it is shown that the STRING ORBIT problem is equivalent to the problems in the Luks equivalence class [Babai 1994]. The Luks equivalence class contains many natural problems concerning permutation groups, and it is believed that the decision problems in it, although in **NP**, are (i) harder than those that are equivalent to the GRAPH ISOMORPHISM problem and (ii) not **NP**-complete [Hoffmann 1982; Babai 1994].

The fact that the two versions, ME and ME2, of the marking equivalence problem discussed above are equivalent to the GRAPH ISOMORPHISM problem is because the considered groups are automorphism groups of graphs, not arbitrary permutation groups. However, deciding whether two markings are equivalent under an *arbitrary* subgroup of $\mathrm{Aut}(N)$ is equivalent to the STRING ORBIT problem as shown below.

Problem 3.16 GENERALIZED MARKING EQUIVALENCE (GME). *Given a net N, a subgroup G of $\mathrm{Aut}(N)$, and two markings M_1 and M_2 of N, are the markings equivalent under G?*

Theorem 3.17 GME *is polynomial time many-one equivalent to* STRING ORBIT.

Proof. The reduction from STRING ORBIT to GME is given first. Assume that $\Sigma = \{1, \ldots, k\}$ is the applied finite alphabet. Given an index set I, two Σ-strings, s and s', on I, and a permutation group G on I, the net N is constructed as follows. The set of places P of the net is simply the index set I. The net has no transitions or arcs, and the initial marking is empty. Obviously, G is a subgroup of $\mathrm{Aut}(N)$. The marking M corresponding to the string s is simply defined by $M(i) = s(i)$ for each place $i \in P$. The marking M' for s' is constructed similarly. It is quite clear that the markings M and M' are equivalent under G if and only if s and s' are G-equivalent.

To reduce the other way, assume a net N, a subgroup G of $\mathrm{Aut}(N)$, and two markings M_1 and M_2 of N. Let

$$K = \max\{k \in \mathbb{N} \mid \exists p \in P : k = M_1(p) \vee k = M_2(p)\}$$

be the maximum number of tokens appearing in any place in the two markings in question. Let $I = \{i_{p,j} \mid p \in P \wedge j \in \mathbb{N} \wedge 1 \leq j \leq \lceil \log K \rceil\}$ be the index set and $\Sigma = \{0, 1\}$ the binary alphabet. The marking M_l, $l \in \{1, 2\}$, is transformed into a Σ-string s_l on I by binary coding the number of tokens in each place p in the index elements of form $i_{p,j}$. The group G' on I is obtained from G by the group isomorphism $\gamma : g \mapsto g'$ such that $g'(i_{p,j}) = i_{g(p),j}$ for each place p and for each $1 \leq j \leq \lceil \log K \rceil$. Now s_1 and s_2 are G'-equivalent if and only if M_1 and M_2 are equivalent under G. □

3.3.2 Finding the Lexicographical Leader Marking

Recall that a canonical representative marking function is a mapping

$$\mathit{canrepr} : \mathbb{M} \to \mathbb{M}$$

such that (i) $\mathit{canrepr}(M) \equiv_G M$ for each marking M, and (ii) $M_1 \equiv_G M_2$ implies $\mathit{canrepr}(M_1) = \mathit{canrepr}(M_2)$. Therefore, given a canonical representative function *canrepr*, one can decide whether two markings M_1 and M_2 are equivalent by simply computing both $\mathit{canrepr}(M_1)$ and $\mathit{canrepr}(M_2)$, and comparing whether they are equal. In this sense, computing any canonical representative function is at least as hard as testing whether two markings are equivalent and thus at least as hard as the GRAPH ISOMORPHISM problem.

Perhaps the most natural choice for the canonical representative marking is the lexicographically greatest (or smallest) marking in the orbit. In order to define lexicographical orders, a *base* (or an *element ordering*) of a net $N = \langle P, T, F, W, M_0 \rangle$ is defined to be an ordered list $\beta = [\beta_1, \ldots, \beta_{|P|+|T|}]$ of the elements in $P \cup T$ such that all the places are listed before the transitions. The *lexicographical ordering* $<_\beta$ of the markings of N under the base β is defined by:

$$M_1 <_\beta M_2$$

if and only if there is an i, $1 \leq i \leq |P|$, such that

1. $M_1(\beta_i) < M_2(\beta_i)$, and
2. for all $1 \leq j < i$, $M_1(\beta_j) = M_2(\beta_j)$.

Define that $M_1 \leq_\beta M_2$ if either $M_1 <_\beta M_2$ or $M_1 = M_2$. For instance, if the set of places is $P = \{p_a, p_b, p_c\}$, the base is $\beta = [p_b, p_c, p_a, \ldots]$, and $M_1 = p_b + 2'p_c$ and $M_2 = 3'p_a + p_b$ are two markings, then $M_2 <_\beta M_1$.

As is proven next, it is not easy to find the lexicographically greatest marking in the orbit of a given marking under a given element ordering. In fact, it is as hard as some classical optimization problems such as the TRAVELING SALESPERSON problem, i.e., $\mathbf{FP}^{\mathbf{NP}}$-complete. As in the MARKING EQUIVALENCE problems discussed above, two versions of the lex-greatest marking problem are defined. The "hard", or general, version is:

Problem 3.18 LEX-GREATEST MARKING (LGM). *Given a net N, a base β of N, and two markings M and M' of N, find the $<_\beta$-greatest marking in the* $\mathrm{Stab}(N, M)$*-orbit of M'.*

In order to classify the complexity of this problem, a decision version of it is defined and classified as follows.

Problem 3.19 LEX-GREATEST MARKING, *decision version* (LGM(D)). *Given a net N, a base β of N, and two markings M and M' of N, is there a marking that is (i) $<_\beta$-greater than M' and (ii) in the* $\mathrm{Stab}(N, M)$*-orbit of M'?*

Lemma 3.20 LGM(D) *is in* **NP**.

Proof. Simply guess a permutation $\sigma \in \mathrm{Sym}(P \cup T)$ and (deterministically) verify in polynomial time that (i) σ is an automorphism of N, (ii) σ stabilizes M, and (iii) $M' <_\beta \sigma(M')$. □

Based on the above lemma, the following is easy to prove.

Theorem 3.21 LGM *is in* $\mathbf{FP}^{\mathbf{NP}}$.

Proof. Let $k = \max_{p \in P} M'(p)$ be the maximum number of tokens in any place in the marking M'. Thus the size of the input for the problem is at least $\Omega(|P| + \log_2 k)$. Clearly, any marking that is in the $\mathrm{Stab}(N, M)$-orbit of M' has at most k tokens in any place. There are $k^{|P|}$ possible markings fulfilling this restriction. With standard binary search, the lexicographically greatest marking in the $\mathrm{Stab}(N, M)$-orbit of M' can be found by using $\log_2 k^{|P|} = |P| \log_2 k$ (a polynomial amount in $|P| + \log_2 k$) queries to an **NP**-oracle deciding the problem LGM(D). □

The "easy" version of the lex-greatest marking problem is:

Problem 3.22 LEX-GREATEST MARKING, *version 2* (LGM2). *Given a 1-safe and live net N, a base β of N, the group* $\mathrm{Aut}(N)$ *and a reachable marking M of N, find the $<_\beta$-greatest marking in the* $\mathrm{Aut}(N)$*-orbit of M.*

Clearly LGM2 $\leq^{\mathrm{p}}_{\mathrm{m}}$ LGM. Again, the lower computational complexity bound of the "easy" version is the same as the upper bound of the "hard" version.

Theorem 3.23 LGM2 *is* $\mathbf{FP}^{\mathbf{NP}}$*-hard.*

Proof. The following $\mathbf{FP}^{\mathbf{NP}}$-complete problem in [Krentel 1988] is reduced to LGM2.

> MAXIMUM SATISFYING ASSIGNMENT (MSA). Given a Boolean formula ϕ over a set $X = \{x_1, \ldots, x_n\}$ of variables, find the lexicographically largest $x_1 \cdots x_n \in \{0, 1\}^n$ that satisfies ϕ or 0 if ϕ is not satisfiable.

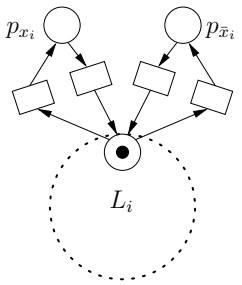
MSA stays $\mathbf{FP}^{\mathbf{NP}}$-complete for Boolean formulae in conjunctive normal form with at most three literals in each clause. Assume such a formula ϕ over a set $\{x_1, \ldots, x_n\}$ of n Boolean variables. First, the possible duplicate literals in each clause and tautological clauses in ϕ are removed. This can be accomplished in polynomial time without affecting the satisfying truth assignments of ϕ. For instance, a clause $x_2 \vee \neg x_7 \vee \neg x_7$ is replaced with $x_2 \vee \neg x_7$ and a clause $x_3 \vee \neg x_5 \vee x_5$ is removed. Assume that the resulting formula has m clauses $c_1, \ldots, c_m$. The set of Boolean variables appearing in a clause c_j is denoted by $\mathrm{vars}(c_j)$, e.g., $\mathrm{vars}(x_2 \vee \neg x_7) = \{x_2, x_7\}$.

A truth assignment $T : X' \to \{0, 1\}$ for a set $X' \subseteq X$ can also be represented by an index-sorted string such that

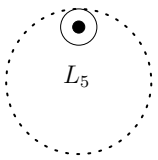
1. x_i is in the string if and only if $x_i \in X'$ and $T(x_i) = 1$ and
2. $\bar{x}_i$ is in the string if and only if $x_i \in X'$ and $T(x_i) = 0$.

For instance, a truth assignment $\{x_1 \mapsto 1, x_3 \mapsto 0\}$ for the Boolean variables x_1, x_3 is denoted by the string $x_1\bar{x}_3$. For a Boolean variable x_i, $1 \leq i \leq n$, and a truth assignment T for a set $X' \subseteq X$, the truth assignment $\mathit{flip}_i(T)$ is the same as T except that the value of x_i is swapped from 0 to 1 or vice versa if $x_i \in X'$. For instance, $\mathit{flip}_3(\{x_1 \mapsto 1, x_3 \mapsto 0\}) = \{x_1 \mapsto 1, x_3 \mapsto 1\}$ and $\mathit{flip}_2(\{x_1 \mapsto 1, x_3 \mapsto 0\}) = \{x_1 \mapsto 1, x_3 \mapsto 0\}$. Obviously, any other truth assignment for X' can be formed by applying a composition of at most n flipping functions to T.

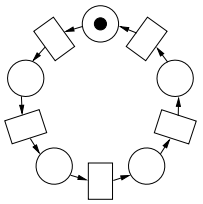
The net N for the formula ϕ is constructed as follows. First, for each Boolean variable $x_i \in X$, the net has the subnet



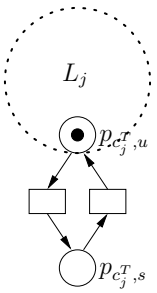
including the places p_{x_i} and $p_{\bar{x}_i}$, where the dotted circle named L_i is an abbreviation for a cycle net consisting of i places and transitions. For instance,



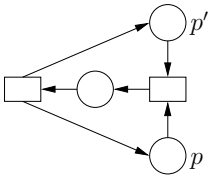
is an abbreviation for



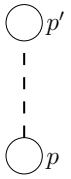
For each clause c_j in ϕ and for each (at most 8) truth assignment T for $\mathrm{vars}(c_j)$, the net has the subnet



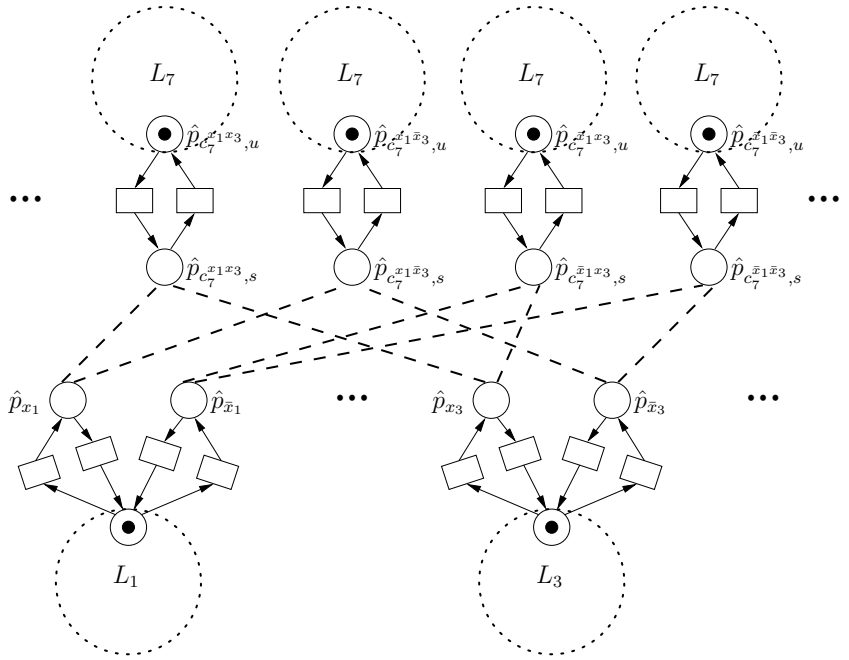
including the places $p_{c_j^T,s}$ and $p_{c_j^T,u}$ (T is represented in the string form), where L_j is as above. Each place p_{x_i} is connected to the place $p_{c_j^T,s}$ if and only if $x_i \in \mathrm{vars}(c_j)$ and $T(x_i) = 1$. Similarly, each place $p_{\bar{x}_i}$ is connected to the place $p_{c_j^T,s}$ if and only if $x_i \in \mathrm{vars}(c_j)$ and $T(x_i) = 0$. This connection between two places p and p' is made by the subnet



abbreviated by



For instance, a part of the net for a formula ϕ having a clause $c_7 = x_1 \vee \neg x_3$ is



The initial marking M_0 for the net is shown in the figures above, i.e., the cycle subnets each have one token in the place connected to the rest of the net. By construction, the net is 1-safe and live.

For each Boolean variable x_i, $1 \leq i \leq n$, there is a unique automorphism Nflip_i of the net corresponding to the truth value flipping flip_i of the value of x_i. That is, $\mathit{Nflip}_i(p_{x_i}) = p_{\bar{x}_i}$, $\mathit{Nflip}_i(p_{\bar{x}_i}) = p_{x_i}$, $\mathit{Nflip}_i(p_{x_k}) = p_{x_k}$ for $k \neq i$, $\mathit{Nflip}_i(p_{\bar{x}_k}) = p_{\bar{x}_k}$ for $k \neq i$, $\mathit{Nflip}_i(p_{c_j^T,s}) = p_{c_j^{\mathit{flip}_i(T)},s}$, $\mathit{Nflip}_i(p_{c_j^T,u}) = p_{c_j^{\mathit{flip}_i(T)},u}$, and the images of other places and transitions are uniquely determined. For instance, Nflip_3 maps p_{x_3} to $p_{\bar{x}_3}$, p_{x_1} to itself, and $p_{c_7^{\bar{x}_1\bar{x}_3},u}$ to $p_{c_7^{\bar{x}_1x_3},u}$. In fact, the group $\mathrm{Aut}(N)$ is generated by the set $\{\mathit{Nflip}_i \mid 1 \leq i \leq n\}$. Notice that (i) $\mathit{Nflip}_i \circ \mathit{Nflip}_j = \mathit{Nflip}_j \circ \mathit{Nflip}_i$ for each $1 \leq i, j \leq n$, (ii) $\mathit{Nflip}_i \circ \mathit{Nflip}_i = \mathbf{I}$ for each $1 \leq i \leq n$, and (iii) the group $\mathrm{Aut}(N)$ is an abelian 2-group of order 2^n. Furthermore, the group $\mathrm{Aut}(N)$ stabilizes the initial marking M_0.

Take a truth assignment T' for $X = \{x_1, \ldots, x_n\}$ formed from the truth assignment T_0 mapping each x_i to 0 by a sequence $\mathit{flip} = \mathit{flip}_{i_1} \circ \cdots \circ \mathit{flip}_{i_k}$ of truth value flippings, i.e., $\mathit{flip}(T_0) = T'$. Define the marking $M_{T'}$ corresponding to T' by:

1. For each $1 \leq i \leq n$, $M_{T'}(p_{x_i}) = T'(x_i)$ and $M_{T'}(p_{\bar{x}_i}) = 1 - T'(x_i)$. That is, the markings of the places p_{x_i} and $p_{\bar{x}_i}$ in $M_{T'}$ uniquely describe the truth assignment T'.
2. For each $1 \leq j \leq m$ and for each truth assignment T for the Boolean variables appearing in the clause c_j,
   (a) $M_{T'}(p_{c_j^T,s}) = 1$ if $\mathit{flip}(T)$ satisfies the clause c_j and 0 otherwise, and
   (b) $M_{T'}(p_{c_j^T,u}) = 0$ if $\mathit{flip}(T)$ satisfies the clause c_j and 1 otherwise.
3. $M_{T'}(p) = 0$ for all other places p in the net.

Obviously, $M_{T'}$ is reachable from the initial marking M_0. It is easy to show that $\mathit{Nflip}_i(M_{T'}) = M_{\mathit{flip}_i(T')}$ for each $1 \leq i \leq n$:

- For the place p_{x_i}, $\mathit{Nflip}_i(M_{T'})(p_{x_i}) = \mathit{Nflip}_i(M_{T'})(\mathit{Nflip}_i(p_{\bar{x}_i})) = M_{T'}(p_{\bar{x}_i}) = 1 - T'(x_i) = (\mathit{flip}_i(T'))(x_i) = M_{\mathit{flip}_i(T')}(p_{x_i})$.
- Similarly for the place $p_{\bar{x}_i}$.
- For each place p_{x_k}, where $i \neq k$, it holds that $\mathit{Nflip}_i(M_{T'})(p_{x_k}) = \mathit{Nflip}_i(M_{T'})(\mathit{Nflip}_i(p_{x_k})) = M_{T'}(p_{x_k}) = T'(x_k) = (\mathit{flip}_i(T'))(x_k) = M_{\mathit{flip}_i(T')}(p_{x_k})$.
- Similarly for each place $p_{\bar{x}_k}$, $i \neq k$.
- For each place $p_{c_j^T,s}$, $\mathit{Nflip}_i(M_{T'})(p_{c_j^T,s}) = M_{T'}(\mathit{Nflip}_i^{-1}(p_{c_j^T,s})) = M_{T'}(\mathit{Nflip}_i(p_{c_j^T,s}))$ because Nflip_i is its own inverse. It now holds that $M_{T'}(\mathit{Nflip}_i(p_{c_j^T,s})) = M_{T'}(p_{c_j^{\mathit{flip}_i(T)},s}) = 1$ if and only if $\mathit{flip}(\mathit{flip}_i(T))$ satisfies the clause c_j and 0 otherwise. But also $M_{\mathit{flip}_i(T')}(p_{c_j^T,s}) = 1$ if and only if $\mathit{flip}_i(\mathit{flip}(T)) = \mathit{flip}(\mathit{flip}_i(T))$ satisfies the clause c_j. Thus $\mathit{Nflip}_i(M_{T'})(p_{c_j^T,s}) = M_{\mathit{flip}_i(T')}(p_{c_j^T,s})$.
- Similarly for each place $p_{c_j^T,u}$.
- All the other places are empty in both $\mathit{Nflip}_i(M_{T'})$ and $M_{\mathit{flip}_i(T')}$.

Thus the orbit of a marking $M_{T'}$ consists exactly of all the markings corresponding to truth assignments for X. Note especially that $M_{T'}(p_{c_j^{T_0},s}) = 1$, where T_0 is the truth assignment mapping all Boolean variables to 0, if and only if the clause c_j is satisfied in the truth assignment T'. Now define the base β for the net by listing (i) first all the m places of form $p_{c_j^{T_0},s}$ (ii) then the places $p_{x_1}, \ldots, p_{x_n}$ in that order, and (iii) finally the rest of the elements of the net. Take a marking $M_{T'}$ corresponding to a truth assignment T'. The lexicographically greatest marking M in the orbit of $M_{T'}$ under β has the following property:

1. $M(p_{c_j^{T_0},s}) = 1$ for all the first m places in β if and only if the formula ϕ is satisfiable, and
2. if this is the case, the markings for next n places of form $p_{x_1}, \ldots, p_{x_n}$ describe the lexicographically greatest satisfying truth assignment.

□

Corollary 3.24 *The problems* LGM *and* LGM2 *are both* $\mathbf{FP}^{\mathbf{NP}}$*-complete.*

The string orbit problem revisited. Recall the STRING ORBIT problem defined in page 32. As a direct consequence of the above theorem, the following problem is also $\mathbf{FP}^{\mathbf{NP}}$-complete: Given a Σ-string s on an ordered index set I and a permutation group G on I, find the lexicographically greatest string that is G-equivalent to s. This holds when G is an abelian 2-group, too. Note that the string canonization algorithm presented in [Babai and Luks 1983] provides a canonical representative (*not* the lexicographical leader under the given base) for the nets and markings used in the proof of Theorem 3.23 in polynomial time.

3.4 SYMMETRIC COVERABILITY

In order to verify the boundedness of a P/T-net, a coverability graph of the net can be constructed [Karp and Miller 1969; Finkel 1990]. A marking M is said to *cover* a marking M' if $M' \leq M$. In order to build the coverability graph, markings are extended to be functions of form $M : P \to (\mathbb{N} \cup \{\omega\})$, where ω is a symbol not in $\mathbb{N}$ and for all $x \in \mathbb{N} \cup \{\omega\}$, $x \leq \omega$. The coverability graph construction can be combined with the symmetry reduction method [Huber et al. 1985a; Petrucci 1990]. The following definitions are from [Schmidt 2000a].

Definition 3.25 *A marking M* symmetrically covers *a marking M', denoted by $M' \leqq M$, if there is a symmetry $\sigma \in \mathrm{Aut}(N)$ such that $M' \leq \sigma(M)$.*

Problem 3.26 SYMMETRIC COVERABILITY. *Given a net N and two of its markings, M and M', does M symmetrically cover M'?*

The algorithm for solving whether two markings are equivalent, presented in [Schmidt 2000a], is extended in the same paper to solve the SYMMETRIC COVERABILITY problem, too. Interestingly, the complexity of SYMMETRIC COVERABILITY jumps from GRAPH ISOMORPHISM to **NP**-completeness, a phenomenon resembling that happening when moving from the GRAPH ISOMORPHISM to SUBGRAPH ISOMORPHISM problem [Garey and Johnson 1979].

Theorem 3.27 SYMMETRIC COVERABILITY *is* **NP***-complete.*

Proof. Obviously SYMMETRIC COVERABILITY is in **NP**. **NP**-hardness is shown by reduction from the **NP**-complete problem CLIQUE asking if an undirected graph $G = \langle V, E \rangle$ has a clique of size k or more (it can be assumed that $k \geq 2$). The graph G is assumed to have a reflexive edge set meaning that all vertices have a self-loop. Construct the net $\hat{N}$ and as in the proof of Theorem 3.13. For the graph $G = \langle V, E \rangle$, construct the marking

$$\hat{M}_G = \sum_{\langle v,v' \rangle \in E} 1' \hat{p}^e_{v,v'}.$$

Take any subset V' of V such that $|V'| = k$ and build the marking $\hat{M}_k = \sum_{v,v' \in V'} 1' \hat{p}^e_{v,v'}$ corresponding to a k-clique. Now clearly $\hat{M}_G$ symmetrically covers $\hat{M}_k$ if and only if G has a clique of size k or more. □

Remark 3.28 *Again, the complexity of* SYMMETRIC COVERABILITY *does not depend on whether the automorphism group of the net in question is given as input. Furthermore, it does not depend on the extension of markings with the ω symbol.*

An elegant way to solve the symmetric coverability problem would be to define a canonical representative function that solves the coverability problem at the same time:

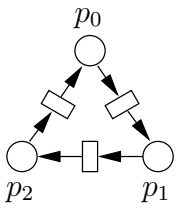


Figure 3.6: A net with no suitable canonical representative function

Definition 3.29 *A canonical representative function canrepr is* suitable for symmetric coverability *if* $\mathit{canrepr}(M') \leq \mathit{canrepr}(M) \Leftrightarrow M' \leqq M$ *for all* $M, M' \in \mathbb{M}$.

Unfortunately, suitable representative functions do not always exist, as is shown in the next example and theorem.

Example 3.30 The function that selects the lexicographically greatest marking in the orbit is *not* a suitable canonical representative function for all nets. For a counter-example, consider the net in Figure 3.6 and assume a base $\beta = [p_0, p_1, p_2, \ldots]$. Now the marking $M = 2'p_0 + 2'p_1 + 0'p_2$ is its own representative $\mathit{canrepr}(M)$, while for $M' = 0'p_0 + 1'p_1 + 2'p_2$ the representative is $\mathit{canrepr}(M') = 2'p_0 + 0'p_1 + 1'p_2$. Now M symmetrically covers M' since $\sigma(M) = 0'p_0 + 2'p_1 + 2'p_2 \geq M'$, where σ maps each p_i to $p_{i+1 \bmod 3}$. But $\mathit{canrepr}(M') \leq \mathit{canrepr}(M)$ does not hold. ♣

Theorem 3.31 *There are nets for which suitable canonical representative functions do not exist.*

Proof. Assume that such functions exist for all nets. Consider again the net N in Figure 3.6. Take the marking $M = 2'p_0 + 2'p_1 + 0'p_2$ and any of its representatives, say $\mathit{canrepr}(M) = M$. Consider two other markings, $M_1 = 2'p_0 + 1'p_1 + 0'p_2$ and $M_2 = 1'p_0 + 2'p_1 + 0'p_2$. Clearly M symmetrically covers both M_1 and M_2. In order to *canrepr* to be suitable for symmetric coverability, it must be that $\mathit{canrepr}(M_1) = M_1$ and $\mathit{canrepr}(M_2) = M_2$ (other representatives lead to a situation in which place p_2 has one or more tokens and thus $\mathit{canrepr}(M)$ would not cover them). Now consider the marking $M' = 2'p_0 + 1'p_1 + 1'p_2$ which symmetrically covers both markings M_1 and M_2. For *canrepr* to be suitable, it must be that $\mathit{canrepr}(M') = M'$ since other representatives do not cover $\mathit{canrepr}(M_1)$. But now $\mathit{canrepr}(M')$ does not cover $\mathit{canrepr}(M_2)$. Thus the initial assumption must be wrong and suitable canonical representative functions do not exist for all nets. □

4 PLACE/TRANSITION NETS: NEW CANONICAL MARKING ALGORITHMS

The previous chapter studied the computational complexity issues concerning the symmetry reduction method for place/transition nets. In addition, algorithms for computing the automorphism group of the net, i.e., for finding the symmetries, were described in Section 3.2.2. This chapter studies the algorithms for the next step in the symmetry reduction method, i.e., for exploiting the symmetries during the reduced reachability graph generation. As described in Section 2.2.2, this requires an algorithm either

1. deciding whether two markings are equivalent under the symmetries, or
2. building canonical representatives for markings.

Some algorithms for the task in the context of P/T-nets are described in [Schmidt 2000a; 2000b]:

- The first algorithm, "iterating the symmetries", applies all the symmetries to the new marking and checks whether the resulting marking has already been visited during the reduced reachability graph construction. The facts that (i) the symmetries are stored in a special form called Schreier-Sims representation (described in Section 4.1.1), and (ii) the set of already visited markings is stored as a prefix sharing decision tree, are exploited to prune the set of symmetries that have to considered.
- The second algorithm, "iterating the states", pairwisely checks the new marking with each already visited marking for equivalence by using the algorithm described in [Schmidt 2000a]. The set of necessary equivalence tests is reduced by using symmetry-respecting hash functions. This approach does not need the pre-calculation of the symmetries of the net.
- The third algorithm, "canonical representatives", computes a (non-canonical) representative for the newly generated marking. This is done by a *limited* search with greedy heuristics in the Schreier-Sims representation of symmetries, trying to find the lexicographically smallest equivalent marking.

The new algorithms described in this chapter follow the canonical representative function approach. That is, they describe how to compute a function

$$\mathit{canrepr} : \mathbb{M} \to \mathbb{M}$$

such that

1. $\mathit{canrepr}(M) \equiv_G M$, and
2. $M_1 \equiv_G M_2$ implies $\mathit{canrepr}(M_1) = \mathit{canrepr}(M_2)$,

where G is the applied symmetry group. All the new algorithms presented require that the symmetry group of the net is known and stored in a standard form called Schreier-Sims representation. This is not a serious drawback because it is beneficial to first compute the symmetry group of the net in

order to see whether there are any non-trivial symmetries, i.e., to see whether the symmetry reduction method can help at all. In addition, the performance of symmetry reduction algorithms may depend on the size of the symmetry group, see [Schmidt 2000b] and Section 4.5.2, and thus knowing it may help in selecting an appropriate algorithm.

The first new algorithm presented in Section 4.2 uses a black box graph canonizer algorithm to produce a canonical representative for a marking. First, the characteristic graph of the marking is build. Characteristic graphs have the property that the characteristic graphs of two markings are isomorphic if and only if the markings are equivalent. Furthermore, the isomorphisms between the characteristic graphs correspond exactly to the symmetries transforming the markings to each other. The canonical version of the characteristic graph of a marking is then obtained by applying a black box graph canonizer, and finally the canonical representative for the marking is obtained by using an isomorphism between the characteristic graph and its canonical version. In Section 7.6, an analogous algorithm is described for high-level Petri nets and similar formalisms.

The second algorithm, presented in Section 4.3, is a backtracking search algorithm in the Schreier-Sims representation of the symmetry group. The algorithm returns the smallest marking produced by symmetries that are "compatible" with the marking in question. The search is pruned (i) by considering only symmetries that are "compatible" with the marking, (ii) by using the smallest already found equivalent marking, and (iii) by exploiting the stabilizers of the marking (which are found during the search). This algorithm is a variant of the backtracking search algorithms developed in computational group theory, see e.g. [Butler 1991]. However, the compatibility definition between symmetries and markings is, to author's knowledge, novel. Moreover, the algorithm can be seen as a complete, canonical version of the "canonical representatives" algorithm described in [Schmidt 2000b] augmented with effective pruning techniques.

The third algorithm presented in Section 4.4 combines the techniques used in Sections 4.2 and 4.3 by "opening" the black box graph canonizer. A standard preprocessing technique of existing graph isomorphism algorithms (see e.g. [McKay 1981; Kreher and Stinson 1999]) is used to produce an ordered partition of the marking in question in a symmetry-respecting way. The partition is then used to prune the backtrack search in the Schreier-Sims representation by considering only symmetries that are compatible with the partition.

The algorithms and results presented in this chapter have been published in [Junttila 2002a].

4.1 PRELIMINARIES

Some common preliminaries for the proposed algorithms are given first.

4.1.1 The Schreier-Sims Representation

Although a permutation group on a set of n elements may have up to $n!$ permutations, there are representations for permutation groups that have size polynomial in n. The following text describes one standard representation form that has some useful properties exploited later in this chapter. For more on permutation group algorithms, see [Butler 1991]. The presentation here is based on [Kreher and Stinson 1999].

Assume a finite set X and a permutation group G on X. For instance, X may be the set $P \cup T$ and G the group $\mathrm{Aut}(N)$ for a P/T-net $N = \langle P, T, F, W, M_0 \rangle$. Assume that $|X| = n$ and order the elements in X in any order $\beta = [x_1, x_2, \ldots, x_n]$. Let

$$\begin{aligned} G_0 &= G \\ G_1 &= \{g \in G_0 \mid g(x_1) = x_1\} \\ G_2 &= \{g \in G_1 \mid g(x_2) = x_2\} \\ &\vdots \\ G_n &= \{g \in G_{n-1} \mid g(x_n) = x_n\}\,. \end{aligned}$$

The groups $G_0, G_1, \ldots, G_n$ are subgroups of G such that

$$G = G_0 \geq G_1 \geq \cdots \geq G_n = \{\mathbf{I}\}$$

where $\mathbf{I}$ denotes the identity permutation. Note that a permutation $g \in G_i$, $0 \leq i \leq n$, fixes each element $x_1, \ldots, x_i$. For each $1 \leq i \leq n$, let $[x_i]_{G_{i-1}} = \{g(x_i) \mid g \in G_{i-1}\}$ denote the orbit of x_i under G_{i-1}. Assume that $[x_i]_{G_{i-1}} = \{x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}\}$ for an $1 \leq n_i \leq n$. For each $1 \leq j \leq n_i$, choose a $h_{i,j} \in G_{i-1}$ such that $h_{i,j}(x_i) = x_{i,j}$ and let $U_i = \{h_{i,1}, h_{i,2}, \ldots, h_{i,n_i}\}$. Now U_i is a left transversal of G_i in G_{i-1}, i.e., $h_{i,j} \circ G_i \neq h_{i,k} \circ G_i$ for $j \neq k$ and $G_{i-1} = h_{i,1} \circ G_i \cup \cdots \cup h_{i,n_i} \circ G_i$, where $h \circ G_i$ denotes the left coset $\{h \circ g \mid g \in G_i\}$. The structure $\vec{G} = [U_1, U_2, \ldots, U_n]$ is a *Schreier-Sims representation* of the group G. Each element in $g \in G$, and only those, can be uniquely written as a composition $g = h_1 \circ h_2 \circ \cdots \circ h_n$, where $h_i \in U_i$, and thus the order of G equals to $|U_1||U_2| \cdots |U_n|$. The ordering $\beta = [x_1, x_2, \ldots, x_n]$ is called the *base* of the representation. It can be and is assumed from now on that each U_i contains the identity permutation $\mathbf{I}$. As each U_i contains at most $n - i + 1$ permutations, there are at most $n(n+1)/2$ permutations in the Schreier-Sims representation $\vec{G} = [U_1, U_2, \ldots, U_n]$. Many operations, such as testing whether a permutation belongs to the group, can be performed in polynomial time by using Schreier-Sims representations. Furthermore, given a generating set of permutations for a group, the Schreier-Sims representation for the group can be calculated in polynomial time.

The ground sets in [Schmidt 2000a; 2000b] are actually Schreier-Sims representations. Thus the algorithm for computing the symmetries of a net presented in [Schmidt 2000a] produces a Schreier-Sims representation of the symmetry group.

Finally, note that a more compact representation consisting of at most $n-1$ permutations could also be used instead of the Schreier-Sims representation [Jerrum 1986].

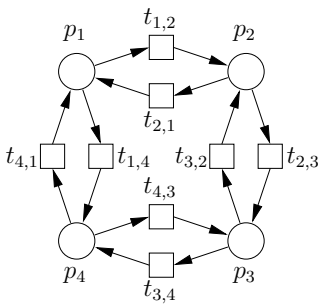


Figure 4.1: An example net

Example 4.1 Consider the net in Figure 4.1. Its symmetry group, call it G, under the base

$$\beta = [p_1, p_2, p_3, p_4, t_{1,2}, t_{2,1}, t_{2,3}, t_{3,2}, t_{3,4}, t_{4,3}, t_{4,1}, t_{1,4}]$$

has a Schreier-Sims representation $\vec{G} = [U_1, U_2, \ldots, U_{|P|+|T|}]$, where

$$\begin{aligned}
U_1 &= \left\{ \begin{array}{l} h_{1,1} = \mathbf{I} \\ h_{1,2} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_2 & p_3 & p_4 & p_1 & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} & t_{1,2} & t_{2,1} \end{pmatrix} \\ h_{1,3} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_3 & p_4 & p_1 & p_2 & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} \end{pmatrix} \\ h_{1,4} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_4 & p_1 & p_2 & p_3 & t_{4,1} & t_{1,4} & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} \end{pmatrix} \end{array} \right\}, \\
U_2 &= \left\{ \begin{array}{l} h_{2,1} = \mathbf{I} \\ h_{2,2} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_1 & p_4 & p_3 & p_2 & t_{1,4} & t_{4,1} & t_{4,3} & t_{4,2} & t_{3,2} & t_{2,3} & t_{2,1} & t_{1,2} \end{pmatrix} \end{array} \right\}, \text{ and} \\
U_i &= \{\mathbf{I}\} \text{ for } 3 \leq i \leq |P| + |T|.
\end{aligned}$$

Therefore, $|G| = 8$. ♣

4.1.2 Compatible Permutations

In addition to the standard Schreier-Sims representation definitions above, some new concepts are needed in the rest of the chapter.

To facilitate the understanding of the following concepts, a Schreier-Sims representation $\vec{G} = [U_1, \ldots, U_n]$ of a permutation group G on a set X under a base $\beta = [x_1, \ldots, x_n]$ can be seen as a tree. The levels of the tree correspond to the base of the representation and each node at a level i has $|U_i|$ children at the level $i + 1$, the edges to the children being labeled with the permutations in U_i. For instance, Figure 4.2 shows (a prefix of) the tree corresponding to the Schreier-Sims representation in Example 4.1.

Consider a path in the tree starting in the root and ending in a node v at a level i. Composing the labels of the edges in the path defines the corresponding permutation $g \in U_1 \circ \cdots \circ U_{i-1} \subseteq G$. Thus the full paths ending in leaf nodes of the tree define exactly the permutations in the group. The node v has $|U_i|$ child nodes, and extending the path to any of them defines an extension permutation of g which is in $g \circ U_i$. The set $\{g(h(x_i)) \mid h \in U_i\}$ is now the set of $|U_i|$ possible images of the ith base element x_i under all the permutations corresponding to the paths going through the node v. Figure 4.3

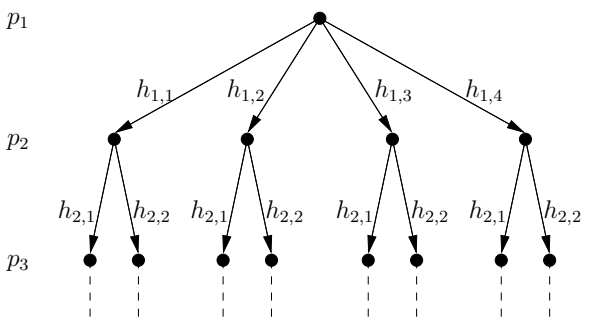


Figure 4.2: Schreier-Sims representation seen as a tree

shows the tree in Figure 4.2 when the base element images are augmented in the edges. For instance, consider the "fourth" path with the edges $h_{1,2}$ and $h_{2,2}$. The image of the first base element p_1 is now $h_{1,2}(p_1) = p_2$ and the second base element p_2 is mapped to $h_{1,2}(h_{2,2}(p_2)) = h_{1,2}(p_4) = p_1$.

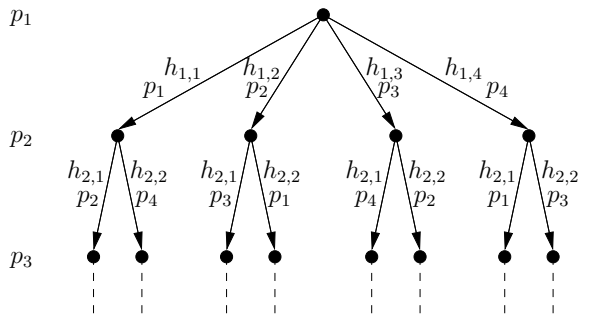


Figure 4.3: Schreier-Sims representation tree augmented with the base element images

Assume that the elements in the permuted set X are associated with natural numbers by a valuation function $pval : X \to \mathbb{N}$. Now the edges in the tree can be augmented with the values of the base element images assigned by *pval*. For instance, Figure 4.4 shows the tree in Figure 4.3 augmented in this way when $pval = \{p_1 \mapsto 1, p_2 \mapsto 0, p_3 \mapsto 0, p_4 \mapsto 0, t_{1,2} \mapsto 0, \ldots, t_{1,4} \mapsto 0\}$.

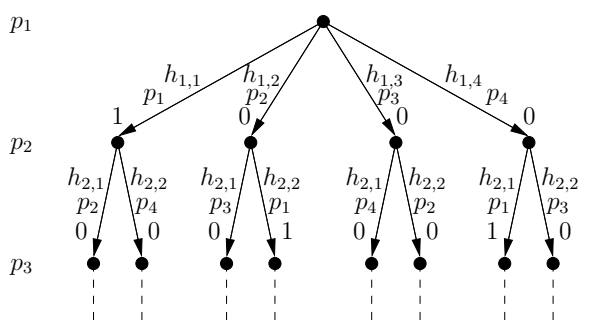
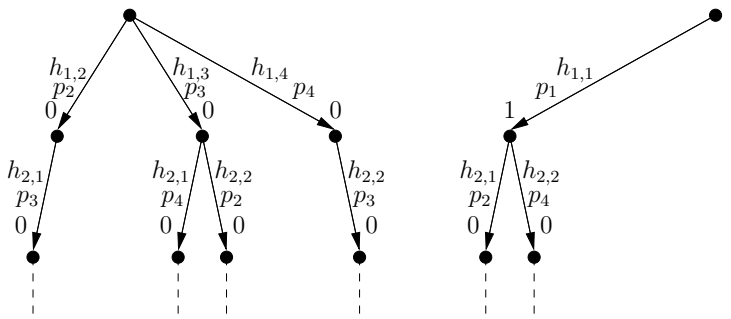


Figure 4.4: Schreier-Sims representation tree augmented with the base element images and their values

Next, consider a node v at level i in the tree. It has $|U_i|$ children and the edges to children are weighted by the valuation *pval* in the way described

above. The weights form a multiset over natural numbers. For instance, the multiset for the root node in Figure 4.4 is $3'0 + 1'1$ and the multiset for the second node from the left in the second level is $1'0 + 1'1$. The idea now is to prune the tree by considering only a nonempty subset of children of each node in the way from root to leaf nodes. This pruning is done by applying a *multiset selector* to the multiset of edge weights leaving the node. The multiset selector chooses a nonempty set of "good" edge weights. Only the children of the node reachable via an edge with "good" weight are then considered and the rest are pruned away. Formally, a multiset selector is a function from nonempty multisets over natural numbers to nonempty sets of natural numbers such that each number in the image set has a non-zero multiplicity in the argument multiset. That is, if *select* is a multiset selector and $n \in \mathit{select}(m)$, then $m(n) \geq 1$. For instance, the *trivial* multiset selector is $\mathit{select}_{\text{trivial}} = \{n \mid m(n) \geq 1\}$, e.g. $\mathit{select}_{\text{trivial}}(3'2 + 2'4 + 2'5 + 4'7) = \{2, 4, 5, 7\}$. For a better example, define the *minimal element* multiset selector $\mathit{select}_{\text{min}}$ such that $\mathit{select}_{\text{min}}(m) = \{n\}$, where n is the smallest number that has non-zero multiplicity in m. Now $\mathit{select}_{\text{min}}(3'2 + 2'4 + 2'5 + 4'7) = \{2\}$. Similarly, the *maximal element* multiset selector $\mathit{select}_{\text{max}}$ would give $\mathit{select}_{\text{max}}(3'2 + 2'4 + 2'5 + 4'7) = \{7\}$. Also define the *minimal element with minimal frequency* multiset selector $\mathit{select}_{\text{minminfreq}}$ such that $\mathit{select}_{\text{minminfreq}}(m) = \{n\}$, where n is the smallest number among those that have the smallest non-zero multiplicity in m. E.g., $\mathit{select}_{\text{minminfreq}}(3'2 + 2'4 + 2'5 + 4'7) = \{4\}$. Similarly, the *maximal element with minimal frequency* multiset selector $\mathit{select}_{\text{maxminfreq}}$ would give $\mathit{select}_{\text{maxminfreq}}(3'2 + 2'4 + 2'5 + 4'7) = \{5\}$. A *function* multiset selector is a multiset selector for which the image set always contains exactly one element. All the other multiset selectors above except $\mathit{select}_{\text{trivial}}$ clearly fulfill this condition. Figure 4.5(a) shows the tree in Figure 4.4 pruned by applying the minimal element multiset selector $\mathit{select}_{\text{min}}$, and Figure 4.5(b) shows the result when the maximal element with minimal frequency multiset selector is applied instead.



(a) $\mathit{select}_{\text{min}}$ (b) $\mathit{select}_{\text{maxminfreq}}$

Figure 4.5: Pruned Schreier-Sims representation trees

The following definition formalizes the above discussed pruning procedure. The permutations corresponding to the full paths in the tree that survive the pruning will be called compatible. Assume a fixed multiset selector *select*, a permutation group G on a set X with $|X| = n$ and a Schreier-Sims representation $\vec{G} = [U_1, \ldots, U_n]$ of G under a base $\beta = [x_1, \ldots, x_n]$.

Definition 4.2 *A permutation $g_1 \circ \cdots \circ g_n \in G$, where $g_j \in U_j$ for each $1 \leq j \leq n$, is* compatible *with a valuation pval : $X \to \mathbb{N}$ if*

$$pval((g_1 \circ \cdots \circ g_{i-1} \circ g_i)(x_i)) \in select\left(\sum_{h \in U_i} 1' pval((g_1 \circ \cdots \circ g_{i-1} \circ h)(x_i))\right)$$

holds for each $1 \leq i \leq n$ (when $i = 1$, $g_1 \circ \cdots \circ g_{i-1} = \mathbf{I}$).

Note that there is always at least one permutation compatible with the valuation. Furthermore, it is straightforward to see that if

- the valuation *pval* is an injection, i.e., $pval(x) = pval(x') \Rightarrow x = x'$ for all elements $x, x' \in X$, and
- *select* is a function multiset selector,

then there is exactly one element in G that is compatible with *pval*. Define the action of G on the valuation functions $pval : X \to \mathbb{N}$ by $g(pval) = pval \circ g^{-1}$ (or equivalently, $(g(pval))(g(x)) = pval(x)$ for each $x \in X$) for each $g \in G$. The following property of compatibility is crucial in latter sections.

Theorem 4.3 *Let $g \in G$. A permutation $\hat{g} \in G$ is compatible with a valuation pval if and only if the permutation $g \circ \hat{g} \in G$ is compatible with the permuted valuation $g(pval)$.*

Proof. Assume that $\hat{g}_1 \circ \cdots \circ \hat{g}_n$ is the unique representation of $\hat{g}$ in the fixed Schreier-Sims representation of G. Similarly, let $\hat{g}'_1 \circ \cdots \hat{g}'_n$ be the unique representation of $\hat{g}' = g \circ \hat{g}$. Fix any i, $1 \leq i \leq n$. It has to be shown that

$$pval((\hat{g}_1 \circ \cdots \circ \hat{g}_i)(x_i)) \in select\left(\sum_{h \in U_i} 1' pval(\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(x_i) \mid h \in U_i\})\right)$$

if and only if

$$(g(pval))((\hat{g}'_1 \circ \cdots \circ \hat{g}'_i)(x_i)) \in select\left(\sum_{h \in U_i} 1' (g(pval))(\{(\hat{g}'_1 \circ \cdots \circ \hat{g}'_{i-1} \circ h)(x_i) \mid h \in U_i\})\right).$$

First, note that $(\hat{g}_1 \circ \cdots \circ \hat{g}_i)(x_i) = \hat{g}(x_i)$ because the "postfix" permutation $\hat{g}_{i+1} \circ \cdots \circ \hat{g}_n$ of $\hat{g}$ fixes x_i. Similarly, $(\hat{g}'_1 \circ \cdots \circ \hat{g}'_i)(x_i) = \hat{g}'(x_i) = g(\hat{g}(x_i))$. Thus $pval((\hat{g}_1 \circ \cdots \circ \hat{g}_i)(x_i)) = pval(\hat{g}(x_i))$ and $(g(pval))((\hat{g}'_1 \circ \cdots \circ \hat{g}'_i)(x_i)) = (g(pval))(\hat{g}'(x_i)) = (pval \circ g^{-1})(g(\hat{g}(x_i))) = pval(\hat{g}(x_i))$, implying

$$pval((\hat{g}_1 \circ \cdots \circ \hat{g}_i)(x_i)) = (g(pval))((\hat{g}'_1 \circ \cdots \circ \hat{g}'_i)(x_i)). \tag{4.1}$$

Second, note that $\{h(x_i) \mid h \in U_i\} = [x_i]_{G_{i-1}}$, i.e., the orbit of x_i under G_{i-1}. Therefore, the set $\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(x_i) \mid h \in U_i\}$ equals to $(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1})([x_i]_{G_{i-1}})$. As the last permutations $\hat{g}_i \circ \cdots \circ \hat{g}_n$ in the representation of $\hat{g}$ belong to the subgroup G_{i-1}, $(\hat{g}_i \circ \cdots \circ \hat{g}_n)([x_i]_{G_{i-1}}) = [x_i]_{G_{i-1}}$. Thus $\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(x_i) \mid h \in U_i\} = (\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1})([x_i]_{G_{i-1}}) = \hat{g}([x_i]_{G_{i-1}})$.

Similarly, the set $\{(\hat{g}'_1 \circ \cdots \circ \hat{g}'_{i-1} \circ h)(x_i) \mid h \in U_i\}$ equals to $\hat{g}'([x_i]_{G_{i-1}}) = (g \circ \hat{g})([x_i]_{G_{i-1}}) = g\left(\hat{g}([x_i]_{G_{i-1}})\right)$ Therefore,

$$\{(\hat{g}'_1 \circ \cdots \circ \hat{g}'_{i-1} \circ h)(x_i) \mid h \in U_i\} = g\left(\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(x_i) \mid h \in U_i\}\right).$$

This implies that

$$\begin{aligned}
\sum_{h \in U_i} 1'(g(\mathit{pval}))(\{(\hat{g}'_1 \circ \cdots \circ \hat{g}'_{i-1} \circ h)(p_{\beta,i}) \mid h \in U_i\}) &= \\
\sum_{h \in U_i} 1'(\mathit{pval} \circ g^{-1})(g\left(\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(p_{\beta,i}) \mid h \in U_i\}\right)) &= \\
\sum_{h \in U_i} 1'\mathit{pval}(\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(p_{\beta,i}) \mid h \in U_i\})
\end{aligned}$$

and thus

$$\begin{aligned}
\mathit{select}(\sum_{h \in U_i} 1'(g(\mathit{pval}))(\{(\hat{g}'_1 \circ \cdots \circ \hat{g}'_{i-1} \circ h)(p_{\beta,i}) \mid h \in U_i\})) &= \\
\mathit{select}(\sum_{h \in U_i} 1'\mathit{pval}(\{(\hat{g}_1 \circ \cdots \circ \hat{g}_{i-1} \circ h)(p_{\beta,i}) \mid h \in U_i\})).
\end{aligned}$$

□

The theory above was developed for arbitrary permutation groups. However, applying it to P/T-nets and their automorphism groups is straightforward. Assume a P/T-net $N = \langle P, T, F, W, M_0 \rangle$ and a subgroup G of $\mathrm{Aut}(N)$. Any Schreier-Sims representation $\vec{G} = [U_1, \ldots, U_{|P|+|T|}]$ of G is from now on assumed to be given under a base $\beta = [p_{\beta,1}, \ldots, p_{\beta,|P|}, t_{\beta,1}, \ldots, t_{\beta,|T|}]$ in which the places are enumerated before transitions. It can be safely assumed that each set U_j, where $|P| + 1 \leq j \leq |P| + |T|$, contains only the identity permutation. If this were not the case, then the subgroup of G stabilizing each place would be non-trivial and the net would contain identical transitions (that is, transitions that consume the same number of tokens from the same places and produce the same number of places to the same places). Such transitions can be safely identified. Under these assumptions, the element valuation functions are called *place valuations* and are restricted to be functions of form $\mathit{pval} : P \to \mathbb{N}$ (it is implicitly defined that $\mathit{pval}(t) = 0$ for each transition $t \in T$). Observe that this definition is exactly the same as for markings, a different name is only used in order to avoid confusions later. The action of permutations in $\mathrm{Aut}(N)$ on place valuations is also defined similarly to that on markings.

Example 4.4 Recall the net in Figure 4.1 and the Schreier-Sims representation of its automorphism group G described in Example 4.1. Assume a place valuation $\mathit{pval} = \{p_1 \mapsto 1, p_2 \mapsto 0, p_3 \mapsto 0, p_4 \mapsto 0\}$ and the minimal element multiset selector $\mathit{select}_{\min}$. Now

$$\begin{aligned}
\mathit{select}_{\min}\left(\sum_{h \in U_1} 1'\mathit{pval}(h(p_{\beta,1}))\right) &= \\
\mathit{select}_{\min}\left(1'\mathit{pval}(p_1) + 1'\mathit{pval}(p_2) + 1'\mathit{pval}(p_3) + 1'\mathit{pval}(p_4)\right) &= \\
\mathit{select}_{\min}\left(1'1 + 1'0 + 1'0 + 1'0\right) &= \\
\mathit{select}_{\min}\left(3'0 + 1'1\right) &= \{0\}
\end{aligned}$$

and thus $pval(\hat{g}_1(p_1)) = 0$ must hold for any permutation $\hat{g} = \hat{g}_1 \circ \cdots \circ \hat{g}_{12}$, $\hat{g}_i \in U_i$ for each $1 \leq i \leq 12$, that is compatible with *pval*. This requirement is fulfilled by $h_{1,2}$, $h_{1,3}$ and $h_{1,4}$.

If $\hat{g}_1 = h_{1,2}$, then

$$\begin{aligned} select_{\min}\left(\sum_{h \in U_2} 1'pval(h_{1,2}(h(p_{\beta,2})))\right) &= \\ select_{\min}\left(1'pval(p_3) + 1'pval(p_1)\right) &= \\ select_{\min}\left(1'1 + 1'0\right) &= \{0\} \end{aligned}$$

and thus $pval(h_{1,2}(\hat{g}_2(p_2))) = 0$ must hold for any permutation $\hat{g} = h_{1,2} \circ \hat{g}_2 \cdots \circ \hat{g}_{12}$ that is compatible with *pval*. This requirement is fulfilled by $h_{2,1}$. Because $U_i = \{\mathbf{I}\}$ for $i \geq 3$, the symmetry $h_{1,2} \circ h_{2,1} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \ldots \\ p_2 & p_3 & p_4 & p_1 & t_{2,3} & \ldots \end{pmatrix}$ is compatible with *pval*.

Similar computations show that the other permutations compatible with *pval* are

$$\begin{aligned} h_{1,3} \circ h_{2,1} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \ldots \\ p_3 & p_4 & p_1 & p_2 & t_{3,4} & \ldots \end{pmatrix}, \\ h_{1,3} \circ h_{2,2} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \ldots \\ p_3 & p_2 & p_1 & p_4 & t_{3,2} & \ldots \end{pmatrix}, \text{ and} \\ h_{1,4} \circ h_{2,2} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \ldots \\ p_4 & p_3 & p_2 & p_1 & t_{4,3} & \ldots \end{pmatrix}. \end{aligned}$$

To sum up, there are 4 permutations that are compatible with *pval*. Note that these permutations correspond to the paths in the pruned tree in Figure 4.5(a).

Observe that if the maximal element with minimal frequency multiset selector were used instead, only 2 permutations, namely $h_{1,1} \circ h_{2,1}$ and $h_{1,1} \circ h_{2,2}$, would be compatible with *pval*. These permutations correspond to the paths in the pruned tree in Figure 4.5(b). ♣

Algorithm 4.1 describes the obvious depth-first backtrack search algorithm enumerating all permutations compatible with a place valuation.

Algorithm 4.1 An algorithm enumerating all compatible permutations

function *compatible_permutations*(pval)
1: Call $backtrack(1, \mathbf{I})$
function $backtrack(l, \hat{g})$
Require: l is the backtracking level
Require: $\hat{g}$ is the currently enumerated compatible permutation
2: **if** $l = |P| + 1$ **then**
3: Report $\hat{g}$
4: **return**
5: Evaluate $S = select(\Sigma_{h \in U_l} 1'pval(\hat{g}(h(p_{\beta,l}))))$
6: **for all** $h \in U_l$ such that $pval(\hat{g}(h(p_{\beta,l}))) \in S$ **do**
7: Call $backtrack(l + 1, \hat{g} \circ h)$
8: **return**

4.2 USING THE CANONICAL VERSION OF THE CHARACTERISTIC GRAPH

The first canonical representative marking function presented is based on the use of graph canonizers, i.e., functions that transform graphs to unique, isomorphic representatives called canonical versions. For instance, the *nauty* tool implements such a function [McKay 1990]. First, the characteristic graph of the marking in question is built. The canonical version of the characteristic graph is then obtained by using the graph canonizer. Finally, the canonical representative for the marking is obtained from a mapping transforming the characteristic graph to its canonical version.

Characteristic Graphs. Consider a P/T-net $N = \langle P, T, F, W, M_0 \rangle$ and the stabilizer group $G = \mathrm{Stab}(N, \hat{M})$ of a marking $\hat{M}$. Usually, $\hat{M}$ is either the initial marking M_0 or the empty marking (in the latter case, $\mathrm{Stab}(N, \hat{M})$ equals to $\mathrm{Aut}(N)$). A *characteristic graph assigner* (under G) is a function that assigns each marking M a graph $\mathcal{G}_M$ (in a fixed class of graphs) such that its vertex set contains $P \cup T$ and for all markings M_1, M_2 of N it holds that

1. if $g \in G$ maps a marking M_1 to M_2, then there is an isomorphism γ from $\mathcal{G}_{M_1}$ to $\mathcal{G}_{M_2}$ such that γ restricted to $P \cup T$ equals to g, and
2. if γ is an isomorphism from $\mathcal{G}_{M_1}$ to $\mathcal{G}_{M_2}$, then (i) $\gamma(P) = P$, (ii) $\gamma(T) = T$, and (iii) γ restricted to $P \cup T$ belongs to G and maps M_1 to M_2.

Then the graph $\mathcal{G}_M$ is called the *characteristic graph* of M. Clearly, two markings are equivalent under G if and only if their characteristic graphs are isomorphic. Thus testing whether two markings are equivalent under G can be done by (i) building their characteristic graphs, and (ii) testing whether the characteristic graphs are isomorphic by using a tool for solving the graph isomorphism problem. Furthermore, the stabilizer group $\mathrm{Stab}(G, M)$ can be easily retrieved from the automorphism group of $\mathcal{G}_M$ by simply restricting it to $P \cup T$.

For the class of directed, vertex and edge labeled graphs it is easy to define characteristic graphs. One can simply define that the characteristic graph of a marking M is the graph $\mathcal{G}_M = \langle V, E, L \rangle$ such that

1. the vertex set is the set of nodes of the net: $V = P \cup T$,
2. the edges are the arcs of the net N: $E = F$, and
3. each place $p \in P$ is labeled with the pair of numbers defined by the markings $\hat{M}$ and M: $L(p) = \langle \hat{M}(p), M(p) \rangle$
4. each transition $t \in T$ is labeled with the text string "T", $L(t) =$ "T", so that it is distinguished from the vertices representing the places, and
5. each edge $f \in F$ is labeled with the arc multiplicity $L(f) = W(f)$.

Note that this construction is similar to the ones in the proofs of Theorems 3.7 and 3.11. It is quite straightforward to see that the requirements for a characteristic graph assigner are fulfilled by the above definition.

For the class of undirected, vertex labeled graphs, some extra vertices and edges are inserted to compensate the lack of edge labels and direction. One can define that the characteristic graph of a marking M is the graph $\mathcal{G}_M = \langle V, E, L \rangle$ such that

1. the vertex set is $V = P \cup T \cup F$,

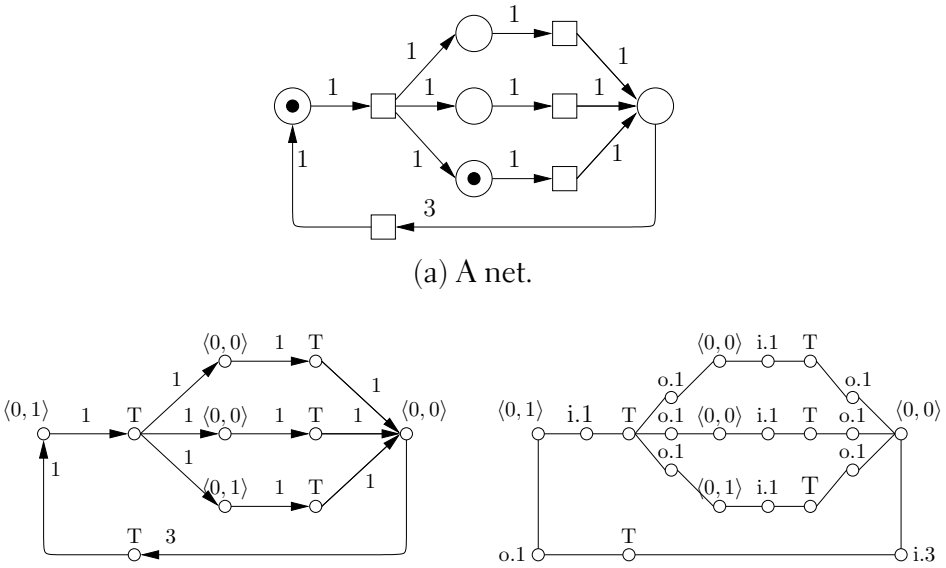
2. for each arc $\langle x, y\rangle \in F$, the edge set E contains the edges $\langle x, \langle x, y\rangle\rangle$ and $\langle\langle x, y\rangle, y\rangle$, and these are the only edges in E,
3. for each place $p \in P$, $L(p) = \langle \hat{M}(p), M(p)\rangle$,
4. for each transition $t \in T$, $L(t)$ is the text string "T", and
5. for each arc $\langle p, t\rangle \in F \cap (P \times T)$, $L(\langle p, t\rangle)$ is the concatenation of the text string "i" (for input arc) and the number $W(\langle p, t\rangle)$ and for each arc $\langle t, p\rangle \in F \cap (T \times P)$, $L(\langle t, p\rangle)$ is the concatenation of the text string "o" (for output arc) and the number $W(\langle t, p\rangle)$.

Now an isomorphism γ from a graph $G_1 = \langle V_1, E_1, L_1\rangle$ to $G_2 = \langle V_2, E_2, L_2\rangle$ is a bijection from V_1 to V_2 such that

1. $\langle v, v'\rangle \in E_1 \Leftrightarrow \langle \gamma(v), \gamma(v')\rangle \in E_2$, and
2. $L_1(v) = L_2(\gamma(v))$ for each vertex $v \in V_1$.

It is again straightforward to see that the requirements for a characteristic graph assigner are fulfilled by the above definition.

Figure 4.6 shows a marked net and its characteristic graphs for both of the graph classes mentioned above (the marking $\hat{M}$ is assumed to be the empty marking).



(a) A net.

(b) Directed, vertex and edge labeled characteristic graph.

(c) Undirected, vertex labeled characteristic graph.

Figure 4.6: A marked net and its characteristic graphs

The characteristic graph assigners defined above can be improved in the case the group $G = \mathrm{Stab}(N, \hat{M})$ is given. Assume that the set of nodes $P \cup T$ of the net is ordered. The orbits of the nodes under G, $[x]_G = \{g(x) \mid g \in G\}$ for each $x \in P \cup T$, inherit the same ordering by e.g. considering the first element in each orbit. Let $\mathit{orbitnum}(x) = i$ if the node $x \in P \cup T$ belongs to the ith orbit. Now the labels of the vertices in the characteristic graph corresponding to the places and transitions can be replaced by (i) $L(p) = \langle \mathit{orbitnum}(p), M(p)\rangle$ for each place p, and (ii) $L(t) = \mathit{orbitnum}(t)$."T" for each transition t. Note that this construction requires that the group G is the stabilizer group of a marking, it does *not* work for arbitrary subgroups of $\mathrm{Aut}(N)$.

Graph canonizers. For a fixed class of graphs, a function $\mathcal{K}$ from graphs to graphs is a *graph canonizer* if for all graphs G, G' it holds that

- $\mathcal{K}(G)$ is isomorphic to G, and
- $\mathcal{K}(G) = \mathcal{K}(G')$ if and only if G and G' are isomorphic.

The graph $\mathcal{K}(G)$ is the *canonical version* of G. It can be assumed that the vertex set of the canonical version of a graph with n vertices is $\{1, 2, \ldots, n\}$ and that a bijective *canonization mapping*, i.e., an isomorphism from G to $\mathcal{K}(G)$, is provided, too.

A graph canonizer can be used for obtaining canonical representative markings, as shown next. First, it is assumed that a Schreier-Sims representation for the group $G = \mathrm{Stab}(N, \hat{M})$ is given. For a marking $M \in \mathbb{M}$, consider the following procedure.

1. Build the characteristic graph $\mathcal{G}_M$.
2. Compute the canonical version $\mathcal{K}(\mathcal{G}_M)$ of $\mathcal{G}_M$ and a canonization mapping γ from $\mathcal{G}_M$ to $\mathcal{K}(\mathcal{G}_M)$.
3. Define the place valuation *pval* by $\forall p \in P : \mathit{pval}(p) = \gamma(p)$, i.e., the place p is associated with the number of the vertex into which the vertex p in the characteristic graph is mapped by γ. Clearly, *pval* is injective.
4. Take the unique element $\hat{g} \in G$ that is compatible with *pval* (under a fixed function multiset selector *select*).
5. Return $\hat{g}^{-1}(M)$ as the representative marking.

Denote the marking $\hat{g}^{-1}(M)$ above by $\mathcal{K}_{\mathbb{M}}(M)$. The fact that $\mathcal{K}_{\mathbb{M}}(M)$ is unique for M despite the indefinite article at item 2 in the process described above (that is, any canonization mapping can be selected) is proven in the following theorem.

Theorem 4.5 *The mapping $\mathcal{K}_{\mathbb{M}}$ is a canonical representative function.*

Proof. Clearly $\mathcal{K}_{\mathbb{M}}(M)$ is equivalent to M under G because $\mathcal{K}_{\mathbb{M}}(M)$ is obtained by applying an element of G to M.

Assume two markings, M_1 and M_2, that are equivalent under G. By definition, their characteristic graphs $\mathcal{G}_{M_1}$ and $\mathcal{G}_{M_2}$, respectively, are isomorphic. Assume that $\mathcal{K}(\mathcal{G}_{M_1})$ (which equals to $\mathcal{K}(\mathcal{G}_{M_2})$) is the canonical version of $\mathcal{G}_{M_1}$ and $\mathcal{G}_{M_2}$. Take any canonization mapping (i.e., isomorphism) γ_1 from $\mathcal{G}_{M_1}$ to $\mathcal{K}(\mathcal{G}_{M_1})$ and γ_2 from $\mathcal{G}_{M_2}$ to $\mathcal{K}(\mathcal{G}_{M_1})$. Now $\gamma_2^{-1} \circ \gamma_1$ is an isomorphism from $\mathcal{G}_{M_1}$ to $\mathcal{G}_{M_2}$ and $\gamma_1^{-1} \circ \gamma_2$ is an isomorphism from $\mathcal{G}_{M_2}$ to $\mathcal{G}_{M_1}$. By the definition of characteristic graphs, $\gamma_2^{-1} \circ \gamma_1$ restricted to $P \cup T$ belongs to G and maps M_1 to M_2 and $\gamma_1^{-1} \circ \gamma_2$ restricted to $P \cup T$ belongs to G and maps M_2 to M_1.

Define the place valuations pval_1 and pval_2 by $\forall p \in P : \mathit{pval}_1(p) = \gamma_1(p)$ and $\forall p \in P : \mathit{pval}_2(p) = \gamma_2(p)$. Now $\big((\gamma_2^{-1} \circ \gamma_1)(\mathit{pval}_1)\big)\,(p) = \mathit{pval}_1((\gamma_2^{-1} \circ \gamma_1)^{-1}(p)) = \mathit{pval}_1(\gamma_1^{-1}(\gamma_2(p))) = \gamma_1(\gamma_1^{-1}(\gamma_2(p))) = \gamma_2(p) = \mathit{pval}_2(p)$, i.e., $\gamma_2^{-1} \circ \gamma_1$ restricted to $P \cup T$ maps pval_1 to pval_2.

Observe that pval_1 and pval_2 are clearly injective functions. Assume that $\hat{g}_1$ is the unique element in G that is compatible with pval_1. By Theorem 4.3, $\hat{g}_1$ is compatible with pval_1 if and only if $(\gamma_2^{-1} \circ \gamma_1) \circ \hat{g}_1$ is compatible with pval_2. Thus $(\gamma_2^{-1} \circ \gamma_1) \circ \hat{g}_1$ is the unique element in G that is compatible with

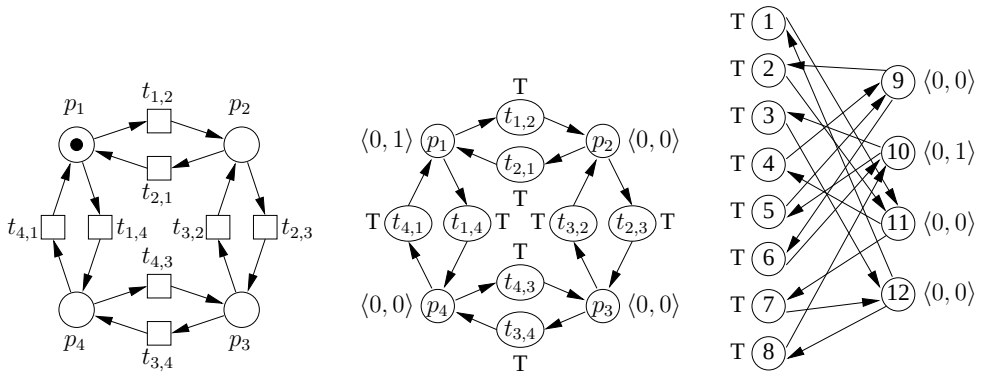


Figure 4.7: A marked net, its characteristic graph, and the canonical version of the characteristic graph

$pval_2$. Now $((\gamma_2^{-1} \circ \gamma_1) \circ \hat{g}_1)^{-1}(M_2) = \hat{g}_1^{-1}((\gamma_1^{-1} \circ \gamma_2)(M_2)) = \hat{g}_1^{-1}(M_1)$ and thus $\mathcal{K}_{\mathbb{M}}(M_1) = \mathcal{K}_{\mathbb{M}}(M_2)$.

The fact that $\mathcal{K}_{\mathbb{M}}(M)$ is uniquely determined follows by considering the case $M_1 = M_2$. □

Example 4.6 Consider the marked version of the net N in Figure 4.1, shown in the left hand side of Figure 4.7. The characteristic graph $\mathcal{G}_M$ of the marking (when $\hat{M}$ is the empty marking) is shown in the middle of Figure 4.7.

Suppose a graph canonizer that produces the canonical version $\mathcal{K}(\mathcal{G}_M)$ of $\mathcal{G}_M$ shown in the right hand side of Figure 4.7. There are two isomorphisms, i.e., canonization mappings, from the characteristic graph $\mathcal{G}_M$ to its canonical version $\mathcal{K}(\mathcal{G}_M)$, namely

$$\begin{array}{rcl} \gamma_1 & = & \left(\begin{smallmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ 10 & 12 & 11 & 9 & 3 & 8 & 1 & 7 & 4 & 2 & 6 & 5 \end{smallmatrix}\right) \text{ and} \\ \gamma_2 & = & \left(\begin{smallmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ 10 & 9 & 11 & 12 & 5 & 6 & 2 & 4 & 7 & 1 & 8 & 3 \end{smallmatrix}\right). \end{array}$$

The corresponding place valuations are

$$\begin{array}{rcl} pval_1 & = & \{p_1 \mapsto 10, p_2 \mapsto 12, p_3 \mapsto 11, p_4 \mapsto 9\} \text{ and} \\ pval_2 & = & \{p_1 \mapsto 10, p_2 \mapsto 9, p_3 \mapsto 11, p_4 \mapsto 12\}, \end{array}$$

respectively. Assuming the Schreier-Sims representation of $\mathrm{Aut}(N)$ used in Example 4.1 and that the minimal element multiset selector is applied,

$$\hat{g}_1 = h_{1,4} \circ h_{2,1} = \left(\begin{smallmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \cdots \\ p_4 & p_1 & p_2 & p_3 & t_{4,1} & \cdots \end{smallmatrix}\right)$$

is the only permutation compatible with $pval_1$ and

$$\hat{g}_2 = h_{1,2} \circ h_{2,2} = \left(\begin{smallmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \cdots \\ p_2 & p_1 & p_4 & p_3 & t_{2,1} & \cdots \end{smallmatrix}\right)$$

is the only permutation compatible with $pval_2$. The canonical representative marking for M is thus

$$\hat{g}_1^{-1}(M) = \hat{g}_2^{-1}(M) = 1'p_2.$$

Finally, note that

– $\mathrm{Stab}(\mathrm{Aut}(N), M) = \{\mathbf{I}, h_{2,2}\}$,
– $\mathrm{Aut}(\mathcal{K}(\mathcal{G}_M)) = \{\mathbf{I}, \left(\begin{smallmatrix}1&2&3&4&5&6&7&8&9&10&11&12\\2&1&5&7&3&8&4&6&12&10&11&9\end{smallmatrix}\right)\}$,
– $\mathrm{Aut}(\mathcal{G}_M) = \gamma_1^{-1} \circ \mathrm{Aut}(\mathcal{K}(\mathcal{G}_M)) \circ \gamma_1 = \gamma_2^{-1} \circ \mathrm{Aut}(\mathcal{K}(\mathcal{G}_M)) \circ \gamma_2 = \left\{\mathbf{I}, \left(\begin{smallmatrix}p_1&p_2&p_3&p_4&t_{1,2}&t_{2,1}&t_{2,3}&t_{3,2}&t_{3,4}&t_{4,3}&t_{4,1}&t_{1,4}\\p_1&p_4&p_3&p_2&t_{1,4}&t_{4,1}&t_{4,3}&t_{3,4}&t_{3,2}&t_{2,3}&t_{2,1}&t_{1,2}\end{smallmatrix}\right)\right\}$, and
– $\mathrm{Stab}(\mathrm{Aut}(N), M)$ equals to $\mathrm{Aut}(\mathcal{G}_M)$ restricted to $P \cup T$ (that is, to $\mathrm{Aut}(\mathcal{G}_M)$ for the class of characteristic graphs used here).

♣

4.3 BACKTRACK SEARCH IN THE SCHREIER-SIMS REPRESENTATION

This section presents representative marking algorithms that are based on selecting a permutation that is compatible with the marking in question. That is, the marking itself is interpreted as a place valuation. A canonical representative marking function is obtained by performing a backtracking search in the Schreier-Sims representation for the lexicographically smallest marking produced by a compatible permutation. Pruning techniques for the search are also discussed.

First, assume a base $\beta = [p_{\beta,1}, \ldots, p_{\beta,|P|}, t_{\beta,1}, \ldots, t_{\beta,|T|}]$ where the places are enumerated before the transitions and a Schreier-Sims representation $\vec{G} = [U_1, \ldots, U_{|P|+|T|}]$ of any subgroup G of $\mathrm{Aut}(N)$ under this base. Similarly, a fixed multiset selector is implicitly assumed throughout this and the following section. Let

$$\mathit{posreps}(M) = \left\{\hat{g}^{-1}(M) \mid \hat{g} \in G \text{ and } \hat{g} \text{ is compatible with } M\right\}$$

denote the set of *possible representative markings* for M. That is, the inverses of the symmetries compatible with the marking are applied to the marking. For equivalent markings, the sets of possible representative markings are the same:

Theorem 4.7 *For each marking $M \in \mathbb{M}$ and for each symmetry $g \in G$, $\mathit{posreps}(M) = \mathit{posreps}(g(M))$.*

Proof. By Theorem 4.3, $\hat{g}$ is compatible with M if and only if $g \circ \hat{g}$ is compatible with $g(M)$. In addition, $(g \circ \hat{g})^{-1}(g(M)) = \hat{g}^{-1}(g^{-1}(g(M))) = \hat{g}^{-1}(M)$. □

Obviously, $M' \in \mathit{posreps}(M)$ implies $M' \equiv_G M$. However, it does *not*, in general, hold that $M \in \mathit{posreps}(M)$. Note that the number of symmetries in G compatible with M is a multiple of $|\mathrm{Stab}(G, M)|$: if $\hat{g}$ is compatible with M, then by Theorem 4.3 the permutation $g \circ \hat{g} \in G$ is compatible with the marking $g(M) = M$ for each stabilizer $g \in \mathrm{Stab}(G, M)$. That is, if $\hat{g}$ is compatible with M, then (and only then) all the permutations in the right coset $\mathrm{Stab}(G, M) \circ \hat{g}$ are compatible with M.

The "hardness" of a marking can be classified as follows. Define that a marking M is

1. *trivial* if there is exactly one permutation in G compatible with the marking M,

2. *easy* if it is not trivial but the set *posreps*(M) contains only one marking,
3. *hard* if it is neither trivial nor easy.

Note that this classification depends on the applied Schreier-Sims representation and multiset selector. It is easy to see that the classification is closed under G, i.e. a marking is trivial/easy/hard if and only if all the markings equivalent to it under G are trivial/easy/hard, respectively. Note that for both trivial and easy markings, the set *posreps*(M) contains only one marking. The difference is that easy markings have several permutations in G that are compatible with the marking.

A very simple (non-canonical) representative marking algorithm would be to simply take an arbitrary permutation $\hat{g} \in G$ that is compatible with the marking M in question and then return $\hat{g}^{-1}(M)$ as the representative marking. Theorem 4.7 guarantees that it is possible, although not guaranteed, that the same representative marking is selected for two equivalent markings. However, for trivial and easy markings, as classified above, the unique canonical representative marking is returned.

A canonical representative marking algorithm can be obtained by first defining a total order between all the markings and then selecting the smallest (or greatest) marking in the set of possible representative markings to be the representative marking. A natural total ordering between the markings is the lexicographical ordering $<_\beta$ induced by the applied base β, defined on page 33. Now define *canrepr*(M) to be the $<_\beta$-smallest marking in the set *posreps*(M). As *posreps*(M) = *posreps*$(g(M))$, *canrepr*(M) = *canrepr*$(g(M))$ for each marking M and for each $g \in G$. Furthermore, *canrepr*$(M) \equiv_G M$. The canonical representative marking *canrepr*(M) for a marking M can be obtained by the depth-first backtracking search shown in Algorithm 4.2, derived from Algorithm 4.1.

Algorithm 4.2 An algorithm finding the smallest marking in *posreps*(M)

function *canrepr*(M)
Require: A global marking $BestMarking$
Set $BestMarking = p \mapsto \infty$ for all $p \in P$
Set $pval(p) = M(p)$ for each place p
Call $backtrack(1, \mathbf{I})$
return $BestMarking$
function $backtrack(l, \hat{g})$
Require: l is the backtracking level
Require: $\hat{g}$ is the currently enumerated compatible permutation
if $l = |P| + 1$ **then**
  if $\hat{g}^{-1}(M) \leq_\beta BestMarking$ **then**
    Set $BestMarking = \hat{g}^{-1}(M)$
  return
Evaluate $S = select(\Sigma_{h \in U_l} 1' pval(\hat{g}(h(p_{\beta,l}))))$
for all $h \in U_l$ such that $pval(\hat{g}(h(p_{\beta,l}))) \in S$ **do**
  Call $backtrack(l + 1, \hat{g} \circ h)$
return

Example 4.8 Recall the net in Figure 4.1 and the Schreier-Sims representation of its automorphism group G described in Example 4.1. In Example 4.4, it was shown that the symmetries

$$\begin{aligned}
h_{1,2} \circ h_{2,1} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \dots \\ p_2 & p_3 & p_4 & p_1 & t_{2,3} & \dots \end{pmatrix}, \\
h_{1,3} \circ h_{2,1} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \dots \\ p_3 & p_4 & p_1 & p_2 & t_{3,4} & \dots \end{pmatrix}, \\
h_{1,3} \circ h_{2,2} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \dots \\ p_3 & p_2 & p_1 & p_4 & t_{3,2} & \dots \end{pmatrix}, \text{ and} \\
h_{1,4} \circ h_{2,2} &= \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \dots \\ p_4 & p_3 & p_2 & p_1 & t_{4,3} & \dots \end{pmatrix}
\end{aligned}$$

are compatible with the marking $M = 1'p_1$ (under the minimal element multiset selector). Thus $\mathit{posreps}(M) = \{1'p_3, 1'p_4\}$ and M is hard. Under the applied base, $1'p_4$ is the lexicographically smallest marking in the set $\mathit{posreps}(M)$. ♣

Pruning with the already fixed prefix. Consider a permutation $g = g_1 \circ \cdots \circ g_i$ in G, where $1 \leq i \leq |P|$ and $g_j \in U_j$ for each $1 \leq j \leq i$. Now each "extended" permutation $\tilde{g} = g_1 \circ \cdots \circ g_i \circ g_{i+1} \circ g_{|P|+|T|}$ in G maps $p_{\beta,1}$ to $g(p_{\beta,1})$, $p_{\beta,2}$ to $g(p_{\beta,2})$, and so on up to and including $p_{\beta,i}$ that is mapped to $g(p_{\beta,i})$. Thus the values of the first i places in $\tilde{g}^{-1}(M)$ are known: $(\tilde{g}^{-1}(M))\,(p_{\beta,1}) = M(\tilde{g}(p_{\beta,1})) = M(g(p_{\beta,1}))$, …, and $(\tilde{g}^{-1}(M))\,(p_{\beta,i}) = M(\tilde{g}(p_{\beta,i})) = M(g(p_{\beta,i}))$. If a marking $M' \in \mathit{posreps}(M)$ such that (i) $M'(p_{\beta,j}) = M(g(p_{\beta,j}))$ for each $1 \leq j < k$ and (ii) $M'(p_{\beta,k}) < M(g(p_{\beta,k}))$ for a $1 \leq k \leq i$ has already been found during the search, one knows that $M' <_\beta \tilde{g}^{-1}(M)$ for all extensions $\tilde{g}$ of g and can therefore skip all such $\tilde{g}$.

To improve the possibilities of this pruning technique to work efficiently, the Schreier-Sims representation can be optimized to have the fixed elements as early as possible in the base. Let $p_{\beta,i}$ be the last element in the base where a place $p_{\beta,j}$, $j \geq i$, may be permuted i.e. $h_{i,l}(p_{\beta,j}) \neq p_{\beta,j}$ for an $h_{i,l} \in U_i$. Now the base can be changed so that $p_{\beta,j}$ is after $p_{\beta,i}$ but before any $p_{\beta,k}$ for which $U_k \supset \{\mathbf{I}\}$.

Finding and pruning with stabilizers. Take any "prefix" permutation $\tilde{g} = g_1 \circ \cdots \circ g_{i-1} \in U_1 \circ \cdots \circ U_{i-1}$ for an $1 \leq i \leq |P|$. Consider two left cosets, $(\tilde{g} \circ g_i) \circ G_{i+1}$ and $(\tilde{g} \circ g_i') \circ G_{i+1}$, where $g_i, g_i' \in U_i$. Let π be a stabilizer of a marking M that (i) fixes each place $\tilde{g}(p_{\beta,1}), \dots, \tilde{g}(p_{\beta,i-1})$, and (ii) maps the place $(\tilde{g} \circ g_i)(p_{\beta,i})$ to $(\tilde{g} \circ g_i')(p_{\beta,i})$. Now, if a permutation g' belongs to the left coset $(\tilde{g} \circ g_i') \circ G_{i+1}$, then $\pi^{-1} \circ g'$ must belong to the left coset $(\tilde{g} \circ g_i) \circ G_{i+1}$ since (i) $(\pi^{-1} \circ g')(p_{\beta,j}) = \pi^{-1}(\tilde{g}(p_{\beta,j})) = \tilde{g}(p_{\beta,j})$ for each $1 \leq j < i$ and (ii) $(\pi^{-1} \circ g')(p_{\beta,i}) = \pi^{-1}((\tilde{g} \circ g_i')(p_{\beta,i})) = (\tilde{g} \circ g_i)(p_{\beta,i})$. Furthermore, for each marking M, $(\pi^{-1} \circ g')^{-1}(M) = (g'^{-1} \circ \pi)(M) = g'^{-1}(M)$. Therefore, the left cosets $(\tilde{g} \circ g_i') \circ G_{i+1}$ and $(\tilde{g} \circ g_i) \circ G_{i+1}$ produce the same markings. In addition, if g is compatible with M, then $\pi^{-1} \circ g$ is compatible with $\pi^{-1}(M) = M$ and therefore the sets of possible representative markings in the left cosets are the same. To sum up, if all the permutations in a left coset $(\tilde{g} \circ g_i) \circ G_{i+1}$ have already been searched and there is a stabilizer π with the above mentioned properties, one can ignore the left coset $(\tilde{g} \circ g_i') \circ G_{i+1}$ as it produces the same possible representative markings.

Stabilizers of markings can be found during the backtrack search on the Schreier-Sims representation. Assume that M' is a marking that has been found earlier during the search by traversing a path $g = g_1 \circ \cdots \circ g_{i-1} \circ g_i \circ g_{i+1} \circ \cdots g_{|P|}$ meaning that $g^{-1}(M) = M'$. For instance, M' could be the lexicographically smallest marking found so far. Assume that the currently traversed path is $g' = g_1 \circ \cdots \circ g_{i-1} \circ g'_i \circ g'_{i+1} \circ \cdots g'_{|P|}$, where $g'_i \neq g_i$. If it holds that $g'^{-1}(M) = M' = g^{-1}(M)$, then $g' \circ g^{-1}$ is a stabilizer of M and (i) $g' \circ g^{-1}$ fixes each $(g_1 \circ \cdots \circ g_j)(p_{\beta,j})$, $1 \leq j < i$, as $(g' \circ g^{-1})((g_1 \circ \cdots \circ g_j)(p_{\beta,j})) = (g' \circ g^{-1})(g(p_{\beta,j})) = g'(p_{\beta,j}) = (g_1 \circ \cdots \circ g_j)(p_{\beta,j})$, and (ii) $g' \circ g^{-1}$ maps $(g_1 \circ \cdots \circ g_{i-1} \circ g_i)(p_{\beta,i}) = g(p_{\beta,i})$ to $g'(p_{\beta,i}) = (g_1 \circ \cdots \circ g_{i-1} \circ g'_i)(p_{\beta,i})$. Thus $g' \circ g^{-1}$ is a stabilizer of M fulfilling the properties discussed above (the prefix $\tilde{g}$ being $g_1 \circ \cdots \circ g_{i-1}$), and the search can be "back-jumped" to the level $i - 1$. This is the most trivial (and easiest to implement) way to prune with the found stabilizers. There are many ways to achieve even larger degree of pruning by composing the found stabilizers, see [Kreher and Stinson 1999; McKay 1981; Butler 1991] and also the discussion in Section 7.4.2.

Transition pruning with stabilizers. Stabilizers of markings can also be used to prune the set of successor markings that have to be visited during the reduced reachability graph generation, see e.g. [Jensen 1995] and the discussion on page 31. As generators of $\mathrm{Stab}(G, M)$ can be found during the search through the Schreier-Sims representation as discussed above, the orbits of the transitions under $\mathrm{Stab}(G, M)$ can be computed during the search, too.

4.4 PARTITION GUIDED SCHREIER-SIMS SEARCH

It is possible to combine the backtracking search in the Schreier-Sims representation described in Section 4.3 with a standard preprocessing technique applied in graph isomorphism algorithms. Assuming a fixed subgroup G of $\mathrm{Aut}(N)$ and given a marking M, an ordered partition of $P \cup T$ is first computed in a way that respects the symmetries in G. The procedure computing the partition for M is based on the use of invariants and is a variant of the standard techniques used in graph isomorphism checking and canonical labeling of graphs, see e.g. [McKay 1981; Kreher and Stinson 1999]. The place valuation corresponding to the partition is then used to prune the search in the Schreier-Sims representation of G. That is, instead of searching through the permutations that are compatible with the marking in question as was done in Section 4.3, the permutations compatible with the constructed place valuation are searched. The hope is that the place valuation is closer to being injective than the original marking, i.e., that it can distinguish more places from each other.

4.4.1 Partition Generators

Assume a net N, a subgroup G of $\mathrm{Aut}(N)$, and a Schreier-Sims representation $\vec{G} = [U_1, \ldots, U_{|P|+|T|}]$ of G under a base $\beta = [p_1, \ldots, p_{|P|}, t_1, \ldots, t_{|T|}]$ in which the places are enumerated before the transitions. Recall the ba-

sic definitions of ordered partitions in Section 2.3 and denote the set of all ordered partitions of $P \cup T$ by $\mathfrak{P}$.

Next, the marking M in question is assigned an ordered partition of $P \cup T$ in a way that respects the symmetries in G. The idea is to try to distinguish between the elements in $P \cup T$ so that distinguishable elements are put in different cells. Formally, define the following.

Definition 4.9 *A function $pg : \mathbb{M} \to \mathfrak{P}$ assigning each marking an ordered partition is a* G-partition generator *if for all markings $M \in \mathbb{M}$ and for all $g \in G$ it holds that $pg(g(M)) = g(pg(M))$.*

That is, for permuted markings, similarly permuted ordered partitions are assigned. A technique for obtaining G-partition generators will be described in Section 4.4.2. Now assume a fixed G-partition generator pg.

An ordered partition can be interpreted as a place valuation by simply assigning each place the cell number in which it appears in the ordered partition. Formally, the place valuation $pval_{\mathfrak{p}}$ corresponding to an ordered partition $\mathfrak{p}$ of $P \cup T$ is defined by

$$pval_{\mathfrak{p}}(p) = incell(\mathfrak{p}, p)$$

for each place $p \in P$. The next lemma shows that the place valuations assigned to equivalent markings in this way are equivalent, too.

Lemma 4.10 *For all $g \in G$ and all markings M, $pval_{pg(g(M))} = g(pval_{pg(M)})$.*

Proof. For each place $p \in P$ it holds that

$$pval_{pg(g(M))}(p) = incell(pg(g(M)), p) = incell(g(pg(M)), p) = incell(pg(M), g^{-1}(p)) = \left(pval_{pg(M)}\right)(g^{-1}(p)) = \left(g(pval_{pg(M)})\right)(p).$$

□

A direct consequence of this is that each stabilizer $g \in \mathrm{Stab}(G, M)$ is a stabilizer of $pval_{pg(M)}$:

Corollary 4.11 *For each $g \in \mathrm{Stab}(G, M)$, $g(pval_{pg(M)}) = pval_{pg(M)}$.*

Thus $\mathrm{Stab}(G, M)$ is a subgroup of $\mathrm{Stab}(G, pval_{pg(M)})$. For all "reasonable" G-partition generators, the stabilizer groups are actually the same.[1]

Lemma 4.12 *If $incell(pg(M), p_1) = incell(pg(M), p_2) \Rightarrow M(p_1) = M(p_2)$ holds for all places $p_1, p_2 \in P$, then $\mathrm{Stab}(G, pval_{pg(M)}) = \mathrm{Stab}(G, M)$.*

Proof. In Corollary 4.11, it is shown that $\mathrm{Stab}(G, M) \subseteq \mathrm{Stab}(G, pval_{pg(M)})$.

Take any permutation $g \in \mathrm{Stab}(G, pval_{pg(M)})$, any place $p \in P$ and assume that $incell(pg(M), p_1) = incell(pg(M), p_2)$ implies $M(p_1) = M(p_2)$ for all places $p_1, p_2 \in P$. It is now shown that

$$(g(M))(p) = M(p)$$

[1] Such "reasonable" cases are obtained by simply applying the marking invariant described in the following subsection during the partition generation process.

meaning that $g \in \mathrm{Stab}(G, M)$ and that $\mathrm{Stab}(G, \mathit{pval}_{\mathit{pg}(M)}) \subseteq \mathrm{Stab}(G, M)$. Since g is a stabilizer of $\mathit{pval}_{\mathit{pg}(M)}$ in G, $g(\mathit{pval}_{\mathit{pg}(M)}) = \mathit{pval}_{\mathit{pg}(M)}$ holds and implies that

$$(g(\mathit{pval}_{\mathit{pg}(M)}))(p) = \mathit{pval}_{\mathit{pg}(M)}(p). \tag{4.2}$$

By the action of g on $\mathit{pval}_{\mathit{pg}(M)}$, $(g(\mathit{pval}_{\mathit{pg}(M)}))(p) = \mathit{pval}_{\mathit{pg}(M)}(g^{-1}(p))$, which combined with (4.2) gives $\mathit{pval}_{\mathit{pg}(M)}(p) = \mathit{pval}_{\mathit{pg}(M)}(g^{-1}(p))$. Applying the definition of $\mathit{pval}_{\mathit{pg}(M)}$ gives $\mathit{incell}(\mathit{pg}(M), p) = \mathit{incell}(\mathit{pg}(M), g^{-1}(p))$. The initial assumption now implies that $M(p) = M(g^{-1}(p))$, which in turn implies that $(g(M))(p) = M(g^{-1}(p)) = M(p)$, concluding the proof. □

After building the ordered partition $\mathit{pg}(M)$ for the marking M and the corresponding place valuation $\mathit{pval}_{\mathit{pg}(M)}$, let

$$\mathit{posreps}(M) = \left\{ \hat{g}^{-1}(M) \mid \hat{g} \in G \text{ and } \hat{g} \text{ is compatible with } \mathit{pval}_{\mathit{pg}(M)} \right\}$$

denote the set of *possible representative markings* for M (recall Section 4.3). Like earlier in Theorem 4.7, it can be proven that for equivalent markings, the sets of possible representative markings coincide.

Theorem 4.13 *For each marking $M \in \mathbb{M}$ and for each symmetry $g \in G$, $\mathit{posreps}(M) = \mathit{posreps}(g(M))$.*

Proof. By Theorem 4.3 and Lemma 4.10, $\hat{g}$ is compatible with $\mathit{pval}_{\mathit{pg}(M)}$ if and only if $g \circ \hat{g}$ is compatible with $g(\mathit{pval}_{\mathit{pg}(M)}) = \mathit{pval}_{\mathit{pg}(g(M))}$. In addition, $(g \circ \hat{g})^{-1}(g(M)) = \hat{g}^{-1}(g^{-1}(g(M))) = \hat{g}^{-1}(M)$. □

Again, $M' \in \mathit{posreps}(M)$ implies $M' \equiv_G M$ and it is *not*, in general, the case that $M \in \mathit{posreps}(M)$. Furthermore, by Theorem. 4.3, a permutation $\hat{g}$ is compatible with $\mathit{pval}_{\mathit{pg}(M)}$ if and only if the permutation $g \circ \hat{g}$ is compatible with $g(\mathit{pval}_{\mathit{pg}(M)}) = \mathit{pval}_{\mathit{pg}(M)}$ for any stabilizer $g \in G$ of $\mathit{pval}_{\mathit{pg}(M)}$. Hence, the number of permutations compatible with $\mathit{pval}_{\mathit{pg}(M)}$ is a multiple of $\left|\mathrm{Stab}(G, \mathit{pval}_{\mathit{pg}(M)})\right|$ (that in all reasonable cases equals to $|\mathrm{Stab}(G, M)|$ by Lemma 4.12).

Now the lexicographically smallest state in $\mathit{posreps}(M)$ can be searched by using the backtrack search shown in Algorithm 4.2 described in Section 4.3 with the obvious changes (i.e., changing the line 2 to refer to the valuation $\mathit{pval}_{\mathit{pg}(M)}$ instead of M). Obviously, the pruning technique based on the fixed prefix is sound, and Corollary 4.11 ensures that the stabilizer pruning technique is also sound.

Similarly to that in Section 4.3, a hardness measure can be defined for markings. Define that a marking M is

1. *trivial* if the partition $\mathit{pg}(M)$ is discrete,
2. *easy* if it is not trivial but the set $\mathit{posreps}(M)$ contains only one marking,
3. *hard* if it is neither trivial nor easy.

Again, this classification depends on the applied (i) Schreier-Sims representation, (ii) G-partition generator, and (iii) multiset selector. Furthermore, the classification is closed under G. Assuming that a function multiset selector is applied, then for each trivial marking M there is a unique permutation compatible with the partition $pg(M)$ and thus the set $posreps(M)$ contains only one marking. On the other hand, easy markings may have several permutations in G that are compatible with the partition. The definition of triviality defined here is stronger than that in Section 4.3 in the sense that there may be markings M for which there is only one permutation compatible with $pg(M)$ although $pg(M)$ is not discrete. The definition here is chosen because it reveals the efficiency of the applied G-partition generator better (more trivial markings, the better). However, a fundamental limitation of G-partition generators is that they cannot distinguish between the elements that are in the same $\mathrm{Stab}(G, M)$-orbit:

Fact 4.14 *If $g \in \mathrm{Stab}(G, M)$ for a marking M, then $pg(g(M)) = g(pg(M))$ implies $pg(M) = g(pg(M))$ and thus each element $x \in P \cup T$ must be in the same cell in the partition $pg(M)$ as the element $g(x)$.*

Thus a trivial marking M has the trivial stabilizer group, i.e., $\mathrm{Stab}(G, M) = \{\mathbf{I}\}$.

Example 4.15 Consider the net in Figure 4.1 and the Schreier-Sims representation $\vec{G}$ of its automorphism group G described in Example 4.1.

Assume a marking $M = 1'p_1$ and a G-partition generator pg mapping M to

$$pg(M) = [\{p_3\}, \{p_2, p_4\}, \{p_1\}, \{t_{1,2}, t_{1,4}\}, \{t_{2,1}, t_{4,1}\}, \{t_{3,2}, t_{3,4}\}, \{t_{2,3}, t_{4,3}\}].$$

By Fact 4.14, this is one of the finest partitions that any G-partition generator can produce since $h_{1,1} \circ h_{2,2} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_1 & p_4 & p_3 & p_2 & t_{1,4} & t_{4,1} & t_{4,3} & t_{3,4} & t_{3,2} & t_{2,3} & t_{2,1} & t_{1,2} \end{pmatrix}$ is a stabilizer of M in G. The corresponding place valuation is

$$pval_{pg(M)} = \{p_1 \mapsto 3, p_2 \mapsto 2, p_3 \mapsto 1, p_4 \mapsto 2\}$$

and the symmetries in G compatible with $pval_{pg(M)}$ are $h_{1,3} \circ h_{2,1}$ and $h_{1,3} \circ h_{2,2}$. Now $(h_{1,3} \circ h_{2,1})^{-1}(M) = 1'p_3$ and $(h_{1,3} \circ h_{2,2})^{-1}(M) = 1'p_3$. Thus $posreps(M) = \{1'p_3\}$. According to the above hardness measure for markings, M is easy.

For the marking $M' = 1'p_2$ (which is equivalent to M as $g = h_{1,2} \circ h_{2,1} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \cdots \\ p_2 & p_3 & p_4 & p_1 & t_{2,3} & \cdots \end{pmatrix}$ maps M to M'), the G-partition generator pg must map M' to $pg(M') = pg(g(M)) = g(pg(M))$, i.e.,

$$pg(M') = [\{p_4\}, \{p_1, p_3\}, \{p_2\}, \{t_{2,3}, t_{2,1}\}, \{t_{3,2}, t_{1,2}\}, \{t_{4,3}, t_{4,1}\}, \{t_{3,4}, t_{1,4}\}].$$

Again, this is one of the finest partitions one can get by using any G-partition generator since $h_{1,3} \circ h_{2,2} = \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & \cdots \\ p_3 & p_2 & p_1 & p_4 & t_{3,2} & \cdots \end{pmatrix}$ is a stabilizer of M' in G. The corresponding place valuation is $pval_{pg(M')} = \{p_1 \mapsto 2, p_2 \mapsto 3, p_3 \mapsto 2, p_4 \mapsto 1\}$ and the symmetries compatible with $pval_{pg(M')}$ are $h_{1,4} \circ h_{2,1}$ and $h_{1,4} \circ h_{2,2}$. Now $(h_{1,4} \circ h_{2,1})^{-1}(M') = 1'p_3$ and $(h_{1,4} \circ h_{2,2})^{-1}(M') = 1'p_3$. Thus $posreps(M') = posreps(M)$ as required by Theorem 4.13. ♣

4.4.2 Partition Refiners and Invariants

The G-partition generators discussed above can be obtained by using G-partition refiners defined below.

Definition 4.16 *A* G-partition refiner *is a function* $\mathcal{R} : \mathbb{M} \times \mathfrak{P} \to \mathfrak{P}$ *such that both*

1. $\mathcal{R}(M, \mathfrak{p}) \preceq \mathfrak{p}$, *and*
2. $\mathcal{R}(g(M), g(\mathfrak{p})) = g(\mathcal{R}(M, \mathfrak{p}))$

hold for all $g \in G$, *for all markings* $M \in \mathbb{M}$, *and for all partitions* $\mathfrak{p} \in \mathfrak{P}$.

That is, the refined partition must be a cell order preserving refinement of the argument partition and for permuted arguments, the result has to be similarly permuted. A direct consequence of the definition is that if a permutation $g \in G$ fixes both a marking M and a partition $\mathfrak{p}$ (i.e., $G(M) = M$ and $g(\mathfrak{p}) = \mathfrak{p}$), then it fixes the refined partition $\mathcal{R}(M, \mathfrak{p})$, too. Two G-partition refiners can be composed:

Lemma 4.17 *The composition* $\mathcal{R}_2 \star \mathcal{R}_1$ *of two* G*-partition refiners* $\mathcal{R}_1$ *and* $\mathcal{R}_2$, *defined by* $(\mathcal{R}_2 \star \mathcal{R}_1)(M, \mathfrak{p}) = \mathcal{R}_2(M, \mathcal{R}_1(M, \mathfrak{p}))$, *is a* G*-partition refiner.*

Proof. Because $(\mathcal{R}_2 \star \mathcal{R}_1)(M, \mathfrak{p}) = \mathcal{R}_2(M, \mathcal{R}_1(M, \mathfrak{p})) \preceq \mathcal{R}_1(M, \mathfrak{p}) \preceq \mathfrak{p}$, $(\mathcal{R}_2 \star \mathcal{R}_1)(M, \mathfrak{p})$ is a cell order preserving refinement of $\mathfrak{p}$. On the other hand, $g((\mathcal{R}_2 \star \mathcal{R}_1)(M, \mathfrak{p})) = g(\mathcal{R}_2(M, \mathcal{R}_1(M, \mathfrak{p}))) = \mathcal{R}_2(g(M), g(\mathcal{R}_1(M, \mathfrak{p}))) = \mathcal{R}_2(g(M), \mathcal{R}_1(g(M), g(\mathfrak{p}))) = (\mathcal{R}_2 \star \mathcal{R}_1)(g(M), g(\mathfrak{p}))$ for each $g \in G$. □

This implies that a finite sequence $\mathcal{R}_n \star \mathcal{R}_{n-1} \star \cdots \star \mathcal{R}_1$ of G-partition refiners, defined by $\mathcal{R}_n(M, \mathcal{R}_{n-1}(M, \cdots (M, \mathcal{R}_1(M, \mathfrak{p})) \ldots))$, is also a G-partition refiner. When a G-partition refiner is applied to the unit partition, the result is a G-partition generator.

Lemma 4.18 *For each* G*-partition refiner* $\mathcal{R}$, *the function* $pg_{\mathcal{R}} : \mathbb{M} \to \mathfrak{P}$ *defined by* $pg_{\mathcal{R}}(M) = \mathcal{R}(M, [P \cup T])$ *is a* G*-partition generator.*

Proof. For each $g \in G$, $pg_{\mathcal{R}}(g(M)) = \mathcal{R}(g(M), [P \cup T]) = \mathcal{R}(g(M), g([P \cup T])) = g(\mathcal{R}(M, [P \cup T]) = g(pg_{\mathcal{R}}(M))$. □

A way to obtain G-partition refiners is based on the use of G-invariants.

Definition 4.19 *A function* $I : \mathbb{M} \times \mathfrak{P} \times \{P \cup T\} \to \mathbb{Z}$ *is a* G-invariant *if*

$$I(M, \mathfrak{p}, x) = I(g(M), g(\mathfrak{p}), g(x)).$$

holds for all $g \in G$, *for all markings* $M \in \mathbb{M}$, *for all ordered partitions* $\mathfrak{p} \in \mathfrak{P}$ *of* $P \cup T$, *and for all nodes* $x \in P \cup T$.

Clearly any G-invariant is also a G'-invariant for any subgroup G' of G. The following are G-invariants for any subgroup G of $\mathrm{Aut}(N)$.

– The *node type invariant* $I_{\text{node type}}$ is defined by

$$I_{\text{node type}}(M, \mathfrak{p}, x) = \begin{cases} 0 & \text{if } x \in P \\ 1 & \text{if } x \in T. \end{cases}$$

– Assume a fixed total order between the places and transitions. Now the orbits of G inherit this order and the *G-orbit invariant* $I_{G\text{-orbit}}$ is defined by $I_{G\text{-orbit}}(M, \mathfrak{p}, x) = \mathit{orbitnum}(x)$, where $\mathit{orbitnum}(x)$ is defined as on page 50.
– The *marking invariant* I_{marking} is defined by

$$I_{\text{marking}}(M, \mathfrak{p}, x) = \begin{cases} M(x) & \text{if } x \in P \\ -1 & \text{if } x \in T. \end{cases}$$

– The *preset* of an element $x \in P \cup T$ is the set ${}^\bullet x = \{x' \mid \langle x', x\rangle \in F\}$ and the *postset* $x^\bullet$ is the set $\{x' \mid \langle x, x'\rangle \in F\}$. The *partition independent weighted in- and out-degree invariants* are defined by

$$I_{\text{in-degree of weight } w}(M, \mathfrak{p}, x) = |\{x' \in {}^\bullet x \mid W(\langle x', x\rangle) = w\}|$$

and

$$I_{\text{out-degree of weight } w}(M, \mathfrak{p}, x) = |\{x' \in x^\bullet \mid W(\langle x, x'\rangle) = w\}|.$$

– The *partition dependent weighted in- and out-degree invariants* are defined by

$$I_{\text{in-degree of weight } w \text{ from cell } c}(M, \mathfrak{p}, x) = \\ |\{x' \in {}^\bullet x \mid W(\langle x', x\rangle) = w \wedge \mathit{incell}(\mathfrak{p}, x') = c\}|$$

and

$$I_{\text{out-degree of weight } w \text{ to cell } c}(M, \mathfrak{p}, x) = \\ |\{x' \in x^\bullet \mid W(\langle x, x'\rangle) = w \wedge \mathit{incell}(\mathfrak{p}, x') = c\}|.$$

Note that the partition independent weighted in- and out-degree invariants and the node type invariant are subsumed by the G-orbit invariant in the sense that if the values of two nodes are equal under the G-orbit invariant, they are equal under these invariants, too. That is, they cannot distinguish elements that the G-orbit invariant cannot.

A partition can be *refined according to an invariant* by splitting the cells according to the values assigned to nodes by the invariant in the partition. Formally, an invariant defines the corresponding partition refiner as follows. For a G-invariant I, define the function $\mathcal{R}_I : \mathbb{M} \times \mathfrak{P} \to \mathfrak{P}$ by $\mathcal{R}_I(M, \mathfrak{p}) = \mathfrak{p}_r$ such that for all $x, x' \in \{P \cup T\}$, for all $\mathfrak{p} \in \mathfrak{P}$, and for all $M \in \mathbb{M}$,

1. $\mathit{incell}(\mathfrak{p}_r, x) = \mathit{incell}(\mathfrak{p}_r, x')$ if and only if $\mathit{incell}(\mathfrak{p}, x) = \mathit{incell}(\mathfrak{p}, x')$ and $I(M, \mathfrak{p}, x) = I(M, \mathfrak{p}, x')$, and
2. $\mathit{incell}(\mathfrak{p}_r, x) < \mathit{incell}(\mathfrak{p}_r, x')$ if and only if either
   (a) $\mathit{incell}(\mathfrak{p}, x) < \mathit{incell}(\mathfrak{p}, x')$, or
   (b) $\mathit{incell}(\mathfrak{p}, x) = \mathit{incell}(\mathfrak{p}, x')$ and $I(M, \mathfrak{p}, x) < I(M, \mathfrak{p}, x')$.

Lemma 4.20 *The function $\mathcal{R}_I$ is a G-partition refiner.*

Proof. The fact that $\mathcal{R}_I(M,\mathfrak{p}) \preceq \mathfrak{p}$ is straightforward to see. Take any $g \in G$, any marking M, and any partition $\mathfrak{p}$. Assume that $\mathcal{R}_I(M,\mathfrak{p}) = \mathfrak{p}_{r,1}$ and $\mathcal{R}_I(g(M),g(\mathfrak{p})) = \mathfrak{p}_{r,2}$. It remains to be shown that $g(\mathfrak{p}_{r,1}) = \mathfrak{p}_{r,2}$. For all $x, x' \in P \cup T$,

$$\begin{aligned}
& incell(g(\mathfrak{p}_{r,1}),x) = incell(g(\mathfrak{p}_{r,1}),x') \\
\Leftrightarrow\ & incell(\mathfrak{p}_{r,1},g^{-1}(x)) = incell(\mathfrak{p}_{r,1},g^{-1}(x')) \\
\Leftrightarrow\ & incell(\mathfrak{p},g^{-1}(x)) = incell(\mathfrak{p},g^{-1}(x')) \text{ and} \\
& I(M,\mathfrak{p},g^{-1}(x)) = I(M,\mathfrak{p},g^{-1}(x')) \\
\Leftrightarrow\ & incell(g(\mathfrak{p}),x) = incell(g(\mathfrak{p}),x') \text{ and} \\
& I(g(M),g(\mathfrak{p}),x) = I(g(M),g(\mathfrak{p}),x') \\
\Leftrightarrow\ & incell(\mathfrak{p}_{r,2},x) = incell(\mathfrak{p}_{r,2},x')
\end{aligned}$$

and thus the cells in $g(\mathfrak{p}_{r,1})$ and in $\mathfrak{p}_{r,2}$ are the same. Similarly, for all $x, x' \in P \cup T$,

$$\begin{aligned}
& incell(g(\mathfrak{p}_{r,1}),x) < incell(g(\mathfrak{p}_{r,1}),x') \\
\Leftrightarrow\ & incell(\mathfrak{p}_{r,1},g^{-1}(x)) < incell(\mathfrak{p}_{r,1},g^{-1}(x')) \\
\Leftrightarrow\ & (a)\quad incell(\mathfrak{p},g^{-1}(x)) < incell(\mathfrak{p},g^{-1}(x')) \text{ or} \\
& (b)\quad incell(\mathfrak{p},g^{-1}(x)) = incell(\mathfrak{p},g^{-1}(x')) \text{ and} \\
& \qquad I(M,\mathfrak{p},g^{-1}(x)) < I(M,\mathfrak{p},g^{-1}(x')) \\
\Leftrightarrow\ & (a)\quad incell(g(\mathfrak{p}),x) < incell(g(\mathfrak{p}),x') \text{ or} \\
& (b)\quad incell(g(\mathfrak{p}),x) = incell(g(\mathfrak{p}),x') \text{ and} \\
& \qquad I(g(M),g(\mathfrak{p}),x) < I(g(M),g(\mathfrak{p}),x') \\
\Leftrightarrow\ & incell(\mathfrak{p}_{r,2},x) < incell(\mathfrak{p}_{r,2},x')
\end{aligned}$$

and thus the cells in $g(\mathfrak{p}_{r,1})$ and in $\mathfrak{p}_{r,2}$ are ordered in the same way. Therefore, $g(\mathfrak{p}_{r,1}) = \mathfrak{p}_{r,2}$. □

Partition refiners with respect to some invariants can also be defined procedurally so that in the resulting partition two nodes are in the same cell if and only if their invariant values in that partition are the same. This is especially the case for the partition dependent weighted in- and out-degree invariants, where the procedure corresponds to the method of computing the so-called equitable partition in [McKay 1981; Kreher and Stinson 1999].

To sum up, a G-partition generator can be obtained by

1. defining a sequence $I_1.I_2.\ldots.I_n$ of G-invariants, and
2. refining the unit partition according to the sequence, meaning that the G-partition refiner sequence $\mathcal{R}_{I_n} \star \mathcal{R}_{I_{n-1}} \star \cdots \star \mathcal{R}_{I_1}$ is applied to it (by Lemmas 4.20, 4.17, and 4.18).

Example 4.21 Consider again the net in Figure 4.1 and the Schreier-Sims representation $\vec{G}$ of its automorphism group G described in Example 4.1.

Assume a marking $M = 1'p_1$. Initially, the partition is

$$\mathfrak{p}_{M,0} = [\{p_1,p_2,p_3,p_4,t_{1,2},\ldots\}].$$

Refining this partition according to the G-orbit invariant yields

$$\mathfrak{p}_{M,1} = [\{p_1,p_2,p_3,p_4\},\{t_{1,2},\ldots\}],$$

and refining according to the marking M gives

$$\mathfrak{p}_{M,2} = [\{p_2, p_3, p_4\}, \{p_1\}, \{t_{1,2}, \ldots\}].$$

Evaluating the invariant $I_{\text{in-degree of weight 1 from cell 1}}$ in the partition $\mathfrak{p}_{M,2}$ gives $I_{\text{in-degree of weight 1 from cell 1}}(M, \mathfrak{p}_{M,2}, p_i) = 0$ for each $1 \leq i \leq 4$, and that $I_{\text{in-degree of weight 1 from cell 1}}(M, \mathfrak{p}_{M,2}, t)$ equals to 0 for $t = t_{1,2}$ and $t = t_{1,4}$ and to 1 for other transitions. Refining $\mathfrak{p}_{M,2}$ thus yields

$$\mathfrak{p}_{M,3} = [\{p_2, p_3, p_4\}, \{p_1\}, \{t_{1,2}, t_{1,4}\}, \{t_{2,1}, t_{2,3}, t_{3,2}, t_{4,3}, t_{3,4}, t_{4,1}\}]$$

Refining this ordered partition according to $I_{\text{in-degree of weight 1 from cell 2}}$ changes nothing and thus $\mathfrak{p}_{M,4} = \mathfrak{p}_{M,4}$. Next, $I_{\text{in-degree of weight 1 from cell 3}}(M, \mathfrak{p}_{M,4}, p)$ equals to 0 for $p = p_1$ and $p = p_3$ and to 1 for $p = p_2$ and $p = p_4$, and $I_{\text{in-degree of weight 1 from cell 3}}(M, \mathfrak{p}_{M,4}, t) = 0$ for all transitions. Thus

$$\mathfrak{p}_{M,5} = [\{p_3\}, \{p_2, p_4\}, \{p_1\}, \{t_{1,2}, t_{1,4}\}, \{t_{2,1}, t_{2,3}, t_{3,2}, t_{4,3}, t_{3,4}, t_{4,1}\}]$$

Refining with $I_{\text{in-degree of weight 1 from cell 4}}$ and $I_{\text{in-degree of weight 1 from cell 5}}$ changes nothing. Next, refining according to $I_{\text{out-degree of weight 1 to cell 1}}$ yields

$$\mathfrak{p}_{M,8} = [\{p_3\}, \{p_2, p_4\}, \{p_1\}, \{t_{1,2}, t_{1,4}\}, \{t_{2,1}, t_{3,2}, t_{3,4}, t_{4,1}\}, \{t_{2,3}, t_{4,3}\}]$$

and refining according to $I_{\text{out-degree of weight 1 to cell 2}}$ yields

$$\mathfrak{p}_{M,9} = [\{p_3\}, \{p_2, p_4\}, \{p_1\}, \{t_{1,2}, t_{1,4}\}, \{t_{2,1}, t_{4,1}\}, \{t_{3,2}, t_{3,4}\}, \{t_{2,3}, t_{4,3}\}].$$

This partition cannot be refined further by any invariant since the permutation $\begin{pmatrix} p_1 & p_2 & p_3 & p_4 & t_{1,2} & t_{2,1} & t_{2,3} & t_{3,2} & t_{3,4} & t_{4,3} & t_{4,1} & t_{1,4} \\ p_1 & p_4 & p_3 & p_2 & t_{1,4} & t_{4,1} & t_{4,3} & t_{3,4} & t_{3,2} & t_{2,3} & t_{2,1} & t_{1,2} \end{pmatrix} \in G$ is a stabilizer of M in G also fixing the partition $\mathfrak{p}_{M,9}$. ♣

4.5 EXPERIMENTAL RESULTS

This section presents some experimental results. The results are obtained by using and extending the *LoLA* reachability analyzer, version 1.0 beta [Schmidt 2000c]. The source code for the extended *LoLA*, including all the nets that are used in the experiments, is available via

`http://www.tcs.hut.fi/~tjunttil/`

4.5.1 Net Classes

The following net classes are used in the experiments.

Mutual exclusion in grid-like networks. These nets are based on the nets in [Schmidt 2000b]. A net "grid d n" models a d-dimensional hypercube of agents with n agents in each dimension. Each agent has two states, critical and non-critical, and can move from the non-critical state to the critical one if none of its neighbors is in the critical state. See Figure 4.8 for the net "grid 3 2" (the dotted lines are drawn only to visualize the three dimensions). The automorphism group of an d dimensional grid net is isomorphic to the automorphism group of an d-dimensional (hyper)cube and has the order $2^d d!$.

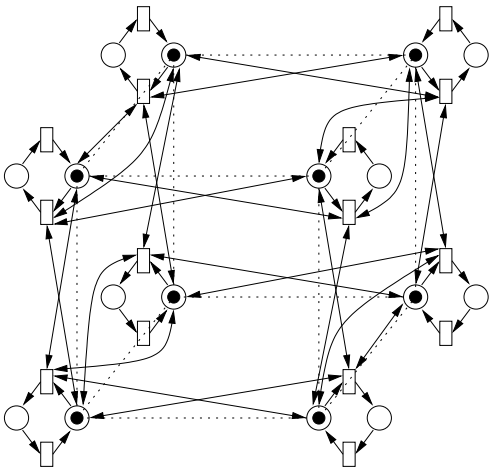


Figure 4.8: A three dimensional grid with two agents per row

Dining philosophers. A version of the classic dining philosophers net. A net "ph n" has n philosophers and the automorphism group of such net is isomorphic to the cyclic group of order n.

Database managers. An unfolding of the colored Petri net presented in [Jensen 1992]. "db n" denotes the net with n managers, having the automorphism group isomorphic to the symmetric group of degree n.

Graph enumeration nets. These nets resemble the one appearing in the proof of Theorem 3.13, inspired by the system in the proof of Theorem 3.4 in [Ip 1996]. Assume a vertex set $V = \{1, \ldots, n\}$ and consider the set of all the directed, unlabeled graphs over V having no self-loops. The following net, call it "digraphs n", enumerates all such graphs in its reachable markings (see Figure 4.9 for an example when $n = 3$). For each vertex $v \in V$, the net has the place p_v. Similarly, for each possible edge $\langle v_1, v_2 \rangle \in V \times V$ such that $v_1 \neq v_2$, the net has the place p_{v_1,v_2}. The purpose is that the places of form p_{v_1,v_2} describe the adjacency matrix of a graph over V and that a place p_{v_1,v_2} contains one token in a marking if and only if the graph corresponding to the marking has an edge $\langle v_1, v_2 \rangle$. For each place p_{v_1,v_2} there is a transition removing one token from it. In addition, each place p_v corresponding to a vertex v is connected to each place of form $p_{v,v'}$ with a gadget shown as a dashed line and explained in Figure 4.9. Similarly, p_v is also connected to each place of form $p_{v',v}$ with a gadget shown as a dotted line and explained in Figure 4.9. These gadgets guarantee that the automorphism group of the net is isomorphic to the permutation group consisting of all permutations of V (i.e., to the symmetric group of degree n). The action of a permutation π of V on the places is such that each p_v is permuted to $p_{\pi(v)}$ and each p_{v_1,v_2} is permuted to $p_{\pi(v_1),\pi(v_2)}$. Thus the action of π corresponds to the usual action of a permutation of the vertex set on the adjacency matrix of a graph. In the initial marking, all the places of form p_{v_1,v_2} corresponding to the possible edges have one token and the others are empty. Thus the set of all reachable markings of the net corresponds to the set of all directed, unlabeled graphs over V having no self-loops. Furthermore, two reachable markings are

equivalent if and only if their corresponding graphs are isomorphic. Thus a minimal reduced reachability graph consisting only of one marking of each orbit has exactly one marking for each class of mutually isomorphic graphs.

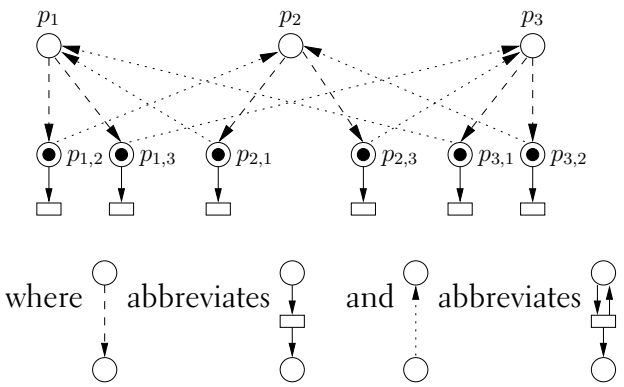


Figure 4.9: A net enumerating all directed graphs without self-loops over three vertices

A similar net, call it "graphs n", enumerating all undirected, unlabeled graphs over n vertices having no self-loops can be constructed by similar principles. See Figure 4.10 for an example when $n = 4$.

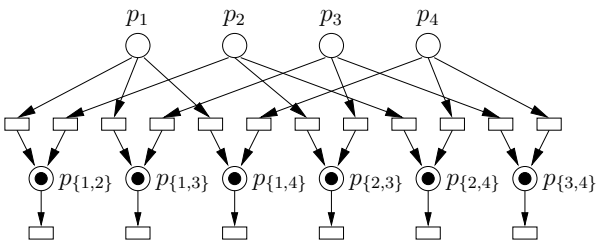


Figure 4.10: A net enumerating all undirected graphs without self-loops over four vertices

Properties of nets. Table 4.1 lists the properties of the nets used in the experiments. The columns $|P|$ and $|T|$ describe the number of places and transitions in the net, respectively, and $|G|$ gives the size of the symmetry group stabilizing the initial marking (the group that is used in the experiments). The number of reachable markings and transition firings as well as the run time of *LoLA* in seconds without the symmetry reduction method are given in the last three columns, respectively. For some nets the number of reachable markings is too large and running *LoLA* would result in running out of memory. In such cases, the run time of *LoLA* is not given but the number of reachable states is given analytically.

4.5.2 Results

The experimental results were obtained in a PC machine with 1GHz AMD Athlon processor and 1 gigabyte of memory, running the Debian Linux operating system. The extended *LoLA* was compiled with the GNU g++ compiler with the -O3 optimization flag switched on. All run-times were obtained

net	$\|P\|$	$\|T\|$	$\|G\|$	reachable markings	reachable edges	*LoLA* time
ph 10	40	30	10	6,726	43,480	1
ph 13	52	39	13	94,642	795,353	4
ph 16	64	48	16	1,331,714	13,774,112	90
db 8	193	128	40,320	17,497	81,664	1
db 9	244	162	362,880	59,050	314,946	3
db 10	301	200	3,628,800	196,831	1,181,000	15
db 20	1201	800	20!	$1+20\times(3^{20-1}) \approx 23 \times 10^9$		
grid 2 5	50	50	8	55,447	688,478	3
grid 3 3	54	54	48	70,633	897,594	4
grid 5 2	64	64	3840	254,475	3,689,792	20
graphs 6	21	45	720	32,678	245,760	1
graphs 7	28	63	5,040	2,097,152	22,020,096	86
graphs 8	36	84	40,320	$2^{\binom{8}{2}} = 2^{28} \approx 268 \times 10^6$		
graphs 9	45	108	362,880	$2^{\binom{9}{2}} = 2^{36} \approx 68 \times 10^9$		
digraphs 3	9	18	6	64	192	1
digraphs 4	16	36	24	4,096	24,576	1
digraphs 5	25	60	120	1,048,576	10,485,760	39
digraphs 6	36	90	720	$2^{6\times(6-1)} = 2^{30} \approx 10^9$		

Table 4.1: Properties of the nets

by the Unix `time` command and are user times rounded up to full seconds unless otherwise stated. The available memory was limited to 900 megabytes and the available time to 24 hours by the Unix `ulimit` command.

The symmetry reduction algorithms in the original *LoLA*, described in [Schmidt 2000b], are numbered as follows: 1 refers to the "iterating the symmetries" algorithm, 2 is the "iterating the states" algorithm, and 3 is the "canonical representative" algorithm[2]. The results of these algorithms are shown in Table 4.2. The current *LoLA* implementation seems to contain some bugs since the algorithms 1 and 2 should both produce minimal reduced reachability graphs but the numbers of the markings in the generated reduced reachability graphs are not the same.

Table 4.3 shows the results of the Schreier-Sims search algorithm described in Section 4.3. The maximal element with minimal frequency multiset selector is used because it seems to usually give the best results. For instance, the minimal element multiset selector gives for some nets bit smaller running times since it can be implemented more efficiently, but in some nets the running times are much worse. Pruning with the fixed prefix, the trivial pruning with the found stabilizers, and the base optimization described on page 55 are applied, too. The pruning of transitions with the found stabilizers was not implemented because the current *LoLA* implementation only stores the symmetry group restricted to the set of places. The "trivial %" and "easy %" columns show the percentage of trivial and easy canonized markings, respectively, as defined in Section 4.3. The "max dead" and "av. dead" columns show the maximum and average number of dead nodes, respec-

[2]Not a canonical representative marking function by the terms used in this work.

net	*LoLA* alg. 1 markings	edges	time	*LoLA* alg. 2 markings	edges	time
ph 10	684	4,421	1	684	4,421	8
ph 13	7,282	61,193	2	7,282	61,193	629
ph 16	83,311	861,696	33	≥83,311	≥861,000	>24h
db 8	37	177	1	37	177	3
db 9	46	250	9	46	250	6
db 10	56	341	118	56	341	11
db 20			>24h	211	2,681	1,477
grid 2 5	7,567	94,143	1	7,471	92,982	183
grid 3 3	2,154	27,620	2	2,103	26,994	62
grid 5 2	296	4,336	7	287	4,237	14
graphs 6	156	1,170	1	152	1,140	1
graphs 7	1,044	10,962	17	1,022	10,731	27
graphs 8	12,346	172,844	2,358	12,095	169,330	3,662
graphs 9	>47,683	>675,000	>24h	>55,400	>792,000	>24h
digraphs 3	16	48	1	16	48	1
digraphs 4	218	1,308	1	215	1,290	1
digraphs 5	9,735	97,357	3	9,567	95,670	1,197
digraphs 6	1,598,555	24,060,959	1,810	>85,469	>908,000	>24h

net	*LoLA* alg. 3 markings	edges	time
ph 10	684	4,421	1
ph 13	7,282	61,193	1
ph 16	83,311	861,696	9
db 8	2,188	10,215	1
db 9	6,562	35,002	1
db 10	19,684	118,109	4
db 20	>399,000	>3,110,000	>418
grid 2 5	14,236	177,007	2
grid 3 3	10,847	136,446	2
grid 5 2	3,020	44,502	1
graphs 6	1,646	11,572	1
graphs 7	37,195	361,478	3
graphs 8	1,536,698	19,805,842	246
graphs 9	>5,128,600	>61,941,000	>801
digraphs 3	16	48	1
digraphs 4	347	2,038	1
digraphs 5	40,078	375,708	3
digraphs 6	>4,581,000	>56,146,000	>512

Table 4.2: Results for the original *LoLA* algorithms

tively, in the search trees for hard markings. As can be seen, practically all markings are usually hard and the number of bad nodes in a search tree can grow quite large. The main reason for this behavior is probably that all the nets are 1-safe, i.e., the number of tokens in a place in each reachable marking is at most one. Thus the multiset selector cannot usually prune the search tree efficiently.

net	markings	edges	time	trivial %	easy %	max dead	av. dead
ph 10	684	4,421	1	7.71	0.50	9	1.38
ph 13	7,282	61,193	1	2.85	0.00	12	2.09
ph 16	83,311	861,696	15	1.04	0.01	15	2.98
db 8	37	177	1	0	6.21	66	24.98
db 9	46	250	1	0	4.80	132	39.49
db 10	56	341	1	0	3.81	259	60.36
db 20	211	2,681	172	0	0.86	40,152	1,844.12
grid 2 5	7,471	92,982	1	0	3.88	7	2.07
grid 3 3	2,103	26,994	1	0	1.59	46	12.14
grid 5 2	288	4,253	1	0	1.03	278	126.94
graphs 6	156	1,170	1	0	0.09	313	109.76
graphs 7	1,044	10,962	3	0	0.01	1,413	272.48
graphs 8	12,346	172,844	82	0	0.00	8,770	580.28
graphs 9	274,668	4,944,024	5,036	0	0.00	70,017	226.80
digraphs 3	16	48	1	29.17	22.92	2	1.17
digraphs 4	218	1,308	1	0	4.05	7	3.15
digraphs 5	9,608	96,080	2	0	0.09	27	7.97
digraphs 6	1,540,944	23,114,160	929	0	0.00	93	17.19

Table 4.3: Results of the plain Schreier-Sims search

Table 4.4 shows the results for the partition guided Schreier-Sims search algorithm described in Section 4.4. The applied partition generator first refines the unit partition according to the orbit and marking invariants and then refines the resulting partition with the partition dependent weighted in- and out-degree invariants until no improvement is achieved. For efficiency reasons, this latter refinement is implemented in a procedural way as discussed on page 62. As can be seen from the results, the amounts of trivial and easy markings are now much higher, compared to the marking guided Schreier-Sims search algorithm discussed above. Furthermore, the hard markings are also easier, and although the number of dead nodes can be still in thousands, on the average it is very low. For nets with small symmetry groups, the overhead of computing the ordered partition sometimes makes the algorithm slower than the marking guided Schreier-Sims search (e.g., the dining philosophers nets and the nets "grid 2 5", "grid 3 3", and "digraphs 6").

Table 4.5 shows the results of the characteristic graph approach described in Section 4.2 when *nauty* (version 2.0 beta 9) [McKay 1990] is used as the graph canonizer. The "trivial %" column shows the percentage of the trivial canonized markings, i.e., markings for which the search tree of *nauty* contains only one node. The "max nodes" and "av. nodes" columns give the maximum and average number of *nauty* search tree nodes, respectively, for

the canonized non-trivial markings. Note that the percentage of the trivial markings encountered is essentially the same as in the partition guided Schreier-Sims search approach discussed above. This not a surprise since the preprocessing technique in *nauty* and the applied partition generator are based on the same ideas (recall Section 4.4). Note that although the search tree sizes of *nauty* are very small in all examples, the running times are high. The bad running times are because of the following:

1. *nauty* does not handle edge labels and is optimized for undirected graphs. P/T-nets are, on the other hand, edge labeled and directed. Thus some extra vertices have to be included in the graphs (recall Section 4.2).
2. While P/T-nets are usually sparse, *nauty* is designed for dense graphs in the sense that the graphs are internally represented as adjacency matrixes. Thus storing a graph with thousands of vertices takes a lot of memory and consequently slows down the refinement routines needed during the search tree traversal in *nauty*.

The results would probably look very different if a graph canonizer designed for (i) sparse, and (ii) vertex and edge labeled directed graphs were used.

As a final note, observe that the proposed Schreier-Sims search algorithms could be approximated (that is, made non-canonical) by performing only a limited search in the Schreier-Sims representation. For instance, an upper limit for the traversed nodes could be set. This would ensure that the time spent in computing a representative for a marking is kept in a reasonable amount, although with the risk that equivalent markings are included in the reduced reachability graph.

net	markings	edges	time	trivial %	easy %	max dead	av. dead
ph 10	684	4,421	1	98.76	1.24	-	-
ph 13	7,282	61,193	4	99.997	0.003	-	-
ph 16	83,311	861,696	66	99.91	0.09	-	-
db 8	37	177	1	0	100	-	-
db 9	46	250	1	0	100	-	-
db 10	56	341	1	0	100	-	-
db 20	211	2,681	27	0	100	-	-
grid 2 5	7,471	92,982	9	90.86	8.71	2	1.19
grid 3 3	2,103	26,994	4	60.82	33.22	16	1.86
grid 5 2	288	4,253	1	2.26	72.00	15	2.30
graphs 6	156	1,170	1	11.11	57.78	40	3.31
graphs 7	1,044	10,962	1	24.70	50.58	196	3.18
graphs 8	12,346	172,844	15	40.52	42.95	535	3.71
graphs 9	274,668	4,944,024	586	57.46	33.92	2,045	3.02
digraphs 3	16	48	1	77.08	22.92	-	-
digraphs 4	218	1,308	1	78.29	21.18	3	2.29
digraphs 5	9,608	96,080	7	89.15	10.22	10	1.52
digraphs 6	1,540,944	23,114,160	2,404	95.68	4.05	34	1.10

Table 4.4: Results of the partition guided Schreier-Sims search

net	markings	edges	time	trivial %	max nodes	av. nodes
ph 10	684	4,421	8	98.76	3	3.00
ph 13	7,282	61,193	201	99.997	3	3.00
ph 16	83,311	861,696	4,866	99.91	3	3.00
db 8	37	177	87	0	36	18.09
db 9	46	250	278	0	45	24.11
db 10	56	341	877	0	55	31.12
db 20			>24h			
grid 2 5	7,471	92,982	2,303	90.86	8	3.13
grid 3 3	2,103	26,994	1,498	60.82	10	3.52
grid 5 2	288	4,253	920	2.26	21	5.71
graphs 6	156	1,170	3	11.11	21	5.47
graphs 7	1,044	10,962	38	24.70	28	5.29
graphs 8	12,346	172,844	1,053	40.52	36	4.83
graphs 9	274,668	4,944,024	49,916	57.46	45	4.26
digraphs 3	16	48	1	77.08	4	3.09
digraphs 4	218	1,308	2	78.29	8	3.23
digraphs 5	9,608	96,080	243	89.15	13	3.23
digraphs 6	>1,028,419	>14,187,000	>24h			

Table 4.5: Results of the characteristic graph approach

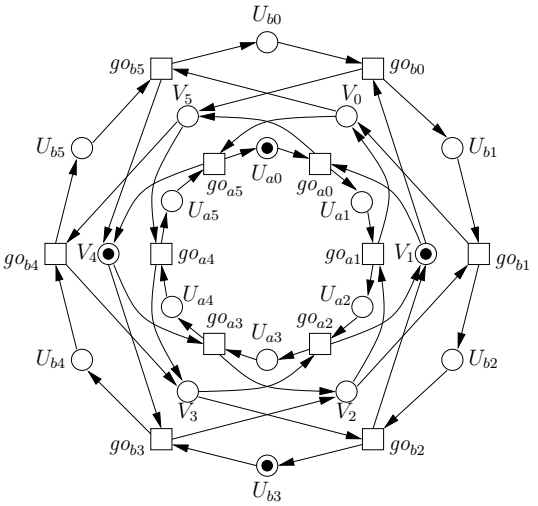
5 DATA SYMMETRIES OF ALGEBRAIC SYSTEM NETS

The place/transition nets discussed in the previous chapters are easy to define and to understand. However, their major drawback in modeling and analyzing complex systems is that the nets tend to grow very large. This is especially the case when use of data is modeled: the tokens in place/transition nets have only one "color", i.e., they do not contain any other information except of being present or absent in a place. Thus data values must be modeled by using extra places. The large size of a net makes it more difficult to understand the net and therefore increases the risk of modeling errors. High-level Petri nets, including net classes such as colored Petri nets [Jensen 1981; 1992], predicate/transition nets [Genrich 1991], well-formed nets [Chiola et al. 1991], many-sorted high-level nets [Billington 1989], and algebraic system nets [Kindler and Reisig 1996; Kindler and Völzer 1998; 2001], have been introduced to solve this problem. In these net classes, tokens are allowed to have many "colors", i.e., to contain data values. This allows more concise model descriptions and easier handling of data, and thus enables the modeler to handle more complex systems.

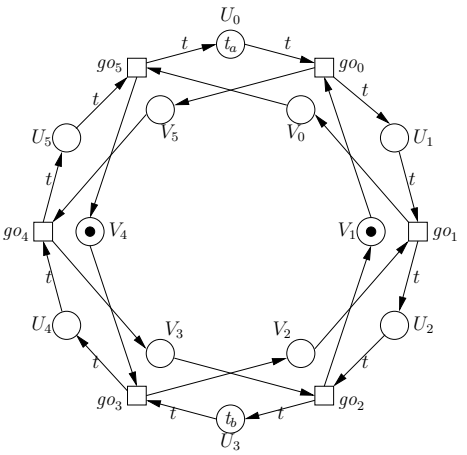
A way to perform state space analysis of high-level Petri nets is to first unfold the high-level net into a corresponding low-level net (for instance, into a place/transition net), and then perform the analysis on that net. This approach enables the use of existing algorithms and tools for low-level nets. For example, if a high-level net is unfolded into a place/transition net, the symmetries can be automatically found and exploited by the algorithms discussed in the previous chapters. However, the main drawback of the unfolding approach is that the unfolded low-level net is often impractically large, or even infinite when infinite data domains are used as token colors. Yet it may be the case that the reachability graph of the net is of manageable size. This can happen because the unfolded low-level net may contain places and transitions that are never actually used during the reachability graph generation (i.e., are dead).

The semantics for high-level nets are usually given explicitly, not by unfoldings into low-level nets. Thus it is also possible to perform the state space analysis directly on the high-level net. This approach does not need the unfolding phase and can thus avoid the problem discussed above. This and the next two chapters study how symmetries can be exploited in this direct analysis approach. First, this and the next chapter discuss how data symmetries are defined and found in a class of high-level Petri nets. Chapter 7 then presents algorithms for the orbit problems in the context of high-level Petri nets and similar formalisms such as the Murφ system description language [Ip and Dill 1996].

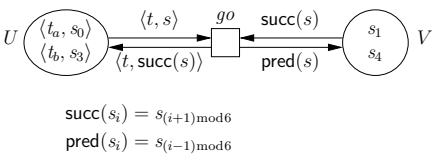
To illustrate the difference between the symmetries of place/transition nets and high-level Petri nets, consider the three nets shown in Figure 5.1. The net in Figure 5.1(a) is the railroad place/transition net already discussed in the previous chapters. The net in Figure 5.1(b) is a high-level net (an algebraic system net) corresponding to the net in Figure 5.1(a) in which the train identities are *folded* together. The idea is that each pair U_{ai} and U_{bi}, where $0 \leq i \leq 5$, of places is folded into one place U_i and the tokens t_a and t_b that



(a) A place/transition net.



(b) A folded version of (a).



(c) Another folded version of (a).

Figure 5.1: A place/transition net and two corresponding high-level Petri nets

can appear in such places carry the identity of the train. Similarly, each pair go_{ai} and go_{bi} of transitions is also folded into one transition go_i. Each transition go_i has one *variable* t which can be *bound* to either t_a or t_b. A transition whose variable is bound is a *transition instance*, and is the entity that may be enabled and fired rather than the transitions themselves. The tokens taken from and added to places during the firing process are determined by the arc annotations. For instance, in the initial marking

$$\{U_0 \mapsto 1't_a, U_3 \mapsto 1't_b, V_1 \mapsto 1'\bullet, V_4 \mapsto 1'\bullet\}$$

depicted in the figure, the transition instance $go_0(t_a)$ (i.e., go_0 when t is bound to t_a) is enabled and firing it leads to the marking

$$\{U_1 \mapsto 1't_a, U_3 \mapsto 1't_b, V_4 \mapsto 1'\bullet, V_5 \mapsto 1'\bullet\}.$$

The net in Figure 5.1(c) is a still further folded version in which the railroad sections are also folded together. The semantics of these high-level nets are formally defined later in this chapter but should be intuitively clear. The reachability graph of the net in Figure 5.1(c) is shown in Figure 5.2 (where each node of form "u, v" denotes the marking $\{U \mapsto u, V \mapsto v\}$).

Note the obvious isomorphism between it and the reachability graph of the place/transition net in Figure 5.1(a) shown in Figure 3.2. In fact, the state spaces of all the three nets in Figure 5.1 are isomorphic. Thus they also have the same (isomorphic) state space symmetries. However, only the rotational structural symmetry of the place/transition net in Figure 5.1(a) is present in the structure of the high-level net in Figure 5.1(b), while the high-level net in Figure 5.1(c) does not have even that symmetry in the structural level. Instead, the missing symmetry information in these high-level nets is hidden in the way the data values are used in the high-level nets. For instance, taking the net in Figure 5.1(c), the rotational state space symmetry is obtained by permuting each data value s_i corresponding to a railroad section to the successor section value $s_{(i+1) \bmod 6}$. Similarly, the state space symmetry produced by swapping the train identities is obtained by permuting the value t_a to t_b and vice versa. The fact that these permutations of data values actually produce state space symmetries is guaranteed by the way the data values are manipulated by the transitions.

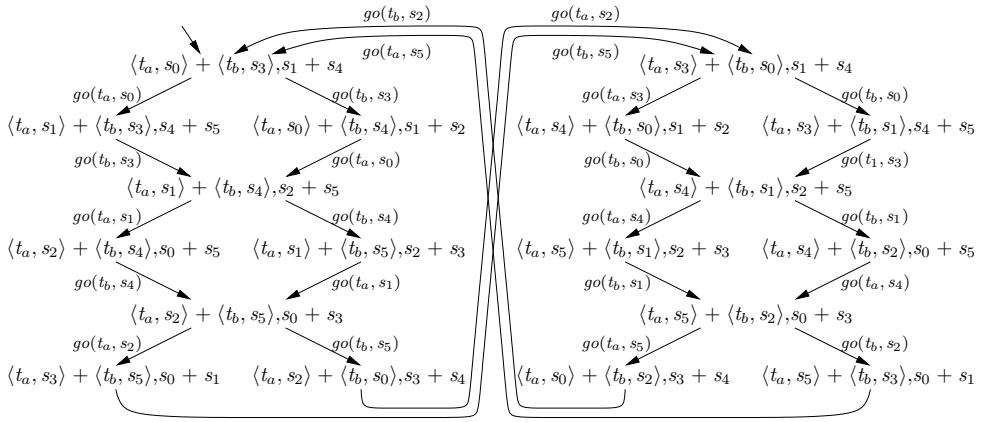


Figure 5.2: The reachability graph of the net in Figure 5.1(c)

This chapter builds a framework for defining such data type based symmetries in a class of high-level nets, namely algebraic system nets (ASNs). The focus is on ASNs because the abstract signatures and algebras employed in ASNs offer a convenient framework for defining both the syntax and semantics of the arc and transition annotations. This is an advantage over colored Petri nets (CP-nets), where the annotations are just general functions. In this sense, CP-nets are more abstract than ASNs, or, ASNs are a more formalistic version of CP-nets. On the other hand, in well-formed nets (WFNs) the syntax and semantics of annotations are fixed in advance. In this sense, ASNs are more flexible than WFNs. Although the focus in this chapter is on algebraic system nets, the results presented here also have relevance to other classes of high-level Petri nets. For instance, Theorem 5.15 in Section 5.3.4 implies that it is not easy (i) to check whether user given data permutations actually produce state space symmetries or (ii) to automatically find data symmetries in arbitrary, reasonably expressive classes of high-level Petri nets (such as CP-nets).

In order to illustrate the use of the developed framework, a class of ASNs called *extended well-formed nets* is defined in the next chapter. In this net

class, the symmetries are induced by defining a special type system. This approach resembles to ones taken in the well-formed nets [Chiola et al. 1991] and in the Murφ description language [Ip and Dill 1996]. The name extended well-formed nets was chosen because the employed type system is richer than that of well-formed nets.

Most of the material in this chapter has been published previously in [Junttila 1998; 1999a; 1999b].

5.1 SIGNATURES AND ALGEBRAS

First, a framework for defining data types and operations on them is given. The framework is based on signature and algebras, commonly used in algebraic specifications [Wirsing 1990]. The definitions in this and next section are based on [Kindler and Reisig 1996; Kindler and Völzer 1998; 2001], except that the introduction of special error values in algebras is by the author.

Signatures and Algebras. First, signatures declare the types and the names of the operations that can be applied on them. Formally, a *signature* $Sig = \langle \mathcal{T}, \mathcal{F} \rangle$ consists of

1. a non-empty set $\mathcal{T}$ of *types*, and
2. a pairwise disjoint family $\mathcal{F} = \{\mathcal{F}_{\sigma,T}\}_{\sigma \in \mathcal{T}^*, T \in \mathcal{T}}$ of *operation names*.

An operation $f \in \mathcal{F}_{T_1.....T_n,T}$ stands for an operation from $T_1, \ldots, T_n$ to T, where $T_1, \ldots, T_n$ are the *argument types* and T is the *range type* of f. The set $\mathcal{F}_{\varepsilon,T}$, where ε is the empty string, is the set of *Sig-constants of type* T. The pairwise disjointness of $\mathcal{F}$ is a mere technicality because the operations can always be renamed. It is only imposed in order to obtain unambiguous interpretation of terms (which are defined later). For instance, one may use the same operation name "add" to denote addition in the contexts of both natural numbers and multiset types as long as it is understood which operation is meant.

Next, algebras concretize signatures by assigning each type a domain and each operation a function. A special error value $\mathfrak{err}$ is included in the definitions in order to allow error handling. Formally, a *Sig-error algebra* (or simply a *Sig-algebra*) $\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$ has the following components:

1. A family $\mathcal{D}^{\mathcal{A}} = \left\{\mathcal{D}_T^{\mathcal{A}}\right\}_{T \in \mathcal{T}}$ of non-empty *domains* for types. It is assumed that no $\mathcal{D}_T^{\mathcal{A}}$ contains the error value $\mathfrak{err}$ while $\tilde{\mathcal{D}}_T^{\mathcal{A}}$ denotes the set $\mathcal{D}_T^{\mathcal{A}} \cup \{\mathfrak{err}\}$.
2. A family $\mathcal{F}^{\mathcal{A}} = \left\{f^{\mathcal{A}}\right\}_{f \in \mathcal{F}}$ of *operations*. For each operation name $f \in \mathcal{F}_{T_1 \ldots T_n,T}$, the operation $f^{\mathcal{A}}$ is a function $f^{\mathcal{A}} : \tilde{\mathcal{D}}_{T_1}^{\mathcal{A}} \times \cdots \times \tilde{\mathcal{D}}_{T_n}^{\mathcal{A}} \to \tilde{\mathcal{D}}_T^{\mathcal{A}}$.

Unless it is otherwise stated, it is implicitly assumed that each operation returns $\mathfrak{err}$ if any of its arguments is $\mathfrak{err}$ (actually, only one exception in Section 6.2 will be considered). An operation $f^{\mathcal{A}}$ is *safe* if $f^{\mathcal{A}}(v_1, \ldots, v_n) = \mathfrak{err}$ implies that at least one of $v_1, \ldots, v_n$ equals to $\mathfrak{err}$.

Variables. A pairwise disjoint family $\mathcal{X} = \{\mathcal{X}_T\}_{T \in \mathcal{T}}$ such that $\mathcal{X} \cap \mathcal{F} = \emptyset$ is called a *family of Sig-variables*. A variable $x \in \mathcal{X}_T$ is said to be *of type* T.

Again, the disjointness of variables and operation names is a technicality only imposed to avoid confusions between constants (which are operations) and variables. An *assignment* α to the variables in $\mathcal{X}$ is a mapping $\alpha : \mathcal{X} \to \mathcal{D}^{\mathcal{A}}$ such that $x \in \mathcal{X}_T$ implies $\alpha(x) \in \mathcal{D}^{\mathcal{A}}_T$. Note that variables cannot be assigned to the error value.

Terms. For a family $\mathcal{X} = \{\mathcal{X}_T\}_{T \in \mathcal{T}}$ of Sig-variables, the set $Terms^{Sig}_T(\mathcal{X})$ of *Sig-terms of type T over $\mathcal{X}$* is the minimal set (in the set inclusion sense) defined inductively by the following rules:

1. $\mathcal{X}_T \subseteq Terms^{Sig}_T(\mathcal{X})$.
2. If $f \in \mathcal{F}_{T_1 \ldots T_n, T}$ and $term_i \in Terms^{Sig}_{T_i}(\mathcal{X})$ for each $1 \leq i \leq n$, then $f(term_1, \ldots, term_n) \in Terms^{Sig}_T(\mathcal{X})$.[1]

The set $Terms^{Sig}_T = Terms^{Sig}_T(\emptyset)$ denotes the set of *Sig-ground terms of type T* and $Terms^{Sig}(\mathcal{X}) = \bigcup_{T \in \mathcal{T}} Terms^{Sig}_T(\mathcal{X})$ is the set of *Sig-terms over $\mathcal{X}$*. Note that terms may also be written in mix-fix notation whenever no confusion can arise. As an example, a term $+(a, b)$ for an operation $+ \in \mathcal{F}_{\mathsf{Int.Int,Int}}$ may be written as $a + b$.

An assignment α to the variables in $\mathcal{X}$ is extended to the corresponding *evaluation of terms over $\mathcal{X}$*, $eval_\alpha$, by the following inductive definition for each $term \in Terms^{Sig}(\mathcal{X})$.

1. If $term = x$ for a variable $x \in \mathcal{X}_T$, then $eval_\alpha(term) = \alpha(x)$.
2. If $term = f(term_1, \ldots, term_n)$ for an operation $f \in \mathcal{F}_{T_1 \ldots T_n, T}$, then $eval_\alpha(term) = f^{\mathcal{A}}(eval_\alpha(term_1), \ldots, eval_\alpha(term_n))$.

Obviously, if a term $term$ is of type T, then $eval_\alpha(term) \in \tilde{\mathcal{D}}^{\mathcal{A}}_T$. Note that for ground terms all evaluations yield the same value because variables are not involved in ground terms. For a ground term $term$ one may thus simply write $eval(term)$ instead of $eval_\alpha(term)$ (for any assignment α). Also notice that the terms composed only of safe operations never evaluate to $\mathfrak{err}$.

Example 5.1 Consider a signature Sig and a Sig-algebra $\mathcal{A}$ including the type Bool for booleans with the domain $\mathcal{D}^{\mathcal{A}}_{\mathsf{Bool}} = \{false, true\}$. Operations for Bool include the ones corresponding to the usual Boolean operations, for instance

1. the constants $\mathsf{false}, \mathsf{true} \in \mathcal{F}_{\varepsilon, \mathsf{Bool}}$ with the interpretations $\mathsf{false}^{\mathcal{A}}() = false$ and $\mathsf{true}^{\mathcal{A}}() = true$,
2. the unary operation $\mathsf{neg} \in \mathcal{F}_{\mathsf{Bool,Bool}}$ with $\mathsf{neg}^{\mathcal{A}}(x) = \neg x$ meaning that $\mathsf{neg}^{\mathcal{A}}(false) = true$ and $\mathsf{neg}^{\mathcal{A}}(true) = false$, and
3. the binary operations $\mathsf{and}, \mathsf{or}, \mathsf{xor}, \mathsf{equiv}, \mathsf{implies} \in \mathcal{F}_{\mathsf{Bool.Bool,Bool}}$ with the obvious interpretations, for instance, $\mathsf{implies}^{\mathcal{A}}(x, y) = x \Rightarrow y$.

Assume a family $\mathcal{X} = \{\mathcal{X}_{\mathsf{Bool}} = \{x, y\}\}$ of Sig-variables. Now the term $\mathsf{and}(x, \mathsf{or}(y, \mathsf{true}))$ over $\mathcal{X}$ can be evaluated under an assignment $\alpha = \{x \mapsto$

[1] For a constant $f \in \mathcal{F}_{\varepsilon, T}$, one may simply write f instead of $f()$.

$true, y \mapsto false\}$, resulting in

$$\begin{aligned} eval_\alpha(\mathsf{and}(x, \mathsf{or}(y, \mathsf{true}))) &= \mathsf{and}^{\mathcal{A}}(eval_\alpha(x), eval_\alpha(\mathsf{or}(y, \mathsf{true}))) \\ &= \mathsf{and}^{\mathcal{A}}(\alpha(x), \mathsf{or}^{\mathcal{A}}(eval_\alpha(y), eval_\alpha(\mathsf{true}))) \\ &= \mathsf{and}^{\mathcal{A}}(true, \mathsf{or}^{\mathcal{A}}(\alpha(y), \mathsf{true}^{\mathcal{A}}())) \\ &= \mathsf{and}^{\mathcal{A}}(true, \mathsf{or}^{\mathcal{A}}(false, true)) \\ &= \mathsf{and}^{\mathcal{A}}(true, true) \\ &= true. \end{aligned}$$

♣

5.2 ALGEBRAIC SYSTEM NETS

First, as Booleans and multisets will have a special role in algebraic system nets, the following is assumed from now on.

Requirement 5.2 *Each signature $Sig = \langle \mathcal{T}, \mathcal{F} \rangle$ and each Sig-algebra $\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$ used in algebraic system nets has to fulfill the following.*

1. *The set $\mathcal{T}$ of types includes the type* Bool *for Booleans having the domain $\mathcal{D}^{\mathcal{A}}_{\mathsf{Bool}} = \{false, true\}$.*
2. *If $T \in \mathcal{T}$ is a type, then* Multiset(T) *is also a type in $\mathcal{T}$ with the domain $\mathcal{D}^{\mathcal{A}}_{\mathsf{Multiset}(T)} = [\mathcal{D}^{\mathcal{A}}_T \to \mathbb{N}]$, i.e., the set of all multisets over the domain of T.*

Basically, an algebraic system net is a Petri net augmented with algebraic annotations.

Definition 5.3 *An* algebraic system net *(ASN) is a tuple*

$$\mathcal{N} = \langle P, T, F, Sig, \mathcal{A}, type, vars, W, guard, m_{init} \rangle,$$

with the following components.

- *P is a finite non-empty set of* places.
- *T is a finite set of* transitions *such that $P \cap T = \emptyset$.*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of* arcs *(the flow relation).*
- *$Sig = \langle \mathcal{T}, \mathcal{F} \rangle$ is a signature.*
- *$\mathcal{A} = \langle \mathcal{D}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}} \rangle$ is a Sig-algebra.*
- *$type : P \to \mathcal{T}$ assigns each place a type. For each place p, the type $type(p)$ is the* type*, the set $\mathcal{D}^{\mathcal{A}}_{type(p)}$ is the* domain*, and the type* Multiset$(type(p))$ *is the* multiset type *of p.*
- *$vars$ associates each transition $t \in T$ with a finite family $vars(t)$ of Sig-variables.*
- *W is an* arc annotation function *assigning each arc $\langle t, p \rangle, \langle p, t \rangle \in F$, where $p \in P$ and $t \in T$, a Sig-term of place's multiset type over transition's variables: $W(t, p), W(p, t) \in Terms^{Sig}_{\mathsf{Multiset}(type(p))}(vars(t))$.*
- *$guard$ assigns each transition $t \in T$ a* guard *that is a Boolean term over transition's variables, i.e., $guard(t) \in Terms^{Sig}_{\mathsf{Bool}}(vars(t))$.*

– *m_{init} is the* symbolic initial marking *assigning each place a Sig-ground term of place's multiset type. That is, $m_{init}(p) \in Terms^{Sig}_{\mathsf{Multiset}(type(p))}$.*

Now assume a fixed ASN $\mathcal{N}$. A *marking* is a function M that assigns each place a multiset over its domain: $M(p) \in \mathcal{D}^{\mathcal{A}}_{\mathsf{Multiset}(type(p))}$ for each $p \in P$. The *set of all markings* is denoted by $\mathbb{M}$. The *initial marking* M_{init} is the evaluation of the symbolic initial marking: $M_{init}(p) = eval(m_{init}(p))$ for each place $p \in P$ (it is assumed that $eval(m_{init}(p)) \neq \mathfrak{err}$).

Let $t \in T$ be a transition. An assignment α to its variables $vars(t)$ is called a *mode* (or a *binding*) for t and the pair $\langle t, \alpha \rangle$, also denoted by t_α, is a *transition instance*. The *set of all transition instances* is denoted by $\mathbb{T}$. For a transition instance t_α, the *input effect function* t^-_α describes the tokens that are removed from the places when the transition instance is fired. It is defined for each place $p \in P$ by

$$t^-_\alpha(p) = \begin{cases} eval_\alpha(W(p,t)) & \text{if } \langle p, t \rangle \in F \\ \emptyset & \text{otherwise.} \end{cases}$$

Similarly, the *output effect function* t^+_α describes the tokens that are produced in the places and is defined for each place $p \in P$ by

$$t^+_\alpha(p) = \begin{cases} eval_\alpha(W(t,p)) & \text{if } \langle t, p \rangle \in F \\ \emptyset & \text{otherwise.} \end{cases}$$

A transition instance t_α is *enabled in a marking* M, denoted by $M\ [t_\alpha\rangle$, if

1. $eval_\alpha(guard(t)) = true$, i.e., the guard evaluates to true,
2. $t^-_\alpha(p) \neq \mathfrak{err}$ and $M(p) \geq t^-_\alpha(p)$ for each place $p \in P$, and
3. $t^+_\alpha(p) \neq \mathfrak{err}$ for each place $p \in P$.

If t_α is enabled in M, it may *fire* and transform M into the new marking M' defined by

$$M'(p) = M(p) - t^-_\alpha(p) + t^+_\alpha(p)$$

for each place $p \in P$. This is denoted by $M\ [t_\alpha\rangle\ M'$. The *state space* of the ASN $\mathcal{N}$ is the LTS

$$\langle \mathbb{M}, \mathbb{T}, [\rangle, M_{init} \rangle,$$

where $[\rangle = \{\langle M_1, t_\alpha, M_2 \rangle \in \mathbb{M} \times \mathbb{T} \times \mathbb{M} \mid M_1\ [t_\alpha\rangle\ M_2\}$.

Example 5.4 The ASN in Figure 5.1(c) can now be formally defined. First, the signature *Sig* and the *Sig*-algebra $\mathcal{A}$ have the following types and operations.

– The type Trains for train identities with the domain $\mathcal{D}^{\mathcal{A}}_{\mathsf{Trains}} = \{t_a, t_b\}$ and the constants $\mathsf{t}_a, \mathsf{t}_b \in \mathcal{F}_{\varepsilon,\mathsf{Trains}}$ with the interpretations ${\mathsf{t}_x}^{\mathcal{A}}() = t_x$ for each $x \in \{a, b\}$.
– The type Secs for the railroad sections with $\mathcal{D}^{\mathcal{A}}_{\mathsf{Secs}} = \{s_0, \ldots, s_5\}$ and the constants $\mathsf{s}_0, \ldots, \mathsf{s}_5 \in \mathcal{F}_{\varepsilon,\mathsf{Secs}}$ with the interpretations ${\mathsf{s}_i}^{\mathcal{A}}() = s_i$ for each $i \in \{0, \ldots, 5\}$. Furthermore, the predecessor and successor operations $\mathsf{pred}, \mathsf{succ} \in \mathcal{F}_{\mathsf{Secs},\mathsf{Secs}}$ are defined by $\mathsf{pred}^{\mathcal{A}}(s_i) = s_{(i-1) \bmod 6}$ and $\mathsf{succ}^{\mathcal{A}}(s_i) = s_{(i+1) \bmod 6}$.

– The structure type $\mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs})$ having the domain

$$\mathcal{D}^{\mathcal{A}}_{\mathsf{Struct}(\mathsf{Trains},\mathsf{Secs})} = \mathcal{D}^{\mathcal{A}}_{\mathsf{Trains}} \times \mathcal{D}^{\mathcal{A}}_{\mathsf{Secs}}$$

and the constructor operation $\mathsf{makeStruct} \in \mathcal{F}_{\mathsf{Trains.Secs},\mathsf{Struct}(\mathsf{Trains},\mathsf{Secs})}$ defined by $\mathsf{makeStruct}^{\mathcal{A}}(t, s) = \langle t, s\rangle$.

– The corresponding multiset types for the types above. For any type T and the corresponding multiset type $\mathsf{Multiset}(T)$ with $\mathcal{D}^{\mathcal{A}}_{\mathsf{Multiset}(T)} = [\mathscr{D}_T \to \mathbb{N}]$, the unit multiset operation $\mathsf{unitMS} \in \mathcal{F}_{T,\mathsf{Multiset}(T)}$ is defined by $\mathsf{unitMS}^{\mathcal{A}}(v) = 1'v$. and the multiset addition operation is denoted by the operation $\mathsf{add} \in \mathcal{F}_{\mathsf{Multiset}(T).\mathsf{Multiset}(T),\mathsf{Multiset}(T)}$.

The net consists of

– two places: $P = \{U, V\}$,
– one transition: $T = \{go\}$, and
– four arcs: $F = \{\langle U, go\rangle, \langle go, U\rangle, \langle V, go\rangle, \langle go, V\rangle\}$.

The types of the places are $type(U) = \mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs})$ and $type(V) = \mathsf{Secs}$, and the transition go has two variables: t of type Trains and s of type Secs. The arcs are annotated as follows:

– $W(\langle U, go\rangle) = \mathsf{unitMS}(\mathsf{makeStruct}(t, s))$,
– $W(\langle go, U\rangle) = \mathsf{unitMS}(\mathsf{makeStruct}(t, \mathsf{succ}(s)))$,
– $W(\langle V, go\rangle) = \mathsf{unitMS}(\mathsf{succ}(s))$, and
– $W(\langle go, V\rangle) = \mathsf{unitMS}(\mathsf{pred}(s))$.

The guard of the transition is the constant true, and the symbolic initial marking is defined by

$$\begin{aligned} m_{init}(U) &= \mathsf{add}(\mathsf{unitMS}(\mathsf{makeStruct}(\mathsf{t}_a, \mathsf{s}_0)), \mathsf{unitMS}(\mathsf{makeStruct}(\mathsf{t}_b, \mathsf{s}_3))) \\ m_{init}(V) &= \mathsf{add}(\mathsf{unitMS}(\mathsf{s}_1), \mathsf{unitMS}(\mathsf{s}_4)). \end{aligned}$$

The annotation terms in Figure 5.1(c) are written in abbreviated informal form, e.g. $\mathsf{unitMS}(\mathsf{makeStruct}(t, \mathsf{succ}(s)))$ is simply written as $\langle t, \mathsf{succ}(s)\rangle$.

The transition instance $\langle go, \{t \mapsto t_a, s \mapsto s_0\}\rangle$, abbreviated by $go(t_a, s_0)$, is enabled in the initial marking $M_{init} = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$ because

1. the guard of go is invariably true,
2. $go^-_{\{t\mapsto t_a, s\mapsto s_0\}}(U) = 1'\langle t_a, s_0\rangle \leq M_{init}(U)$, and
3. $go^-_{\{t\mapsto t_a, s\mapsto s_0\}}(V) = 1's_1 \leq M_{init}(V)$.

Firing it leads to the marking $\{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$. The entire reachability graph of the net is shown in Figure 5.2. ♣

Example 5.5 By augmenting the signature and algebra in the previous example with (i) a type Token having the domain $\mathcal{D}^{\mathcal{A}}_{\mathsf{Token}} = \{\bullet\}$ and (ii) the corresponding multiset type $\mathsf{Multiset}(\mathsf{Token})$, the net in Figure 5.1(b) can be described as an ASN, too. ♣

5.3 DATA SYMMETRIES

It is now defined how the domains of data types can be permuted and how these permutations act on the markings and transition instances of ASNs. A sufficient condition is then presented for the domain permutations to actually produce state space symmetries. The computational complexity of verifying whether a domain permutation fulfills this condition is then studied. Since this problem turns out to be **co-NP**-complete even for very simple cases, an approximation of this condition is also presented. This approximated condition is extensively used in Chapter 6.

5.3.1 Domain Permutations

A *domain permutation* ψ^T for a type $T \in \mathcal{T}$ is a permutation of its domain, i.e., a member of $\mathrm{Sym}(\mathcal{D}_T^{\mathcal{A}})$. A domain permutation ψ^T is implicitly extended to the error domain $\tilde{\mathcal{D}}_T^{\mathcal{A}}$ by $\psi^T(\mathfrak{err}) = \mathfrak{err}$, meaning that error values are never permuted. A *domain permutation* for a set $\mathcal{T}' \subseteq \mathcal{T}$ of types is a family $\psi^{\mathcal{T}'} = \{\psi^T\}_{T \in \mathcal{T}'}$ of domain permutations for the types in $\mathcal{T}'$. A *domain permutation group* for $\mathcal{T}'$ is a non-empty set $\Psi^{\mathcal{T}'}$ of domain permutations for $\mathcal{T}'$ forming a group under the type-wise function composition operator $*$ on domain permutations for $\mathcal{T}'$ defined by

$$\{\psi_1^T\}_{T \in \mathcal{T}'} * \{\psi_2^T\}_{T \in \mathcal{T}'} = \{\psi_3^T\}_{T \in \mathcal{T}'} \Leftrightarrow \forall T \in \mathcal{T}' : \psi_1^T \circ \psi_2^T = \psi_3^T.$$

A domain permutation (group) for the set $\mathcal{T}$ of all types is also called a domain permutation (group) for $\mathcal{A}$ or simply a domain permutation (group). Thus a domain permutation $\psi^{\mathcal{T}}$ may simply be written by ψ and a domain permutation group $\Psi^{\mathcal{T}}$ by Ψ.

Requirement 5.6 *As Booleans and multisets have a special role in ASNs (recall Requirement 5.2), the following is required from each domain permutation $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ used in ASNs.*

1. *Booleans are not permuted: $\psi^{\mathsf{Bool}}(x) = x$ for each $x \in \{false, true\}$.*
2. *The domain permutation for each multiset type $\mathsf{Multiset}(T)$ is defined by the domain permutation for the type T as follows: for each multiset $m \in \mathscr{D}_{\mathsf{Multiset}(T)}$ over $\mathscr{D}_T$, $\psi^{\mathsf{Multiset}(T)}(m)$ is the multiset fulfilling $\left(\psi^{\mathsf{Multiset}(T)}(m)\right)(\psi^T(v)) = m(v)$ for each $v \in \mathscr{D}_T$. That is, an element $v \in \mathscr{D}_T$ has multiplicity n in m if and only if $\psi^T(v)$ has multiplicity n in $\psi^{\mathsf{Multiset}(T)}(m)$.*

This requirement has some direct consequences on the usual multiset operations.

Lemma 5.7 *Let m_1, m_2 be two multisets in the domain of a multiset type $\mathsf{Multiset}(T)$, and let $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ be a domain permutation fulfilling Requirement 5.6. Then the following hold.*

1. $\psi^{\mathsf{Multiset}(T)}(\emptyset) = \emptyset$.
2. *$m_1 \leq m_2$ if and only if $\psi^{\mathsf{Multiset}(T)}(m_1) \leq \psi^{\mathsf{Multiset}(T)}(m_2)$.*
3. $\psi^{\mathsf{Multiset}(T)}(m_1 + m_2) = \psi^{\mathsf{Multiset}(T)}(m_1) + \psi^{\mathsf{Multiset}(T)}(m_2)$.

4. *If* $m_2 \leq m_1$, *then*

$$\psi^{\mathsf{Multiset}(T)}(m_1 - m_2) = \psi^{\mathsf{Multiset}(T)}(m_1) - \psi^{\mathsf{Multiset}(T)}(m_2).$$

5. $\psi^{\mathsf{Multiset}(T)}(n \cdot m_1) = n \cdot \psi^{\mathsf{Multiset}(T)}(m_1)$ *for each natural number* n.

Proof. Item 1. $\left(\psi^{\mathsf{Multiset}(T)}(\emptyset)\right)(\psi^T(v)) = \emptyset(v) = 0$ for each $v \in \mathscr{D}_T$.

Item 2. $m_1 \leq m_2$ if and only if $m_1(v) \leq m_2(v)$ for all $v \in \mathscr{D}_T$ if and only if $\left(\psi^{\mathsf{Multiset}(T)}(m_1)\right)(\psi^T(v)) \leq \left(\psi^{\mathsf{Multiset}(T)}(m_2)\right)(\psi^T(v))$ for all $v \in \mathscr{D}_T$ if and only if $\left(\psi^{\mathsf{Multiset}(T)}(m_1)\right)(v) \leq \left(\psi^{\mathsf{Multiset}(T)}(m_2)\right)(v)$ for all $v \in \mathscr{D}_T$ by the bijectivity of ψ^T if and only if $\psi^{\mathsf{Multiset}(T)}(m_1) \leq \psi^{\mathsf{Multiset}(T)}(m_2)$.

Item 3. For each $v \in \mathscr{D}_T$,

$$\begin{aligned}
\left(\psi^{\mathsf{Multiset}(T)}(m_1 + m_2)\right)(\psi^T(v)) &= \\
(m_1 + m_2)(v) &= \\
m_1(v) + m_2(v) &= \\
\left(\psi^{\mathsf{Multiset}(T)}(m_1)\right)(\psi^T(v)) + \left(\psi^{\mathsf{Multiset}(T)}(m_2)\right)(\psi^T(v)) &= \\
\left(\psi^{\mathsf{Multiset}(T)}(m_1) + \psi^{\mathsf{Multiset}(T)}(m_2)\right)(\psi^T(v)).
\end{aligned}$$

Item 4. By item 1, it holds that $m_2 \leq m_1$ if and only if $\psi^{\mathsf{Multiset}(T)}(m_2) \leq \psi^{\mathsf{Multiset}(T)}(m_1)$. The rest is similar to item 3.

Item 5. For each $v \in \mathscr{D}_T$,

$$\begin{aligned}
\left(\psi^{\mathsf{Multiset}(T)}(n \cdot m_1)\right)(\psi^T(v)) &= \\
(n \cdot m_1)(v) &= \\
n \times m_1(v) &= \\
n \times \left(\psi^{\mathsf{Multiset}(T)}(m_1)\right)(\psi^T(v)) &= \left(n \cdot \psi^{\mathsf{Multiset}(T)}(m_1)\right)(\psi^T(v)).
\end{aligned}$$

□

5.3.2 Actions of Domain Permutations

Domain permutations act on variable assignments by simply permuting the assigned values. That is, a domain permutation $\psi = \left\{\psi^T\right\}_{T \in \mathcal{T}}$ acts on a variable assignment α to $\mathcal{X} = \{\mathcal{X}_T\}_{T \in \mathcal{T}}$ by $\psi(\alpha) : x \mapsto \psi^T(\alpha(x))$ for each variable $x \in \mathcal{X}_T \in \mathcal{X}$.

Similarly, a domain permutation $\psi = \left\{\psi^T\right\}_{T \in \mathcal{T}}$ acts on the markings by permuting the multisets assigned to places, or formally, $\psi(M) : p \mapsto \psi^{\mathsf{Multiset}(type(p))}(M(p))$ for each marking M and each place $p \in P$.

Finally, a domain permutation $\psi = \left\{\psi^T\right\}_{T \in \mathcal{T}}$ acts on the transition instances by permuting the mode assignment and leaving the transition name intact. That is, $\psi(t_\alpha) = t_{\psi(\alpha)}$ for each transition instance t_α.

Example 5.8 Recall the net described in Figure 5.1(c) and in Example 5.4, and consider a domain permutation ψ in which

$$\begin{aligned}
\psi^{\mathsf{Trains}} &= \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right), \\
\psi^{\mathsf{Secs}} &= \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_2 & s_3 & s_4 & s_5 & s_0 & s_1 \end{smallmatrix}\right), \\
\psi^{\mathsf{Struct}(\mathsf{Trains},\mathsf{Secs})} &= \langle t_x, s_i \rangle \mapsto \langle \psi^{\mathsf{Trains}}(t_x), \psi^{\mathsf{Secs}}(s_i) \rangle
\end{aligned}$$

and the domain permutations for the Boolean type Bool and multiset types are defined as required in Requirement 5.6. This domain permutation corresponds to the swapping of the train identities and rotating the railroad sections two steps. Now the initial marking

$$M = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$$

is mapped to

$$\psi(M) = \{U \mapsto \langle t_b, s_2\rangle + \langle t_a, s_5\rangle, V \mapsto s_3 + s_0\}.$$

Furthermore, the transition instance $go_{\{t\mapsto t_a, s\mapsto s_0\}}$, which is enabled in M, is mapped to $\psi(go_{\{t\mapsto t_a, s\mapsto s_0\}}) = go_{\{t\mapsto \psi^{\mathsf{Trains}}(t_a), s\mapsto \psi^{\mathsf{Secs}}(s_0)\}} = go_{\{t\mapsto t_b, s\mapsto s_2\}}$, which is enabled in $\psi(M)$. Firing $go_{\{t\mapsto t_a, s\mapsto s_0\}}$ in M leads to the marking $M' = \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$, while firing $go_{\{t\mapsto t_b, s\mapsto s_2\}}$ in $\psi(M)$ leads to the marking $\{U \mapsto \langle t_b, s_3\rangle + \langle t_a, s_5\rangle, V \mapsto s_0 + s_1\} = \psi(M')$. Thus the state space symmetry equation

$$M\ [go_{\{t\mapsto t_a, s\mapsto s_0\}}\rangle\ M' \Leftrightarrow \psi(M)\ [\psi(go_{\{t\mapsto t_a, s\mapsto s_0\}})\rangle\ \psi(M')$$

holds for this particular domain permutation and for these markings and transition instances.

To see that the state space symmetry equation does not necessarily hold for arbitrary domain permutations, consider a domain permutation ψ_{bad} in which

$$\begin{aligned}
\psi_{\text{bad}}^{\mathsf{Trains}} &= \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right), \\
\psi_{\text{bad}}^{\mathsf{Secs}} &= \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_0 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \\
\psi_{\text{bad}}^{\mathsf{Struct(Trains,Secs)}} &= \langle t_x, s_i\rangle \mapsto \langle \psi^{\mathsf{Trains}}(t_x), \psi^{\mathsf{Secs}}(s_i)\rangle
\end{aligned}$$

corresponding to the swapping of the train identities and swapping of the zeroth and first railroad sections. The transition instance $go_{\{t\mapsto t_a, s\mapsto s_0\}}$ is still enabled in the initial marking $M = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$ but the transition instance $\psi_{\text{bad}}(go_{\{t\mapsto t_a, s\mapsto s_0\}}) = go_{\{t\mapsto t_b, s\mapsto s_1\}}$ is *not* enabled in the marking $\psi_{\text{bad}}(M) = \{U \mapsto \langle t_b, s_1\rangle + \langle t_a, s_3\rangle, V \mapsto s_0 + s_4\}$. This is because the domain permutation ψ_{bad} is not "compatible" (defined formally in the following subsection) with some of the arc annotation terms. ♣

5.3.3 Term Compatibility

As shown in Example 5.8 above, not all domain permutations produce state space symmetries. In the following, a sufficient condition ensuring this is presented.

A term and a domain permutation are said to be compatible if it holds for each variable assignment on the variables appearing in the term that for the permuted variable assignment the evaluation result of the term is similarly permuted. Formally:

Definition 5.9 *A term* $term \in Terms_T^{Sig}(\mathcal{X})$ *and a domain permutation* $\psi = \{\psi^T\}_{T\in\mathcal{T}}$ *are* compatible *if*

$$eval_{\psi(\alpha)}(term) = \psi^T(eval_\alpha(term))$$

holds for each assignment α to the variables in $\mathcal{X}$. The term term is compatible with a domain permutation group Ψ if it is compatible with all the domain permutations in the group.

Note that a term consisting only of a variable is by the definition compatible with all possible domain permutations because $eval_{\psi(\alpha)}(x) = \big(\psi(\alpha)\big)(x) = \psi^T(\alpha(x)) = \psi^T(eval_\alpha(x))$ for any variable x of a type T.

The following definition gives a sufficient condition for a domain permutation to produce state space symmetries, as proven in the theorem below. Similar conditions for colored Petri nets have been presented previously in [Jensen 1995, Definition 3.16] and in [Chiola et al. 1997, Definition 2.8].

Definition 5.10 *A domain permutation $\psi = \{\psi^T\}_{T\in\mathcal{T}}$ is* compatible with the ASN $\mathcal{N}$ *if it is compatible with all the transition guard and arc annotation terms appearing in the net. Similarly, a domain permutation group Ψ is compatible with the net if all the domain permutations in it are.*

Theorem 5.11 *If a domain permutation $\psi = \{\psi^T\}_{T\in\mathcal{T}}$ is compatible with the net, then the state space symmetry equation*

$$M_1\ [t_\alpha\rangle\ M_2 \Leftrightarrow \psi(M_1)\ [\psi(t_\alpha)\rangle\ \psi(M_2)$$

holds.

Proof. Take a marking M and a transition instance t_α. Since the guard $guard(t)$ is a Boolean term compatible with ψ and the Booleans are not permuted,

$$eval_\alpha(guard(t)) = \psi^{\mathsf{Bool}}(eval_\alpha(guard(t))) = eval_{\psi(\alpha)}(guard(t)). \quad (5.1)$$

Because arc annotation terms are compatible with ψ, it can be shown that

$$t^-_{\psi(\alpha)}(p) = \psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p)) \quad (5.2)$$

for each place p. Namely, if $\langle p,t\rangle \in F$, then $t^-_{\psi(\alpha)}(p) = eval_{\psi(\alpha)}(W(p,t)) = \psi^{\mathsf{Multiset}(type(p))}(eval_\alpha(W(p,t))) = \psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p))$. If $\langle p,t\rangle \notin F$, then $t^-_{\psi(\alpha)}(p) = \emptyset = \psi^{\mathsf{Multiset}(type(p))}(\emptyset) = \psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p))$ by Lemma 5.7. Similar arguments show that

$$t^+_{\psi(\alpha)}(p) = \psi^{\mathsf{Multiset}(type(p))}(t^+_\alpha(p)) \quad (5.3)$$

for each place p.

Now the transition instance t_α is enabled in M if and only if $\psi(t_\alpha) = t_{\psi(\alpha)}$ is enabled in $\psi(M)$:

1. By (5.1), the guard $guard(t)$ evaluates to true under α if and only if it does under $\psi(\alpha)$.
2. For each place p, $t^-_\alpha(p) \neq \mathfrak{err}$ if and only if $\psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p)) = t^-_{\psi(\alpha)}(p) \neq \mathfrak{err}$ as $\mathfrak{err}$ is never permuted and by (5.2). Furthermore, by Lemma 5.7, $M(p) \geq t^-_\alpha(p)$ if and only if $\psi^{\mathsf{Multiset}(type(p))}(M(p)) \geq \psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p))$ if and only if $(\psi(M))\,(p) \geq t^-_{\psi(\alpha)}(p)$ by the definition of the action of ψ on markings and by (5.2).

3. Similarly, $t^+_\alpha(p) \neq \mathfrak{err}$ if and only if $\psi^{\mathsf{Multiset}(type(p))}(t^+_\alpha(p)) = t^+_{\psi(\alpha)}(p) \neq \mathfrak{err}$.

In the case the transition instances are enabled, the successor markings are equivalent:

$$\begin{aligned}(\psi(M))\,(p) - t^-_{\psi(\alpha)}(p) + t^+_{\psi(\alpha)}(p) &= \\ \psi^{\mathsf{Multiset}(type(p))}(M(p)) - \psi^{\mathsf{Multiset}(type(p))}(t^-_\alpha(p)) + \psi^{\mathsf{Multiset}(type(p))}(t^+_\alpha(p)) &= \\ \psi^{\mathsf{Multiset}(type(p))}\left(M(p) - t^-_\alpha(p) + t^+_\alpha(p)\right)\end{aligned}$$

by applying (5.2), (5.3), and Lemma 5.7. □

Example 5.12 Recall the domain permutation ψ_{bad} in Example 5.8. The term $\mathsf{unitMS}(\mathsf{succ}(s))$ annotating the arc from the transition *go* to the place V in Figure 5.1(c) is not compatible with ψ_{bad} because

$$eval_{\{t\mapsto t_a, s\mapsto s_0\}}(\mathsf{unitMS}(\mathsf{succ}(s))) = 1's_1$$

but

$$\begin{aligned}eval_{\psi_{\text{bad}}(\{t\mapsto t_a, s\mapsto s_0\})}(\mathsf{unitMS}(\mathsf{succ}(s))) &= \\ eval_{\{t\mapsto t_b, s\mapsto s_1\}}(\mathsf{unitMS}(\mathsf{succ}(s))) &= \\ 1's_2 \;\neq\; \psi^{\mathsf{Multiset(Secs)}}_{\text{bad}}(1's_1) &= 1's_0.\end{aligned}$$

Thus ψ_{bad} is not compatible with the net in Figure 5.1(c). ♣

In addition to ensuring that a domain permutation group produces state space symmetries (Theorem 5.11 above), Definition 5.9 can also be applied when analyzing whether an atomic proposition of a temporal logic formula is invariant with respect to the applied symmetry group (recall the definitions in Section 2.2.3). Assume that the ASN under the study has the set $P = \{p_1, \ldots, p_n\}$ of places. Define the family $\hat{P}$ of corresponding *place variables* that includes the variable $\hat{p}_i$ of the multiset type $\mathsf{Multiset}(type(p_i))$ for each place p_i. Now an atomic proposition can be defined to be a Boolean term over the variables in $\hat{P}$, i.e., a term $term \in Terms^{Sig}_{\mathsf{Bool}}(\hat{P})$. An atomic proposition *term* is defined to hold in a marking M if and only if it evaluates to true "in the marking" meaning that $eval_{\{\hat{p}_1\mapsto M(p_1),\ldots,\hat{p}_n\mapsto M(p_n)\}}(term) = true$. If the atomic proposition is compatible with a domain permutation group, then it is invariant under the corresponding state space permutations, too.

Theorem 5.13 *Assume that an atomic proposition $term \in Terms^{Sig}_{\mathsf{Bool}}(\hat{P})$ is compatible with a domain permutation group Ψ. Then it is invariant under Ψ, i.e., it holds in a marking M if and only if it holds in the marking $\psi(M)$ for each $\psi \in \Psi$.*

Proof. The atomic proposition *term* holds in a marking $\psi(M)$ if and only if

$$\begin{aligned}eval_{\{\hat{p}_1\mapsto(\psi(M))(p_1),\ldots,\hat{p}_n\mapsto(\psi(M))(p_n)\}}(term) &= \\ eval_{\{\hat{p}_1\mapsto\psi^{\mathsf{Multiset}(type(p_1))}(M(p_1)),\ldots,\hat{p}_n\mapsto\psi^{\mathsf{Multiset}(type(p_n))}(M(p_n))\}}(term) &= \\ eval_{\psi(\{\hat{p}_1\mapsto M(p_1),\ldots,\hat{p}_n\mapsto M(p_n)\})}(term) &= \\ \psi^{\mathsf{Bool}}(eval_{\{\hat{p}_1\mapsto M(p_1),\ldots,\hat{p}_n\mapsto M(p_n)\}}(term)) &= \\ eval_{\{\hat{p}_1\mapsto M(p_1),\ldots,\hat{p}_n\mapsto M(p_n)\}}(term) &= true\end{aligned}$$

by applying (i) the definitions of how domain permutations act on markings and variable assignments, (ii) the fact that *term* is compatible with ψ, and (ii) the fact that Booleans are not permuted. Therefore, the atomic proposition *term* holds in a marking M if and only if it holds in the marking $\psi(M)$. □

An application of this theorem and examples of atomic propositions will be described in Section 6.3.

5.3.4 Complexity of Deciding Term Compatibility

In the light of Definition 5.10 and Theorems 5.11 and 5.13 above, deciding whether a term and a domain permutation are compatible is an important task. For instance, given an ASN and a domain permutation specified by the user, it is desirable to be able to check whether the domain permutation is compatible with all the transition and arc annotations in the net. Formally, the term compatibility problem is defined as:

Problem 5.14 TERM COMPATIBILITY. *Given a term and a domain permutation, are they compatible?*

Unfortunately, the term compatibility problem is not easy even for very simple, *fixed* algebras.

Theorem 5.15 *For a fixed algebra,* TERM COMPATIBILITY *is* **co-NP***-complete.*

Proof. Assume a very simple algebra, namely the Boolean algebra with one type, `Bool` with the domain $\{false, true\}$, and a truth-functionally complete set of operations (such as $\{\wedge, \vee, \neg\}$ with the usual interpretations), recall Example 5.1. Take any Boolean formula f over a set $\mathcal{X}$ of Boolean variables. Define $f' = x \vee \neg f$, where x is a new Boolean variable not in $\mathcal{X}$ and let $\mathcal{X}'$ be $\mathcal{X}$ augmented with x. Clearly f' can be expressed as a term over $\mathcal{X}'$ in the Boolean algebra. Consider the only non-identity domain permutation ψ in the algebra: the one flipping the truth values *true* and *false*. Now f' is evaluated to true for all assignments on $\mathcal{X}'$ such that $x \mapsto true$. But, in order f' to be compatible with ψ, this implies that for all assignments permuted by ψ, i.e., for all assignments where $x \mapsto false$, f' must evaluate to *false*. Therefore $\neg f$ must be false for all assignments on $\mathcal{X}$ meaning that f has to evaluate to *true* for all assignments on $\mathcal{X}$. Thus f is valid if and only if f' is compatible with ψ. Since VALIDITY is a **co-NP**-complete problem, TERM COMPATIBILITY is **co-NP**-hard.

The membership of TERM COMPATIBILITY in **co-NP** follows by noticing that one can simply guess a disqualifying assignment α to the variables occurring in the term *term* in non-deterministic polynomial time and then calculate and compare $eval_{\psi(\alpha)}(term)$ and $\psi^T(eval_\alpha(term))$ in deterministic polynomial time. "Yes" is then returned if they were not equal and "No" otherwise. This non-deterministic polynomial time computation clearly accepts the complement of TERM COMPATIBILITY.

Although the Booleans are not allowed to be permuted in algebras used in ASNs, the proof can be made to work by just assuming any other type whose operations can simulate the Boolean operations. □

As the proof above shows, TERM COMPATIBILITY is a hard problem even for fixed algebras with finite domains. This is because of the quantification over the variable assignments in the definition of term compatibility. In the case a term only involves a small number of variables, each having a small domain, it may be feasible in practice to decide term compatibility by simply enumerating all the variable assignments and checking the term evaluation results for each of them.

Obviously, deciding whether a term is compatible with a domain permutation *group* is at least as hard as deciding whether it is compatible with a single domain permutation. Of course, an algorithm for deciding term compatibility with a domain permutation can be used for deciding compatibility with a domain permutation group by simply applying it to each domain permutation in the group. However, by representing domain permutation groups by means of generating sets, this task can be made easier as shown below.

Definition 5.16 *A domain permutation group Ψ is* generated *by a set of domain permutations if all the domain permutations in Ψ, and only those, can be written as a finite composition of domain permutations and their inverses appearing in the set.*

Note that if Ψ is generated by a set S, then S is a subset of Ψ.

Example 5.17 Recall the net described in Figure 5.1(c) and studied in Examples 5.4 and 5.8. Consider the domain permutation group Ψ in which each domain permutation ψ fulfills the following rules:

- $\psi^{\mathsf{Trains}} \in \mathrm{Sym}(\mathcal{D}^{\mathcal{A}}_{\mathsf{Trains}})$,
- $\psi^{\mathsf{Secs}} = (\mathsf{succ}^{\mathcal{A}})^k$ for some k,
- $\psi^{\mathsf{Struct(Trains,Secs)}} : \langle t_x, s_i \rangle \mapsto \langle \psi^{\mathsf{Trains}}(t_x), \psi^{\mathsf{Secs}}(s_i) \rangle$, and
- the domain permutations for the Boolean type Bool and multiset types are defined as required in Requirement 5.6.

That is, a domain permutation in the group may swap the train identities and rotate the railroad sections. The group has $2! \times 6 = 12$ domain permutations. It can be generated by two domain permutations in Ψ, namely

- ψ_1 in which $\psi_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)$ and $\psi_1^{\mathsf{Secs}} = \mathbf{I}$, and
- ψ_2 in which $\psi_2^{\mathsf{Trains}} = \mathbf{I}$ and $\psi_2^{\mathsf{Secs}} = \mathsf{succ}^{\mathcal{A}}$.

The following lemma and corollary state that, in order to check whether a term is compatible with a domain permutation group, it is sufficient to check that the term is compatible with each domain permutation in a generating set of the group.

Lemma 5.18 *Assume that a term $term \in Terms_T^{Sig}(\mathcal{X})$ is compatible with domain permutations $\psi = \{\psi^T\}_{T \in \mathcal{T}}$, $\psi_1 = \{\psi_1^T\}_{T \in \mathcal{T}}$, and $\psi_2 = \{\psi_2^T\}_{T \in \mathcal{T}}$. Then term is also compatible with the inverse domain permutation ψ^{-1} and the composition domain permutation $\psi_1 * \psi_2$.*

Proof. Inverses. Take any assignment α to the variables in $\mathcal{X}$ and assume that $eval_{\psi^{-1}(\alpha)}(term) = v$ for some $v \in \mathscr{D}_T$. Because ψ is compatible with $term$, $eval_\alpha(term) = eval_{(\psi * \psi^{-1})(\alpha)}(term) = eval_{\psi(\psi^{-1}(\alpha))}(term)$ equals to $\psi^T(eval_{\psi^{-1}(\alpha)}(term)) = \psi^T(v)$. Therefore, $eval_{\psi^{-1}(\alpha)}(term) = v = \psi^{T^{-1}}(\psi^T(v)) = \psi^{T^{-1}}(eval_\alpha(term))$.

Composition. For each assignment α it holds that $eval_{(\psi_1 * \psi_2)(\alpha)}(term) = eval_{(\psi_1(\psi_2(\alpha))}(term) = \psi_1^T(eval_{\psi_2(\alpha)}(term)) = \psi_1^T(\psi_2^T(eval_\alpha(term))) = (\psi_1^T \circ \psi_2^T)(eval_\alpha(term))$. □

Corollary 5.19 *Assume that a domain permutation group Ψ is generated by a set S of domain permutations. Then a term $term \in Terms_T^{Sig}(\mathcal{X})$ is compatible with Ψ if and only if it is compatible with each $\psi \in S$.*

Now consider an arbitrary fixed algebra with finitely many types, each type having a finite domain. Furthermore, assume that the following tasks can be computed in deterministic polynomial time: (i) given an assignment to the variables in a finite family of variables and a term over the variables, evaluate the term, and (ii) given a domain permutation, apply it to an element in the domain of a type or to a variable assignment over a finite family of variables.

Problem 5.20 TERM COMPATIBILITY 2 *(*TC2*). Given a term term and a domain permutation group Ψ by means of a generating set S, is term compatible with Ψ?*

Theorem 5.21 *Under a fixed algebra fulfilling the assumptions made above,* TC2 *is* **co-NP***-complete.*

Proof. The **co-NP**-hardness follows directly from the proof of Theorem 5.15 (setting Ψ to consist of the the identity mapping and the truth value flipping permutation ψ, f' is compatible with Ψ if and only if f is valid).

As Ψ is given by a generating set, one can solve the problem TC2 by (i) non-deterministically guessing a disqualifying domain permutation ψ in the generating set S, (ii) non-deterministically choosing a disqualifying assignment, and (iii) proceeding in the same way as in the inclusion part of Theorem 5.15. Thus TC2 is in **co-NP**. □

The existence of compatible domain permutations for a term is another interesting question.

Problem 5.22 EXISTENCE OF COMPATIBLE DOMAIN PERMUTATIONS. *Given a term term, is there a non-identity domain permutation ψ such that term is compatible with ψ?*

By the above theorems, this problem is apparently not easy since even verifying whether a single domain permutation is compatible with the term is hard. Notice the difference to place/transition nets described in Chapter 3: it is easy to verify whether a permutation of places and transitions is an automorphism of a place/transition net.

5.3.5 Approximating Term Compatibility

In the light of results presented above, an approximation scheme for term compatibility is highly desirable. One such scheme can be achieved by considering a lower level compatibility, namely that between domain permutations and individual operations.

Definition 5.23 *An operation $f^{\mathcal{A}}$, where $f \in \mathcal{F}_{T_1 \ldots T_n, T}$, and a domain permutation $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ are* compatible *if*

$$f^{\mathcal{A}}(\psi^{T_1}(v_1), \ldots, \psi^{T_n}(v_n)) = \psi^T(f^{\mathcal{A}}(v_1, \ldots, v_n))$$

holds for all $v_1 \in \tilde{\mathcal{D}}^{\mathcal{A}}_{T_1}, \ldots, v_n \in \tilde{\mathcal{D}}^{\mathcal{A}}_{T_n}$. An operation f is compatible with a domain permutation group Ψ if it is compatible with each domain permutation in Ψ.

Hence, an operation is compatible with a domain permutation if for permuted input arguments it will give a similarly permuted output. An immediate consequence of the definition is that a domain permutation compatible with a constant f (recall that constants are operations) cannot permute its value to other values since $f^{\mathcal{A}}() = \psi^T(f^{\mathcal{A}}())$ must hold. Also note that in the usual case when an operation returns $\mathfrak{err}$ whenever and only when any of the arguments is $\mathfrak{err}$, it suffices to consider the compatibility under the non-error arguments since $\mathfrak{err}$ is never permuted. It is easy to prove that terms composed only of variables and compatible operations are compatible.

Lemma 5.24 *Let $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ be a domain permutation and let $term \in Terms^{Sig}_T(\mathcal{X})$ be a term such that the operations appearing in it are compatible with ψ. Then $eval_{\psi(\alpha)}(term) = \psi^T(eval_\alpha(term))$ holds for all assignments α on $\mathcal{X}$, i.e., term is compatible with ψ.*

Proof. By induction on the structure of the term.

Induction base. If $term = x$, where x is a variable of type T, then $eval_{\psi(\alpha)}(term) = \psi(\alpha)(x) = \psi^T(\alpha(x)) = \psi^T(eval_\alpha(term))$.

Induction hypothesis. Let $term_i \in Terms^{Sig}_{T_i}(\mathcal{X})$, $1 \leq i \leq n$, be terms compatible with ψ (that is, $eval_{\psi(\alpha)}(term_i) = \psi^{T_i}(eval_\alpha(term_i))$).

Induction step. Assume $term = f(term_1, \ldots, term_n) \in Terms^{Sig}_T(\mathcal{X})$, where $f \in \mathcal{F}_{T_1, \ldots, T_n, T}$. Now

$$\begin{aligned} eval_{\psi(\alpha)}(term) &= f^{\mathcal{A}}(eval_{\psi(\alpha)}(term_1), \ldots, eval_{\psi(\alpha)}(term_n)) \\ &= f^{\mathcal{A}}(\psi^{T_1}(eval_\alpha(term_1)), \ldots, \psi^{T_n}(eval_\alpha(term_n))) \\ &= \psi^T(f^{\mathcal{A}}(eval_\alpha(term_1), \ldots, eval_\alpha(term_n))) \\ &= \psi^T(eval_\alpha(term)). \end{aligned}$$

by applying the induction hypothesis and the fact that $f^{\mathcal{A}}$ is compatible with ψ. □

Example 5.25 The requirement that the operations appearing in a term are compatible is a sufficient condition for the term to be compatible. To see that it is not a necessary one, consider a term $\mathsf{equals}(\mathsf{c}(), \mathsf{d}())$, where c and d are

constants of a type T and $\mathsf{equals} \in \mathcal{F}_{T.T,\mathsf{Bool}}$ is such that $\mathsf{equals}^{\mathcal{A}}(x, y) = true$ if and only if x and y are the same element. The constants c and d are not compatible with any domain permutation permuting the elements $\mathsf{c}^{\mathcal{A}}()$ and $\mathsf{d}^{\mathcal{A}}()$ in $\mathcal{D}_T^{\mathcal{A}}$. But the term itself is compatible with all domain permutations that leave the domain of Bool intact. This is because $\mathsf{c}^{\mathcal{A}}()$ and $\mathsf{d}^{\mathcal{A}}()$ are the same element if and only if $\psi^T(\mathsf{c}^{\mathcal{A}}())$ and $\psi^T(\mathsf{d}^{\mathcal{A}}())$ are the same element (by the bijective nature of permutations). ♣

Remark 5.26 *The definition of compatibility between domain permutations and operations is similar to that of signature isomorphisms in [Wirsing 1990]. That is, if all operations are compatible with a domain permutation, then the domain permutation is a Sig-isomorphism from the algebra $\mathcal{A}$ to itself.*

Approximating term compatibility through operation compatibility is especially convenient when defining net classes for automatic reachability analyzers. That is, following the approaches taken in [Chiola et al. 1991] and in [Ip and Dill 1996], the type system of a net description language is such that it allows definition of special type classes to which only certain restricted operations can be applied. Operations are analyzed by hand *once* during the language design phase in order to find out what kind of domain permutations are compatible with them. Thus checking the compatibility of a term corresponds to checking that only the allowed compatible operations appear in it. This approach is illustrated the next chapter.

6 EXTENDED WELL-FORMED NETS

In order to illustrate the theory developed in the previous chapter, a subclass of algebraic system nets is now defined. The proposed net class is called *extended well-formed nets* (EWF-nets) because it is inspired by and extends the class of well-formed nets [Chiola et al. 1991]. See [Mäkelä 2001b; 2001a; 2002] for a discussion on implementing a reachability analyzer for a high-level Petri net class similar to EWF-nets.

The EWF-nets use signatures and algebras fulfilling certain rules. First, the types are partitioned into (i) primitive types, such as integers, Booleans, enumeration types, integer sub-ranges, process identifiers, and so on, and (ii) structured types, such as lists, sets and so on, built over the primitive types. To facilitate easy description and automatic verification of data symmetries, an approach similar to that used in well-formed nets [Chiola et al. 1991] and in the Murφ system [Ip and Dill 1996] is used. That is, the set of primitive types is further partitioned into the classes of ordered, cyclic, and unordered primitive types.[1] The idea is that the domains of ordered primitive types may not be permuted, cyclic primitive types allow cyclic permutations, and the domains of unordered primitive types can be permuted arbitrarily. The domains and domain permutations of structured types are uniquely defined by those for the primitive types. The compatibility of data manipulation operations on types is then classified once in the net class definition phase. This enables semi-automatic detection of symmetries by the following procedure.

1. The user (i.e., the modeler of the net) declares the primitive types to be either ordered, cyclic, or unordered.
2. The reachability analyzer tool verifies that only compatible operations appear in the arc and transition guard annotations. This is a very simple syntactical check.
3. If this is the case, the domain permutations will actually produce state space symmetries. These symmetries can be automatically exploited by the algorithms described in the next chapter.

The main difference of this procedure to the so-called permutation symmetry approach described in [Jensen 1995, Section 3.3] is that the compatibility of the data manipulation operations is classified once in the net class definition phase. This makes symmetry verification, the step 2 above, very simple, thus eliminating the need for building algorithms checking that the domain permutations are compatible with the net annotations (which, in the light of the complexity results in Section 5.3.4, can be computationally quite hard).

6.1 TYPE SYSTEM

The types, domains, and domain permutations used in EWF-nets are now defined. The same type system will also be used in the next chapter studying the algorithms for the orbit problems under data symmetries.

[1] Unordered primitive types are called scalar sets in the Murφ terminology.

6.1.1 Types

First, a set $\mathcal{T}_0$ of *primitive types* is assumed. Based on primitive types, the set $\mathcal{T}$ of *types* is defined by the grammar

$$T ::= T_0 \mid \mathsf{List}(T) \mid \mathsf{Struct}(T, \ldots, T) \mid \mathsf{Set}(T) \mid \mathsf{Multiset}(T) \mid \mathsf{AssocArray}(T, T) \mid \mathsf{Union}(T, \ldots, T)$$

where T_0 ranges over $\mathcal{T}_0$. The types in $\mathcal{T} \setminus \mathcal{T}_0$ are called *structured types over* $\mathcal{T}_0$.

Next, each primitive type $T \in \mathcal{T}_0$ is associated with a domain $\mathscr{D}_T$. Based on these, the domains of structured types are naturally defined by the following inductive rules:

$$\begin{array}{ll} \mathscr{D}_{\mathsf{List}(T)} = \mathscr{D}_T^* & \mathscr{D}_{\mathsf{Struct}(T_1,\ldots,T_n)} = \mathscr{D}_{T_1} \times \cdots \times \mathscr{D}_{T_n} \\ \mathscr{D}_{\mathsf{Set}(T)} = \wp(\mathscr{D}_T) & \mathscr{D}_{\mathsf{AssocArray}(T_1,T_2)} = [\mathscr{D}_{T_1} \rightsquigarrow \mathscr{D}_{T_2}] \\ \mathscr{D}_{\mathsf{Multiset}(T)} = [\mathscr{D}_T \to \mathbb{N}] & \mathscr{D}_{\mathsf{Union}(T_1,\ldots,T_n)} = \bigcup_{1 \le i \le n} \{T_i\} \times \mathscr{D}_{T_i} \end{array}$$

where $\wp(A)$ denotes the power set of the set A, and $[A \rightsquigarrow B]$ denotes the set of all partial functions from A to B.[2] For instance, if Int and Bool are primitive types with the domains $\mathscr{D}_{\mathsf{Int}} = \mathbb{Z}$ and $\mathscr{D}_{\mathsf{Bool}} = \mathbb{B} = \{\mathit{false}, \mathit{true}\}$, respectively, then List(Struct(Bool, Int, Int)) is a type with the domain $(\mathbb{B} \times \mathbb{Z} \times \mathbb{Z})^*$. Note that arrays can be defined by means of association arrays: if Int[1–10] is a primitive type with the domain $\mathscr{D}_{\mathsf{Int[1–10]}} = \{1, 2, \ldots, 10\}$, then the type AssocArray(Int[1–10], Int) corresponds to a 10-element array of integers (with the possibility for undefined array elements). Also notice that an element in the domain of an union type is a pair consisting of a type name and an element of that type. This enables one to retrieve the type of an element in an union in the case the domains of the unionized types are overlapping. For instance, consider the union type $\mathsf{Union}(T_1, T_2)$, where $T_1 = \mathsf{Struct}(\mathsf{Int}, \mathsf{Int})$ and $T_2 = \mathsf{List}(\mathsf{Int})$. Now the structure element $\langle T_1, \langle 3, 6\rangle\rangle$ is distinguished from the list element $\langle T_2, \langle 3, 6\rangle\rangle$.

Recall that the domain permutations for multisets used in ASNs must fulfill certain rules defined in Requirement 5.6. Similarly, the domain permutations for structured types are naturally defined by the domain permutations for the primitive types.

Requirement 6.1 *The following rules must hold for each domain permutation* $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ *used in EWF-nets.*

- $\psi^{\mathsf{List}(T)}(\langle v_1, \ldots, v_n\rangle) = \langle \psi^T(v_1), \ldots, \psi^T(v_n)\rangle$,
- $\psi^{\mathsf{Struct}(T_1,\ldots,T_n)}(\langle v_1, \ldots, v_n\rangle) = \langle \psi^{T_1}(v_1), \ldots, \psi^{T_n}(v_n)\rangle$,
- $\psi^{\mathsf{Set}(T)}(V) = \{\psi^T(v) \mid v \in V\}$,
- $\psi^{\mathsf{Multiset}(T)}(m) : \psi^T(v) \mapsto m(v)$,
- $\psi^{\mathsf{AssocArray}(T_1,T_2)}(a) = \{\langle \psi^{T_1}(v_1), \psi^{T_2}(v_2)\rangle \mid \langle v_1, v_2\rangle \in a\}$, *and*
- $\psi^{\mathsf{Union}(T_1,\ldots,T_n)}(\langle T_i, v\rangle) = \langle T_i, \psi^{T_i}(v)\rangle$.

That is, each domain permutation $\psi = \{\psi^T\}_{T \in \mathcal{T}}$ can be uniquely expressed by the domain permutation $\{\psi^T\}_{T \in \mathcal{T}_0}$ for the primitive types only.

[2] A partial function from a set A to a set B is a subset f of $A \times B$ such that each $a \in A$ appears at most once as the first component of pairs in f.

6.1.2 Permutable Primitive Types

In order to exploit symmetries by using the theory devised in Section 5.3 (especially in Section 5.3.5), the set of primitive types, $\mathcal{T}_0$, is partitioned into three subclasses: *ordered*, *cyclic*, and *unordered* primitive types. The following limitations are set on the domains and domain permutations of the types in these classes. First, the domains of the primitive types must comply with the following rules.

1. The domain of each ordered primitive type T is an arbitrary set.
2. The domain of each cyclic primitive type T is a finite set of form $\mathscr{D}_T = \{v_0, v_1, \ldots, v_{n-1}\}$ for an $n \geq 1$, associated with a cyclic successor function $succ_T$ such that $succ_T(v_i) = v_{i+1 \bmod n}$.
3. The domain of each unordered primitive type T is a finite set.

Second, a domain permutation $\psi^{\mathcal{T}} = \{\psi^T\}_{T \in \mathcal{T}}$ is *allowed* if it fulfills the following rules in addition to those described in Requirement 6.1.

1. For each ordered primitive type T, $\psi^T = \mathbf{I}$, i.e., the identity permutation.
2. For each cyclic primitive type T, $\psi^T = succ_T^k$ for a $0 \leq k < |\mathscr{D}_T|$, i.e., a rotation of $\mathscr{D}_T$ defined by the successor function.
3. For each unordered primitive type T, $\psi^T \in \mathrm{Sym}(\mathscr{D}_T)$, i.e., an arbitrary permutation of $\mathscr{D}_T$.

From now on, an allowed domain permutation is denoted by using the symbol θ rather than ψ. The set of all allowed domain permutations, denoted by Θ, is a domain permutation group. Cyclic and unordered primitive types are also called *permutable primitive types* and the set of such types of denoted by $\mathcal{T}_P$. Clearly, each allowed domain permutation can be uniquely described by giving the domain permutations for the permutable primitive types only.

In addition, it is assumed that the type Bool for Booleans with the domain $\mathscr{D}_{\mathsf{Bool}} = \{false, true\}$ is an ordered primitive type. Thus the type system clearly fulfills the Requirements 5.2 and 5.6. Furthermore, it is assumed that the type Nat for natural numbers with the domain $\mathscr{D}_{\mathsf{Nat}} = \{0, 1, 2, \ldots\}$ is an ordered primitive type.

Example 6.2 Consider again the railroad net in Figure 5.1(c) (recall Examples 5.4 and 5.17). Assume that the primitive type Secs is declared to be cyclic with the successor function as defined in Example 5.4, and that the primitive type Trains is unordered. Then there are $|\mathscr{D}_{\mathsf{Secs}}| \times |\mathscr{D}_{\mathsf{Trains}}|! = 6 \times 2! = 12$ allowed domain permutations, one of them being

$$\theta = \left(\theta^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_2 & s_3 & s_4 & s_5 & s_0 & s_1 \end{smallmatrix}\right), \theta^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right)$$

from Example 5.8. It maps the initial state

$$M_{init} = \{U \mapsto \langle t_a, s_0 \rangle + \langle t_b, s_3 \rangle, V \mapsto s_1 + s_4\}$$

to

$$\theta(M_{init}) = \{U \mapsto \langle t_a, s_5 \rangle + \langle t_b, s_2 \rangle, V \mapsto s_0 + s_3\}.$$

The domain permutation ψ_{bad} in Example 5.8 is not an allowed domain permutation because the permutation for the cyclic primitive type Secs is not a power of the successor function. ♣

6.2 OPERATIONS

In the following, operations for the type classes defined above are introduced and their compatibility with the allowed domain permutations is analyzed according to Definition 5.23. It turns out that this analysis is in most cases quite straightforward. Some of the operations have erroneous invocations in which case the special error element $\mathfrak{err}$ is returned. In fact, these operations are the reason why error algebras had to be introduced in the first place. For net classes simpler than EWF-nets, normal algebras would have sufficed.

Equality Testing
For each type T, the equality testing operation $\mathsf{equals} \in \mathcal{F}_{T.T,\mathsf{Bool}}$ is defined by $\mathsf{equals}(v, v) = \mathit{true}$ if $v = v'$ and false otherwise. Thus one can, for example, compare two lists $l, l' \in \mathscr{D}_{\mathsf{List}(T)}$ for equality by using the operation $\mathsf{equals} \in \mathcal{F}_{\mathsf{List}(T).\mathsf{List}(T),\mathsf{Bool}}$. Since all the allowed domain permutations are bijections and Bool is an ordered primitive type, $\mathsf{equals}(\theta^T(v), \theta^T(v')) = \mathit{true}$ if and only if $\mathsf{equals}(v, v') = \mathit{true}$ if and only if $\theta^{\mathsf{Bool}}(\mathsf{equals}(v, v')) = \mathit{true}$. Thus equals is compatible with all allowed domain permutations.

If-Then-Else
For each type T, the standard if-then-else operation $\mathsf{ite}_T \in \mathcal{F}_{\mathsf{Bool}.T.T,T}$ is defined by

$$\mathsf{ite}_T(b, v_1, v_2) = \begin{cases} v_1 & \text{if } b = \mathit{true} \\ v_2 & \text{if } b = \mathit{false}, \text{ and} \\ \mathfrak{err} & \text{if } b = \mathfrak{err}. \end{cases}$$

The ite_T operation may also be written in mix-fix notation as $\mathsf{if} \cdot \mathsf{then} \cdot \mathsf{else} \cdot$. The operation is compatible with each allowed domain permutation θ because the Booleans and the error values are not permuted. For instance, $\mathsf{ite}_T(\theta^{\mathsf{Bool}}(\mathit{true}), \theta^T(v_1), \theta^T(v_2)) = \mathsf{ite}_T(\mathit{true}, \theta^T(v_1), \theta^T(v_2)) = \theta^T(v_1) = \theta^T(\mathsf{ite}_T(\mathit{true}, v_1, v_2))$.

Finally, observe that $\mathsf{ite}_T(\mathit{true}, v_1, \mathfrak{err}) = v_1$ and $\mathsf{ite}_T(\mathit{false}, \mathfrak{err}, v_2) = v_2$, making an exception to the implicit assumption that an operation returns $\mathfrak{err}$ if any of its arguments is $\mathfrak{err}$.

Ordered Primitive Types
As the domains of ordered primitive types are not permuted by allowed domain permutations, all the operations involving only them are compatible with all allowed domain permutations. For instance, the natural number constants $\mathtt{0}, \mathtt{1}, \ldots \in \mathcal{F}_{\varepsilon,\mathsf{Nat}}$ defined by $\mathsf{n}() = n$ are such operations. So are the Boolean operations described in Example 5.1. Furthermore, the usual operations for natural numbers such as

1. $\mathsf{plus} \in \mathcal{F}_{\mathsf{Nat}.\mathsf{Nat},\mathsf{Nat}}$ defined by $\mathsf{plus}(v, v') = v + v'$,
2. $\mathsf{minus} \in \mathcal{F}_{\mathsf{Nat}.\mathsf{Nat},\mathsf{Nat}}$ defined by $\mathsf{minus}(v, v') = v - v'$ if $v \geq v'$ and $\mathfrak{err}$ otherwise, and
3. $\mathsf{less} \in \mathcal{F}_{\mathsf{Nat}.\mathsf{Nat},\mathsf{Bool}}$ defined by $\mathsf{less}(v, v') = \mathit{true}$ if $v < v'$ and false otherwise,

are also compatible with Θ.

Cyclic Primitive Types

Assume a cyclic primitive type T with a domain $\mathscr{D}_T = \{v_0, v_1, \ldots, v_{n-1}\}$, associated with the successor function $succ_T(v_i) = v_{i+1 \bmod n}$. Since the domain of T may be permuted by the allowed domain permutations, the constants $\mathsf{v}_i \in \mathcal{F}_{\varepsilon,T}$ defined by $\mathsf{v}_i() = v_i$ for $1 \leq i \leq n$ are *not* compatible with Θ. However, the parameterized successor element operation $\mathsf{successor} \in \mathcal{F}_{\mathsf{Nat},T,T}$ defined by $\mathsf{successor}(m, v) = succ_T^m(v)$ is compatible with Θ since for each allowed domain permutation θ in which $\theta^T = succ_T^k$ for a $0 \leq k < n$,

$$\begin{aligned} \mathsf{successor}(\theta^{\mathsf{Nat}}(m), \theta^T(v)) &= \mathsf{successor}(m, \theta^T(v)) \\ &= succ_T^m(\theta^T(v)) \\ &= succ_T^m(succ_T^k(v)) \\ &= succ_T^k(succ_T^m(v)) \\ &= \theta^T(\mathsf{successor}(m, v)). \end{aligned}$$

Define that an element $v \in \mathscr{D}_T$ is the *k-successor of an element* $v' \in \mathscr{D}_T$ if k is the smallest integer such that $succ_T^k(v') = v$. Clearly, for each $v, v' \in \mathscr{D}_T$ and each allowed domain permutation θ, v is the k-successor of v' if and only if $\theta^T(v)$ is the k-successor of $\theta^T(v')$. Thus the *distance operation* $\mathsf{dist} \in \mathcal{F}_{T,T,\mathsf{Nat}}$, defined by $\mathsf{dist}(v, v') = k$ if v' is the k-successor of v, is compatible with Θ.

By using the operations above, it is possible to simulate more complex operations. For instance, assume that $term, term_2, term_3$ are terms of type T. Now the immediate successor of the element described by $term$ can be defined by the term $\mathsf{successor}(\mathtt{1}, term)$, abbreviated by $\mathsf{successor}(term)$, and the predecessor element can be defined by the term $\mathsf{successor}(\mathtt{n\text{-}1}, term)$, abbreviated by $\mathsf{predecessor}(term)$. Similarly, $\mathsf{between}(term, term_1, term_2)$ abbreviates the Boolean term $\mathsf{less}(\mathsf{dist}(term_1, term), \mathsf{dist}(term_1, term_2))$.

Unordered Primitive Types

Since the domain of an unordered primitive type T can be permuted arbitrarily by allowed domain permutations, there are not many compatible operations for it. One such exception is the equality testing operation discussed above. Especially, again, the constants $\mathsf{v} \in \mathcal{F}_{\varepsilon,T}$ defined by $\mathsf{v}() = v$ for each $v \in \mathscr{D}_T$ are not compatible with Θ. However, note that unordered (like ordered and cyclic) primitive types can be used as elements in sets, lists, etc. by the operations described below.

In some cases, it may be necessary to include some special, unpermuted elements in the domain of an unordered (or cyclic) primitive type T. For instance, a special "undefined" value is often required to handle the case when the value of a variable is not defined. As this kind of elements are not permuted, they can be accessed by using constants and other operations not allowed for permutable elements. Including unpermuted elements can be handled in two ways. The first one is by specifying an ordered primitive type T' that includes all such unpermuted elements and using the union type $\mathsf{Union}(T, T')$ instead of T. The other way of achieving the same is to (i) extend the domain of T to include the unpermuted elements and (ii) define the allowed domain permutations for T to permute only the permutable sub-

set of the domain. The latter approach requires that some obvious, trivial changes are made in the algorithms for the orbit problems presented in the next chapter.

Quantification

Before presenting operations for structured types, an important way of defining new operations based on the existing ones is introduced by an example. Consider that one wants to check whether a Boolean condition holds for all the elements of a type T. The Boolean condition itself can be expressed as a Boolean term. Now define an operation

$$\text{“for all } z : T \text{ it holds } term\text{''} \in \mathcal{F}_{T_1 \ldots T_n, \mathsf{Bool}},$$

where $term$ is a Boolean term over the family of variables consisting of the variable z of type T and the variables $z_1, \ldots, z_n$ of types $T_1, \ldots, T_n$, respectively. In order to have well-foundedness, i.e., non-cyclic definitions, the term $term$ may not contain the operation “for all $x : T$ it holds $term$” itself. The idea is that the variable z is the universally quantified variable, while the free variables $z_1, \ldots, z_n$ in $term$ are bound by the arguments for the operation. Thus “for all $x : T$ it holds $term$”$(v_1, \ldots, v_n)$ evaluates to

- $true$ if $eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term) = true$ for all $v \in \mathscr{D}_T$,
- $\mathfrak{err}$ if $eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term) = \mathfrak{err}$ for a $v \in \mathscr{D}_T$, and
- $false$ otherwise.

The important thing is that if the term $term$ is compatible with all allowed domain permutations, then the operation is, too. The compatibility of $term$ in turn can be derived by checking that only compatible operations appear in it.

Lemma 6.3 *If the term $term$ is compatible with an allowed domain permutation θ, then the operation “for all $x : T$ it holds $term$” is, too.*

Proof. It suffices to note that $eval_{\{z_1 \mapsto \theta^{T_1}(v_1), \ldots, z_n \mapsto \theta^{T_n}(v_n), z \mapsto \theta^{T}(v)\}}(term) = eval_{\theta(\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\})}(term) = \theta^{\mathsf{Bool}}(eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term)) = eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term)$ as $term$ is compatible with θ and the Booleans are not permuted. Thus $\theta^{\mathsf{Bool}}(\text{“for all } x : T \text{ it holds } term\text{''}(v_1, \ldots, v_n)) = \text{“for all } x : T \text{ it holds } term\text{''}(v_1, \ldots, v_n)$ always yields the same values as $\text{“for all } x : T \text{ it holds } term\text{''}(\theta^{T_1}(v_1), \ldots, \theta^{T_n}(v_n))$. □

As an example, consider the operation

$$\begin{array}{l}\text{“for all } z : \mathsf{Struct}(\mathsf{Bool}, \mathsf{PIds}) \text{ it holds} \\ \quad \mathsf{implies}(\mathsf{greaterOrEqual}(\mathsf{multiplicity}(z_1, z), 1), \\ \qquad \mathsf{equals}(\mathsf{getField2}(z), z_2))\text{''}\end{array}$$

in $\mathcal{F}_{\mathsf{Multiset(Struct(Bool,PIds)).PIds,Bool}}$. It evaluates to true if and only if the multiset given as the first argument contains only elements of type Struct(Bool, PIds) whose second field is equal to the second argument (the operations appearing in the term are formally defined later). The operation can be written in an informal abbreviated form if it is understood what is meant. For instance,

$$\forall z \text{ of } \mathsf{Struct}(\mathsf{Bool}, \mathsf{PIds}) : (x_1(z) \geq 1 \Rightarrow \mathsf{getField2}(z) = x_2)$$

denotes the term formed from the operation when the arguments for it are the variables x_1 and x_2.

Nesting operations defined this way can be somewhat cumbersome, at least in the formal way. For instance, consider the quantified Boolean expression

$$\forall y_1 \text{ of } \mathsf{Bool} : (\forall y_2 \text{ of } \mathsf{Bool} : (y_2 \wedge y_1) \vee (\neg y_1 \wedge y_2)),$$

easily written in the informal abbreviated form. The innermost quantifier can be expressed by a Boolean operation

$$\text{“for all } z : \mathsf{Bool} \text{ it holds } (z \wedge z_1) \vee (\neg z_1 \wedge z)''$$

in $\mathcal{F}_{\mathsf{Bool},\mathsf{Bool}}$. Thus z corresponds to the variable y_2 and z_1 corresponds to the variable y_1. Now the whole expression is the constant term

$$\text{“for all } z : \mathsf{Bool} \text{ it holds “for all } z : \mathsf{Bool} \text{ it holds } (z \wedge z_1) \vee (\neg z_1 \wedge z)''(z)''$$

in $\mathcal{F}_{\varepsilon,\mathsf{Bool}}$. Note that the outermost term is

$$\text{“for all } z : \mathsf{Bool} \text{ it holds } (z \wedge z_1) \vee (\neg z_1 \wedge z)''(z),$$

i.e., a term consisting of one unary operation and the variable z (the variable z_1 only appears in the name of the operation, and is thus considered to be merely a text string).

Lists

For a list type $\mathsf{List}(T)$, the following operations are compatible with all allowed domain permutations.

- Operation: $\mathsf{emptyList} \in \mathcal{F}_{\varepsilon,\mathsf{List}(T)}$
  Definition: $\mathsf{emptyList}() = \langle\rangle$
- Operation: $\mathsf{isEmpty} \in \mathcal{F}_{\mathsf{List}(T),\mathsf{Bool}}$
  Definition: $\mathsf{isEmpty}(l) = \mathit{true}$ if $l = \langle\rangle$ and false otherwise
- Operation: $\mathsf{length} \in \mathcal{F}_{\mathsf{List}(T),\mathsf{Nat}}$
  Definition: $\mathsf{length}(\langle v_1, v_2, \ldots, v_n\rangle) = n$
- Operations: $\mathsf{getElement} \in \mathcal{F}_{\mathsf{List}(T).\mathsf{Nat},T}$ and
  $\mathsf{setElement} \in \mathcal{F}_{\mathsf{List}(T).\mathsf{Nat}.T,\mathsf{List}(T)}$
  Definitions: $\mathsf{getElement}(\langle v_1, ..., v_n\rangle, i) = \begin{cases} v_i & \text{if } 1 \le i \le n \\ \mathfrak{err} & \text{otherwise} \end{cases}$
  $\mathsf{setElement}(\langle v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n\rangle, i, v) =$
  $\begin{cases} \langle v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n\rangle & \text{if } 1 \le i \le n \\ \mathfrak{err} & \text{otherwise} \end{cases}$
- Operations: $\mathsf{addFirst}, \mathsf{addLast} \in \mathcal{F}_{\mathsf{List}(T).T,\mathsf{List}(T)}$
  Definitions: $\mathsf{addFirst}(\langle v_1, \ldots, v_n\rangle, v) = \langle v, v_1, \ldots, v_n\rangle$
  $\mathsf{addLast}(\langle v_1, \ldots, v_n\rangle, v) = \langle v_1, \ldots, v_n, v\rangle$

– Operations: removeBeginning, removeEnd $\in \mathcal{F}_{\mathsf{List}(T).\mathsf{Nat},\mathsf{List}(T)}$
Definitions:

$$\mathsf{removeBeginning}(\langle v_1,\ldots,v_n\rangle, l) = \begin{cases} \langle v_{l+1},\ldots,v_n\rangle & \text{if } l < n \\ \langle\rangle & \text{if } l \geq n \end{cases}$$

$$\mathsf{removeEnd}(\langle v_1,\ldots,v_n\rangle, l) = \begin{cases} \langle v_1,\ldots,v_{n-l}\rangle & \text{if } l < n \\ \langle\rangle & \text{if } l \geq n \end{cases}$$

A version which returns $\mathfrak{err}$ if $l > n$ could also be defined.

– Operation: concatenate $\in \mathcal{F}_{\mathsf{List}(T).\mathsf{List}(T),\mathsf{List}(T)}$
Definition: $\mathsf{concatenate}(\langle v_1, v_2,\ldots,v_n\rangle, \langle u_1, u_2,\ldots,u_m\rangle) = \langle v_1, v_2,\ldots,v_n, u_1, u_2,\ldots,u_m\rangle$

Note that the usual types of stacks and first-in first-out buffers (FIFOs) are covered by the above definition of lists. If $term$ is a term of type $\mathsf{List}(T)$, one may use $\mathsf{first}(term)$ and $\mathsf{last}(term)$ to denote the terms $\mathsf{getElement}(term, \mathtt{1})$ and $\mathsf{getElement}(term, \mathsf{length}(term))$, respectively. Similarly, one may write $\mathsf{removeFirst}(term)$ to abbreviate the term $\mathsf{removeBeginning}(term, \mathtt{1})$ and $\mathsf{removeLast}(term)$ for $\mathsf{removeEnd}(term, \mathtt{1})$.

Structures

For a structure sort $\mathsf{Struct}(T_1,\ldots,T_n)$, only the element manipulation and construction operations are needed.

– Operation: $\mathsf{getField}i \in \mathcal{F}_{\mathsf{Struct}(T_1,\ldots,T_n),T_i}$, for each $1 \leq i \leq n$
Definition: $\mathsf{getField}i(\langle v_1,\ldots,v_n\rangle) = v_i$

– Operation: $\mathsf{setField}i \in \mathcal{F}_{\mathsf{Struct}(T_1,\ldots,T_n).T_i,\mathsf{Struct}(T_1,\ldots,T_n)}$, $1 \leq i \leq n$
Definition: $\mathsf{setField}i(\langle v_1,\ldots,v_{i-1}, v_i, v_{i+1}\ldots,v_n\rangle, v_i') = \langle v_1,\ldots,v_{i-1}, v_i', v_{i+1}\ldots,v_n\rangle$

– Operation: makeStruct $\in \mathcal{F}_{T_1.\ldots.T_n,\mathsf{Struct}(T_1,\ldots,T_n)}$
Definition: $\mathsf{makeStruct}(v_1,\ldots,v_n) = \langle v_1,\ldots,v_n\rangle$

These operations are clearly compatible with all allowed domain permutations.

Sets

Assume a set type $\mathsf{Set}(T)$ and let $V, V_1, V_2 \in \wp(\mathscr{D}_T)$ be sets belonging to the domain of $\mathsf{Set}(T)$. The compatibility of the following operations with all allowed domain permutations is quite evident by noticing that, for each allowed domain permutation $\theta = \{\theta^T\}_{T\in\mathcal{T}}$, an element $v \in \mathscr{D}_T$ is in the set V if and only if the element $\theta^T(v)$ is in the set $\theta^{\mathsf{Set}(T)}(V)$.

– Operation: cardinality $\in \mathcal{F}_{\mathsf{Set}(T),\mathsf{Nat}}$
Definition: $\mathsf{cardinality}(V) = |V|$ if $|V|$ is finite and $\mathfrak{err}$ otherwise

– Operation: isIn $\in \mathcal{F}_{T.\mathsf{Set}(T),\mathsf{Bool}}$
Definition: $\mathsf{isIn}(v, V) = \mathit{true}$ if $v \in V$ and false otherwise

– Operation: "all $z : T$ such that $term$" $\in \mathcal{F}_{T_1\ldots T_n,\mathsf{Set}(T)}$ where $term$ is a Boolean term over the family $\mathcal{X}$ of variables consisting of the variable z of type T and the variables $z_1,\ldots,z_n$ of types $T_1,\ldots,T_n$, respectively

Definition: "all $z : T$ such that $term$"$(v_1, \ldots, v_n) =$
$$\left\{v \in \mathscr{D}_T \mid eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term) = true\right\}$$
Returns the set of elements of type T for which the term $term$ evaluates to true under the values given as the arguments (cf. the quantification operation defined earlier). For example, if PIds is a primitive type with $\mathscr{D}_{\mathsf{PIds}} = \{pid_1, pid_2, pid_3\}$, then the operation

$$\text{"all } z : \mathsf{Struct}(\mathsf{PIds}, \mathsf{PIds}) \text{ such that } \mathsf{equals}(\mathsf{getField1}(z), z_1)\text{"}$$

in $\mathcal{F}_{\mathsf{PIds},\mathsf{Set}(\mathsf{Struct}(\mathsf{PIds},\mathsf{PIds}))}$ gives under the the argument pid_2 the set $\{\langle pid_2, pid_1\rangle, \langle pid_2, pid_2\rangle, \langle pid_2, pid_3\rangle\}$.
If the term $term$ is compatible with all allowed domain permutations, the operation is, too. Again, in order to avoid cyclic definitions, the term $term$ may not contain the operation "all $z : T$ such that $term$".

– Operation: $\mathsf{toMultiSet} \in \mathcal{F}_{\mathsf{Set}(T),\mathsf{Multiset}(T)}$
Definition: $\mathsf{toMultiSet}(V) = \Sigma_{v \in V} 1'v$

Other typical set operations can again be abbreviated by the ones listed above. Let $term_1, term_2$ be terms of type $\mathsf{Set}(T)$. Now $\mathsf{emptySet}()$ is an abbreviation for the term "all $z : T$ such that false"() returning the empty set, $\mathsf{fullSet}()$ is an abbreviation for the term "all $z : T$ such that true"() returning the set $\mathscr{D}_T$, $\mathsf{complement}(term_1)$ is an abbreviation for the term

$$\text{"all } z : T \text{ such that } \mathsf{not}(\mathsf{isIn}(z, z_1))\text{"}(term_1),$$

$\mathsf{union}(term_1, term_2)$ is an abbreviation for

$$\text{"all } z : T \text{ such that } \mathsf{or}(\mathsf{isIn}(z, z_1), \mathsf{isIn}(z, z_2))\text{"}(term_1, term_2),$$

$\mathsf{intersection}(term_1, term_2)$ is an abbreviation for

$$\text{"all } z : T \text{ such that } \mathsf{and}(\mathsf{isIn}(z, z_1), \mathsf{isIn}(z, z_2))\text{"}(term_1, term_2),$$

and $\mathsf{setMinus}(term_1, term_2)$ is an abbreviation for

$$\text{"all } z : T \text{ such that } \mathsf{and}(\mathsf{isIn}(z, z_1), \mathsf{not}(\mathsf{isIn}(z, z_2)))\text{"}(term_1, term_2).$$

Multisets
Assume a multiset type $\mathsf{Multiset}(T)$. The compatibility of the following operations with all allowed domain permutations is partly proven in Lemma 5.7 and can be easily seen by the fact that, for each allowed domain permutation $\theta = \left\{\theta^T\right\}_{T \in \mathcal{T}}$, an element $v \in \mathscr{D}_T$ appears n times in a multiset $m \in \mathscr{D}_{\mathsf{Multiset}(T)}$ if and only if the element $\theta^T(v)$ appears n times in the multiset $\theta^{\mathsf{Multiset}(T)}(m)$.

– Operation: $\mathsf{multiplicity} \in \mathcal{F}_{\mathsf{Multiset}(T).T,\mathsf{Nat}}$
Definition: $\mathsf{multiplicity}(m, v) = m(v)$
– Operation: $\mathsf{add} \in \mathcal{F}_{\mathsf{Multiset}(T).\mathsf{Multiset}(T),\mathsf{Multiset}(T)}$
Definition: $\mathsf{add}(m_1, m_2) = m_1 + m_2$
– Operation: $\mathsf{minus} \in \mathcal{F}_{\mathsf{Multiset}(T).\mathsf{Multiset}(T),\mathsf{Multiset}(T)}$

Definition: $\mathsf{minus}(m_1, m_2) = \begin{cases} m_1 - m_2 & \text{if } m_2 \leq m_1 \\ \mathfrak{err} & \text{otherwise} \end{cases}$

- Operation: $\mathsf{times} \in \mathcal{F}_{\mathsf{Nat.Multiset}(T),\mathsf{Multiset}(T)}$
  Definition: $\mathsf{times}(n, m) = n \cdot m$
- Operation: $\mathsf{lessOrEqual} \in \mathcal{F}_{\mathsf{Multiset}(T).\mathsf{Multiset}(T),\mathsf{Bool}}$
  Definition: $\mathsf{lessOrEqual}(m_1, m_2) = \mathit{true}$ if and only if $m_1 \leq m_2$
- Operation: $\mathsf{unitMS} \in \mathcal{F}_{T,\mathsf{Multiset}(T)}$
  Definition: $\mathsf{unitMS}(v) = 1'v$
- Operation: all
  An abbreviation for toMultiSet("all $x : T$ such that true"). Returns the multiset consisting of one copy of each element in the domain of T.
- Operation: $\mathsf{size} \in \mathcal{F}_{\mathsf{Multiset}(T),\mathsf{Nat}}$
  Definition: $\mathsf{size}(m) = \sum_{v \in \mathscr{D}_T} m(v)$ if the sum is finite and $\mathfrak{err}$ if it is not.

  Returns the number of all elements in a multiset i.e. its size. In the case of an infinite multiset, the error return value was chosen because introducing ∞ in the domain of Nat could cause some inconveniences elsewhere. In practice, all multisets can be expected to be finite.
- Operation: $\mathsf{toSet} \in \mathcal{F}_{\mathsf{Multiset}(T),\mathsf{Set}(T)}$
  Definition: $\mathsf{toSet}(m) = \{v \in \mathscr{D}_T \mid m(v) \geq 1\}$
- Operation: "construct $term$" $\in \mathcal{F}_{T_1 \ldots T_n,\mathsf{Multiset}(T)}$, $term$ being a term of type Nat over the family $\mathcal{X}$ of variables consisting of the variable z of type T and the variables $z_1, \ldots, z_n$ of types $T_1, \ldots, T_n$, respectively
  Definition: "construct $term$"$(v_1, \ldots, v_n) = \sum_{v \in \mathscr{D}_T} eval_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term)'v$ or $\mathfrak{err}$ if $term$ evaluates to $\mathfrak{err}$ for the assignment $\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}$ for a $v \in \mathscr{D}_T$

  Returns the multiset where the multiplicity of an element is computed by applying the term $term$. If the term $term$ is compatible with all allowed domain permutations, the operation is, too (by recalling the fact that Nat is an ordered primitive type).
  For instance, the operation

  "construct if equals(z, z_1) then 0 else multiplicity(z_2, z)"

  in $\mathcal{F}_{T.\mathsf{Multiset}(T),\mathsf{Multiset}(T)}$ for a type T returns the multiset over T in which an element v has multiplicity 0 if it equals to the first argument and the same multiplicity as in the multiset given as the second argument otherwise. That is, if $\mathscr{D}_T = \{v_1, v_2, v_3\}$, x_1 is a variable of type T assigned to v_2, and x_2 is a variable of type $\mathsf{Multiset}(T)$ assigned to $2'v_1 + 5'v_2 + 3'v_3$, then

  "construct if equals(z, z_1) then 0 else multiplicity(z_2, z)"$(x_1, x_2) = 2'v_1 + 0'v_2 + 3'v_3$.

  Similarly, "construct 1"$() = 1'v_1 + 1'v_2 + 1'v_3$ and "construct 0" is the constant operation for the empty multiset.

Association Arrays

Let $T = \mathsf{AssocArray}(T_1, T_2)$ be an association array type. The compatibility of operations below with each allowed domain permutation θ can be established by noticing that an element $v_1 \in \mathscr{D}_{T_1}$ is associated with an element $v_2 \in \mathscr{D}_{T_2}$ by an association array $a \in \mathscr{D}_T$ if and only if the element $\theta^{T_1}(v_1)$ is associated with the element $\theta^{T_2}(v_2)$ by the association array $\theta^T(a)$.

- Operation: $\mathsf{emptyAssocArray} \in \mathcal{F}_{\varepsilon,\mathsf{AssocArray}(T_1,T_2)}$
  Definition: $\mathsf{emptyAssocArray}() = \emptyset$
- Operation: $\mathsf{isDefined} \in \mathcal{F}_{\mathsf{AssocArray}(T_1,T_2).T_1,\mathsf{Bool}}$
  Definition: $\mathsf{isDefined}(a, v) = \mathit{true}$ if $a(v)$ is defined and false otherwise
- Operations: $\mathsf{get} \in \mathcal{F}_{\mathsf{AssocArray}(T_1,T2).T_1,T_2}$,
  $\mathsf{set} \in \mathcal{F}_{\mathsf{AssocArray}(T_1,T2).T_1.T_2,\mathsf{AssocArray}(T_1,T2)}$, and
  $\mathsf{unset} \in \mathcal{F}_{\mathsf{AssocArray}(T_1,T2).T_1,\mathsf{AssocArray}(T_1,T2)}$
  Definitions: $\mathsf{get}(a, i) = \begin{cases} a(i) & \text{if } a(i) \text{ is defined} \\ \mathfrak{err} & \text{otherwise} \end{cases}$
  $\mathsf{set}(a, i, v) = a[i \mapsto v]$
  $\mathsf{unset}(a, i) = \begin{cases} a \setminus \{\langle i, a(i)\rangle\} & \text{if } a(i) \text{ is defined} \\ a & \text{otherwise} \end{cases}$
- Operation: "initialize to $term$" $\in \mathcal{F}_{T'_1 \ldots T'_n,\mathsf{AssocArray}(T_1,T_2)}$, where $term$ is a term of type T_2 over the family $\mathcal{X}$ of variables consisting of the variable z of type T_1 and the variables $z_1, \ldots, z_n$ of types $T'_1, \ldots, T'_n$, respectively
  Definition: "initialize to $term$"$(v_1, \ldots, v_n) =$
  $\{\langle v, \mathit{eval}_{\{z_1 \mapsto v_1, \ldots, z_n \mapsto v_n, z \mapsto v\}}(term)\rangle \mid v \in \mathscr{D}_{T_1}\}$.
  Returns the association array in which the image of an element is computed by applying the term $term$. If the term $term$ is compatible with all allowed domain permutations, the operation is, too.
  For instance, assume that Ring is a cyclic primitive type with the domain $\mathscr{D}_{\mathsf{Ring}} = \{r_0, r_1, r_2\}$ and the obvious successor function. Now the operation "initialize to $\mathsf{makeStruct}(\mathsf{successor}(z), z_1)$" from Nat to $\mathsf{AssocArray}(\mathsf{Ring}, \mathsf{Struct}(\mathsf{Ring}, \mathsf{Nat}))$ returns with the argument 2 the association array $\{r_0 \mapsto \langle r_1, 2\rangle, r_1 \mapsto \langle r_2, 2\rangle, r_2 \mapsto \langle r_0, 2\rangle\}$.

Unions

The following operations for union types are clearly compatible with all allowed domain permutations.

- Operation: $\mathsf{make}_T \in \mathcal{F}_{T,\mathsf{Union}(T_1,\ldots,T_n)}$ for each $T \in \{T_1, \ldots, T_n\}$
  Definition: $\mathsf{make}_T(v) = \langle T, v\rangle$
  Creates an union element out of a constituent type element.
- Operation: $\mathsf{isOf}T \in \mathcal{F}_{\mathsf{Union}(T_1,\ldots,T_n),\mathsf{Bool}}$ for each $T \in \{T_1, \ldots, T_n\}$
  Definition: $\mathsf{isOf}T(\langle T', v\rangle) = \mathit{true}$ if $T' = T$ and false otherwise
  Checks the type of the union element.
- Operation: $\mathsf{castTo}T \in \mathcal{F}_{\mathsf{Union}(T_1,\ldots,T_n),T}$ for each $T \in \{T_1, \ldots, T_n\}$
  Definition: $\mathsf{castTo}T(\langle T', v\rangle) = v$ if $T' = T$ and $\mathfrak{err}$ otherwise
  "Casts" an union element back to its base type or returns error if types mismatch.

6.3 EXAMPLES

In the following, some very simple examples of EWF-nets are given. More complicated examples can be obtained by translating existing well-formed and colored Petri nets into extended well-formed nets. For instance, the colored Petri net model of Lamport's fast mutual exclusion algorithm presented in [Jørgensen and Kristensen 1999] can be interpreted as an EWF-net in a very straightforward way. Furthermore, the same symmetries of the model that are used in [Jørgensen and Kristensen 1999] can be defined in the corresponding EWF-net model by simply declaring the process identifier type PID to be unordered. For more examples of modeling and analyzing systems with high-level Petri nets, see e.g. [Jensen 1997; Reisig and Rozenberg 1998b].

Railroad net. The railroad net in Figure 5.1(c) (also recall Examples 5.4 and 6.2) is an EWF-net. Its symmetries can be exploited by just declaring the type Secs to be cyclic and the type Trains to be unordered primitive types. Now the following LTL property can be used to verify that two trains are never in the same railroad section:

$$\mathbf{AG}(\forall x_1 \text{ of } \mathsf{Struct}(\mathsf{Trains},\mathsf{Secs}) : \forall x_2 \text{ of } \mathsf{Struct}(\mathsf{Trains},\mathsf{Secs}) : (\hat{U}(x_1) \geq 1 \wedge \hat{U}(x_2) \geq 1 \wedge \mathsf{getField1}(x_1) \neq \mathsf{getField1}(x_2)) \Rightarrow \mathsf{getField2}(x_1) \neq \mathsf{getField2}(x_2)),$$

where $\hat{U}$ is the place variable of type Multiset(Struct(Trains, Secs)) corresponding the marking in the place U as described in Section 5.3.3. As the atomic proposition in the formula only uses operations that are compatible with all the allowed domain permutations, it is invariant with respect to the symmetries of the net. Therefore, the formula can be checked directly on a reduced reachability graph of the net. Similarly, the following LTL property states that two trains can never be in the consecutive railroad sections:

$$\mathbf{AG}(\forall x_1 \text{ of } \mathsf{Struct}(\mathsf{Trains},\mathsf{Secs}) : \forall x_2 \text{ of } \mathsf{Struct}(\mathsf{Trains},\mathsf{Secs}) : (\hat{U}(x_1) \geq 1 \wedge \hat{U}(x_2) \geq 1 \wedge \mathsf{getField1}(x_1) \neq \mathsf{getField1}(x_2)) \Rightarrow \mathsf{getField2}(x_1) \neq \mathsf{successor}(\mathsf{getField2}(x_2))).$$

Distributed Database Net. Figure 6.1 shows an EWF-net version of the well-known distributed database system [Jensen 1992]. The type PIds for process identifiers is an unordered primitive type with the domain $\mathscr{D}_{\mathsf{PIds}} = \{pid_1, \ldots, pid_n\}$ for some n, and the arc annotation term Mes(s) is an abbreviation for

$$\text{"all } z : \mathsf{Struct}(\mathsf{PIds},\mathsf{PIds}) \text{ such that } \mathsf{getField1}(z) = z_1 \wedge \mathsf{getField2}(z) \neq z_1\text{"}(s).$$

That is, Mes(s) returns the set of all pairs of elements in PIds such that the first element equals to s and the second does not. As usual, the annotations in the figure are in an abbreviated informal form, e.g. the arc annotation s is formally unitMS(s) and Mes(s) means toMultiset(Mes(s)).

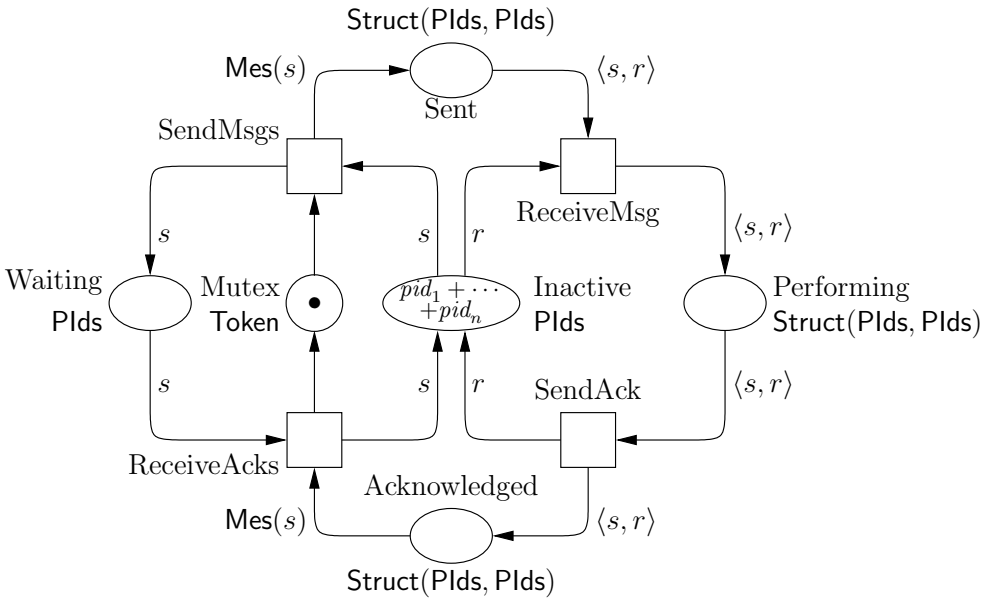


Figure 6.1: The distributed database net

7 ALGORITHMS FOR DATA SYMMETRIES

In this chapter, new algorithms are developed for the orbit problems under data symmetries, i.e., symmetries that are produced by symmetric use of data values. The studied framework is so general that it covers the extended well-formed nets described in the previous chapter, the original well-formed nets [Chiola et al. 1991], the Murφ verification system [Ip and Dill 1996], as well as the most commonly used instances of colored Petri nets (the so-called permutation symmetries in [Jensen 1995]). In the framework, a system is considered to consist of a finite set of typed state variables, a state being an assignment to the variables. The applied type system is the same as for the extended well-formed nets, covering the type systems in Murφ, well-formed nets, and the structured colors in [Jensen 1995, Section 3.3]. Symmetries of such systems are produced in the same way as in the above mentioned formalisms, i.e., by permuting the values of certain primitive types. The transition relation of a system is implicitly assumed to be induced by the applied formalism.

The first proposed algorithm family is based on building an ordered partition of the elements for each permutable primitive type appearing in a system state. The partition family is built in a symmetry-respecting way so that equivalent states are assigned equivalent partitions. The partitions can then be exploited to prune the set of symmetries that have to be considered when comparing whether two states are equivalent or when building a representative for the state. The partition family for a state is iteratively build via a process that refines the current partition by applying symmetry-respecting invariants to it. This approach resembles the one taken in Section 4.4 and also the preprocessing step in graph isomorphism algorithms [McKay 1981; Kreher and Stinson 1999]. As already mentioned in Section 1.1, using symmetry-respecting partitions to prune the set of symmetries is already used, for instance, in [Jensen 1995; Ip 1996; Sistla et al. 2000; Lorentsen 2002]. However, the approach presented in this chapter offers the following improvements.

1. The process of building the partition for a state, as well as the invariants needed in the process, are formally and rigorously defined.
2. Both unordered and cyclic primitive types are handled in the same unified way.
3. The invariants proposed in this chapter can handle all the structured types. Moreover, some very powerful invariants are proposed for types of certain forms.

In addition, a novel improvement inspired by graph isomorphism and canonization algorithms is presented. In this approach, the partition for a state, built as described above, is used as the root for a finite search tree. The search tree is built by iteratively splitting a cell in the partition and refining the resulting partition until a discrete partition is obtained. By considering only the leaf nodes in the search tree, i.e., the discrete partitions, the set of symmetries that have to be considered when solving the orbit problems can be further reduced.

The second algorithm family for solving the orbit problems is based on assigning a state the corresponding characteristic graph, i.e., on translating the state into a graph that captures the symmetries of the state. Thus the task of comparing whether two states are equivalent can be solved by testing whether the corresponding characteristic graphs are isomorphic. Similarly, one can obtain a canonical representative for a state by using the canonical version of its characteristic graph. Therefore, one can apply existing graph isomorphism and canonization algorithms, such as the *nauty* tool [McKay 1990], for solving the orbit problems. This resembles the approach taken in Section 4.2.

Some of the proposed algorithms are implemented in the Murφ tool. The experimental results show that the new algorithms are competitive against the previous ones implemented in Murφ.

The material in this chapter has been published in [Junttila 2002b].

7.1 AN ABSTRACT SYSTEM CLASS

First, an abstract system model is introduced. The model covers the well-formed nets [Chiola et al. 1991], the Murφ system [Ip and Dill 1996], and the extended well-formed nets described in the previous chapter in the sense that each system described with one of these formalisms can be transformed into the model. The main benefit of the model is that the details of the actual transition relation (the semantics of the actual formalism) are abstracted away. Those details play no role in the contributions of this chapter.

The applied type system is as described in Section 6.1 in the previous chapter. That is, the set $\mathcal{T}$ of types is partitioned into (i) primitive types such as Booleans, integers, process identifiers, and so on, and (ii) structured types such as lists, sets, association arrays etc. built on the primitive types. Furthermore, the set $\mathcal{T}_0$ of primitive types is partitioned into ordered, cyclic, and unordered ones. The cyclic and unordered primitive types are commonly called permutable and the set of such types is denoted by $\mathcal{T}_P$.

A *system* is defined to be a tuple $\mathfrak{S} = \langle \mathcal{X}, \longrightarrow, \mathfrak{s}_0 \rangle$, where

- $\mathcal{X}$ is a finite, non-empty set of *state variables*. Each $x \in \mathcal{X}$ is associated with a type T_x. A *state* $\mathfrak{s}$ is a mapping associating each $x \in \mathcal{X}$ with an element in the domain of its type T_x: $\mathfrak{s}(x) \in \mathscr{D}_{T_x}$. The *set of all states* is denoted by $\mathcal{S}$.
- $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* describing how the states may evolve to others, and
- $\mathfrak{s}_0 \in \mathcal{S}$ is the *initial state*.

The *state space* of the system $\mathfrak{S}$ is the unlabeled transition system (rooted graph) $\langle \mathcal{S}, \longrightarrow, \mathfrak{s}_0 \rangle$. To see the connection between this system model and the formalisms mentioned above, first consider a Murφ description of a system. Translation to the system model is easy since the Murφ description consists of (i) a type system similar to that in Section 6.1, (ii) a set of state variables, and (iii) a set of rules that transform the values of state variables, inducing the transition relation. Similarly, a well-formed net (extended or not) consists of (i) a type system, (ii) a set of places (which can be seen as state variables of multiset types), and (iii) a set of transitions connected to places with

arcs. The semantics of well-formed nets describe how the transitions modify the values of places (state variables) and thus induce a transition relation.

Example 7.1 Recall the EWF-net in Figure 5.1(c) (reprinted in Figure 7.1), discussed in Examples 5.4, 5.8, and 6.2. It can be seen as a system consisting of two state variables, U of type $\mathsf{Multiset}(\mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs}))$ and V of type $\mathsf{Multiset}(\mathsf{Secs})$, where Trains is an unordered primitive type with the domain $\mathscr{D}_{\mathsf{Trains}} = \{t_a, t_b\}$ and Secs is a cyclic primitive type with the domain $\mathscr{D}_{\mathsf{Secs}} = \{s_0, \ldots, s_5\}$. ♣

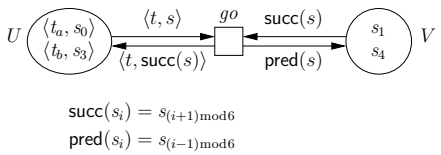


Figure 7.1: An EWF-net for a railroad system

Example 7.2 Figure 7.2 shows a Murφ version of the mutual exclusion program discussed in Example 2.1. The corresponding system has one state variable s of type $\mathsf{AssocArray}(\mathsf{PIds}, \mathsf{Loc})$, where PIds is an unordered primitive type with the domain $\mathscr{D}_{\mathsf{PIds}} = \{pid_1, \ldots, pid_P\}$ and Loc is an ordered primitive type with the domain $\mathscr{D}_{\mathsf{Loc}} = \{N, T, C\}$. The transition relation is induced by the rules in the figure and the invariant "Mutual exclusion" states the correctness property to be checked by Murφ. ♣

The symmetries of systems are produced by the corresponding group Θ of allowed domain permutations also described in Section 6.1. An allowed domain permutation $\theta = \{\theta^T\}_{T \in \mathcal{T}}$ acts on the states by permuting the values assigned to the state variables, i.e., $\theta(\mathfrak{s}) : x \mapsto \theta^{T_x}(\mathfrak{s}(x))$ for each state $\mathfrak{s}$. Under a subgroup Θ' of Θ, two states, $\mathfrak{s}$ and $\mathfrak{s}'$, are Θ'-*equivalent* if there is an allowed domain permutation $\theta \in \Theta'$ such that $\theta(\mathfrak{s}) = \mathfrak{s}'$. Under the group Θ, one can simply say that Θ-equivalent states are *equivalent*. The fact that the allowed domain permutations produce state space symmetries, i.e., that the state space symmetry equation

$$\mathfrak{s} \longrightarrow \mathfrak{s}' \Leftrightarrow \theta(\mathfrak{s}) \longrightarrow \theta(\mathfrak{s}')$$

holds for each $\theta \in \Theta$, is ensured by the restrictions on the allowed data manipulation operations applied in the above mentioned formalisms (see the previous chapter).

7.1.1 Stabilizers and Storing Subgroups

The concept of stabilizers is now briefly recalled. An allowed domain permutation $\theta = \{\theta^T\}_{T \in \mathcal{T}}$ *fixes* (or *stabilizes*) an element $v \in \mathscr{D}_T$ of a type T if $\theta^T(v) = v$. The *stabilizer (sub)group* of v in a subgroup Θ' of Θ is

$$\mathrm{Stab}(\Theta', v) = \{\theta \mid \theta \in \Theta' \text{ and } \theta \text{ is a stabilizer of } v\}\,.$$

```
const
  P: 2;

type
  PIds: scalarset(P);
  Loc: enum{N,T,C};
var
  s: Array[PIds] of Loc;

Startstate
  Begin
    for i:PIds do s[i] := N; end;
  End;

Ruleset i:PIds do
  Rule "t"
    s[i] = N ==> s[i] := T;
  EndRule;
  Rule "e"
    s[i] = T & forall j:PIds do s[j] != C end ==> s[i] := C;
  EndRule;
  Rule "l"
    s[i] = C ==> s[i] := N;
  EndRule;
EndRuleset;

Invariant "Mutual exclusion"
  forall i:PIds Do s[i] = C ->
    forall j:PIds Do
      i != j -> s[j] != C
    End
  End
```

Figure 7.2: A Murφ version of the mutual exclusion program in Example 2.1

Similarly for states, θ is a *stabilizer* of a state $\mathfrak{s}$ if $\theta(\mathfrak{s}) = \mathfrak{s}$. Clearly this is equivalent to the requirement that $\theta^{T_x}(\mathfrak{s}(x)) = \mathfrak{s}(x)$ for each state variable $x \in \mathcal{X}$. Given a subgroup Θ' of Θ, the *stabilizer group of a state $\mathfrak{s}$ in Θ'* is

$$\mathrm{Stab}(\Theta', \mathfrak{s}) = \{\theta \in \Theta' \mid \theta(\mathfrak{s}) = \mathfrak{s}\}\,.$$

Obviously, $\mathrm{Stab}(\Theta', \mathfrak{s}) = \bigcap_{x \in \mathcal{X}} \mathrm{Stab}(\Theta', \mathfrak{s}(x))$. Stabilizers can also be calculated iteratively: assuming that the state variables are $x_1, \ldots, x_n$, let $\Theta_1 = \mathrm{Stab}(\Theta', \mathfrak{s}(x_1))$, $\Theta_2 = \mathrm{Stab}(\Theta_1, \mathfrak{s}(x_2))$,..., and $\Theta_n = \mathrm{Stab}(\Theta_{n-1}, \mathfrak{s}(x_n))$. Now $\Theta_n = \mathrm{Stab}(\Theta', \mathfrak{s})$. The group $\mathrm{Stab}(\Theta, \mathfrak{s})$ is simply called the *stabilizer group of the state $\mathfrak{s}$*. The following theorem relates the stabilizer groups of equivalent states.

Theorem 7.3 *Assume that an allowed domain permutation $\theta \in \Theta$ maps a state $\mathfrak{s}_1$ to $\mathfrak{s}_2$, i.e., $\theta(\mathfrak{s}_1) = \mathfrak{s}_2$. Then*

1. $\mathrm{Stab}(\Theta, \mathfrak{s}_2) = \theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) * \theta^{-1}$, *where* $\theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) * \theta^{-1} = \{\theta * \theta' * \theta^{-1} \mid \theta' \in \mathrm{Stab}(\Theta, \mathfrak{s}_1)\}$, *and*

2. *the left coset* $\theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) = \{\theta * \theta' \mid \theta' \in \mathrm{Stab}(\Theta, \mathfrak{s}_1)\}$ *is the set of all allowed domain permutations mapping* $\mathfrak{s}_1$ *to* $\mathfrak{s}_2$.

As a direct consequence, (i) $|\mathrm{Stab}(\Theta, \mathfrak{s}_1)| = |\mathrm{Stab}(\Theta, \mathfrak{s}_2)|$*, (ii) there are* $|\mathrm{Stab}(\Theta, \mathfrak{s}_1)|$ *allowed domain permutations mapping* $\mathfrak{s}_1$ *to* $\mathfrak{s}_2$*, and (iii) there are* $|\Theta|/|\mathrm{Stab}(\Theta, \mathfrak{s}_1)|$ *states that are equivalent to* $\mathfrak{s}_1$.

Proof. Part 1. For each $\theta' \in \mathrm{Stab}(\Theta, \mathfrak{s}_1)$, $(\theta * \theta' * \theta^{-1})(\mathfrak{s}_2) = (\theta * \theta')(\mathfrak{s}_1) = \theta(\mathfrak{s}_1) = \mathfrak{s}_2$ and thus $\theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) * \theta^{-1} \subseteq \mathrm{Stab}(\Theta, \mathfrak{s}_2)$. For each $\theta'' \in \mathrm{Stab}(\Theta, \mathfrak{s}_2)$, $\theta'' = \theta * \theta^{-1} * \theta'' * \theta * \theta^{-1} \in \theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) * \theta^{-1}$ because $\theta^{-1} * \theta'' * \theta \in \mathrm{Stab}(\Theta, \mathfrak{s}_1)$ and thus $\mathrm{Stab}(\Theta, \mathfrak{s}_2) \subseteq \theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1) * \theta^{-1}$.

Part 2. For each $\theta * \theta' \in \theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1)$, $(\theta * \theta')(\mathfrak{s}_1) = \theta(\mathfrak{s}_1) = \mathfrak{s}_2$. On the other hand, if $\theta''(\mathfrak{s}_1) = \mathfrak{s}_2$, then $(\theta^{-1} * \theta'')(\mathfrak{s}_1) = \theta^{-1}(\mathfrak{s}_2) = \mathfrak{s}_1$ implies that $\theta^{-1} * \theta'' \in \mathrm{Stab}(\Theta, \mathfrak{s}_1)$ and $\theta * (\theta^{-1} * \theta'') = \theta''$ belongs to the left coset $\theta * \mathrm{Stab}(\Theta, \mathfrak{s}_1)$. □

Example 7.4 Recall the EWF-net in Figure 7.1 and Example 7.1. Consider the initial state

$$\mathfrak{s}_0 = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}.$$

The stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s}_0)$ has two members:

$$\theta_1 = \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right)$$

and

$$\theta_2 = \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).$$

♣

Note that although the group of allowed domain permutations Θ can be very large, there is no need to represent it explicitly — it is implicitly represented by the knowledge of which primitive types are cyclic or unordered. However, it is not so easy to represent a subgroup of Θ, for instance the stabilizer group of a state. Fortunately, there are efficient data structures for representation of permutation groups, for instance the Schreier-Sims representation discussed in Section 4.1.1. In order to use those data structures, one only has to rename the domains of permutable primitive types to be mutually disjoint. Now any domain permutation (group) can be represented by a permutation (group) on the set $\bigcup_{T \in \mathcal{T}_P} \mathscr{D}_T$.

7.2 VALUE TREES AND CHARACTERISTIC GRAPHS

Before proceeding to the algorithms for the orbit problems, some new concepts have to be defined.

An element of a complex structured type can be easily illustrated by its "parse tree" that is here called a value tree. Formally, for a type T and an element $v \in \mathscr{D}_T$, the *value tree* $\mathcal{VT}(T, v)$ is an edge weighted tree that has the node $T::v$ as its root. The children of the root node are defined inductively as follows.

– For a primitive type T, the root node $T::v$ has no children.

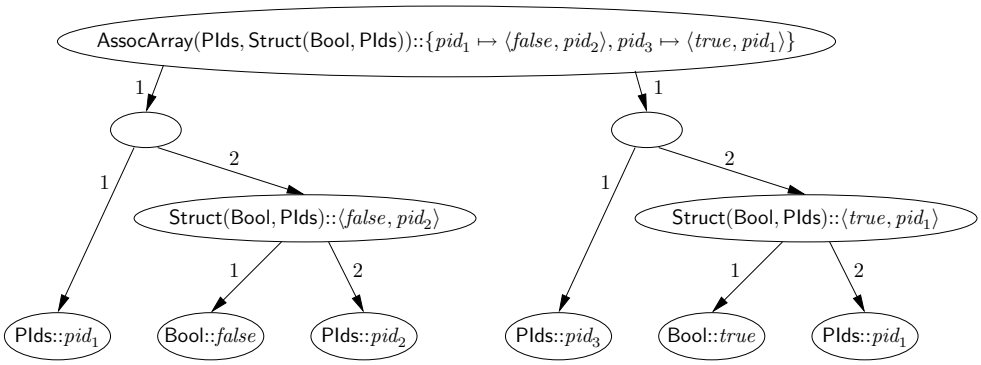


Figure 7.3: A value tree

- A root node $\mathsf{List}(T)::\langle v_1, \ldots, v_n\rangle$, has as its children the value trees $\mathcal{VT}(T, v_i)$, $1 \leq i \leq n$, the edge to each $\mathcal{VT}(T, v_i)$ having weight i.
- A root node $\mathsf{Struct}(T_1, \ldots, T_n)::\langle v_1, \ldots, v_n\rangle$, has as its children the value trees $\mathcal{VT}(T_i, v_i)$, $1 \leq i \leq n$, the edge to each $\mathcal{VT}(T_i, v_i)$ having weight i.
- A root node $\mathsf{Set}(T)::V$ has as its children the value trees $\mathcal{VT}(T, v)$ for each $v \in V$, the edge to each such $\mathcal{VT}(T, v)$ having weight 1.
- A root node $\mathsf{Multiset}(T)::m$ has as its children the trees $\mathcal{VT}(T, v)$ for each $v \in \mathscr{D}_T$ with $m(v) \geq 1$, the edge to each such $\mathcal{VT}(T, v)$ having weight $m(v)$.
- A root node $\mathsf{AssocArray}(T_1, T_2)::a$ has, for each $\langle v_1, v_2\rangle \in a$, the following tree as its child with the edge to it having weight 1. The child tree consists of an anonymous root node with two children: the value tree $\mathcal{VT}(T_1, v_1)$ with the edge to it having weight 1 and the value tree $\mathcal{VT}(T_2, v_2)$ with the edge to it having weight 2.
- A root node $\mathsf{Union}(T_1, \ldots, T_n)::\langle T_i, v_i\rangle$ has the value tree $\mathcal{VT}(T_i, v_i)$ as its only child, the edge to it having weight 1.

Example 7.5 Figure 7.3 shows the value tree for the element

$$\{pid_1 \mapsto \langle false, pid_2\rangle, pid_3 \mapsto \langle true, pid_1\rangle\}$$

of type $\mathsf{AssocArray}(\mathsf{PIds}, \mathsf{Struct}(\mathsf{Bool}, \mathsf{PIds}))$, where PIds is a primitive type with the domain $\mathscr{D}_{\mathsf{PIds}} = \{pid_1, pid_2, pid_3, pid_4\}$ and Bool is a primitive type with the domain $\mathscr{D}_{\mathsf{Bool}} = \{false, true\}$. ♣

It is straightforward to see that value trees have the following property:

Fact 7.6 *If there is a path* $T::v \xrightarrow{w_1} n_1 \xrightarrow{w_2} n_2 \cdots n_k \xrightarrow{w_{k+1}} T'::v'$ *from the root node* $T::v$ *to a leaf node* $T'::v'$ *in a value tree* $\mathcal{VT}(T, v)$*, then for each allowed domain permutation* θ*, there is a path* $T::\theta^T(v) \xrightarrow{w_1} \theta(n_1) \xrightarrow{w_2} \theta(n_2) \cdots \theta(n_k) \xrightarrow{w_{k+1}} T'::\theta^{T'}(v')$ *from the root node* $T::\theta^T(v)$ *to a leaf node* $T'::\theta^{T'}(v')$ *in the value tree* $\mathcal{VT}(T, \theta^T(v))$ *(where* $\theta(n_i)$ *is an anonymous node if* n_i *is, and* $T_i::\theta^{T_i}(v_i)$ *if* $n_i = T_i::v_i$*).*

Assume a state variable x of type T. The *value tree of* x *in a state* $\mathfrak{s}$ consists of the root node x that has the value tree $\mathcal{VT}(T, \mathfrak{s}(x))$ as its only child, the edge to it having weight 1. By combining the values trees of all state

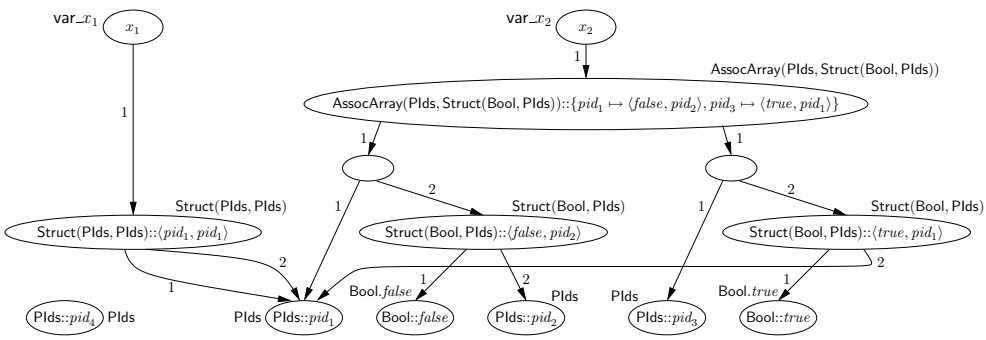


Figure 7.4: A characteristic graph

variables in a state, the characteristic graph for the state can be constructed (cf. Section 4.2).

Definition 7.7 *The* characteristic graph *of a state $\mathfrak{s}$ is the vertex labeled and edge weighted directed graph $\mathcal{G}_\mathfrak{s}$ obtained as follows.*

1. *Take the disjoint union of the value trees of each state variable x in the state $\mathfrak{s}$.*
2. *For each primitive type T and each element $v \in \mathscr{D}_T$, merge all the nodes $T{::}v$ into one node.*
3. *For each permutable primitive type T, if there is no node $T{::}v$ for an element $v \in \mathscr{D}_T$, include it in the graph.*
4. *For each cyclic primitive type T, add a directed edge of weight 1 from each node $T{::}v$ to its successor node $T{::}succ_T(v)$*
5. *Label nodes as follows:*
   (a) *Each node $T{::}v$ for a permutable primitive type T is labeled with T.*
   (b) *Each node $T{::}v$ for an ordered primitive type T is labeled with $T.v$.*
   (c) *Each node $T{::}v$ for a non-primitive type T is labeled with T.*
   (d) *Each node x corresponding to a state variable x is labeled with* var_x.

Example 7.8 Recall the previous example and assume that Bool is an ordered primitive type and PIds is an unordered primitive type. Figure 7.4 now shows the characteristic graph of a state $\mathfrak{s}$ over two state variables: (i) x_1 of type Struct(PIds, PIds) having the value $\mathfrak{s}(x_1) = \langle pid_1, pid_1\rangle$, and (ii) x_2 of type AssocArray(PIds, Struct(Bool, PIds)) having the value $\mathfrak{s}(x_2) = \{pid_1 \mapsto \langle false, pid_2\rangle, pid_3 \mapsto \langle true, pid_1\rangle\}$. Note especially that there are two edges from the node Struct(PIds, PIds)::$\langle pid_1, pid_1\rangle$ to the node PIds::pid_1, and that there is an isolated node PIds::pid_4. ♣

Since isomorphisms between two vertex labeled, edge weighted directed graphs have to preserve node labels and edge weights, it is quite straightforward to see that characteristics graphs have the following properties:

Fact 7.9 *For each allowed domain permutation θ, there is an isomorphism γ from the characteristic graph $\mathcal{G}_\mathfrak{s}$ of a state $\mathfrak{s}$ to the characteristic graph $\mathcal{G}_{\theta(\mathfrak{s})}$ of the state $\theta(\mathfrak{s})$ such that for each permutable primitive type T and for each element $v \in \mathscr{D}_T$, $\theta^T(v) = v' \Leftrightarrow \gamma(T{::}v) = T{::}v'$.*

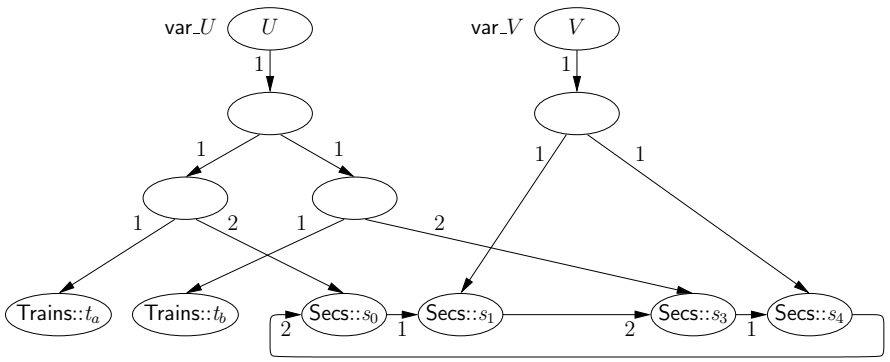


Figure 7.5: A modified characteristic graph

Fact 7.10 *If there is an isomorphism γ from the characteristic graph $\mathcal{G}_{\mathfrak{s}}$ of a state $\mathfrak{s}$ to the characteristic graph $\mathcal{G}_{\mathfrak{s}'}$ of a state $\mathfrak{s}'$, then there is a unique allowed domain permutation θ mapping $\mathfrak{s}$ to $\mathfrak{s}'$ such that for each permutable primitive type T and for each $v \in \mathscr{D}_T$, $\gamma(T{::}v) = T{::}v' \Leftrightarrow \theta^T(v) = v'$.*

From these two facts it follows directly that *the characteristic graphs of two states are isomorphic if and only if the states are equivalent.* Furthermore, the stabilizer subgroup of a state in Θ can be easily extracted from the automorphism group of the characteristic graph for the state. Now consider the following problem.

Problem 7.11 STATE EQUIVALENCE. *Given two states, are they equivalent?*

The proof of Theorem 4 in [Ip and Dill 1996] shows that the STATE EQUIVALENCE problem is at least as hard as the GRAPH ISOMORPHISM problem. With the following modifications to characteristic graphs, it can be shown that the STATE EQUIVALENCE problem is actually *as hard as* the GRAPH ISOMORPHISM problem. These modifications remove the "superfluous" nodes corresponding to the elements of permutable primitive types not appearing in a state. First, remove item 3 in Definition 7.7 and replace item 4 with

4 For each cyclic primitive type T, and for each node $T{::}v$ in the graph do the following. If $k > 0$ is the smallest integer such that $T{::}succ_T^k(v)$ is a node in the graph, add a directed edge of weight k from $T{::}v$ to $T{::}succ_T^k(v)$.

It is easy to see that the modified characteristic graphs of two states are isomorphic if and only if the states are equivalent. Furthermore, the size of the modified characteristic graph is linear in the size of the state (when states are coded in a standard, uncompressed way, used in the examples for instance).

Corollary 7.12 STATE EQUIVALENCE *is polynomial time many-one equivalent to* GRAPH ISOMORPHISM.

Example 7.13 Recall the EWF-net in Figure 7.1. Figure 7.5 shows the modified characteristic graph for the state $\{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$. (For the sake of simplicity, some node names and labels that should be obvious are omitted). ♣

7.3 A BASIC PARTITION BASED ALGORITHM

A basic partition based representative state algorithm is now presented. The idea is that, given a state for which a representative is to be computed,

1. first assign the state a partitioning of the domains of the permutable primitive types in a symmetry-respecting way,
2. then select an allowed domain permutation that is "compatible" with the partitioning, and
3. return the state permuted with the selected domain permutation as the representative.

This process is very similar to the one presented in Section 4.4 for place/transition nets. However, the technical details are different because states are now much more complex data structures than the markings of place/transition nets. For instance, the invariants used in building the partitions are quite different. For the sake of simplicity, in the rest of the chapter it is assumed that the applied symmetry group is the group Θ of all allowed domain permutations. If a subgroup of Θ were considered instead, the definitions given in this and following sections (e.g., the compatibility definition between symmetries and partitions) would have to be reformulated to resemble those used in Chapter 4.

First, recall the definitions and notations for ordered partitions described in Section 2.3. An *ordered permutable primitive type partition* is a family $\mathfrak{p} = \{\mathfrak{p}^T\}_{T \in \mathcal{T}_P}$, where each $\mathfrak{p}^T$ is an ordered partition of the domain $\mathscr{D}_T$. The set of *all ordered permutable primitive type partitions* is denoted by $\mathfrak{P}$. The definitions for ordered partitions are naturally extended to ordered permutable primitive type partitions. That is, $\mathfrak{p}$ is *discrete* (*unit*) if all its constituent partitions are discrete (unit). Similarly, a domain permutation $\psi = \{\psi^T\}_{T \in \mathcal{T}_P}$ acts on a partition $\mathfrak{p} = \{\mathfrak{p}^T\}_{T \in \mathcal{T}_P}$ by $\psi(\mathfrak{p}) = \{\psi^T(\mathfrak{p}^T)\}_{T \in \mathcal{T}_P}$ and $\{\mathfrak{p}_1^T\}_{T \in \mathcal{T}_P} \preceq \{\mathfrak{p}_2^T\}_{T \in \mathcal{T}_P}$ if $\mathfrak{p}_1^T \preceq \mathfrak{p}_2^T$ for each $T \in \mathcal{T}_P$. As only ordered partitions will be used in the following, the prefix "ordered" is usually omitted and one simply speaks of partitions. For convenience, the prefix "permutable primitive type" may also be omitted whenever no confusion can arise.

A state $\mathfrak{s}$ is associated with a partition by using a function that respects the group of allowed domain permutations (i.e., symmetries of the system).

Definition 7.14 *A function $pg : \mathcal{S} \to \mathfrak{P}$ that maps each state to a permutable primitive type partition is a* partition generator *if*

$$pg(\theta(\mathfrak{s})) = \theta(pg(\mathfrak{s}))$$

holds for all allowed domain permutations $\theta \in \Theta$ and for all states $\mathfrak{s} \in \mathcal{S}$.

That is, for permuted states the partition assigned by *pg* should be similarly permuted. A way to produce such functions will be developed in Section 7.3.1.

The compatibility condition between allowed domain permutations and partitions is given by the following definition. As the group Θ of all allowed domain permutations is considered, the definition is much simpler than the corresponding Definition 4.2 for place/transition nets.

Definition 7.15 *An allowed domain permutation* $\left\{\theta^T\right\}_{T\in\mathcal{T}_P}$ *is* compatible *with a partition* $\left\{\mathfrak{p}^T\right\}_{T\in\mathcal{T}_P}$ *if*

- *For each cyclic primitive type* T *with* $\mathscr{D}_T = \{v_0, \ldots, v_{n-1}\}$ *and* $\mathfrak{p}^T = [C_1^T, \ldots, C_m^T]$, θ^T *is such that it maps an element* $v \in C_1^T$ *to* v_0.
- *For each unordered primitive type* T *with* $\mathscr{D}_T = \{v_1, \ldots, v_n\}$ *and* $\mathfrak{p}^T = [C_1^T, \ldots, C_m^T]$, θ^T *must fulfill the following: if* $incell(\mathfrak{p}^T, v_i) < incell(\mathfrak{p}^T, v_j)$, *then for the permuted elements* $v_{i'} = \theta^T(v_i)$ *and* $v_{j'} = \theta^T(v_j)$ *it holds that* $i' < j'$. *That is, the* n_1 *elements in the first cell* C_1^T *are mapped to* $v_1, \ldots, v_{n_1}$, *the* n_2 *elements in the second cell* C_2^T *are mapped to* $v_{n_1+1}, \ldots, v_{n_1+n_2}$, *and so on.*

Obviously, for each partition there is at least one allowed domain permutation compatible with it.

The following lemma and theorem provide results corresponding to Theorems 4.3 and 4.13, respectively, for place/transition nets.

Lemma 7.16 *For each allowed domain permutation* θ *it holds that an allowed domain permutation* $\hat{\theta}$ *is compatible with a partition* $\mathfrak{p}$ *if and only if the allowed domain permutation* $\hat{\theta} * \theta^{-1}$ *is compatible with the partition* $\theta(\mathfrak{p})$.

Proof. It suffices to prove the "only if" direction because the compatibility of $\hat{\theta} * \theta^{-1}$ with $\theta(\mathfrak{p})$ then implies the compatibility of $(\hat{\theta} * \theta^{-1}) * (\theta^{-1})^{-1} = \hat{\theta}$ with $\theta^{-1}(\theta(\mathfrak{p})) = \mathfrak{p}$.

Let $\theta = \left\{\theta^T\right\}_{T\in\mathcal{T}_P}$, $\hat{\theta} = \left\{\hat{\theta}^T\right\}_{T\in\mathcal{T}_P}$, and $\mathfrak{p} = \left\{\mathfrak{p}^T\right\}_{T\in\mathcal{T}_P}$ such that $\mathfrak{p}^T = [C_1^T, \ldots, C_{c_T}^T]$.

For a cyclic primitive type T, assume that $\hat{\theta}^T$ maps a $v_i \in C_1^T$ to v_0, i.e., $\hat{\theta}^T(v_i) = v_0$. Observe that $\hat{\theta}^T = \hat{\theta}^T \circ \theta^{T^{-1}} \circ \theta^T$ and therefore $(\hat{\theta}^T \circ \theta^{T^{-1}})(\theta^T(v_i)) = v_0$. But now $\theta^T(v_i)$ is in the first cell $\theta^T(C_1^T)$ for the type T in the partition $\theta(\mathfrak{p})$ and thus $\hat{\theta} * \theta^{-1}$ fulfills the compatibility requirement w.r.t. $\theta(\mathfrak{p})$ for the type T.

For an unordered primitive type T, assume that for $v_i, v_j \in \mathscr{D}_T$ it holds that $incell(\theta^T(\mathfrak{p}^T), v_i) < incell(\theta^T(\mathfrak{p}^T), v_j)$. Thus $incell(\mathfrak{p}^T, \theta^{T^{-1}}(v_i)) < incell(\mathfrak{p}^T, \theta^{T^{-1}}(v_j))$ holds, too. As $\hat{\theta}$ is compatible with $\mathfrak{p}$, $\hat{\theta}^T(\theta^{T^{-1}}(v_i)) = v_{i'} = (\hat{\theta}^T \circ \theta^{T^{-1}})(v_i)$ and $\hat{\theta}^T(\theta^{T^{-1}}(v_j)) = v_{j'} = (\hat{\theta}^T \circ \theta^{T^{-1}})(v_j)$ such that $i' < j'$, and thus $\hat{\theta} * \theta^{-1}$ fulfills the compatibility requirement w.r.t. $\theta(\mathfrak{p})$ for the type T. □

Next, assuming a fixed partition generator *pg*, define

$$posreps(\mathfrak{s}) = \left\{\hat{\theta}(\mathfrak{s}) \mid \hat{\theta} \text{ is compatible with } pg(\mathfrak{s})\right\}$$

to denote the *set of possible representative states for a state* $\mathfrak{s}$. For two equivalent states, the sets of possible representative states are the same.

Theorem 7.17 *For each state* $\mathfrak{s}$ *and for each allowed domain permutation* θ, $posreps(\mathfrak{s}) = posreps(\theta(\mathfrak{s}))$.

Proof. By Lemma 7.16, $\hat{\theta}$ is compatible with $pg(\mathfrak{s})$ if and only if $\hat{\theta} * \theta^{-1}$ is compatible with $\theta(pg(\mathfrak{s})) = pg(\theta(\mathfrak{s}))$. Thus $\hat{\theta}(\mathfrak{s}) \in posreps(\mathfrak{s})$ if and only if $(\hat{\theta} * \theta^{-1})(\theta(\mathfrak{s})) = \hat{\theta}(\mathfrak{s}) \in posreps(\theta(\mathfrak{s}))$. □

Example 7.18 Consider the state $\mathfrak{s} = \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$ for the railroad system net in Figure 7.1 (recall Example 7.1). Assume a partition generator *pg* that produces the partition

$$pg(\mathfrak{s}) = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},4} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},4} = [\{t_a, t_b\}]\right).$$

for $\mathfrak{s}$. Having the fixed ordering $s_0 < s_1 < \cdots < s_5$ between the railroad sections and $t_a < t_b$ between the train identities, the four possible allowed domain permutations compatible with the partition are

$$\begin{array}{rcl}
\theta_1 & = & \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \\
\theta_2 & = & \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \\
\theta_3 & = & \left(\theta_3^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_3^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\
\theta_4 & = & \left(\theta_4^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_4^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).
\end{array}$$

The corresponding possible representative states for $\mathfrak{s}$ are:

$$\begin{array}{rcl}
\theta_1(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \mathfrak{s}, \\
\theta_2(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\}, \\
\theta_3(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_5\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_3\}, \text{ and} \\
\theta_4(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_5\rangle, V \mapsto s_2 + s_3\}.
\end{array}$$

Now consider the state $\mathfrak{s}' = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_4\rangle, V \mapsto s_1 + s_2\}$ obtained from $\mathfrak{s}$ by rotating the railroad sections 3 steps and swapping the train identities, i.e., by applying

$$\theta = \left(\theta^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).$$

Since $\mathfrak{s}' = \theta(\mathfrak{s})$, the partition generator *pg* must assign the partition $\theta(pg(\mathfrak{s}))$ to $\mathfrak{s}'$, i.e.,

$$pg(\mathfrak{s}') = \theta(pg(\mathfrak{s})) = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s}',4} = [\{s_3, s_5\}, \{s_1, s_2\}, \{s_0, s_4\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s}',4} = [\{t_a, t_b\}]\right).$$

The four possible allowed domain permutations compatible with the partition are

$$\begin{array}{rcl}
\theta_{1'} & = & \left(\theta_{1'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta_{1'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_2 * \theta^{-1}, \\
\theta_{2'} & = & \left(\theta_{2'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta_{2'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_1 * \theta^{-1}, \\
\theta_{3'} & = & \left(\theta_{3'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{3'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_4 * \theta^{-1}, \text{ and} \\
\theta_{4'} & = & \left(\theta_{4'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{4'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_3 * \theta^{-1}.
\end{array}$$

The corresponding possible representative states for $\mathfrak{s}'$ are:

$$\begin{array}{rcl}
\theta_{1'}(\mathfrak{s}') & = & \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\} = \theta_2(\mathfrak{s}), \\
\theta_{2'}(\mathfrak{s}') & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \theta_1(\mathfrak{s}), \\
\theta_{3'}(\mathfrak{s}') & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_5\rangle, V \mapsto s_2 + s_3\} = \theta_4(\mathfrak{s}), \text{ and} \\
\theta_{4'}(\mathfrak{s}') & = & \{U \mapsto \langle t_a, s_5\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_3\} = \theta_3(\mathfrak{s}).
\end{array}$$

Thus the sets of possible representative states for $\mathfrak{s}$ and $\mathfrak{s}'$ are the same as expected because of Lemma 7.16, Theorem 7.17, and the fact that the states $\mathfrak{s}$ and $\mathfrak{s}'$ are equivalent. ♣

The partition generators have the same basic limitations as the G-partition generators for place/transition nets, recall Fact 4.14.

Fact 7.19 *Let $\theta = \{\theta^T\}_{T \in \mathcal{T}_P}$ be a stabilizer of a state $\mathfrak{s}$. Then $pg(\theta(\mathfrak{s})) = \theta(pg(\mathfrak{s}))$ implies $pg(\mathfrak{s}) = \theta(pg(\mathfrak{s}))$ for any partition generator pg. Thus each stabilizer of $\mathfrak{s}$ respects the cells in $pg(\mathfrak{s})$, meaning that if $v \in \mathcal{D}_T$ belongs to the cell C_i^T in the partition $pg(\mathfrak{s})$, then $\theta^T(v)$ belongs to the cell C_i^T, too.*

Optimal partition generator functions, i.e., functions that produce minimal partitions whose cells are as small as possible, are probably not, in general, computable in polynomial time. For if such functions could always be computed efficiently, one would know by the fact above whether the group $\mathrm{Stab}(\Theta, \mathfrak{s})$ is non-trivial (has other elements besides the identity): if a partition has a cell with more than one element for some primitive type, then $\mathrm{Stab}(\Theta, \mathfrak{s})$ is non-trivial. Combined with the construction in the proof of Theorem 3.4 in [Ip 1996], the non-triviality of $\mathrm{Stab}(\Theta, \mathfrak{s})$ would reveal that a graph has non-trivial automorphisms. For this task no polynomial-time algorithms are currently known.

Assuming a fixed partition generator *pg* and a total order on the set $\mathcal{S}$ of states, a canonical representative for a state $\mathfrak{s}$ can be obtained by the following procedure.

1. Build the partition $pg(\mathfrak{s})$.
2. Search through all the allowed domain permutations that are compatible with $pg(\mathfrak{s})$. Let θ be such an allowed domain permutation that results in the smallest state $\theta(\mathfrak{s})$.
3. Return $\theta(\mathfrak{s})$ as the representative.

That is, the smallest state in $posreps(\mathfrak{s})$ is returned as the canonical representative state (the canonicity is ensured by Theorem 7.17). Searching through all the allowed domain permutations compatible with the partition $pg(\mathfrak{s})$ is quite straightforward to do systematically. As in Section 4.3, the search could be pruned by the stabilizers of the state and by the knowledge of which elements are fixed at some point in the search. However, since the states are more complex structures than the markings of place/transition nets, especially the second pruning technique is not so easily implementable and also depends on the applied total order between the states. Because of this, it is much simpler in practice to generate only a possibly non-canonical representative for a state $\mathfrak{s}$ by just selecting an arbitrary allowed domain permutation θ that is compatible with the partition $pg(\mathfrak{s})$ and returning the state $\theta(\mathfrak{s})$ as the representative. This procedure is described in Algorithm 7.1. Note that for some states, even this simple procedure is enough to generate canonical representatives as discussed in Section 7.4.3.

7.3.1 Partition Refiners and Invariants

It is now shown how partition generators defined above can be built in a way similar to that used in Section 4.4.2. First, partition refiners are introduced.

Algorithm 7.1 A representative algorithm based on partitions

Input: A state $\mathfrak{s}$
Output: A representative state that is equivalent to $\mathfrak{s}$
Require: A partition generator *pg*
1: Compute the partition $\mathfrak{p} = pg(\mathfrak{s})$
2: Choose any allowed domain permutation θ that is compatible with $\mathfrak{p}$
3: Return $\theta(\mathfrak{s})$ as the representative state

They are functions that, given a state and a partition, return a cell order preserving refinement of the partition in a symmetry-respecting way.

Definition 7.20 *A* partition refiner *is a function $\mathcal{R} : \mathcal{S} \times \mathfrak{P} \to \mathfrak{P}$ such that for all states $\mathfrak{s} \in \mathcal{S}$ and for all partitions $\mathfrak{p} \in \mathfrak{P}$ it holds that (i) $\mathcal{R}(\mathfrak{s}, \mathfrak{p}) \preceq \mathfrak{p}$ and (ii) $\theta(\mathcal{R}(\mathfrak{s}, \mathfrak{p})) = \mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$ for all allowed domain permutations θ.*

Again, the composition $\mathcal{R}_2 \star \mathcal{R}_1$ of two partition refiners $\mathcal{R}_1$ and $\mathcal{R}_2$, defined by $(\mathcal{R}_2 \star \mathcal{R}_1)(\mathfrak{s}, \mathfrak{p}) = \mathcal{R}_2(\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p}))$, is a partition refiner (see Lemma 4.17). Thus any finite sequence $\mathcal{R}_n \star \mathcal{R}_{n-1} \star \cdots \star \mathcal{R}_1$ of partition refiners, defined by $\mathcal{R}_n(\mathfrak{s}, \mathcal{R}_{n-1}(\mathfrak{s}, \cdots (\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p})) \ldots))$, is also a partition refiner. When a partition refiner is applied to the unit partition, the result is a partition generator.

Theorem 7.21 *For a partition refiner $\mathcal{R}$, the function $pg_{\mathcal{R}}(\mathfrak{s}) = \mathcal{R}(\mathfrak{s}, \mathfrak{p}_0)$, where $\mathfrak{p}_0 = \left\{\mathfrak{p}_0^T = [\mathscr{D}_T]\right\}_{T \in \mathcal{T}_P}$ is the unit partition, is a partition generator.*

Proof. Observe that $\theta(\mathfrak{p}_0) = \mathfrak{p}_0$ for any allowed domain permutation θ. Thus $pg_{\mathcal{R}}(\theta(\mathfrak{s})) = \mathcal{R}(\theta(\mathfrak{s}), \mathfrak{p}_0) = \mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}_0)) = \theta(\mathcal{R}(\mathfrak{s}, \mathfrak{p}_0)) = \theta(pg_{\mathcal{R}}(\mathfrak{s}))$. □

Now the task of building partition generators is reduced to building partition refiners. This task is accomplished by using invariants. An invariant is a function that tries to distinguish between the elements of a permutable primitive type under a given state and partition. It must distinguish the elements in a way that respects the allowed domain permutations, i.e., under a permuted state and partition, the invariant should distinguish the similarly permuted elements.

Definition 7.22 *An* invariant *for a permutable primitive type T is a function I from the domain $\mathscr{D}_T \times \mathcal{S} \times \mathfrak{P}$ such that for all elements $v \in \mathscr{D}_T$, for all states $\mathfrak{s} \in \mathcal{S}$, for all partitions $\mathfrak{p} \in \mathfrak{P}$, and for all allowed domain permutations $\theta \in \Theta$, it holds that*

$$I(v, \mathfrak{s}, \mathfrak{p}) = I(\theta^T(v), \theta(\mathfrak{s}), \theta(\mathfrak{p})).$$

The codomain of I is assumed to be a set with a total order $<$.

An invariant I is *partition independent* if it does not depend on the partition argument, otherwise it is *partition dependent*. Invariants can also be defined for types instead of states:

Definition 7.23 *A* type invariant for a permutable primitive type T in a type T' *is a function I from the domain $\mathscr{D}_T \times \mathscr{D}_{T'} \times \mathfrak{P}$ such that for all elements*

$v \in \mathscr{D}_T$, for all elements $v' \in \mathscr{D}_{T'}$, for all partitions $\mathfrak{p} \in \mathfrak{P}$, and for all allowed domain permutations $\theta \in \Theta$, it holds that

$$I(v, v', \mathfrak{p}) = I(\theta^T(v), \theta^{T'}(v'), \theta(\mathfrak{p})).$$

Again, the codomain of I is assumed to be a set with a total order $<$.

Type invariants can be interpreted as invariants:

Lemma 7.24 *If I is a type invariant for a permutable primitive type T in a type T' and x is a state variable of type T', then $I_x(v, \mathfrak{s}, \mathfrak{p}) = I(v, \mathfrak{s}(x), \mathfrak{p})$ is an invariant for T.*

Proof. For all $\theta \in \Theta$, $I_x(\theta^T(v), \theta(\mathfrak{s}), \theta(\mathfrak{p})) = I(\theta^T(v), (\theta(\mathfrak{s}))(x), \theta(\mathfrak{p})) = I(\theta^T(v), \theta^{T'}(\mathfrak{s}(x)), \theta(\mathfrak{p})) = I(v, \mathfrak{s}(x), \mathfrak{p}) = I_x(v, \mathfrak{s}(x), \mathfrak{p})$. □

Example 7.25 For each primitive type T and for each type T', the function

$$\sharp_T^{T'} : \mathscr{D}_T \times \mathscr{D}_{T'} \to \mathbb{N} \cup \{\infty\},$$

read "the element v of type T appears $\sharp_T^{T'}(v, v')$ times in the element v' of type T'", is defined by the following rules:

1. If T' is primitive type, then

$$\sharp_T^{T'}(v, v') = \begin{cases} 1 & \text{if } T = T' \text{ and } v = v' \\ 0 & \text{otherwise.} \end{cases}$$

2. $\sharp_T^{\mathsf{List}(T')}(v, \langle v'_1, \ldots, v'_n \rangle) = \sum_{1 \le i \le n} \sharp_T^{T'}(v, v'_i)$
3. $\sharp_T^{\mathsf{Struct}(T'_1, \ldots, T'_n)}(v, \langle v'_1, \ldots, v'_n \rangle) = \sum_{1 \le i \le n} \sharp_T^{T'_i}(v, v'_i)$
4. $\sharp_T^{\mathsf{Set}(T')}(v, V') = \sum_{v' \in V'} \sharp_T^{T'}(v, v')$
5. $\sharp_T^{\mathsf{Multiset}(T')}(v, m) = \sum_{v' \in \mathscr{D}_{T'}} m(v') \times \sharp_T^{T'}(v, v')$
6. $\sharp_T^{\mathsf{AssocArray}(T'_1, T'_2)}(v, a) = \sum_{\langle v'_1, v'_2 \rangle \in a} (\sharp_T^{T'_1}(v, v'_1) + \sharp_T^{T'_2}(v, v'_2))$
7. $\sharp_T^{\mathsf{Union}(T'_1, \ldots, T'_n)}(v, \langle T'_i, v' \rangle) = \sharp_T^{T'_i}(v, v')$

It is easy to see that $\sharp_T^{T'}(v, v') = \sharp_T^{T'}(\theta^T(v), \theta^{T'}(v'))$ for all allowed domain permutations θ. Now the function $I_{\sharp T \text{ in } T'}(v, v', \mathfrak{p}) = \sharp_T^{T'}(v, v')$ is a partition independent type invariant for T in T'. If x is a state variable of type T', then the corresponding invariant is $I_{\sharp T \text{ in } x}(v, \mathfrak{s}, \mathfrak{p}) = I_{\sharp T \text{ in } T'}(v, \mathfrak{s}(x), \mathfrak{p})$, i.e., the number of times v appears in the value of x in the state $\mathfrak{s}$. ♣

More invariants will be introduced later. Given an invariant for a permutable primitive type T and a partition $\mathfrak{p}$, the partition may be refined according to the invariant by splitting the cells of the partition for T so that each new cell contains all the elements in the original cell that are assigned to the same value by the invariant.

Definition 7.26 *Given an invariant I for a permutable primitive type T, define the function $\mathcal{R}_I : \mathcal{S} \times \mathfrak{P} \to \mathfrak{P}$ by $\mathcal{R}_I(\mathfrak{s}, \mathfrak{p}) = \mathfrak{p}_{\text{ref}}$, where*

1. *for any permutable primitive type $T' \neq T$, $\mathfrak{p}_{\text{ref}}^{T'} = \mathfrak{p}^{T'}$, and*
2. *the partition $\mathfrak{p}_{\text{ref}}^{T}$ is the one such that for all $v, v' \in \mathscr{D}_T$,*

(a) $incell(\mathfrak{p}^T_{\text{ref}}, v) = incell(\mathfrak{p}^T_{\text{ref}}, v')$ *if and only if* $incell(\mathfrak{p}^T, v) = incell(\mathfrak{p}^T, v')$ *and* $I(v, \mathfrak{s}, \mathfrak{p}) = I(v', \mathfrak{s}, \mathfrak{p})$, *and*
(b) $incell(\mathfrak{p}^T_{\text{ref}}, v) < incell(\mathfrak{p}^T_{\text{ref}}, v')$ *if and only if either*
  i. $incell(\mathfrak{p}^T, v) < incell(\mathfrak{p}^T, v')$, *or*
  ii. $incell(\mathfrak{p}^T, v) = incell(\mathfrak{p}^T, v')$ *and* $I(v, \mathfrak{s}, \mathfrak{p}) < I(v', \mathfrak{s}, \mathfrak{p})$.

Lemma 7.27 *The function $\mathcal{R}_I$ is a partition refiner.*

Proof. Similar to the proof of Lemma 4.20. □

When the partition refiner $\mathcal{R}_I$ is applied to a partition $\mathfrak{p}$ in a state $\mathfrak{s}$, i.e., partition $\mathfrak{p}$ is replaced by $\mathcal{R}_I(\mathfrak{s}, \mathfrak{p})$, $\mathfrak{p}$ is said to be *refined according to* I. Given a sequence $I_1.I_2.\ldots.I_n$ of invariants (for arbitrary primitive types), a partition $\mathfrak{p}$ is said to be *refined according to the sequence* to mean that the partition refiner sequence $\mathcal{R}_{I_n} \star \mathcal{R}_{I_{n-1}} \star \cdots \star \mathcal{R}_{I_1}$ is applied to it. To sum up, a partition generator can be obtained by

1. defining a sequence $I_1.I_2.\ldots.I_n$ of invariants, and
2. refining the unit partition according to the sequence (by Lemma 7.27 and Theorem 7.21).

Example 7.28 Consider the state $\mathfrak{s} = \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$ for the railroad system net in Figure 7.1 (cf. Example 7.18). Initially, the partition is the unit partition

$$\mathfrak{p}_{\mathfrak{s},0} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},0} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},0} = [\{t_a, t_b\}]\right).$$

The partition is now refined according to the sequence

$$I_{\sharp\mathsf{Trains\ in}\ U}.I_{\sharp\mathsf{Trains\ in}\ V}.I_{\sharp\mathsf{Secs\ in}\ U}.I_{\sharp\mathsf{Secs\ in}\ V}$$

of invariants described in Example 7.25. Refining the partition for Trains according to the invariant $I_{\sharp\mathsf{Trains\ in}\ U}$ leads to

$$\mathfrak{p}_{\mathfrak{s},1} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},1} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},1} = [\{t_a, t_b\}]\right),$$

i.e., does not change anything since both t_a and t_b appear once in the value of U. Similarly, refining the partition for Trains according to the invariant $I_{\sharp\mathsf{Trains\ in}\ V}$ changes nothing. Refining the partition for Secs according to the invariant $I_{\sharp\mathsf{Secs\ in}\ U}$ leads to

$$\mathfrak{p}_{\mathfrak{s},3} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},3} = [\{s_0, s_2, s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},3} = [\{t_a, t_b\}]\right),$$

distinguishing the railroad sections s_1 and s_3 from the others because they appear once in the value of U while the others do not. Further refining according to the invariant $I_{\sharp\mathsf{Secs\ in}\ V}$ gives

$$\mathfrak{p}_{\mathfrak{s},4} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},4} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},4} = [\{t_a, t_b\}]\right).$$

Applying the same sequence of invariants to the other state $\mathfrak{s}' = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_4\rangle, V \mapsto s_1 + s_2\}$ in Example 7.18 gives the partition

$$\mathfrak{p}_{\mathfrak{s}',4} = \theta(\mathfrak{p}_{\mathfrak{s},4}) = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s}',4} = [\{s_3, s_5\}, \{s_1, s_2\}, \{s_0, s_4\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s}',4} = [\{t_a, t_b\}]\right).$$

♣

7.3.2 Some Useful Invariants

A Successor Based Invariant for Cyclic Primitive Types
A partition *dependent* invariant is now introduced. Using this invariant, it is possible to exploit the partition produced so far to obtain further partition refinement. For a cyclic primitive type T, consider the function

$$I_{T,\text{succ}}(v, \mathfrak{s}, \mathfrak{p}) = incell(\mathfrak{p}^T, succ_T(v))$$

returning the cell number of the *successor* element of the element v in the partition $\mathfrak{p}$. That is, $I_{T,\text{succ}}$ distinguishes between two elements if their successors are already distinguished in the partition $\mathfrak{p}$.

Lemma 7.29 *The function $I_{T,\text{succ}}$ is an invariant.*

Proof. Assume that $succ_T(v)$ belongs to the ith cell in the partition $\mathfrak{p}^T$. Then for any allowed domain permutation $\theta = \{\theta^T\}_{T \in \mathcal{T}_P}$ in which $\theta^T = succ_T^k$ for a $1 \leq k \leq |\mathscr{D}_T|$, the element $succ_T(\theta^T(v)) = succ_T(succ_T^k(v)) = succ_T^k(succ_T(v)) = \theta^T(succ_T(v))$ belongs to the ith cell in the partition $\theta(\mathfrak{p}^T)$. □

Therefore, if the initial partition is already refined according to an invariant sequence, the resulting partition may be further refined by applying the invariant $I_{T,\text{succ}}$. The resulting partition may again be further refined by the same invariant until no refinement happens, i.e., until a fixed point is reached (in other words, the sequence of length $|\mathscr{D}_T|$ of invariant $I_{T,\text{succ}}$ is applied). Note that, while $I_{T,\text{succ}}$ is partition dependent, it does not depend on the state argument.

Example 7.30 Reconsider the state

$$\mathfrak{s} = \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$$

and the partition

$$\mathfrak{p}_{\mathfrak{s},4} = \left(\mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Secs}} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right)$$

for it given in Examples 7.18 and 7.28. Evaluating the invariant $I_{\mathsf{Secs},\text{succ}}$ in the partition gives

$$\begin{array}{ll} I_{\mathsf{Secs},\text{succ}}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 3, & I_{\mathsf{Secs},\text{succ}}(s_1, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 1, \\ I_{\mathsf{Secs},\text{succ}}(s_2, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 3, & I_{\mathsf{Secs},\text{succ}}(s_3, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 2, \\ I_{\mathsf{Secs},\text{succ}}(s_4, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 2, \text{ and} & I_{\mathsf{Secs},\text{succ}}(s_5, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 1. \end{array}$$

Refining according to this results in the partition

$$\mathfrak{p}_{\mathfrak{s},5} = \left(\mathfrak{p}_{\mathfrak{s},5}^{\mathsf{Secs}} = [\{s_0, s_2\}, \{s_5\}, \{s_4\}, \{s_1\}, \{s_3\}], \mathfrak{p}_{\mathfrak{s},5}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

Further evaluating the invariant $I_{\mathsf{Secs},\text{succ}}$ in this partition gives

$$\begin{array}{ll} I_{\mathsf{Secs},\text{succ}}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 4, & I_{\mathsf{Secs},\text{succ}}(s_1, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 1, \\ I_{\mathsf{Secs},\text{succ}}(s_2, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 5, & I_{\mathsf{Secs},\text{succ}}(s_3, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 3, \\ I_{\mathsf{Secs},\text{succ}}(s_4, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 2, \text{ and} & I_{\mathsf{Secs},\text{succ}}(s_5, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 1. \end{array}$$

Refining according to this results in the partition

$$\mathfrak{p}_{\mathfrak{s},6} = \left(\mathfrak{p}_{\mathfrak{s},6}^{\mathsf{Secs}} = [\{s_0\},\{s_2\},\{s_5\},\{s_4\},\{s_1\},\{s_3\}], \mathfrak{p}_{\mathfrak{s},6}^{\mathsf{Trains}} = [\{t_a,t_b\}]\right).$$

Now there are only two domain permutations compatible with the partition (compared to four in Example 7.18):

$$\begin{array}{rcl} \theta_1 & = & \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\ \theta_2 & = & \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right). \end{array}$$

The corresponding possible representative states for $\mathfrak{s}$ are:

$$\begin{array}{rcl} \theta_1(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \mathfrak{s}, \text{ and} \\ \theta_2(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\}. \end{array}$$

♣

Ordered Structured Types

Consider a structured type T' composed only of primitive types, lists, structures and unions. Now the value tree (recall Section 7.2) for any $v' \in \mathscr{D}_{T'}$ is ordered in the sense that the children of each node can be totally ordered by the edge labelings. Therefore, it is possible to uniquely number the nodes in the value tree, for instance in a depth-first manner. Now each element v of a primitive subtype T of T' that appears in the element v' can be associated with a unique number, e.g. the smallest number of those nodes of form $T{::}v$ in the tree. The elements of T' not appearing in v' can be associated with the number 0. For instance, consider the value tree shown in Figure 7.6 for the element $l = \langle\langle v_3, 3, u_1\rangle, \langle v_3, 2, u_3\rangle\rangle$ of type $\mathsf{List}(\mathsf{Struct}(T_1, \mathsf{Int}, T_2))$, where T_1 is an unordered primitive type with $\mathscr{D}_{T_1} = \{v_1, v_2, v_3, v_4\}$ and T_2 is a cyclic primitive type with $\mathscr{D}_{T_2} = \{u_1, u_2, u_3, u_4\}$. The depth-first numbering of nodes is shown in boldface font in the figure. Thus the elements of T_1 are associated with integers by the mapping $\{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 1, v_4 \mapsto 0\}$ and those of T_2 by $\{u_1 \mapsto 3, u_2 \mapsto 0, u_3 \mapsto 7, u_4 \mapsto 0\}$. Define the function $I_{\text{dfs-numbering of } T \text{ in } T'} : \mathscr{D}_T \times \mathscr{D}_{T'} \times \mathfrak{P} \to \mathbb{N}$ to be the mapping described above. Based on Fact 7.6, it should be obvious that it is a partition independent type invariant with the following property: if two elements, v_1 and v_2, of type T appear in the element v', then $I(v_1, v', \mathfrak{p}) \neq I(v_2, v', \mathfrak{p})$. Therefore, refining a partition according to such an invariant leads to a partition in which all the elements appearing in the element v' are in their own cells. For cyclic primitive types, the resulting partition should be further refined by using the successor based invariant described above.

The above procedure does not work for structured types composed of sets, multisets or association arrays. This is because the value tree is not ordered in the above sense and therefore a unique numbering cannot assigned to the nodes as above. However, this restriction can be circumvented in some special cases. For instance, consider an association array where the domain of the first type (the type whose elements are associated with elements of the second type) is not permuted by allowed domain permutations, e.g., a type $\mathsf{AssocArray}(\mathtt{Int[1\text{-}3]}, \mathsf{Struct}(T_1, \mathsf{Int}))$, where $\mathtt{Int[1\text{-}3]}$ with the domain $\mathscr{D}_{\mathtt{Int[1\text{-}3]}} = \{1, 2, 3\}$ is an ordered primitive type. This type corresponds to

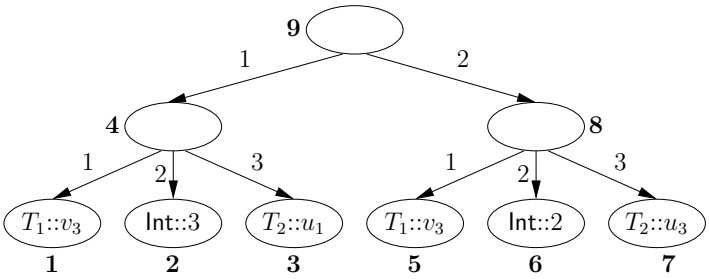


Figure 7.6: An ordered value tree

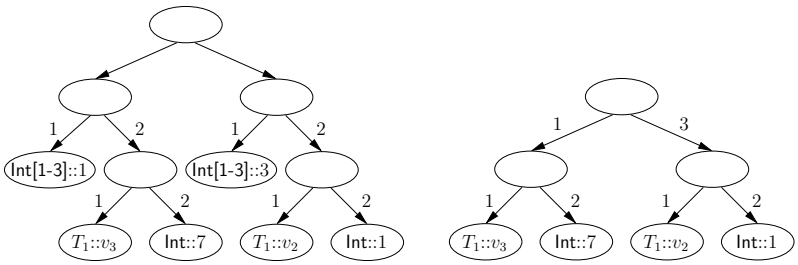


Figure 7.7: Mapping an unordered value tree to an ordered one

a normal array of size 3 (with possibility for undefined elements), and the elements in it are totally ordered. In this kind of case the value tree can be modified to be ordered, as shown in Figure 7.7 for an element $\{1 \mapsto \langle v_3, 7\rangle, 3 \mapsto \langle v_2, 1\rangle\}$, and the above procedure for producing type invariants can be applied.

Although state variables of the "easy" structured types described above are common in Murφ descriptions, in high-level Petri nets the state variables are of multiset types which are not handled by the above procedure. Yet the above procedure can be applied to multisets over the "easy" structured types in some important special cases: if a multiset contains only one element or all the elements in the multiset have different multiplicities, then the value tree becomes ordered and the above procedure works fine. The same applies to set types in the case a set contains only one element.

There is an important special case that often occurs in high-level Petri nets: a state variable of type $\mathsf{Multiset}(T)$, where T is a permutable primitive type. Define the partition independent type invariant $I_{\text{multiplicity}} : \mathscr{D}_T \times \mathscr{D}_{\mathsf{Multiset}(T)} \times \mathfrak{P} \to \mathbb{N}$ by $I_{\text{multiplicity}}(v, m, \mathfrak{p}) = m(v)$. In the case T is an unordered primitive type, $I_{\text{multiplicity}}$ has the property that if a partition is refined according to this invariant, resulting in a partition $\mathfrak{p}_1$, then $\theta_2^{\mathsf{Multiset}(T)}(m) = \theta_3^{\mathsf{Multiset}(T)}(m)$ for all allowed domain permutations θ_2 and θ_3 that are compatible with partitions $\mathfrak{p}_2 \preceq \mathfrak{p}_1$ and $\mathfrak{p}_3 \preceq \mathfrak{p}_1$, respectively. Thus $I_{\text{multiplicity}}$ in a sense canonizes the multiset value m.

Hash-Like Invariants

The invariants introduced so far have been quite simple. More complicated special invariants could be easily defined, but there are too many of them to cover all imaginable cases. For instance, assuming a state variable x of type $\mathsf{Multiset}(\mathsf{Struct}(\mathsf{Int}, T))$, where Int with $\mathscr{D}_{\mathsf{Int}} = \{0, 1, 2, \ldots\}$ is an ordered primitive type, the function $I_{T,x,\langle 3,?\rangle}$ for the type T defined by

$I_{T,x,\langle 3,?\rangle}(v,\mathfrak{s},\mathfrak{p}) = \mathfrak{s}(x)(\langle 3,v\rangle)$, i.e., the number of $\langle 3,v\rangle$-elements in the value of x in the state $\mathfrak{s}$, is an invariant. The more complicated the types of the state variables get, the more complicated the possible invariants get, too. It is now shown how to calculate a general purpose invariant that depends on the structure of a state in a larger degree than the previous ones. It is also partition dependent. Moreover, calculating the invariant is relatively easy: it resembles the way one would compute a hash value for a structured object.

For each primitive type T, a function

$$\mathfrak{g}_T(v,T',v',\mathfrak{p})$$

over four arguments is defined. The first argument is an element v in the domain of the type T, the second argument is a type T', the third argument is an element v' in the domain of the type T', and the last argument is a partition. The first argument v is the element for which the "hash value" is computed, while the second and third arguments describe the object in which this computation is performed. The fourth argument gives the current partition. The function $\mathfrak{g}_T$ is defined recursively top-down on the structure of the second argument type T': the value depends on the values of the subtypes of T'. In the leaves, when T' is a primitive type, the function has a value depending on (i) the relationship between the types T and T', (ii) the relationship between the values as the first and third argument, and (iii) the partition $\mathfrak{p}$.

Firstly, an *associative and commutative* binary operation $\oplus$ on $\mathbb{Z}$ is assumed. Furthermore, for each type T, $h_T : \mathbb{Z} \to \mathbb{Z}$ and $h_{T,n} : \mathbb{Z}^n \to \mathbb{Z}$ are assumed to be arbitrary functions unless otherwise stated. The inductive definition of the function $\mathfrak{g}_T$ now is:

1. For an ordered primitive type T', $\mathfrak{g}_T(v,T',v',\mathfrak{p}) = h_{T'}(v')$, where $h_{T'}$ is a function from $\mathscr{D}_{T'}$ to $\mathbb{Z}$.
2. For a cyclic primitive type T',

$$\mathfrak{g}_T(v,T',v',\mathfrak{p}) = \begin{cases} h_{T'}(\mathit{incell}(\mathfrak{p}^{T'},v')) & \text{if } T \neq T' \\ h_{T',2}(k,\mathit{incell}(\mathfrak{p}^{T'},v')) & \text{if } T = T' \text{ and } v' \text{ is the } k\text{-successor of } v. \end{cases}$$

3. For an unordered primitive type T',

$$\mathfrak{g}_T(v,T',v',\mathfrak{p}) = \begin{cases} h_{T',2}(\mathit{incell}(\mathfrak{p}^{T'},v'),0) & \text{if } T \neq T' \text{ or } T = T' \wedge v \neq v' \\ h_{T',2}(\mathit{incell}(\mathfrak{p}^{T'},v'),1) & \text{if } T = T' \wedge v = v'. \end{cases}$$

4. For a list type $T' = \mathsf{List}(T_1)$,

$$\mathfrak{g}_T(v,T',\langle v_1,\ldots,v_n\rangle,\mathfrak{p}) = h_{T',n}(\mathfrak{g}_T(v,T_1,v_1,\mathfrak{p}),\ldots,\mathfrak{g}_T(v,T_1,v_n,\mathfrak{p})).$$

5. For a structure type $T' = \mathsf{Struct}(T_1,\ldots,T_n)$,

$$\mathfrak{g}_T(v,T',\langle v_1,\ldots,v_n\rangle,\mathfrak{p}) = h_{T',n}(\mathfrak{g}_T(v,T_1,v_1,\mathfrak{p}),\ldots,\mathfrak{g}_T(v,T_n,v_n,\mathfrak{p})).$$

6. For a set type $T' = \mathsf{Set}(T_1)$,

$$\mathfrak{g}_T(v,T',v',\mathfrak{p}) = h_{T'}\left(\bigoplus_{v''\in v'} \mathfrak{g}_T(v,T_1,v'',\mathfrak{p})\right).$$

7. For a multiset type $T' = \mathsf{Multiset}(T_1)$,

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = h_{T'}\left(\bigoplus_{v'' \in \mathscr{D}_{T_1}, v'(v'') \geq 1} h_{T',2}(v'(v''), \mathfrak{g}_T(v, T_1, v'', \mathfrak{p}))\right).$$

8. For an association array type $T' = \mathsf{AssocArray}(T_1, T_2)$,

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = h_{T'}\left(\bigoplus_{\langle v_1, v_2 \rangle \in v'} h_{T',2}(\mathfrak{g}_T(v, T_1, v_1, \mathfrak{p}), \mathfrak{g}_T(v, T_2, v_2, \mathfrak{p}))\right).$$

9. For an union type $T' = \mathsf{Union}(T_1, \ldots, T_n)$,

$$\mathfrak{g}_T(v, T', \langle T_i, v' \rangle, \mathfrak{p}) = h_{T'}(\mathfrak{g}_T(v, T_i, v', \mathfrak{p})).$$

Lemma 7.31 *For each allowed domain permutation* $\theta = \left\{\theta^T\right\}_{T \in \mathcal{T}}$,

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})).$$

Proof. By induction on the structure of T'.

Induction base.

1. For an ordered primitive type T',

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T'}(\theta^{T'}(v')) \\ &= h_{T'}(v') \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}) \end{aligned}$$

since $\theta^{T'}(v') = v'$ for an ordered primitive type T'.

2. Let T' be a cyclic primitive type.
   (a) If $T \neq T'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T'}(\mathit{incell}(\theta^{T'}(\mathfrak{p}^{T'}), \theta^{T'}(v'))) \\ &= h_{T'}(\mathit{incell}(\mathfrak{p}^{T'}, v')) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}). \end{aligned}$$

   (b) If $T = T'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(k, \mathit{incell}(\theta^{T'}(\mathfrak{p}^{T'}), \theta^{T'}(v'))) \\ &= h_{T',2}(k, \mathit{incell}(\mathfrak{p}^{T'}, v')) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}) \end{aligned}$$

   because v' is the k-successor of v if and only if $\theta^T(v')$ is the k-successor of $\theta^T(v)$.

3. Let T' be an unordered primitive type.
   (a) If $T \neq T'$ or $T = T' \wedge v \neq v'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(\mathit{incell}(\theta^{T'}(\mathfrak{p}^{T'}), \theta^{T'}(v')), 0) \\ &= h_{T',2}(\mathit{incell}(\mathfrak{p}^{T'}, v'), 0) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}) \end{aligned}$$

   because in the case $T = T'$, $v \neq v'$ if and only if $\theta^T(v) \neq \theta^T(v')$.

(b) If $T = T' \wedge v = v'$, then

$$\begin{aligned}\mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(incell(\theta^{T'}(\mathfrak{p}^{T'}), \theta^{T'}(v')), 1)\\ &= h_{T',2}(incell(\mathfrak{p}^{T'}, v'), 1)\\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p})\end{aligned}$$

because $v = v'$ if and only if $\theta^T(v) = \theta^T(v')$.

Induction hypothesis. Assume that the lemma holds for types $T_1,\ldots,T_n$.
Induction step.

– For a list type $T' = \mathsf{List}(T_1)$,

$$\begin{aligned}\mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(\langle v_1, \ldots, v_n\rangle), \theta(\mathfrak{p})) &=\\ \mathfrak{g}_T(\theta^T(v), T', \langle\theta^{T_1}(v_1), \ldots, \theta^{T_1}(v_n)\rangle, \theta(\mathfrak{p})) &=\\ h_{T',n}(\mathfrak{g}_T(\theta^T(v), T_1, \theta^{T_1}(v_1), \theta(\mathfrak{p})), \ldots, \mathfrak{g}_T(\theta^T(v), T_1, \theta^{T_1}(v_n), \theta(\mathfrak{p}))) &=\\ h_{T',n}(\mathfrak{g}_T(v, T_1, v_1, \mathfrak{p}), \ldots, \mathfrak{g}_T(v, T_1, v_n, \mathfrak{p})) &=\\ \mathfrak{g}_T(v, T', \langle v_1, \ldots, v_n\rangle, \mathfrak{p}).\end{aligned}$$

– For a structure type $T' = \mathsf{Struct}(T_1, \ldots, T_n)$, the proof is similar to the list case above.
– For a set type $T' = \mathsf{Set}(T_1)$,

$$\begin{aligned}\mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &=\\ h_{T'}\left(\bigoplus_{v''\in\theta^{T'}(v')} \mathfrak{g}_T(\theta^T(v), T_1, v'', \theta(\mathfrak{p}))\right) &=\\ h_{T'}\left(\bigoplus_{v''\in\theta^{T'}(v')} \mathfrak{g}_T(v, T_1, \theta^{T_1{}^{-1}}(v''), \mathfrak{p})\right) &=\\ h_{T'}\left(\bigoplus_{v'''\in v'} \mathfrak{g}_T(v, T_1, v''', \mathfrak{p})\right) &=\\ \mathfrak{g}_T(v, T', v', \mathfrak{p})\end{aligned}$$

by using the commutativity and associativity of $\oplus$, and by noticing that for all $v' \in \mathscr{D}_{\mathsf{Set}(T_1)}$ and all $v'' \in \mathscr{D}_{T_1}$, $v'' \in v' \Leftrightarrow \theta^{T_1}(v'') \in \theta^{\mathsf{Set}(T_1)}(v')$.
– Assume that $T' = \mathsf{Multiset}(T_1)$. Now an element $v'' \in \mathscr{D}_{T_1}$ has multiplicity n in a multi-set $v' \in \mathscr{D}_{\mathsf{Multiset}(T_1)}$ if and only if the element $\theta^{T_1}(v'')$ has multiplicity n in the multi-set $\theta^{T'}(v')$. The rest of the proof is similar to the previous case.
– Let $T' = \mathsf{AssocArray}(T_1, T_2)$. Now for each $v' \in \mathscr{D}_{\mathsf{AssocArray}(T_1,T_2)}$, a pair $\langle v_1, v_2\rangle \in v'$ if and only if $\langle\theta^{T_1}(v_1), \theta^{T_2}(v_2)\rangle \in \theta^{T'}(v')$. The rest of the proof is similar to the case $T' = \mathsf{Set}(T_1)$.
– For an union type $T' = \mathsf{Union}(T_1, \ldots, T_n)$,

$$\begin{aligned}\mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(\langle T_i, v'\rangle), \theta(\mathfrak{p})) &= \mathfrak{g}_T(\theta^T(v), T', \langle T_i, \theta^{T_i}(v')\rangle, \theta(\mathfrak{p}))\\ &= h_{T'}(\mathfrak{g}_T(\theta^T(v), T_i, \theta^{T_i}(v'), \theta(\mathfrak{p})))\\ &= h_{T'}(\mathfrak{g}_T(v, T_i, v', \mathfrak{p}))\\ &= \mathfrak{g}_T(v, T', \langle T_i, v'\rangle, \mathfrak{p}).\end{aligned}$$

□

Corollary 7.32 *For a permutable primitive type T and for a type T',*

$$I_{T,\textit{hash in } T'}(v, v', \mathfrak{p}) = \mathfrak{g}_T(v, T', v', \mathfrak{p})$$

is a type invariant. Similarly, if x is a state variable of type T', then

$$I_{T,\textit{hash in } x}(v, \mathfrak{s}, \mathfrak{p}) = \mathfrak{g}_T(v, T', \mathfrak{s}(x), \mathfrak{p})$$

is an invariant.

Example 7.33 Consider again the state $\mathfrak{s} = \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$ for the railroad system net in Figure 7.1, recall Examples 7.18, 7.28 and 7.30. Let the commutative and associative operation $\oplus$ above be the integer addition operation, and let

$$\begin{array}{ll}
h_{\mathsf{Trains},2}(1,0) = 374, & h_{\mathsf{Trains},2}(2,0) = 1374, \\
h_{\mathsf{Trains},2}(1,1) = 242 \cdot 374, & h_{\mathsf{Trains},2}(2,1) = 242 \cdot 1374, \\
h_{\mathsf{Secs},2}(k,1) = (k+1) \cdot 837, & h_{\mathsf{Secs},2}(k,2) = (k+1) \cdot 274, \\
h_{\mathsf{Secs},2}(k,3) = (k+1) \cdot 97, & h_{\mathsf{Secs},2}(k,4) = (k+1) \cdot 4732, \\
h_{\mathsf{Secs},2}(k,5) = (k+1) \cdot 194, & h_{\mathsf{Secs},2}(k,6) = (k+1) \cdot 958, \\
h_{\mathsf{Multiset(Struct(Trains,Secs)}}(x) = x, & h_{\mathsf{Multiset(Struct(Trains,Secs)),2}}(x,y) = x \cdot y, \\
h_{\mathsf{Struct(Trains,Secs),2}}(x,y) = x \cdot \lfloor \frac{y}{2} \rfloor. &
\end{array}$$

Initially, the partition is

$$\mathfrak{p}_{\mathfrak{s},0} = \left(\mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Secs}} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

Evaluating the invariant $I_{\mathsf{Secs},\text{hash in } U}$ in the partition gives

$$\begin{aligned}
& I_{\mathsf{Secs},\text{hash in } U}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) = \\
& \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Multiset}(\mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs})), \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, \mathfrak{p}_{\mathfrak{s},0}) = \\
& 1 \cdot \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs}), \langle t_a, s_1\rangle, \mathfrak{p}_{\mathfrak{s},0}) \\
& +1 \cdot \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs}), \langle t_b, s_3\rangle, \mathfrak{p}_{\mathfrak{s},0}) = \\
& 1 \cdot (\mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Trains}, t_a, \mathfrak{p}_{\mathfrak{s},0}) \cdot \lfloor \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Secs}, s_1, \mathfrak{p}_{\mathfrak{s},0})/2 \rfloor) \\
& +1 \cdot (\mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Trains}, t_b, \mathfrak{p}_{\mathfrak{s},0}) \cdot \lfloor \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Secs}, s_3, \mathfrak{p}_{\mathfrak{s},0})/2 \rfloor) = \\
& 1 \cdot (h_{\mathsf{Trains}}(1,0) \cdot \lfloor h_{\mathsf{Secs},2}(1,1)/2 \rfloor) + 1 \cdot (h_{\mathsf{Trains}}(1,0) \cdot \lfloor h_{\mathsf{Secs},2}(3,1)/2 \rfloor) = \\
& 1 \cdot (374 \cdot \lfloor (2 \cdot 837)/2 \rfloor) + 1 \cdot (374 \cdot \lfloor (4 \cdot 837)/2 \rfloor) = \\
& 1 \cdot (374 \cdot 837) + 1 \cdot (374 \cdot 1674) = \\
& 939114,
\end{aligned}$$

and

$$\begin{aligned}
I_{\mathsf{Secs},\text{hash in } U}(s_1, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= 625702, \\
I_{\mathsf{Secs},\text{hash in } U}(s_2, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= 1252152, \\
I_{\mathsf{Secs},\text{hash in } U}(s_3, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= 938740, \\
I_{\mathsf{Secs},\text{hash in } U}(s_4, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= 1565190, \text{ and} \\
I_{\mathsf{Secs},\text{hash in } U}(s_5, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= 1251778.
\end{aligned}$$

Now the partition is refined to

$$\mathfrak{p}_{\mathfrak{s},1} = \left(\mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Secs}} = [\{s_1\},\{s_3\},\{s_0\},\{s_5\},\{s_2\},\{s_4\}], \mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Trains}} = [\{t_a,t_b\}]\right).$$

Evaluating $I_{\mathsf{Secs},\text{hash in } V}$ in this partition yields no further information since the partition for Secs is already discrete. Evaluating $I_{\mathsf{Trains},\text{hash in } U}$ in the partition gives $I_{\mathsf{Trains},\text{hash in } U}(t_a,\mathfrak{s},\mathfrak{p}_{\mathfrak{s},1}) = 37883582$ and $I_{\mathsf{Trains},\text{hash in } U}(t_b,\mathfrak{s},\mathfrak{p}_{\mathfrak{s},1}) = 12555928$, refining the partition to

$$\mathfrak{p}_{\mathfrak{s},2} = \left(\mathfrak{p}_{\mathfrak{s},2}^{\mathsf{Secs}} = [\{s_1\},\{s_3\},\{s_0\},\{s_5\},\{s_2\},\{s_4\}], \mathfrak{p}_{\mathfrak{s},2}^{\mathsf{Trains}} = [\{t_b\},\{t_a\}]\right).$$

Now there is only one allowed domain permutation compatible with $\mathfrak{p}_{\mathfrak{s},2}$, namely

$$\theta = \left(\theta^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_5 & s_0 & s_1 & s_2 & s_3 & s_4 \end{smallmatrix}\right), \theta^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right),$$

and the corresponding representative state is

$$\theta(\mathfrak{s}) = \{U \mapsto \langle t_a, s_2\rangle + \langle t_b, s_0\rangle, V \mapsto s_3 + s_4\}.$$

♣

Note that the h-functions defined in the above example are not probably optimal since they are quite similar. Although they suffice for demonstrative purposes, in a real implementation some better bit-level manipulation operations should be applied instead in order to reduce the possibility of value collision. The h-functions may also, for instance, employ pseudo-random numbers to obtain relative independence from each other. The main thing to take care of is that the operation $\oplus$ is commutative and associative.

7.4 IMPROVEMENTS BASED ON SEARCH TREES

Recall the Algorithm 7.1 for producing representative states. Given a state $\mathfrak{s}$, the partition $pg(\mathfrak{s})$ is first produced, an arbitrary allowed domain permutation θ compatible with it is then selected, and finally the state $\theta(\mathfrak{s})$ is returned as the representative. In the case the partition $pg(\mathfrak{s})$ has a non-singleton cell for a permutable primitive type, there may be many compatible allowed domain permutations, and thus, potentially but not necessarily, many possible representative states for $\mathfrak{s}$. Especially, when $pg(\mathfrak{s})$ has a non-singleton cell of size n for an unordered primitive type, the choice of which element will be the "first" one does not affect the $n-1$ choices taken for the rest of the elements (except that they cannot be the "first" element). Nor does it affect in any way the choices that have to be made for other non-singleton cells. This section presents an improvement that can reduce the set of possible representative states. In this approach, the choices may affect or eliminate the choices yet to be taken. The idea for the approach is borrowed from the standard algorithms for the graph isomorphism problem [McKay 1981; Kreher and Stinson 1999].

First, a fixed partition generator pg and a fixed partition refiner $\mathcal{R}$ are assumed.

Definition 7.34 *The* search tree *of a state $\mathfrak{s}$ and a partition $\mathfrak{p} = \{\mathfrak{p}^T\}_{T\in\mathcal{T}_P}$ is a tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ defined by the following inductive rules.*

1. *If each partition $\mathfrak{p}^T$ in $\mathfrak{p}$ is discrete, then the tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ is the single leaf node $\mathfrak{p}$.*
2. *Otherwise, let $\mathfrak{p}^T = [C_1^T, \ldots, C_n^T]$ be the first non-discrete partition in $\mathfrak{p}$ (according to some fixed ordering between the permutable primitive types). Let $C_i^T = \{v_{i,1}, v_{i,2}, \ldots\}$ be the first non-singleton cell in $\mathfrak{p}^T$. The tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ then consists of the root node $\mathfrak{p}$ which has as its children the trees $\mathcal{ST}(\mathfrak{s},\mathcal{R}(\mathfrak{s},\mathfrak{p}_j))$, where for each $1 \leq j \leq \left|C_i^T\right|$ the partition $\mathfrak{p}_j$ is the same as $\mathfrak{p}$ except that the partition for the type T is*

$$\mathfrak{p}_j^T = [C_1^T, \ldots, C_{i-1}^T, \{v_{i,j}\}, C_i^T \setminus \{v_{i,j}\}, C_{i+1}^T, \ldots, C_n^T].$$

In other words, for each element in the first non-discrete cell C_i^T, the cell is split in two parts by distinguishing the element into its own cell. The child $\mathcal{ST}(\mathfrak{s},\mathcal{R}(\mathfrak{s},\mathfrak{p}_j))$ above is called the $v_{i,j}$-child of the node $\mathfrak{p}$ and the edge from $\mathfrak{p}$ to it is labeled with $T.v_{i,j}$. One may use $\mathfrak{p} \xrightarrow{T.v} \mathfrak{p}'$ to denote that $\mathfrak{p}'$ is a v-child of $\mathfrak{p}$.

The search tree $\mathcal{ST}(\mathfrak{s})$ *of a state $\mathfrak{s}$ is the search tree $\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{s}))$.*

Algorithm 7.1 is now modified as follows. Given a state $\mathfrak{s}$, *travel along one, arbitrary path in the search tree $\mathcal{ST}(\mathfrak{s})$ until a leaf node (discrete partition) $\mathfrak{p}$ is encountered,* take the unique allowed domain permutation θ that is compatible with $\mathfrak{p}$ and return $\theta(\mathfrak{s})$ as the representative. The resulting algorithm is shown in Algorithm 7.2.

Algorithm 7.2 A representative algorithm based on search trees

Input: A state $\mathfrak{s}$
Output: A representative state that is equivalent to $\mathfrak{s}$

1: Build the partition $\mathfrak{p} = pg(\mathfrak{s})$
2: Choose any path in the search tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ ending in a discrete partition $\mathfrak{p}'$
3: Let θ be the unique allowed domain permutation compatible with $\mathfrak{p}'$
4: Return $\theta(\mathfrak{s})$ as the representative state

Example 7.35 Consider the state $\mathfrak{s} = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$ for the railroad system net in Figure 7.1. Refining the initial partition with the invariant sequence $I_{\sharp\mathsf{Trains\ in}\ U}.I_{\sharp\mathsf{Trains\ in}\ V}.I_{\sharp\mathsf{Secs\ in}\ U}.I_{\sharp\mathsf{Secs\ in}\ V}$ (i.e., applying the partition generator) gives the partition

$$\mathfrak{p} = \left(\mathfrak{p}^{\mathsf{Secs}} = [\{s_2, s_5\}, \{s_1, s_4\}, \{s_0, s_3\}], \mathfrak{p}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

This partition is the best one can get by using any partition generator function in the sense that the elements in any cell in it cannot be distinguished by any such function. This is because the allowed domain permutation

$$\theta \;=\; \left(\theta^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right)$$

is a stabilizer of $\mathfrak{s}$ in Θ (recall Fact 7.19). The four domain permutations

compatible with the partition are

$$\begin{array}{rcl}
\theta_1 & = & \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \\
\theta_2 & = & \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \\
\theta_3 & = & \left(\theta_3^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_3^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\
\theta_4 & = & \left(\theta_4^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_4^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).
\end{array}$$

The corresponding two possible representative states for $\mathfrak{s}$ are:

$$\begin{array}{rcl}
\theta_1(\mathfrak{s}) = \theta_4(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\} \text{ and} \\
\theta_2(\mathfrak{s}) = \theta_3(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_4\rangle, V \mapsto s_2 + s_5\}.
\end{array}$$

Assume the partition refiner $\mathcal{R}$ that is induced by the invariant sequence that first contains six $I_{\mathsf{Secs,succ}}$ invariants and after those enough hash-like invariants described in Section 7.3.2. The search tree $\mathcal{ST}(\mathfrak{s})$ has $\mathfrak{p}$ as the root node. The cell $\{s_2, s_5\}$ is now the first non-singleton cell in $\mathfrak{p}$ and thus $\mathfrak{p}$ is split to

$$\mathfrak{p}_{1,1} = \left(\mathfrak{p}_{1,1}^{\mathsf{Secs}} = [\{s_2\}, \{s_5\}, \{s_1, s_4\}, \{s_0, s_3\}], \mathfrak{p}_{1,1}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right)$$

and

$$\mathfrak{p}_{2,1} = \left(\mathfrak{p}_{2,1}^{\mathsf{Secs}} = [\{s_5\}, \{s_2\}, \{s_1, s_4\}, \{s_0, s_3\}], \mathfrak{p}_{2,1}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right),$$

respectively. Refining these with the $I_{\mathsf{Secs,succ}}$ invariants gives

$$\mathfrak{p}_{1,2} = \left(\mathfrak{p}_{1,2}^{\mathsf{Secs}} = [\{s_2\}, \{s_5\}, \{s_1\}, \{s_4\}, \{s_0\}, \{s_3\}], \mathfrak{p}_{1,2}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right)$$

and

$$\mathfrak{p}_{2,2} = \left(\mathfrak{p}_{2,2}^{\mathsf{Secs}} = [\{s_5\}, \{s_2\}, \{s_4\}, \{s_1\}, \{s_3\}, \{s_0\}], \mathfrak{p}_{2,2}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right),$$

respectively. Refining these with the invariant $I_{\mathsf{Secs,hash\ in}\ U}$ or $I_{\mathsf{Secs,hash\ in}\ V}$ improves nothing since the partitions for Secs are already discrete. However, refining the partitions with the $I_{\mathsf{Trains,hash\ in}\ U}$ invariant, by using the functions of Example 7.33, yields the partitions

$$\mathfrak{p}_{1,3} = \left(\mathfrak{p}_{1,3}^{\mathsf{Secs}} = [\{s_2\}, \{s_5\}, \{s_1\}, \{s_4\}, \{s_0\}, \{s_3\}], \mathfrak{p}_{1,3}^{\mathsf{Trains}} = [\{t_a\}, \{t_b\}]\right)$$

and

$$\mathfrak{p}_{2,3} = \left(\mathfrak{p}_{2,3}^{\mathsf{Secs}} = [\{s_5\}, \{s_2\}, \{s_4\}, \{s_1\}, \{s_3\}, \{s_0\}], \mathfrak{p}_{2,3}^{\mathsf{Trains}} = [\{t_b\}, \{t_a\}]\right),$$

respectively. These two partitions are the two leaf nodes of the search tree $\mathcal{ST}(\mathfrak{s})$, shown in Figure 7.8, and the allowed domain permutations compatible with them are

$$\begin{array}{rcl}
\theta_{1,3} & = & \left(\theta_{1,3}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_{1,3}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_1 \text{ and} \\
\theta_{2,3} & = & \left(\theta_{2,3}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{2,3}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_4.
\end{array}$$

The corresponding representative state for $\mathfrak{s}$ is:

$$\theta_{1,3}(\mathfrak{s}) = \theta_{2,3}(\mathfrak{s}) = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}.$$

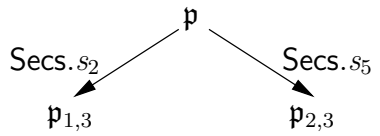


Figure 7.8: A search tree

7.4.1 Properties of Search Trees

Some properties of search trees are listed below.

Theorem 7.36 *For each allowed domain permutation θ, a partition $\mathfrak{p}_{child}$ is a v-child of the root node of the search tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ if and only if the partition $\theta(\mathfrak{p}_{child})$ is a $\theta^T(v)$-child of the root node of the search tree $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$.*

Proof. If $\mathfrak{p}$ is discrete, then $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ has no children. But now $\theta(\mathfrak{p})$ is also discrete and $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$ has no children.

Clearly, $\mathfrak{p}^T = [C_1^T,\ldots,C_n^T]$ is the first non-discrete partition in $\mathfrak{p}$ if and only if $\theta^T(\mathfrak{p}^T) = [\theta^T(C_1^T),\ldots,\theta^T(C_n^T)]$ is the first non-discrete partition in $\theta(\mathfrak{p})$. Furthermore, $C_i^T = \{v_{i,1}, v_{i,2}, \ldots\}$ is the first non-singleton cell in $\mathfrak{p}^T$ if and only if $\theta^T(C_i^T) = \{\theta^T(v_{i,1}), \theta^T(v_{i,2}), \ldots\}$ is the first non-singleton cell in $\theta^T(\mathfrak{p}^T)$. Now the root node $\mathfrak{p}$ of the tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ has as its children the nodes $\mathcal{R}(\mathfrak{s},\mathfrak{p}_j)$, where for each $1 \le j \le \left|C_i^T\right|$ the partition $\mathfrak{p}_j$ is the same as $\mathfrak{p}$ except that the partition for T is

$$\mathfrak{p}_j^T = [C_1^T,\ldots,C_{i-1}^T,\{v_{i,j}\},C_i^T \setminus \{v_{i,j}\},C_{i+1}^T,\ldots].$$

But the root node $\theta(\mathfrak{p})$ of the tree $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$ has as its children the nodes $\mathcal{R}(\theta(\mathfrak{s}),\mathfrak{p}_{j'})$, where for each $1 \le j \le \left|\theta^T(C_i^T)\right| = \left|C_i^T\right|$ the partition $\mathfrak{p}_{j'}$ is the same as $\theta(\mathfrak{p})$ except that the partition for T is

$$\mathfrak{p}_{j'}^T = [\theta^T(C_1^T),\ldots,\theta^T(C_{i-1}^T),\{\theta^T(v_{i,j})\},\theta^T(C_i^T) \setminus \{\theta^T(v_{i,j})\},\theta^T(C_{i+1}^T),\ldots]$$

which equals to $\theta(\mathfrak{p}_j^T)$. Thus the root node $\theta(\mathfrak{p})$ of the tree $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$ has as its children the nodes $\mathcal{R}(\theta(\mathfrak{s}),\theta(\mathfrak{p}_j)) = \theta(\mathcal{R}(\mathfrak{s},\mathfrak{p}_j))$, meaning that $\mathfrak{p}_{\text{child}}$ is a $v_{i,j}$-child of $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ if and only if $\theta(\mathfrak{p}_{\text{child}})$ is a $\theta^T(v_{i,j})$-child of $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$. □

Corollary 7.37 *For each allowed domain permutation θ,*

$$\mathfrak{p} \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_n.v_n} \mathfrak{p}_n$$

is a path in the search tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ if and only if

$$\theta(\mathfrak{p}) \xrightarrow{T_1.\theta^{T_1}(v_1)} \theta(\mathfrak{p}_1) \cdots \xrightarrow{T_n.\theta^{T_n}(v_n)} \theta(\mathfrak{p}_n)$$

is a path in the search tree $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$.

Corollary 7.38 *For each allowed domain permutation θ, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{ST}(\mathfrak{s},\mathfrak{p})$ if and only if the partition $\theta(\mathfrak{p}')$ is a node in the search tree $\mathcal{ST}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$.*

Since $\theta(pg(\mathfrak{s})) = pg(\theta(\mathfrak{s}))$ for the partition generator pg, the above results generalize to search trees for states. For instance:

Corollary 7.39 *For each allowed domain permutation θ, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{ST}(\mathfrak{s})$ if and only if the partition $\theta(\mathfrak{p}')$ is a node in the search tree $\mathcal{ST}(\theta(\mathfrak{s}))$.*

Corollary 7.40 *For each stabilizer $\theta \in \mathrm{Stab}(\Theta, \mathfrak{s})$ of a state $\mathfrak{s}$ in Θ, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{ST}(\mathfrak{s})$ if and only if the partition $\theta(\mathfrak{p}')$ is.*

Since $\mathcal{R}(\mathfrak{s}, \mathfrak{p}) \preceq \mathfrak{p}$ holds for the partition refiner $\mathcal{R}$ used in the construction of search trees, some additional properties also hold. First of all, each descendant of a node is a cell order preserving refinement of the node. Furthermore, all the nodes in a search tree are mutually distinct partitions. It is also easy to verify that if θ is compatible with a partition $\mathfrak{p}_1$, then θ is compatible with any partition $\mathfrak{p}_2$ such that $\mathfrak{p}_1 \preceq \mathfrak{p}_2$.[1] Therefore, *the number of possible representative states for Algorithm 7.2 is at most that for Algorithm 7.1* (when the same partition generator is applied). In addition, by Corollary 7.40, it holds that the number of leaf nodes in the search tree $\mathcal{ST}(\mathfrak{s})$ is a multiple of $|\mathrm{Stab}(\Theta, \mathfrak{s})|$.

Given a discrete partition $\mathfrak{p}$, there is a unique allowed domain permutation, denote it by $\hat{\theta}_{\mathfrak{p}}$, that is compatible with it. Thus the set of leaf nodes in a search tree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$ defines the non-empty set of *possible representative states* by

$$posreps(\mathcal{ST}(\mathfrak{s}, \mathfrak{p})) = \left\{ \hat{\theta}_{\mathfrak{p}}(\mathfrak{s}) \mid \mathfrak{p} \text{ is a leaf node in } \mathcal{ST}(\mathfrak{s}, \mathfrak{p}) \right\}.$$

Define $posreps(\mathcal{ST}(\mathfrak{s})) = posreps(\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{p})))$.

Lemma 7.41 *For each allowed domain permutation θ, $posreps(\mathcal{ST}(\mathfrak{s}, \mathfrak{p})) = posreps(\mathcal{ST}(\theta(\mathfrak{s}), \theta(\mathfrak{p})))$.*

Proof. For each allowed domain permutation θ,

1. by Corollary 7.38, a partition $\mathfrak{p}$ is a leaf node in $\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$ if and only if $\theta(\mathfrak{p})$ is a leaf node in $\mathcal{ST}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$, and
2. by Lemma 7.16, $\hat{\theta}$ is compatible with $\mathfrak{p}$ if and only if $\hat{\theta} * \theta^{-1}$ is compatible with the partition $\theta(\mathfrak{p})$.

Thus $(\hat{\theta} * \theta^{-1})(\theta(\mathfrak{s})) = \hat{\theta}(\mathfrak{s}) \in posreps(\mathcal{ST}(\theta(\mathfrak{s}), \theta(\mathfrak{p})))$ if and only if $\hat{\theta}(\mathfrak{s}) \in posreps(\mathcal{ST}(\mathfrak{s}, \mathfrak{p}))$. □

Corollary 7.42 *For each allowed domain permutation θ, $posreps(\mathcal{ST}(\mathfrak{s})) = posreps(\mathcal{ST}(\theta(\mathfrak{s})))$*

Thus the sets of states from which Algorithm 7.2 selects the representative are the same for equivalent states.

[1] This would not necessarily hold if an arbitrary subgroup of Θ were used together with a compatibility definition similar to the one in Chapter 4.

7.4.2 Producing Canonical Representative States

Although Algorithm 7.2 is better than Algorithm 7.1, it does not necessarily produce canonical representative markings. In order to accomplish this, assume a total order $<$ on the set $\mathcal{S}$ of states. Given a state $\mathfrak{s}$, one can now select the smallest state in the set *posreps*($\mathcal{ST}(\mathfrak{s})$) to be the representative state. This can be done by performing a depth-first search in the search tree $\mathcal{ST}(\mathfrak{s})$. This procedure produces canonical representative states because *posreps*($\mathcal{ST}(\mathfrak{s})$) = *posreps*($\mathcal{ST}(\theta(\mathfrak{s}))$) for any allowed domain permutation θ by Corollary 7.42. The problem is that the search tree can have exponentially many nodes, at least it has $|\mathrm{Stab}(\Theta, \mathfrak{s})|$ nodes by Corollary 7.40. Fortunately, the search in the search tree can be pruned.

Pruning by Image Restriction. Assume that $\mathfrak{s}_{best}$ is the smallest state in *posreps*($\mathcal{ST}(\mathfrak{s})$) found so far during the search tree traversal. If the current partition whose children are not yet traversed is $\mathfrak{p}$ and it can be deduced that all the states in *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$) must be larger than $\mathfrak{s}_{best}$, then one can backtrack the search, i.e., skip the subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$ of $\mathcal{ST}(\mathfrak{s})$. Deducing that all the states in *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$) must be larger than $\mathfrak{s}_{best}$ can be done by the following observations. First, if an allowed domain permutation $\hat{\theta}_1$ is compatible with a descendant $\mathfrak{p}_1$ of $\mathfrak{p}$ in the search tree, then $\hat{\theta}_1$ is also compatible with $\mathfrak{p}$. Therefore, if a state is in *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$), then it must be produced from $\mathfrak{s}$ by applying an allowed domain permutation θ fulfilling the following rules: (i) if $\mathfrak{p}^T = [C_1^T, \ldots, C_c^T]$ for an unordered primitive type T with $\mathscr{D}_T = \{v_1, \ldots, v_n\}$, then θ must map C_1^T to $\{v_1, \ldots, v_{|C_1^T|}\}$, C_2^T to $\{v_{|C_1^T|+1}, \ldots, v_{|C_1^T|+|C_2^T|}\}$ and so on, and (ii) if $\mathfrak{p}^T = [C_1^T, \ldots, C_c^T]$ for a cyclic primitive type T with $\mathscr{D}_T = \{v_0, \ldots, v_{n-1}\}$, then θ must map an element in C_1^T to v_0. Thus the possible images of the elements of permutable primitive types are restricted by $\mathfrak{p}$ and one may be able to deduce that all the states in *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$) must be larger than $\mathfrak{s}_{best}$. Of course, this deduction step depends on the selected total order $<$ on the states.

Pruning with Stabilizers. This technique is adapted from the standard graph isomorphism algorithms, see e.g. [McKay 1981; Kreher and Stinson 1999] and also compare to the stabilizer pruning technique discussed in Section 4.3. Consider the root node $\mathfrak{p}$ of a subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p})$ in the search tree $\mathcal{ST}(\mathfrak{s})$. Assume that it has two children, e.g., $\mathfrak{p} \xrightarrow{T.v} \mathfrak{p}_1$ and $\mathfrak{p} \xrightarrow{T.v'} \mathfrak{p}_2$. If there is a stabilizer θ of $\mathfrak{s}$ that (i) respects $\mathfrak{p}$, i.e., $\theta(\mathfrak{p}) = \mathfrak{p}$, and (ii) maps v to v', i.e., $\theta^T(v) = v'$, then $\theta(\mathfrak{p}_1) = \mathfrak{p}_2$ by Theorem 7.36. Now *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_1)$) = *posreps*($\mathcal{ST}(\theta(\mathfrak{s}), \theta(\mathfrak{p}_1))$) = *posreps*($\mathcal{ST}(\mathfrak{s}, \theta(\mathfrak{p}_1))$) = *posreps*($\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_2)$), meaning that the possible representative states in the subtrees $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_1)$ and $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_2)$ are the same. Therefore, if the subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_1)$ is already traversed, there is no need to traverse the subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_2)$.

As in Section 4.3, stabilizers of a state can be found during the search tree traversal. Assume that a leaf node $\mathfrak{p}_1$ has already been visited in the search tree. If currently visited leaf node is $\mathfrak{p}_2$ and $\hat{\theta}_{\mathfrak{p}_2}(\mathfrak{s}) = \hat{\theta}_{\mathfrak{p}_1}(\mathfrak{s})$, then $\hat{\theta}_{\mathfrak{p}_2}^{-1} * \hat{\theta}_{\mathfrak{p}_1}$ is a stabilizer of $\mathfrak{s}$. Of course, the natural candidate for the partition $\mathfrak{p}_1$ to be remembered and compared against during the search tree traversal

is the partition $\mathfrak{p}$ for which the state $\hat{\theta}_{\mathfrak{p}}(\mathfrak{s})$ is the smallest encountered so far. Because for every leaf node $\mathfrak{p}$ in the search tree and for every stabilizer θ there is the corresponding leaf node $\theta(\mathfrak{p})$ in the search tree and $\hat{\theta}_{\theta(\mathfrak{p})} = \hat{\theta}_{\mathfrak{p}} * \theta^{-1}$ implying $(\hat{\theta}_{\mathfrak{p}} * \theta^{-1})^{-1} * \hat{\theta}_{\mathfrak{p}} = \theta$, every stabilizer can be encountered during the search tree traversal.

Finding the stabilizers is not enough: recall that in order to prune the child $\mathfrak{p} \xrightarrow{T.v'} \mathfrak{p}_2$ of a node $\mathfrak{p}$ and only traverse the child $\mathfrak{p} \xrightarrow{T.v} \mathfrak{p}_1$, one must have a stabilizer that (i) respects $\mathfrak{p}$, i.e., $\theta(\mathfrak{p}) = \mathfrak{p}$, and (ii) maps v to v'. There are two well-known strategies for storing the stabilizers found during the search tree traversal and finding such that fulfill the above requirement, see [Kreher and Stinson 1999; McKay 1981].

The first is to use a Schreier-Sims representation for storing the group of stabilizers generated by the stabilizers found so far. Assume that the current search node $\mathfrak{p}$, whose v, v'-children are to be pruned, is reached from the root node of the search tree via a path $pg(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_n.v_n} \mathfrak{p}$. If there is a stabilizer θ that fixes all the elements $v_1, \ldots, v_n$, then θ maps $\mathfrak{p}$ to itself. Thus one must find whether there is a stabilizer stored in the group of stabilizers found so far fixing each $v_1, \ldots, v_n$ and mapping v to v'. This can be accomplished by using an operation called base change on the Schreier-Sims representation of the stabilizers found so far. Although this is can be done in polynomial time in the size of the union of the domains of the permutable primitive types (the number of permuted elements), it can still be quite time consuming and requires non-trivial algorithms.

The other approach does not store the stabilizers at all. Assume that a leaf node $\mathfrak{p}_{n,1}$ has already been visited by traversing a path $pg(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_i.v_i} \mathfrak{p}_i \xrightarrow{T_{i+1}.v_{i+1,1}} \mathfrak{p}_{i+1,1} \cdots \xrightarrow{T_{n,1}.v_{n,1}} \mathfrak{p}_{n,1}$ and that the whole subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$ has already been traversed. Suppose now that the currently visited leaf node is $\mathfrak{p}_{n,2}$ and the path to it is $pg(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_i.v_i} \mathfrak{p}_i \xrightarrow{T_{i+1}.v_{i+1,2}} \mathfrak{p}_{i+1,2} \cdots \xrightarrow{T_{n,2}.v_{n,2}} \mathfrak{p}_{n,2}$, i.e., the node $\mathfrak{p}_i$ is the latest common ancestor of $\mathfrak{p}_{n,1}$ and $\mathfrak{p}_{n,2}$. If $\hat{\theta}_{\mathfrak{p}_{n,2}}(\mathfrak{s}) = \hat{\theta}_{\mathfrak{p}_{n,1}}(\mathfrak{s})$, then $\theta = \hat{\theta}^{-1}_{\mathfrak{p}_{n,2}} * \hat{\theta}_{\mathfrak{p}_{n,1}}$ is a stabilizer of $\mathfrak{s}$. If it also holds that θ maps $\mathfrak{p}_{n,1}$ to $\mathfrak{p}_{n,2}$, then θ maps each $\mathfrak{p}_j$, $1 \leq j \leq i$, to itself (as $\mathfrak{p}_{n,1} \preceq \mathfrak{p}_j$ and $\mathfrak{p}_{n,2} \preceq \mathfrak{p}_j$) and $v_{i+1,1}$ to $v_{i+1,2}$. This implies that θ maps $\mathfrak{p}_{i+1,1}$ to $\mathfrak{p}_{i+1,2}$ and the subtrees $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$ and $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_{i+1,2})$ have the same possible representative states. Therefore, one can immediately skip the rest of the subtree $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_{i+1,2})$ as $\mathcal{ST}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$ has already been traversed. Furthermore, if a partition $\mathfrak{p}_j$, $1 \leq j \leq i$, has a v-child and a v'-child and a power of θ maps v to v', then the possible representative states in the v- and v'-subtrees of $\mathfrak{p}_j$ are the same.

7.4.3 A Relative Hardness Measure for States

In a way similar to that in Sections 4.3 and 4.4, a hardness measure for states can now be defined. The measure depends on the selected partition generator *pg* and on the partition refiner $\mathcal{R}$. The set of states, $\mathcal{S}$, are divided into three classes:

- A state $\mathfrak{s}$ is *trivial* if the search tree $\mathcal{ST}(\mathfrak{s})$ contains only one node, i.e., $pg(\mathfrak{s})$ is a discrete partition.

– A state $\mathfrak{s}$ is *easy* if it is not trivial and for any two leaf nodes (discrete partitions) $\mathfrak{p}_1$ and $\mathfrak{p}_2$ in the search tree $\mathcal{ST}(\mathfrak{s})$, it holds that there is a stabilizer $\theta \in \mathrm{Stab}(\Theta, \mathfrak{s})$ such that $\theta(\mathfrak{p}_1) = \mathfrak{p}_2$.
– A state $\mathfrak{s}$ is *hard* if it is neither trivial nor easy.

These classes are closed under symmetries:

Lemma 7.43 *If a state $\mathfrak{s}$ is trivial/easy/hard, then $\theta(\mathfrak{s})$ is also trivial/easy/hard for any allowed domain permutation θ.*

Proof. If a state $\mathfrak{s}$ is trivial, then the search tree $\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{s}))$ contains only one node, i.e., $pg(\mathfrak{s})$ is a discrete partition. But now for each allowed domain permutation θ, it must be that $pg(\theta(\mathfrak{s})) = \theta(pg(\mathfrak{s}))$ is also a discrete partition, and thus the search tree $\mathcal{ST}(\theta(\mathfrak{s}), pg(\theta(\mathfrak{s})))$ contains only one node and $\theta(\mathfrak{s})$ is also trivial.

Now assume that a state $\mathfrak{s}$ is easy and take any allowed domain permutation θ. The search tree $\mathcal{ST}(\theta(\mathfrak{s}), pg(\theta(\mathfrak{s})))$ must contain more than one node: if it contained only one node, $\theta(\mathfrak{s})$ would be trivial and, by the previous case, $\theta^{-1}(\theta(\mathfrak{s})) = \mathfrak{s}$ would also be trivial, which contradicts the assumption that $\mathfrak{s}$ is easy. Take any two leaf nodes, say $\mathfrak{p}_{1'}$ and $\mathfrak{p}_{2'}$, in the search tree $\mathcal{ST}(\theta(\mathfrak{s}), pg(\theta(\mathfrak{s})))$. By Corollary 7.39, $\theta^{-1}(\mathfrak{p}_{1'})$ and $\theta^{-1}(\mathfrak{p}_{2'})$ are leaf nodes in the search tree $\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{s}))$. Since $\mathfrak{s}$ is easy, there is a stabilizer θ_{stab} of $\mathfrak{s}$ such that $\theta_{\text{stab}}(\theta^{-1}(\mathfrak{p}_{1'})) = \theta^{-1}(\mathfrak{p}_{2'})$. Now $\theta * \theta_{\text{stab}} * \theta^{-1}$ is a stabilizer of $\theta(\mathfrak{s})$ and $(\theta * \theta_{\text{stab}} * \theta^{-1})(\mathfrak{p}_{1'}) = \theta(\theta_{\text{stab}}(\theta^{-1}(\mathfrak{p}_{1'}))) = \theta(\theta^{-1}(\mathfrak{p}_{2'})) = \mathfrak{p}_{2'}$. Thus $\theta(\mathfrak{s})$ is also easy.

If a state $\mathfrak{s}$ is hard, then the state $\theta(\mathfrak{s})$ must also be hard for any allowed domain permutation θ. For if $\theta(\mathfrak{s})$ were trivial (easy), then by the previous cases $\theta^{-1}(\theta(\mathfrak{s})) = \mathfrak{s}$ would also be trivial (easy), which contradicts the assumption that $\mathfrak{s}$ is hard. □

An algorithm is said to *produce a canonical representative for a state* $\mathfrak{s}$ if it holds that for all allowed domain permutations θ, the algorithm produces the same representative state for $\mathfrak{s}$ and $\theta(\mathfrak{s})$. That is, if two states are equivalent, then the algorithm will produce the same representative state for them.

Theorem 7.44 *If a state $\mathfrak{s}$ is trivial, then Algorithm 7.1 produces a canonical representative for it.*

Proof. Since $\mathfrak{s}$ is trivial, the partition $pg(\mathfrak{s})$ is a discrete partition, there is a unique allowed domain permutation $\hat{\theta}$ compatible with $pg(\mathfrak{s})$, and the unique representative state is $\hat{\theta}(\mathfrak{s})$. For any allowed domain permutation θ, $\theta(\mathfrak{s})$ is also trivial, the partition $pg(\theta(\mathfrak{s})) = \theta(pg(\mathfrak{s}))$ is discrete, $\hat{\theta} * \theta^{-1}$ is compatible with $\theta(pg(\mathfrak{s}))$ by Lemma 7.16, and the representative state for $\theta(\mathfrak{s})$ is $(\hat{\theta} * \theta^{-1})(\theta(\mathfrak{s})) = \hat{\theta}(\mathfrak{s})$. □

Theorem 7.45 *If a state $\mathfrak{s}$ is trivial or easy, then Algorithm 7.2 produces a canonical representative for it.*

Proof. The case in which $\mathfrak{s}$ is trivial follows directly from Theorem 7.44. Now assume that $\mathfrak{s}$ is easy.

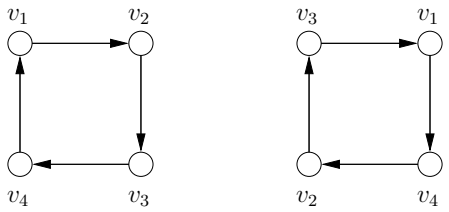


Figure 7.9: Two isomorphic graphs

First, it is shown that choosing any path in the search tree $\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{s}))$ leads to the same representative state for $\mathfrak{s}$. Take any two leaf nodes, say $\mathfrak{p}_1$ and $\mathfrak{p}_2$, in the search tree $\mathcal{ST}(\mathfrak{s}, pg(\mathfrak{s}))$. Since $\mathfrak{s}$ is easy, there is a stabilizer θ of $\mathfrak{s}$ mapping $\mathfrak{p}_1$ to $\mathfrak{p}_2$, i.e., $\theta(\mathfrak{p}_1) = \mathfrak{p}_2$. If $\hat{\theta}$ is the unique allowed domain permutation compatible with $\mathfrak{p}_1$, then $\hat{\theta} * \theta^{-1}$ is the unique allowed domain permutation compatible with $\theta(\mathfrak{p}_1) = \mathfrak{p}_2$ by Lemma 7.16. But now $(\hat{\theta} * \theta^{-1})(\mathfrak{s}) = \hat{\theta}(\theta^{-1}(\mathfrak{s})) = \hat{\theta}(\mathfrak{s})$ since θ^{-1} is a stabilizer of $\mathfrak{s}$ because θ is.

Because the state $\theta(\mathfrak{s})$ for any allowed domain permutation θ is also easy and the sets of possible representative states for $\mathfrak{s}$ and $\theta(\mathfrak{s})$ are the same, i.e., $posreps(\mathcal{ST}(\mathfrak{s})) = posreps(\mathcal{ST}(\theta(\mathfrak{s})))$, by Corollary 7.42, the same representative state is produced for both $\mathfrak{s}$ and $\theta(\mathfrak{s})$. □

7.4.4 A Sidetrack on Equivalence Testing of States

Consider the problem of determining whether two states, say $\mathfrak{s}$ and $\mathfrak{s}'$, are equivalent. Of course, given a *canonical* representative function, this task is easy: compute the canonical representatives of the two states in question and check whether they are equal. The other obvious (but highly inefficient) solution is to test for each allowed domain permutation θ whether $\theta(\mathfrak{s}) = \mathfrak{s}'$. It is now shown how this approach can be improved by using the techniques introduced in this section.

Assuming a partition generator pg, the definition of partition generators directly implies the following: if θ is an allowed domain permutation mapping a state $\mathfrak{s}$ to a state $\mathfrak{s}'$, then it must map the partition $pg(\mathfrak{s})$ to the partition $pg(\mathfrak{s}')$. Based on this, it is sufficient to test whether $\theta(\mathfrak{s}) = \mathfrak{s}$ only for those allowed domain permutations θ that map the partition $pg(\mathfrak{s})$ to the partition $pg(\mathfrak{s}')$. Of course, if the cell structures of the partitions differ, i.e., there is a primitive type T such that the partitions for it in $pg(\mathfrak{s})$ and $pg(\mathfrak{s}')$ differ in the number of cells or in the size of the corresponding cells, it can be directly concluded that there are no allowed domain permutations mapping $pg(\mathfrak{s})$ to $pg(\mathfrak{s}')$ and thus $\mathfrak{s}$ and $\mathfrak{s}'$ are not equivalent. This approach of testing whether two states are equivalent is not new, but has already been used in e.g. [Jensen 1995; Lorentsen 2002]. However, the invariants described earlier in this chapter, needed in building the partitions, are more powerful than those used in [Jensen 1995; Lorentsen 2002]. A similar approach is also taken in [Sistla et al. 2000], where symmetry-respecting signatures (partitions) are first built for states to be tested and then random permutations mapping the signatures to each other are generated to find out whether there is a permutation mapping the states to each other. That is, an incomplete probabilistic algorithm is used.

Example 7.46 Consider a system that has a state variable G (for graph) of type $\mathsf{Set}(\mathsf{Struct}(\mathsf{Vertices}, \mathsf{Vertices}))$, where $\mathsf{Vertices}$ is an unordered primitive type with the domain $\mathscr{D}_{\mathsf{Vertices}} = \{v_1, v_2, v_3, v_4\}$. Take the states

$$\mathfrak{s} = \{G \mapsto \{\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_1\rangle\}\}$$

and

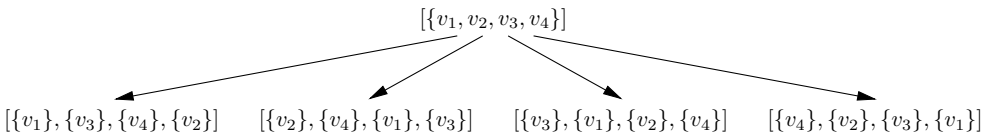
$$\mathfrak{s}' = \{G \mapsto \{\langle v_3, v_1\rangle, \langle v_1, v_4\rangle, \langle v_4, v_2\rangle, \langle v_2, v_3\rangle\}\}$$

corresponding to the directed graphs shown in Figure 7.9. The states are equivalent since $\theta = \left\{\theta^{\mathsf{Vertices}} = \left(\begin{smallmatrix} v_1 & v_2 & v_3 & v_4 \\ v_3 & v_1 & v_4 & v_2 \end{smallmatrix}\right)\right\}$ maps $\mathfrak{s}$ to $\mathfrak{s}'$. After applying *any* partition generator *pg* to the states, it must be that the partition for $\mathsf{Vertices}$ is $\mathfrak{p}^{\mathsf{Vertices}} = [\{v_1, v_2, v_3, v_4\}]$ in both partitions $pg(\mathfrak{s})$ and $pg(\mathfrak{s}')$. This follows from Fact 7.19 by observing that the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$ is generated by $\left(\theta^{\mathsf{Vertices}} = \left(\begin{smallmatrix} v_1 & v_2 & v_3 & v_4 \\ v_2 & v_3 & v_4 & v_1 \end{smallmatrix}\right)\right)$ while $\mathrm{Stab}(\Theta, \mathfrak{s}')$ is generated by $\left(\theta^{\mathsf{Vertices}} = \left(\begin{smallmatrix} v_1 & v_2 & v_3 & v_4 \\ v_4 & v_3 & v_1 & v_2 \end{smallmatrix}\right)\right)$.

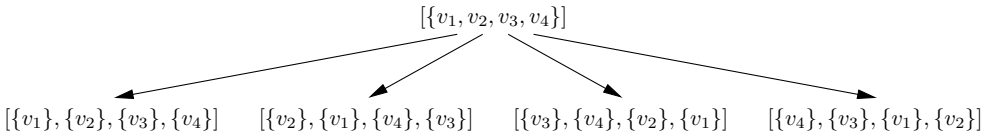
There are $4! = 24$ allowed domain permutations mapping the partition $\mathfrak{p}^{\mathsf{Vertices}}$ in $pg(\mathfrak{s})$ to the (same) partition $\mathfrak{p}^{\mathsf{Vertices}}$ in $pg(\mathfrak{s}')$. However, by Theorem 7.3, there are only $|\mathrm{Stab}(\Theta, \mathfrak{s})| = 4$ allowed domain permutations mapping $\mathfrak{s}$ to $\mathfrak{s}'$. This example can be extended to graphs with n vertices in which there are $n!$ allowed domain permutations mapping the partitions to each other but only n of them mapping the states to each other. Thus $n! - n$ of $n!$, i.e., almost all allowed domain permutations will fail in the equivalence testing approach described above. ♣

The above equivalence test can be improved by using search trees. Assume two states, $\mathfrak{s}_1$ and $\mathfrak{s}_2$. Take any path $pg(\mathfrak{s}_1) \xrightarrow{T_1, v_{1,1}} \mathfrak{p}_{1,1} \ldots \xrightarrow{T_n, v_{1,n}} \mathfrak{p}_{1,n}$ in the search tree $\mathcal{ST}(\mathfrak{s}_1)$ ending in a discrete partition $\mathfrak{p}_{1,n}$. Let $\hat{\theta}_1$ be the allowed domain permutation compatible with the leaf partition $\mathfrak{p}_{1,n}$. If $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are equivalent, then there is an allowed domain permutation θ mapping $\mathfrak{s}_1$ to $\mathfrak{s}_2$ and consequently (by Corollary 7.39) a leaf node $\theta(\mathfrak{p}_{1,n})$ in the search tree $\mathcal{ST}(\mathfrak{s}_2, pg(\mathfrak{s}_2))$. Then by Lemma 7.16, $\hat{\theta}_1 * \theta^{-1}$ is compatible with $\theta(\mathfrak{p}_{1,n})$ and $(\hat{\theta}_1 * \theta^{-1})(\mathfrak{s}_2) = (\hat{\theta}_1 * \theta^{-1})(\theta(\mathfrak{s}_1)) = \hat{\theta}_1(\mathfrak{s}_1)$. Furthermore, if $\hat{\theta}_2$ is compatible with a discrete partition $\mathfrak{p}_2$ in the search tree $\mathcal{ST}(\mathfrak{s}_2, pg(\mathfrak{s}_2))$ and $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\mathfrak{s}_2)$, then $(\hat{\theta}_2^{-1} * \hat{\theta}_1)(\mathfrak{s}_1) = \mathfrak{s}_2$ and the states are equivalent. Therefore, in order to check whether $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are equivalent, perform a backtracking search in the search tree $\mathcal{ST}(\mathfrak{s}_2)$ starting from the root node to find whether there is a leaf node $\mathfrak{p}_2$ in it such that the allowed domain permutation $\hat{\theta}_2$ compatible with $\mathfrak{p}_2$ maps $\mathfrak{s}_2$ to $\hat{\theta}_1(\mathfrak{s}_1)$. The states $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are equivalent if and only if such a leaf node can be found. To prune the search tree, note that if θ maps $\mathfrak{s}_1$ to $\mathfrak{s}_2$, then by Corollary 7.37 there is a path $\theta(pg(\mathfrak{s}_1)) \xrightarrow{T_1, \theta^{T_1}(v_{1,1})} \theta(\mathfrak{p}_{1,1}) \ldots \xrightarrow{T_n, \theta^{T_n}(v_{1,n})} \theta(\mathfrak{p}_{1,n})$ in the search tree $\mathcal{ST}(\mathfrak{s}_2)$ and by Lemma 7.16, $\hat{\theta}_1 * \theta^{-1}$ is compatible with $\theta(\mathfrak{p}_{1,n})$ and $(\hat{\theta}_1 * \theta^{-1})(\mathfrak{s}_2) = (\hat{\theta}_1 * \theta^{-1})(\theta(\mathfrak{s}_1)) = \hat{\theta}_1(\mathfrak{s}_1)$. Therefore, if a path $pg(\mathfrak{s}_2) \xrightarrow{T_1, v_{2,1}} \mathfrak{p}_{2,1} \ldots \xrightarrow{T_k, v_{2,k}} \mathfrak{p}_{2,k}$, $k < n$, is currently being traversed in the search tree $\mathcal{ST}(\mathfrak{s}_2)$, and the cell structures of $\mathfrak{p}_{1,k}$ and $\mathfrak{p}_{2,k}$ differ (meaning that there cannot be any θ mapping $\mathfrak{p}_{1,k}$ to $\mathfrak{p}_{2,k}$), there is no need to traverse the children of the node $\mathfrak{p}_{2,k}$. Naturally, this algorithm can be made probabilistic by traversing the paths in the search tree $\mathcal{ST}(\mathfrak{s}_2)$ randomly.

Example 7.47 (Example 7.46 continued) Applying any reasonably efficient invariants, such as those described in Section 7.3.2, in the partition refiner $\mathcal{R}$ will make the search tree for the state $\mathfrak{s}$ to look something like this:



and the search tree for the state $\mathfrak{s}'$ is thus:



Now the domain permutation $\hat{\theta}_1 = \left(\hat{\theta}_1^{\mathsf{Vertices}} = \left(\begin{smallmatrix} v_1 & v_2 & v_3 & v_4 \\ v_1 & v_4 & v_2 & v_3 \end{smallmatrix}\right)\right)$ compatible with the leftmost leaf node $\left(\mathfrak{p}^{\mathsf{Vertices}} = [\{v_1\}, \{v_3\}, \{v_4\}, \{v_2\}]\right)$ of the search tree for $\mathfrak{s}$ maps $\mathfrak{s}$ to $\hat{\theta}_1(\mathfrak{s}) = \{G \mapsto \{\langle v_1, v_4\rangle, \langle v_4, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_1\rangle\}\}$. Now taking *any* leaf node in the search tree for $\mathfrak{s}'$, the allowed domain permutation compatible with it maps $\mathfrak{s}'$ to $\hat{\theta}_1(\mathfrak{s})$. For instance, the allowed domain permutation $\hat{\theta}_2 = \left(\hat{\theta}_2^{\mathsf{Vertices}} = \left(\begin{smallmatrix} v_1 & v_2 & v_3 & v_4 \\ v_2 & v_1 & v_4 & v_3 \end{smallmatrix}\right)\right)$ compatible with the leaf node $\left(\mathfrak{p}^{\mathsf{Vertices}} = [\{v_2\}, \{v_1\}, \{v_4\}, \{v_3\}]\right)$ of the search tree for $\mathfrak{s}'$ maps $\mathfrak{s}'$ to $\hat{\theta}_2(\mathfrak{s}') = \{G \mapsto \{\langle v_4, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_1\rangle, \langle v_1, v_4\rangle\}\} = \hat{\theta}_1(\mathfrak{s})$. ♣

As the above example shows, using search trees can bring exponential savings in the state equivalence test approach.

7.5 HANDLING LARGE AND INFINITE UNORDERED PRIMITIVE TYPES

So far, it has been assumed that the domains of unordered primitive types (scalar sets) are finite. However, in some cases it would be convenient to have unordered primitive types with very large or even infinite domains. For instance, modeling unbounded resources such as process identifiers would require the domain to be infinite. Without restrictions, infinite domains cause problems in algorithms presented above because partitions are assumed to be ordered lists of subsets of domains. For instance, if a partition contains two cells that have infinitely many elements and an invariant would distinguish infinitely many elements in both of these cells, then the partition refined according to the invariant would result in an ordered list that first has infinitely many cells (refined from the first original cell) and *after* that, yet infinitely many cells (refined from the second original cell). This is absurd and would require partitions to be something else than ordered lists or redefinition of the invariant partitioning process. Likewise, having a cell with infinitely many elements not as the last cell invalidates the Definition 7.15 of compatible allowed domain permutations.

However, these problem can be circumvented by assuming *finite states* in the sense that *only finitely many elements in the infinite domain of each unordered primitive type actually appear in a given state*. This is a plausible assumption since infinitely many elements appearing in a state would also

cause some other problems, starting with the problem of how to represent states. Now consider the allowed domain permutation θ that only swaps two elements v and v' of type T *not* appearing in the state $\mathfrak{s}$. Clearly $\theta(\mathfrak{s}) = \mathfrak{s}$ and $pg(\theta(\mathfrak{s})) = \theta(pg(\mathfrak{s}))$ implies $pg(\mathfrak{s}) = \theta(pg(\mathfrak{s}))$, meaning that the elements v and v' must belong to the same cell in the partition assigned to $\mathfrak{s}$ by any partition generator pg. Therefore, partition generators cannot distinguish between the elements not appearing in a state. Furthermore, if θ' maps the state $\mathfrak{s}$ to $\mathfrak{s}'$, then $\theta' * \theta$ also maps $\mathfrak{s}$ to $\mathfrak{s}'$ meaning that it does not matter how the non-appearing elements are permuted among themselves. Thus one may conclude that *the elements of unordered primitive types that do not appear in the state in question can be ignored.* An algorithmic view of this is to first apply the following invariant for each unordered primitive type T during the computation of the partition generator.

Definition 7.48 *The invariant $I_{T,\text{appears}}(v, \mathfrak{s}, \mathfrak{p})$ is defined to be 0 if the element v of a type T appears in the value of any state variable in the state $\mathfrak{s}$ (meaning that $I_{\sharp T \text{ in } x}(v, \mathfrak{s}, \mathfrak{p}) \geq 1$ for a $x \in \mathcal{X}$), and 1 otherwise.*

This splits the elements in the domain of T in two cells: those that appear in the state $\mathfrak{s}$ (a finite set under the assumption made above) and those that do not (an infinite set). The latter cell is then ignored. Because it was chosen that the elements appearing in the state are assigned the value 0 by $I_{T,\text{appears}}$ (i.e., have a smaller value than those not appearing in the state), the n elements appearing in the state are in the first cell and are thus "compressed" to be the first n elements in the domain by any allowed domain permutation compatible with the partition produced this way.

Another view of the same idea is to first apply an allowed domain permutation that "compresses" the elements appearing in the domains of infinite unordered primitive types and then use the algorithms for finite domains described previously without modification. That is, for a finite state $\mathfrak{s}_1$, take any allowed domain permutation θ_1 that, for each infinite unordered primitive type T, maps the n elements in $\mathscr{D}_T$ appearing in the state $\mathfrak{s}_1$ to the first n elements in the domain $\mathscr{D}_T$. Similarly for another finite state $\mathfrak{s}_2$. Now the states $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are equivalent if and only if $\mathfrak{s}'_1 = \theta_1(\mathfrak{s}_1)$ and $\mathfrak{s}'_2 = \theta_2(\mathfrak{s}_2)$ are equivalent and if they are, there is an allowed domain permutation that (i) maps $\theta_1(\mathfrak{s}_1)$ to $\theta_2(\mathfrak{s}_2)$ and (ii) for each infinite unordered primitive type fixes all the elements not appearing in $\theta_1(\mathfrak{s}_1)$ or in $\theta_2(\mathfrak{s}_2)$. One can now reduce the domains of all infinite unordered primitive types to finite sets consisting only of the elements that appear in the state $\theta_1(\mathfrak{s}_1)$ or in $\theta_2(\mathfrak{s}_2)$. Now $\theta_1(\mathfrak{s}_1)$ and $\theta_2(\mathfrak{s}_2)$ are equivalent under the allowed domain permutation group for the reduced domains if and only if they are under the original allowed domain permutation group. Furthermore, if $\theta_{1'}(\mathfrak{s}'_1) = \theta_{2'}(\mathfrak{s}'_2)$, where $\theta_{1'}$ and $\theta_{2'}$ are allowed domain permutations under the reduced domains, then $\theta_{1'}(\theta_1(\mathfrak{s}_1)) = \theta_{2'}(\theta_2(\mathfrak{s}_2))$ when $\theta_{1'}$ and $\theta_{2'}$ are interpreted as if they were allowed domain permutations for the unreduced domains. Thus the (canonical) representatives computed under the reduced domains can be directly used as (canonical) representatives for the original states.

7.6 ALGORITHMS BASED ON CHARACTERISTIC GRAPHS

It is now illustrated how characteristic graphs of states described in Section 7.2 can be used for deciding whether two states are equivalent and for building a canonical representative for a state. The approach presented here is quite similar to the one described in Section 4.2 for place/transition nets.

Assuming an algorithm for deciding whether two vertex labeled, edge weighted directed graphs are isomorphic, the obvious algorithm for deciding whether two states are equivalent under the group Θ of all allowed domain permutations is

1. to build the characteristic graphs $\mathcal{G}_{\mathfrak{s}}$ and $\mathcal{G}_{\mathfrak{s}'}$ for the two states $\mathfrak{s}$ and $\mathfrak{s}'$ in question, and
2. then check whether $\mathcal{G}_{\mathfrak{s}}$ and $\mathcal{G}_{\mathfrak{s}'}$ are isomorphic.

In the case of a graph isomorphism algorithm only supporting a weaker form of graphs, say vertex labeled undirected graphs, one has to transform the characteristic graphs into that graph class by replacing edges with additional, appropriately labeled vertices (as illustrated in Section 4.2).

It is now shown how to obtain a canonical representative function for states, provided that a canonizer for graphs is available (compare with the approach taken in Section 4.2). Recall that a *canonizer for graphs* is formally a function $\mathcal{K}$ from graphs to graphs such that

1. for each graph G, G and $\mathcal{K}(G)$ are isomorphic, and
2. if two graphs G and G' are isomorphic, then $\mathcal{K}(G) = \mathcal{K}(G')$.

The graph $\mathcal{K}(G)$ is called the *canonical version* of G. Furthermore, it is assumed that the graph canonizer produces graphs that have the vertex set drawn from $\{1, 2, \ldots\}$. That is, if G has a finite vertex set V, then the canonical version $\mathcal{K}(G)$ has the vertex set $\{1, 2, \ldots, |V|\}$. In addition, it is assumed that an isomorphism κ from G to $\mathcal{K}(G)$ is provided.

A graph canonizer $\mathcal{K}$ is extended to $\mathcal{K}_{\mathcal{S}}$ operating on states as follows. Given a state $\mathfrak{s}$, consider its characteristic graph $\mathcal{G}_{\mathfrak{s}}$. Assume that κ is a mapping from the vertices of $\mathcal{G}_{\mathfrak{s}}$ to the vertices of its canonical version $\mathcal{K}(\mathcal{G}_{\mathfrak{s}})$. Take the allowed domain permutation $\theta = \{\theta^T\}_{T \in \mathcal{T}_P}$ that is *compatible* with κ, meaning that the following rules are fulfilled.

- For each cyclic primitive type T with $\mathscr{D}_T = \{v_0, \ldots, v_{n-1}\}$, consider the set $\{\kappa(T::v) \mid v \in \mathscr{D}_T\}$ of κ-images of the nodes in the characteristic graph corresponding to the elements in the domain of T. Now θ^T is the one that maps the element $v \in \mathscr{D}_T$ having the smallest value $\kappa(T::v)$ in the set to v_0.
- For each unordered primitive type T with $\mathscr{D}_T = \{v_1, \ldots, v_n\}$, θ^T is the one that maps an element $v \in \mathscr{D}_T$ to v_i if and only if v has the ith smallest value $\kappa(T::v)$ in the set $\{\kappa(T::v) \mid v \in \mathscr{D}_T\}$.

Denote the state $\theta(\mathfrak{s})$ by $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$. The whole algorithm is shown in Algorithm 7.3. The next theorem establishes the correctness of the algorithm.

Theorem 7.49 *The function $\mathcal{K}_{\mathcal{S}}$ is a canonical representative function.*

Algorithm 7.3 A canonical representative algorithm based on characteristic graphs

Input: A state $\mathfrak{s}$
Output: A canonical representative state for $\mathfrak{s}$
Require: A graph canonizer $\mathcal{K}$
1: Build the characteristic graph $\mathcal{G}_{\mathfrak{s}}$
2: Compute a mapping κ from $\mathcal{G}_{\mathfrak{s}}$ to its canonical version $\mathcal{K}(\mathcal{G}_{\mathfrak{s}})$
3: Take the allowed domain permutation θ that is compatible with κ
4: Return $\theta(\mathfrak{s})$ as the canonical representative state

Proof. Obviously, for any state $\mathfrak{s}$, $\mathfrak{s}$ and $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$ are equivalent since $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$ is obtained from $\mathfrak{s}$ by using an allowed domain permutation.

Assume two equivalent states, $\mathfrak{s}_1$ and $\mathfrak{s}_2$. It now has to be proven that $\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_1) = \mathcal{K}_{\mathcal{S}}(\mathfrak{s}_2)$. Take

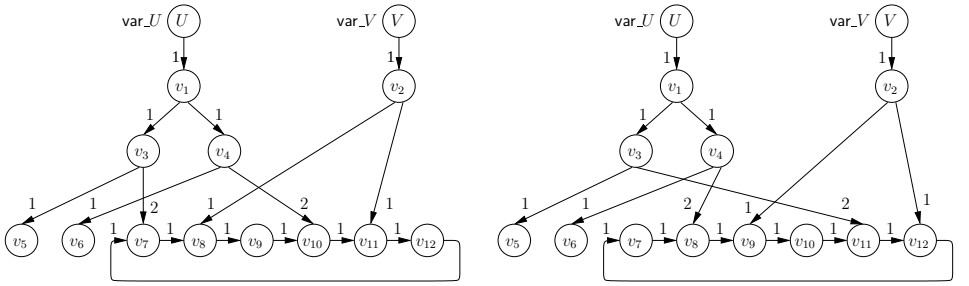
1. the characteristic graphs $\mathcal{G}_{\mathfrak{s}_1} = \langle V_1, \ldots \rangle$ and $\mathcal{G}_{\mathfrak{s}_2} = \langle V_2, \ldots \rangle$ (which are isomorphic since $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are equivalent),
2. their canonical versions $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_1})$ and $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$ (which are equal since $\mathcal{G}_{\mathfrak{s}_1}$ and $\mathcal{G}_{\mathfrak{s}_2}$ are isomorphic),
3. any isomorphism κ_1 from $\mathcal{G}_{\mathfrak{s}_1}$ to $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_1})$ and any isomorphism κ_2 from $\mathcal{G}_{\mathfrak{s}_2}$ to $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$, and
4. the two allowed domain permutations $\theta_{\kappa_1} = \left\{\theta^T_{\kappa_1}\right\}_{T \in \mathcal{T}_P}$ and $\theta_{\kappa_2} = \left\{\theta^T_{\kappa_2}\right\}_{T \in \mathcal{T}_P}$ that are compatible with κ_1 and κ_2, respectively.

Showing that $\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_1) = \mathcal{K}_{\mathcal{S}}(\mathfrak{s}_2)$ equals to showing that $\theta_{\kappa_1}(\mathfrak{s}_1) = \theta_{\kappa_2}(\mathfrak{s}_2)$. For this, it suffices to prove that ${\theta_{\kappa_2}}^{-1} * \theta_{\kappa_1} = \left\{{\theta^T_{\kappa_2}}^{-1} \circ \theta^T_{\kappa_1}\right\}_{T \in \mathcal{T}_P}$ maps $\mathfrak{s}_1$ to $\mathfrak{s}_2$. First, note that ${\kappa_2}^{-1} \circ \kappa_1$ is an isomorphism from the characteristic graph $\mathcal{G}_{\mathfrak{s}_1}$ to the characteristic graph $\mathcal{G}_{\mathfrak{s}_2}$. By Fact 7.10, there is an allowed domain permutation $\theta = \left\{\theta^T\right\}_{T \in \mathcal{T}_P}$ mapping $\mathfrak{s}_1$ to $\mathfrak{s}_2$ such that for each permutable primitive type T and each $v \in \mathscr{D}_T$ it holds that $({\kappa_2}^{-1} \circ \kappa_1)(T{::}v) = T{::}v' \Leftrightarrow \theta^T(v) = v'$. It now suffices to show that ${\theta_{\kappa_2}}^{-1} * \theta_{\kappa_1} = \theta$. Also notice that for any permutable primitive type T, the image set $\kappa_1(\{T{::}v \mid v \in \mathscr{D}_T\})$ of the nodes in the characteristic graph $\mathcal{G}_{\mathfrak{s}_1}$ corresponding to the elements of the type must equal to the image set $\kappa_2(\{T{::}v \mid v \in \mathscr{D}_T\})$ of the nodes in the characteristic graph $\mathcal{G}_{\mathfrak{s}_2}$ since the isomorphisms κ_1 and κ_2 must respect node types. The following two cases must be considered.

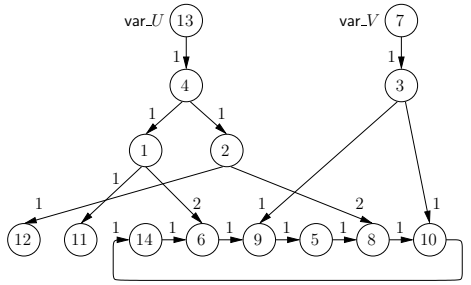
1. Let T be a cyclic primitive type with $\mathscr{D}_T = \{v_0, \ldots, v_{n-1}\}$. Let $v' \in \mathscr{D}_T$ be the element for which $\kappa_1(T{::}v') = \min_{v \in \mathscr{D}_T} \kappa_1(T{::}v)$. Similarly, let $v'' \in \mathscr{D}_T$ be the element for which $\kappa_2(T{::}v'') = \min_{v \in \mathscr{D}_T} \kappa_2(T{::}v)$. Therefore, $\theta^T_{\kappa_1}(v') = v_0 = \theta^T_{\kappa_2}(v'')$ and ${\theta^T_{\kappa_2}}^{-1} \circ \theta^T_{\kappa_1}$ is the one that maps v' to v''. But now also $\kappa_1(T{::}v') = \kappa_2(T{::}v'')$, meaning that $({\kappa_2}^{-1} \circ \kappa_1)(T{::}v') = T{::}v''$ and thus θ^T must equal to ${\theta^T_{\kappa_2}}^{-1} \circ \theta^T_{\kappa_1}$.
2. Assume that T is an unordered primitive type with $\mathscr{D}_T = \{v_1, \ldots, v_n\}$. Let $v' \in \mathscr{D}_T$ be the element having the ith smallest value $\kappa_1(T{::}v')$ among the vertices of form $T{::}v$ in the vertex set set V_1. Similarly, let $v'' \in \mathscr{D}_T$ be the element having the ith smallest value $\kappa_2(T{::}v'')$ among the vertices of form $T{::}v$ in the vertex set set V_2. Therefore,

$\theta^T_{\kappa_1}(v') = v_i = \theta^T_{\kappa_2}(v'')$ and ${\theta^T_{\kappa_2}}^{-1} \circ \theta^T_{\kappa_1}$ maps v' to v''. But now also $\kappa_1(T::v') = \kappa_2(T::v'')$ and thus $({\kappa_2}^{-1} \circ \kappa_1)(T::v') = T::v''$.

□



(a) $\mathcal{G}_{\mathfrak{s}_1}$ (b) $\mathcal{G}_{\mathfrak{s}_2}$



(c) $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_1}) = \mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$

Figure 7.10: Two characteristic graphs and their common canonical version

Example 7.50 Recall the net in Figure 7.1, discussed in Example 7.1. Consider the states $\mathfrak{s}_1 = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$ and $\mathfrak{s}_2 = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}$. The states are equivalent since both

$$\theta_1 = \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right)$$

and

$$\theta_2 = \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right)$$

map $\mathfrak{s}_1$ to $\mathfrak{s}_2$. The characteristic graphs $\mathcal{G}_{\mathfrak{s}_1}$ and $\mathcal{G}_{\mathfrak{s}_2}$ of the states are depicted in Figures 7.10(a) and 7.10(b), respectively. In the figures, the following common abbreviations for vertex names are used: $v_5 = \mathsf{Trains}::t_a$, $v_6 = \mathsf{Trains}::t_b$, $v_7 = \mathsf{Secs}::s_0$, $v_8 = \mathsf{Secs}::s_1$, $v_9 = \mathsf{Secs}::s_2$, $v_{10} = \mathsf{Secs}::s_3$, $v_{11} = \mathsf{Secs}::s_4$, and $v_{12} = \mathsf{Secs}::s_5$. Assume that a graph canonizer produces the canonical version $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_1}) = \mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$ shown in Figure 7.10(c) for these characteristic graphs.

There are two isomorphisms from $\mathcal{G}_{\mathfrak{s}_1}$ to $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_1})$:

$$\begin{aligned} \kappa_{1,1} &= \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 1 & 2 & 11 & 12 & 6 & 9 & 5 & 8 & 10 & 14 \end{smallmatrix}\right) \text{ and} \\ \kappa_{1,2} &= \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 2 & 1 & 12 & 11 & 8 & 10 & 14 & 6 & 9 & 5 \end{smallmatrix}\right). \end{aligned}$$

The two allowed domain permutations compatible with these isomorphisms are

$$\begin{aligned} \theta_{\mathfrak{s}_1,1} &= \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_1,1} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_1,1} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) \text{ and} \\ \theta_{\mathfrak{s}_1,2} &= \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_1,2} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_1,2} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \end{aligned}$$

respectively. The canonical representative state for $\mathfrak{s}_1$ is thus

$$\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_1) = \theta_{\mathfrak{s}_1,1}(\mathfrak{s}_1) = \theta_{\mathfrak{s}_1,2}(\mathfrak{s}_1) = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}.$$

Similarly, there are two isomorphisms from $\mathcal{G}_{\mathfrak{s}_2}$ to $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$:

$$\begin{array}{rcl} \kappa_{2,1} & = & \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 1 & 2 & 11 & 12 & 5 & 8 & 10 & 14 & 6 & 9 \end{smallmatrix}\right) \text{ and} \\ \kappa_{2,2} & = & \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 2 & 1 & 12 & 11 & 14 & 6 & 9 & 5 & 8 & 10 \end{smallmatrix}\right). \end{array}$$

The two allowed domain permutations compatible with these isomorphisms are

$$\begin{array}{rcl} \theta_{\mathfrak{s}_2,1} & = & \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_2,1} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_2,1} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) \text{ and} \\ \theta_{\mathfrak{s}_2,2} & = & \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_2,2} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_2,2} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \end{array}$$

respectively. The canonical representative state for $\mathfrak{s}_2$ is thus

$$\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_2) = \theta_{\mathfrak{s}_2,1}(\mathfrak{s}_2) = \theta_{\mathfrak{s}_2,2}(\mathfrak{s}_2) = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}$$

equaling to $\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_1)$. ♣

Finally, note that Algorithms 7.1 and 7.3 can be combined as follows. Given a state $\mathfrak{s}$, first compute the partition $pg(\mathfrak{s})$ for it by using a fixed partition generator pg. If $pg(\mathfrak{s})$ is discrete, then return the state $\hat{\theta}(\mathfrak{s})$ as the canonical representative state, where $\hat{\theta}$ is the allowed domain permutation compatible with $pg(\mathfrak{s})$. If $pg(\mathfrak{s}) = \{\mathfrak{p}^T\}_{T \in \mathcal{T}_P}$ is not discrete, build the characteristic graph $\mathcal{G}_{\mathfrak{s}}$. Then change the label of each vertex of form $T::v$ for a permutable primitive type T from T to $T.incell(\mathfrak{p}^T, v)$, and proceed to the line 2 of Algorithm 7.3.

7.7 SOME EXPERIMENTAL RESULTS

The Algorithms 7.1, 7.2, and 7.3 proposed in this chapter have been implemented in the version 3.1 of the Murφ tool [Dill 1996]. The source code for the extended Murφ is available via

`http://www.tcs.hut.fi/~tjunttil/`

The Murφ tool has been selected for the experiments for two reasons:

1. Murφ already includes some symmetry reduction algorithms described in [Ip and Dill 1996; Ip 1996]. Thus Murφ already contains some common routines needed in the symmetry reduction method, for instance, routines for permuting the states. This makes the implementation of new algorithms easier. Furthermore, the already implemented algorithms offer a good benchmark for new algorithms.
2. The standard Murφ distribution includes some complex, "real-life" system descriptions exhibiting symmetry.

This section presents some experimental results on the example systems contained in the Murφ distribution as well as on some others.

The original Murφ tool has four representative algorithms, described by the help page of the tool and in [Ip 1996]. The first one simply applies all the allowed domain permutations to the state in question and returns the smallest state obtained as the canonical representative state. The other three algorithms first build an ordered partition as in Section 7.3 by using some invariants and then apply all, 10, or 1, respectively, domain permutations compatible with the partition to the state and return the smallest state found this way as the representative state. The Algorithm 7.1 in this chapter is basically the last Murφ algorithm except that more powerful invariants for building the partition are used. In more detail, the partition generator in Algorithms 7.1 and 7.2 is obtained as follows. First, the invariants for ordered structured types described in Section 7.3.2 are applied on state variables if possible. Then, for the other state variables, the hash-like invariants described in Section 7.3.2 are applied until no refinement occurs. In Algorithm 7.2, the applied partition refiner is produced by refining with the hash-like invariants.

In addition to the example systems in the Murφ distribution, the following graph enumeration systems inspired by the proof of Theorem 3.4 in [Ip 1996] are used. Figure 7.11 shows a Murφ program called graphs5.m. It

```
const
 nof_vertices: 5;
type
 Vertex: scalarset(nof_vertices);
var
 edges: Array[Vertex] of Array[Vertex] of boolean;
Startstate
 Begin
  for i:Vertex do for j:Vertex do
   if(i!=j) then edges[i][j] := true; else edge s[i][j]:=false; end;
  end; end;
 End;

Ruleset i:Vertex do
Ruleset j:Vertex do
 Rule "Delete edge"
  edges[i][j]=TRUE ==> edges[i][j] := FALSE; edges[j][i] := FALSE;
 EndRule;
EndRuleset;
EndRuleset;

Invariant "dummy"
  TRUE
```

Figure 7.11: A system enumerating undirected graphs

has the unordered primitive type (scalar set) called Vertex with the domain of size 5 for the vertices of a graph, and one state variable called *edges* of type AssocArray(Vertex, AssocArray(Vertex, Bool)) with the intuition that each vertex is associated with each vertex and a Boolean value describing whether there is an edge from the first vertex to the second one. The initial state is such that all the edges except self-loops are in it. The transition (rule) "Delete Edge" then removes one (undirected) edge, meaning that the reachability graph of the system consists of all the self-loopless (undirected) graphs with 5 vertices. Consequently, a minimal symmetry reduced reacha-

bility graph consists of all such graphs *up to isomorphism*. Changing the rule "Delete Edge" into

```
Ruleset i:Vertex do
Ruleset j:Vertex do
  Rule "Delete edge"
    edges[i][j]=TRUE ==> edges[i][j] := FALSE;
  EndRule;
EndRuleset;
EndRuleset;
```

results in the system called digraphs5.m, enumerating all the directed graphs of 5 vertices.

Table 7.1 shows the data of the experiments, run in an AMD Athlon 1GHz processor powered PC machine under the Linux operating system. The running times reported are in seconds. Note that the Murφ algorithms 1 and 2 as well as Algorithm 7.3 are canonical representative functions and thus the state columns for these algorithms give the minimum size of the reduced reachability graph.

As the Murφ examples (from adash to n_peterson) show, Algorithm 7.1 is quite fast and produces almost minimal reduced reachability graphs in these examples. In some cases it produces considerably smaller number of states than the original Murφ algorithm 4, which is due to the use of more powerful invariants, especially the hash-like invariants described in Section 7.3.2. Usually it slightly outperforms (in terms of generated states) even the Murφ algorithm 3 which has an advantage of trying 10 permutations instead of just selecting an arbitrary one. Interestingly, the Algorithm 7.2 produces minimal reduced reachability graphs for these instances although it is not a canonical representative algorithm. Furthermore, it is not significantly slower than Algorithm 7.1.

In the graph enumeration problems (graph$n$ and digraph$n$), the Algorithms 7.1 and 7.2 perform very well, producing reachability graphs that are reasonably close to the minimal ones. Again, especially the Algorithm 7.2 produces nearly optimal results in reasonably short time. The Murφ algorithms 2–4 do not perform very well because the invariants implemented in the standard Murφ tool cannot do anything in these systems. Note that although the number of states in the reduced reachability graphs can be very small, the number of times the representative function is called can be much bigger. For instance, on the problem instance graph8 the Algorithm 7.2 produces a reachability graph with 12376 states but with 346528 edges (executed transitions), meaning that the representative function is actually called 346528 times during the reachability graph generation.

The Algorithm 7.3 is also implemented by using the *nauty* tool (version 2.0 beta 9) [McKay 1990] as the graph canonizer. The bad running time results shown in Table 7.1 are probably due to the same reasons as described in Section 4.5.2. First, the characteristics graphs of states can be quite large. The graphs can be large to begin with, and in addition, as *nauty* is specially optimized for undirected graphs having no edge weights, some nodes have to be added in the graphs in order to use *nauty*. As an example, the *nauty* version of the characteristic graph of a state in the eadash instance has 2,768 vertices. Furthermore, the *nauty* tool is designed for dense graphs — the

graphs are represented as adjacency matrixes. Thus a characteristic graph of a state in the eadash instance takes almost one megabyte of memory to represent. This considerably slows down the partition refinement algorithms in *nauty*. Therefore, even though the search tree for the characteristic graph of a state in *nauty* is usually very small, it may take a lot of time to compute it. The results for Algorithm 7.3 might look quite different if a graph canonizer designed for directed, edge weighted, and sparse graphs were available.

system name	Murφ algorithm 1		Murφ algorithm 2		Murφ algorithm 3		Murφ algorithm 4	
	states	time	states	time	states	time	states	time
adash	10466	7	10466	7	10466	7	10471	7
cache3	31433	88	31433	8	31433	5	31433	5
eadash	133426	524	133426	374	133480	423	191088	378
ldash	254743	542	254743	403	254974	423	314194	447
mcslock1	23636	3	23636	3	23645	3	24668	3
mcslock2	540219	57	540219	63	540219	64	542071	61
list6	23410	7	23410	2	23410	2	23446	2
n_peterson	163298	5341	163298	42	163298	42	163298	42
digraphs3	16	1	16	1	16	1	64	1
digraphs4	218	1	218	1	554	1	4096	1
digraphs5	9608	111	9608	116	142113	392	>381000	>1h
graphs5	34	1	34	1	183	1	1024	1
graphs6	156	34	156	23	5408	12	32768	14
graphs7	1044	1963	1044	2008	>105000	>1h	>141000	>1h
graphs8	>210	>1h	>210	>1h	>257000	>1h	>335000	>1h

system name	Algorithm 7.1		Algorithm 7.2		Algorithm 7.3	
	states	time	states	time	states	time
adash	10466	7	10466	7	10466	9766
cache3	31433	5	31433	5	31433	556
eadash	133439	312	133426	311	>200	>1h
ldash	254755	356	254743	354	>1030	>1h
mcslock1	23644	2	23636	2	23636	33
mcslock2	540220	47	540219	47	540219	735
list6	23410	2	23410	2	23410	62
n_peterson	163298	32	163298	35	163298	420
digraphs3	16	1	16	1	16	1
digraphs4	228	1	218	1	218	1
digraphs5	9832	5	9616	5	9608	54
graphs5	40	1	34	1	34	1
graphs6	243	1	156	1	156	5
graphs7	1683	5	1046	4	1044	63
graphs8	19601	99	12376	67	12346	1556

Table 7.1: Some experimental results

7.8 RELATED WORK

The algorithms in the Murφ tool were already discussed in the previous sections, see especially Section 7.7. The main difference between Algorithm 7.1 and the Murφ algorithms is that more powerful invariants are applied. Especially, the hash-like invariants in Section 7.3.2 are novel. Furthermore,

also cyclic primitive types (non-reflexive ring symmetries in the Murφ terminology) are handled in the same unified way. The Algorithms 7.2 and 7.3 presented in this chapter are novel.

In [Huber et al. 1985b; Jensen 1995; Lorentsen 2002], a very elementary version of the partition refinement process is given and applied to checking whether two markings of a colored Petri net are equivalent. Especially, no other structured types than those of form Multiset(T), where T is an unordered primitive type, are taken into account when refining partitions.

The approach taken in [Sistla et al. 2000] is discussed in Section 7.4.4.

Another kind of approach based on computational group theory is presented in [Lorentsen and Kristensen 2001]. The idea there is that, given a state $\mathfrak{s}$, first compute the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$ and then check all the $|\Theta|/|\mathrm{Stab}(\Theta, \mathfrak{s})|$ left coset representative permutations of $\mathrm{Stab}(\Theta, \mathfrak{s})$ in Θ and select the smallest state obtained as the canonical representative state. When $|\mathrm{Stab}(\Theta, \mathfrak{s})|$ is large, substantial savings can be obtained compared to the approach in which all the permutations in Θ are tested. However, whenever $|\mathrm{Stab}(\Theta, \mathfrak{s})|$ is very small, no such large savings are obtained; especially in systems in which most of the reachable states have the trivial stabilizer group, i.e., $|\mathrm{Stab}(\Theta, \mathfrak{s})| = 1$, all the permutations are tested in most of the cases. Note that the states for which $|\mathrm{Stab}(\Theta, \mathfrak{s})|$ is very small are also the states for which the symmetry reduction method has the largest reduction possibility: the number $|\Theta|/|\mathrm{Stab}(\Theta, \mathfrak{s})|$ of equivalent states that can be ignored is large. An advantage of this algorithm is that, as it gives the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$, some transitions starting from $\mathfrak{s}$ can be pruned away (never executed) because they will lead to equivalent successor states. However, note that computing the group $\mathrm{Stab}(\Theta, \mathfrak{s})$ is in general as hard as finding the automorphism group of a graph (a task for which no polynomial time algorithms are known). In [Lorentsen and Kristensen 2001], the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$ is basically found iteratively by letting $\Theta_1 = \mathrm{Stab}(\Theta, \mathfrak{s}(x_1))$, $\Theta_2 = \mathrm{Stab}(\Theta_1, \mathfrak{s}(x_2))$, ..., and $\Theta_n = \mathrm{Stab}(\Theta_{n-1}, \mathfrak{s}(x_n))$, where $x_1, \ldots, x_n$ are the state variables. Now $\Theta_n = \mathrm{Stab}(\Theta, \mathfrak{s})$. The backtracking algorithm presented in [Butler 1991] is used to compute each of the groups $\mathrm{Stab}(\Theta_i, \mathfrak{s}(x_{i+1}))$. However, one could compute the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$ and the lexicographically smallest state equivalent to $\mathfrak{s}$ (i.e., a canonical representative state for $\mathfrak{s}$) *at the same time* by the following procedure. Assuming the state variables $x_1, \ldots, x_n$, initialize the left coset $\theta_0 * \Theta_0$ to be $\mathbf{I} * \Theta$, where $\mathbf{I}$ is the identity domain permutation. Let θ_i be a domain permutation in the coset $\theta_{i-1} * \Theta_{i-1}$ that has minimal $\theta_i(\mathfrak{s}(x_i))$ (the domains of types are assumed to be totally ordered). The coset after the ith round is then $\theta_i * \Theta_i$, where $\Theta_i = \mathrm{Stab}(\Theta_{i-1}, \mathfrak{s}(x_i))$. Now $\theta_n * \Theta_n$ is a *canonical labeling coset*, where $\theta_n(\mathfrak{s})$ is the lexicographically smallest state equivalent to $\mathfrak{s}$ and $\Theta_n = \mathrm{Stab}(\Theta, \mathfrak{s})$. In this approach, a modified version of the backtracking algorithm presented in [Butler 1991] is not only used to compute each of the groups $\Theta_i = \mathrm{Stab}(\Theta_{i-1}, \mathfrak{s}(x_i))$ but also the domain permutation θ_i in the coset $\theta_{i-1} * \Theta_{i-1}$ that has minimal $\theta_i(\mathfrak{s}(x_i))$.

In [Chiola et al. 1991], an algorithm is presented for computing a symbolic representative marking for each encountered marking in the context of well-formed nets. To author's understanding, the symbolic representative for a marking there is in some sense the smallest equivalent marking aug-

mented with the information telling which elements of permutable primitive types can be freely interchanged (i.e., with the stabilizers of the marking that are produced by single transpositions of elements of permutable primitive types).

8 CONCLUSIONS

The following sums up the main achievements of this thesis.

The computational complexity of sub-tasks arising in the symmetry reduction method for place/transition nets is established. Finding the symmetries of a net is shown to be equivalent to the well-known problem of finding the automorphisms of a graph. The task of deciding whether two markings are equivalent under the symmetries is shown to be in general equivalent to the problem of deciding whether two graphs are isomorphic. Interestingly, this result holds independently of whether the symmetry group of the net is given as input. Finding the lexicographically greatest marking in the orbit of a marking (a canonical representative for the marking) is shown to be $\mathbf{FP}^{\mathbf{NP}}$-complete and thus equivalent to many classical optimization problems. It is also shown that deciding whether a marking symmetrically covers another is an **NP**-complete problem and that the symmetric coverability problem cannot be combined with the canonical representative approach in a straightforward way.

New algorithms for producing canonical representatives for markings of place/transition nets are described. The algorithms use a standard representation for permutation groups to store and search through the symmetries of a net. The first algorithm maps the marking to be canonized to a corresponding characteristic graph and then applies a black box graph canonizer to obtain a canonical form of the characteristic graph. The canonical representative of the marking is then derived from it. The second proposed algorithm is a variant of the backtracking search algorithms applied in computational group theory. It searches through the group representation of the symmetries of the net in order to find a canonical representative for the marking. The set of symmetries that have to be considered during the search is pruned (i) by applying a novel compatibility definition between the markings and symmetries, (ii) by using the best representative marking found earlier during the search, and (iii) by the stabilizers of the marking found during the search. The third algorithm combines the first and second ones by "opening" the black box graph canonizer of the first algorithm. This is done by first computing an ordered partition of the net elements for the marking to be canonized. This corresponds to the preprocessing step used in many algorithms for the graph isomorphism problem. The partition is then used to prune the search in the group presentation of net's symmetries (i.e., the second algorithm). The experimental results show that the proposed algorithms are competitive against the previous ones implemented in the *LoLA* tool. In addition to place/transition nets, the proposed algorithms could also be applied in explicit state model checking of other system formalisms in which the symmetries permute the components of the system but not the values of the components.

Data symmetries, i.e., symmetries that are produced by symmetric use of data values, of the high-level Petri net class of algebraic system nets are also studied. It is defined how the permutations of data values used in a net produce corresponding permutations in its state space. A sufficient condition for the arc and transition annotations appearing in the net is defined in order to

ensure that the produced state space permutations are actually symmetries. Because the complexity analysis shows that verifying the condition is computationally expensive, an approximation rule for the condition is also given. This developed theory is illustrated by defining a concrete high-level Petri net class of extended well-formed nets. The type system applied in such nets allows the use of many common high-level data structures such as sets, association arrays and lists. The symmetries of such nets can be found by using syntactical restrictions similar to those in well-formed nets and in the Murφ system. In addition to high-level Petri nets, the developed theory could also be applied when developing new system description formalisms involving high-level data structures. Especially, it allows one to analyze in the formalism definition phase what kind of data symmetries are compatible with the applied data manipulation operations.

New algorithms for the orbit problems under data symmetries are described. The studied framework covers the well-formed nets (both the extended and original ones), the Murφ system, as well as the most commonly used instances of colored Petri nets. The first algorithm family is based on building an ordered partitioning of the elements of the permutable primitive types appearing in a state in a symmetry-respecting way. The partitioning is then used to prune the set of symmetries that have to be considered when comparing whether two states are equivalent or when building a representative for the state. The difference to the similar work, e.g. [Jensen 1995; Ip 1996], is that (i) the partition building process is rigorously defined, (ii) both unordered and cyclic primitive types are handled in a uniform way, and (iii) also some very expressive invariants, needed in the partition building process, are proposed. Furthermore, a novel improvement based on building a partition refinement search tree, inspired by the algorithms for graph isomorphism checking and canonization, is proposed. The second proposed algorithm family is based on transforming states into corresponding characteristic graphs and then performing the equivalence checking and canonization on the graphs instead of states. This approach is similar to the one proposed for place/transition nets in this thesis. Some of the proposed algorithms are implemented in the Murφ tool and the experimental results show that they are competitive against the previous ones. The proposed approximation algorithms, returning a possibly non-canonical representative for a state, also seem to work quite well in the sense that they produce almost always canonical representatives in the experimented system instances. In addition to high-level Petri nets and the Murφ system, the proposed algorithms could also be applied to model checking of software systems as discussed e.g. in [Bošnački et al. 2002; Derepas and Gastin 2001; Lerda and Visser 2001; Iosif 2002] (see the discussion in Section 1.2). As an example, consider a system consisting of several concurrent processes, possibly of different types and each having a set of local variables. Furthermore, assume that there are some global variables as well as a shared memory with no pointer arithmetics allowed (e.g. a Java-like heap memory in which each memory location contains a structure or an array of elements). This kind of system can be easily interpreted as a system of the typed state variable form assumed by the proposed algorithms. First, an association array state variable associates each process identifier with the state of the process (a structure consisting of the type, program counter, and

local variables of the process). Secondly, the global variables are simply interpreted as state variables. And thirdly, the shared memory is interpreted as a state variable of an association array type, associating each memory location to its contents. Assuming that the process identifiers and memory locations can be permuted, i.e., are unordered primitive types, the proposed algorithms can now be applied for producing representative states.

8.1 FUTURE WORK

Some potential future research and implementation topics are listed below.

An Efficient Graph Canonizer for Sparse, Vertex and Edge Labeled Directed Graphs. In Sections 4.2 and 7.6, two quite similar approaches for producing canonical representative states are introduced. In the approaches, the state to be canonized is first transformed into the corresponding characteristic graph. A canonical version of the characteristic graph is then produced by applying a black box graph canonizer algorithm, and the canonical representative state is derived from the canonical version. The experimental results in Sections 4.5.2 and 7.7 show that the current state-of-the-art graph canonizer, the *nauty* tool, does not perform very well in the tested instances. This is because *nauty* seems to be designed and specially optimized for graphs that (i) are dense and undirected, and (ii) do not have edge labels or weights. It would be interesting to see how this characteristic graph approach would perform if a graph canonizer designed especially for sparse, directed, and edge labeled/weighted graphs were applied instead.

The Babai-Luks Algorithm for Place/Transition Nets. An alternative for producing canonical representative markings for place/transition nets not discussed in Chapter 4 is the string canonization algorithm in [Babai and Luks 1983]. The algorithm does the canonization orbit-wise, and also exploits the imprimitivity of groups. However, the algorithm seems to involve more complex permutation group algorithms and thus implementing it is left as a future challenge.

Combining Structural and Data Symmetries. In some cases, it would be necessary to combine structural symmetries (like those used in place/transition nets) with data symmetries (like those used in extended well-formed nets). For instance, the net in Figure 5.1(b) has both types of symmetry. The definitions and orbit problem algorithms for such "mixed" symmetries will probably resemble the fusion of those for place/transition nets and extended well-formed nets.

Practical Use of "Partial" Symmetries. As discussed in Section 1.2, there are some approaches concerning the use of the symmetry reduction method on systems that are only partially symmetric. For instance, in [Emerson and Trefler 1999; Emerson et al. 2000], conditions weaker than the standard state space symmetry condition, yet ensuring that applying the symmetry reduction method is sound, are defined. However, the issue of how to automat-

ically find system description level information that produces such partial symmetries is not addressed. The approach presented in [Sistla and Godefroid 2001], in which the transition constraints causing the asymmetry are tracked during the reduced reachability graph generation, is possibly easier to apply in practice. For instance, the approach could probably be applied to data symmetries of high-level Petri nets or the Murφ system by allowing the use of symmetry breaking (incompatible) operations on permutable primitive types. However, defining and implementing the approach in these formalisms may be a non-trivial task and requires further study.

Bibliography

AJAMI, K., HADDAD, S., AND ILIÉ, J.-M. 1998. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In *Tools and Algorithms for the Construction and Analysis of Systems; 4th International Conference, TACAS'98*, B. Steffen, Ed. Lecture Notes in Computer Science, vol. 1384. Springer, 52–67.

BABAI, L. 1994. Automorphism groups, isomorphism, reconstruction. Tech. Rep. TR-94-10, University of Chicago, Department of Computer Science. Also as chapter 27 of the *Handbook of Combinatorics*, North-Holland, 1995.

BABAI, L. AND LUKS, E. M. 1983. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. ACM, 171–183.

BARNER, S. AND GRUMBERG, O. 2002. Combining symmetry reduction and under-approximation for symbolic model checking. In *Computer Aided Verification: 14th International Conference, CAV 2002*, E. Brinksma and K. G. Larsen, Eds. Lecture Notes in Computer Science, vol. 2404. Springer, 93–106.

BILLINGTON, J. 1989. Many-sorted high-level nets. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM89)*. IEEE Computer Society Press, 166–179. Reprinted in [Jensen and Rozenberg 1991], pages 123–136.

BOŠNAČKI, D. 2002a. A nested depth first search algorithm for model checking with symmetry reduction. See Peled and Vardi [2002], 65–80.

BOŠNAČKI, D. 2002b. Partial order and symmetry reductions for discrete time. In *Proc. of Workshop on Real-Time Tools, RT-TOOLS 2002*, P. Petterson and W. Yi, Eds. Dept. of Information Technology, Uppsala University.

BOŠNAČKI, D., DAMS, D., AND HOLENDERSKI, L. 2000. Symmetric Spin. In *SPIN Model Checking and Software Verification: 7th International SPIN Workshop*, K. Havelund, J. Penix, and W. Visser, Eds. Lecture Notes in Computer Science, vol. 1885. Springer, 1–19.

BOŠNAČKI, D., DAMS, D., AND HOLENDERSKI, L. 2001. A heuristic for symmetry reductions with scalarsets. In *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliviera and P. Zave, Eds. Lecture Notes in Computer Science, vol. 2021. Springer, 518–533.

BOŠNAČKI, D., DAMS, D., AND HOLENDERSKI, L. 2002. Symmetric Spin. *International Journal on Software Tools for Technology Transfer 4*, 92–106.

BUTLER, G. 1991. *Fundamental Algorithms for Permutation Groups*. Lecture Notes in Computer Science, vol. 559. Springer.

CHIOLA, G., DUTHEILLET, C., FRANCESCHINIS, G., AND HADDAD, S. 1991. On well-formed coloured nets and their symbolic reachability graph. See Jensen and Rozenberg [1991], 373–396.

CHIOLA, G., DUTHEILLET, C., FRANCESCHINIS, G., AND HADDAD, S. 1993. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers 42,* 11 (Nov.), 1343–1360.

CHIOLA, G., DUTHEILLET, C., FRANCESCHINIS, G., AND HADDAD, S. 1997. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science 176,* 39–65.

CLARKE, E. M., EMERSON, E. A., JHA, S., AND SISTLA, A. P. 1998. Symmetry reductions in model checking. See Hu and Vardi [1998], 147–158.

CLARKE, E. M., ENDERS, R., FILKORN, T., AND JHA, S. 1996. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design 9,* 1/2 (Aug.), 77–104.

CLARKE, E. M., FILKORN, T., AND JHA, S. 1993. Exploiting symmetry in temporal logic model checking. See Courcoubetis [1993], 450–462.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking.* The MIT Press, Cambridge, Massachusetts.

COURCOUBETIS, C., Ed. 1993. *Computer Aided Verification, 5th International Conference, CAV'93.* Lecture Notes in Computer Science, vol. 697. Springer.

DEREPAS, F. AND GASTIN, P. 2001. Model checking systems of replicated processes with Spin. See Dwyer [2001], 235–251.

DESEL, J. AND REISIG, W. 1998. Place/transition Petri nets. See Reisig and Rozenberg [1998a], 122–173.

DILL, D. L. 1996. The Murφ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96,* R. Alur and T. Henzinger, Eds. Lecture Notes in Computer Science, vol. 1102. Springer, 390–393.

DWYER, M., Ed. 2001. *Model Checking Software: 8th International SPIN Workshop.* Lecture Notes in Computer Science, vol. 2057. Springer.

EMERSON, E. A., HAVLICEK, J. W., AND TREFLER, R. J. 2000. Virtual symmetry reduction. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society, 121–131.

EMERSON, E. A., JHA, S., AND PELED, D. 1997. Combining partial order and symmetry reductions. In *Tools and Algorithms for the Construction and Analysis of Systems; Third International Workshop, TACAS'97,* E. Brinksma, Ed. Lecture Notes in Computer Science, vol. 1217. Springer, 19–34.

EMERSON, E. A. AND SISTLA, A. P. 1993. Symmetry and model checking. See Courcoubetis [1993], 463–478.

EMERSON, E. A. AND SISTLA, A. P. 1995. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. In *Computer Aided Verification: 7th International Conference, CAV'95,* P. Wolper, Ed. Lecture Notes in Computer Science, vol. 939. Springer, 309–324.

EMERSON, E. A. AND SISTLA, A. P. 1996. Symmetry and model checking. *Formal Methods in System Design 9,* 1/2 (Aug.), 105–131.

EMERSON, E. A. AND SISTLA, A. P. 1997. Utilizing symmetry when model checking under fairness assumptions: An automata-theoretic approach. *ACM Transactions on Programming Languages and Systems 19,* 4 (July), 617–638.

EMERSON, E. A. AND TREFLER, R. J. 1998. Model checking real-time properties of symmetric systems. In *Mathematical Foundations of Computer Science 1998*, L. Brim, J. Gruska, and J. Zlatuška, Eds. Lecture Notes in Computer Science, vol. 1450. Springer, 427–436.

EMERSON, E. A. AND TREFLER, R. J. 1999. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods, CHARME'99*, L. Pierre and T. Kropf, Eds. Lecture Notes in Computer Science, vol. 1703. Springer, 142–156.

ESPARZA, J. 1998. Decidability and complexity of Petri net problems — an introduction. See Reisig and Rozenberg [1998a], 374–428.

FINKEL, A. 1990. The minimal coverability graph for Petri nets. In *11th Internation Conference on Application and Theory of Petri Nets.* Paris, 1–21.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco.

GENRICH, H. J. 1991. Predicate/transition nets. See Jensen and Rozenberg [1991], 3–43.

GODEFROID, P. 1997. Model checking for programming languages using VeriSoft. In *POPL '97; Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 174–186.

GODEFROID, P. 1999. Exploiting symmetry when model-checking software. In *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII/PSTV XIX'99*, J. Wu, S. T. Chanson, and Q. Gao, Eds. Kluwer Academic Publishers, 257–275.

GYURIS, V. AND SISTLA, A. P. 1997. On-the-fly model checking under fairness that exploits symmetry. In *Computer Aided Verification: 9th International Conference, CAV'97*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer, 232–243.

GYURIS, V. AND SISTLA, A. P. 1999. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design 15,* 3 (Nov.), 217–238.

HADDAD, S., ILIÉ, J.-M., AND AJAMI, K. 2000. A model checking method for partially symmetric systems. In *Formal Techniques for Distributed System Development, FORTE/PSTV 2000,* T. Bolognesi and D. Latella, Eds. Kluwer, 121–136.

HADDAD, S., ILIÉ, J.-M., TAGHELIT, M., AND ZOUARI, B. 1995. Symbolic reachability graph and partial symmetries. In *Application and Theory of Petri Nets 1995; Proceedings of the 16th International Conference; Turin, Italy, June 1995*, G. D. Michelis and M. Diaz, Eds. Lecture Notes in Computer Science, vol. 935. Springer, 238–257.

HOFFMANN, C. M. 1982. *Group-Theoretic Algorithms and Graph Isomorphism.* Lecture Notes in Computer Science, vol. 136. Springer.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering 23,* 5 (May), 279–295.

HU, A. J. AND VARDI, M. Y., Eds. 1998. *Computer Aided Verification: 10th International Conference, CAV'98.* Lecture Notes in Computer Science, vol. 1427. Springer.

HUBER, P., JENSEN, A. M., JEPSEN, L. O., AND JENSEN, K. 1985a. Towards reachability trees for high-level Petri nets. In *Advances in Petri Nets 1984,* G. Rozenberg, Ed. Lecture Notes in Computer Science, vol. 188. Springer, 215–233.

HUBER, P., JENSEN, A. M., JEPSEN, L. O., AND JENSEN, K. 1985b. Towards reachability trees for high-level Petri nets. Tech. Rep. DAIMI PB 174, Datalogisk Afdeling, Matematisk Institut, Aarhus Universitet. May.

IOSIF, R. 2001. Exploiting heap symmetries in explicit-state model checking of software. In *16th Annual International Conference on a Automated Software Engineering (ASE 2001).* IEEE, 254–261.

IOSIF, R. 2002. Symmetry reduction criteria for software model checking. In *Model Checking Software: 9th International SPIN Workshop,* D. Bošnački and S. Leue, Eds. Lecture Notes in Computer Science, vol. 2318. Springer, 22–41.

IP, C. N. 1996. State reduction methods for automatic formal verification. Ph.D. thesis, Department of Computer Science, Stanford University.

IP, C. N. AND DILL, D. L. 1996. Better verification through symmetry. *Formal Methods in System Design 9,* 1/2 (Aug.), 41–76.

JENSEN, K. 1981. Coloured Petri nets and the invariant-method. *Theoretical Computer Science 14,* 317–336.

JENSEN, K. 1992. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1, Basic Concepts*, Second ed. Monographs in Theoretical Computer Science. Springer.

JENSEN, K. 1995. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer.

JENSEN, K. 1996. Condensed state spaces for symmetrical coloured Petri nets. *Formal Methods in System Design 9,* 1/2 (Aug.), 7–40.

JENSEN, K. 1997. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer.

JENSEN, K. AND ROZENBERG, G., Eds. 1991. *High-level Petri Nets; Theory and Application.* Springer.

JERRUM, M. 1986. A compact representation for permutation groups. *Journal of Algorithms 7,* 1 (Mar.), 60–78.

JØRGENSEN, J. B. AND KRISTENSEN, L. M. 1998. *Design/CPN OE/OS Graph Manual.* Computer Science Department, University of Aarhus. Version 1.1.

JØRGENSEN, J. B. AND KRISTENSEN, L. M. 1999. Computer aided verification of Lamport's fast mutual exclusion algorithm using colored Petri nets and occurrence graphs with symmetries. *IEEE Transactions on Parallel and Distributed Systems 10,* 7 (July), 714–732.

JUNTTILA, T. 1999a. Detecting and exploiting data type symmetries of algebraic system nets during reachability analysis. Research Report A57, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. Dec.

JUNTTILA, T. 2000. Computational complexity of the Place/Transition-net symmetry reduction method. Research Report A59, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. Apr.

JUNTTILA, T. 2002a. New canonical representative marking algorithms for place/transition-nets. Research Report A75, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. Oct.

JUNTTILA, T. 2002b. Symmetry reduction algorithms for data symmetries. Research Report A72, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. May.

JUNTTILA, T. A. 1998. Towards well-formed algebraic system nets. In *Workshop Concurrency, Specification & Programming*, H.-D. Burkhard, L. Czaja, and P. Starke, Eds. Number 110 in Informatik-Bericht. Humboldt-Universität zu Berlin, 116–127.

JUNTTILA, T. A. 1999b. Finding symmetries of algebraic system nets. *Fundamenta Informaticae 37,* 3 (Feb.), 269–289.

JUNTTILA, T. A. 2001. Computational complexity of the Place/Transition-net symmetry reduction method. *Journal of Universal Computer Science 7,* 4, 307–326.

KARP, R. M. AND MILLER, R. E. 1969. Parallel program schemata. *Journal of Computer and System Sciences 3,* 2 (May), 147–195.

KINDLER, E. AND REISIG, W. 1996. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter 51,* 16–31.

KINDLER, E. AND VÖLZER, H. 1998. Flexibility in algebraic nets. In *Application and Theory of Petri Nets 1998; Proceedings of the 19th International Conference, ICATPN'98*, J. Desel and M. Silva, Eds. Lecture Notes in Computer Science, vol. 1420. Springer, 345–364.

KINDLER, E. AND VÖLZER, H. 2001. Algebraic nets with flexible arcs. *Theoretical Computer Science 262,* 1–2 (July), 285–310.

KÖBLER, J., SCHÖNING, U., AND TORÁN, J. 1993. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser, Boston, USA.

KREHER, D. L. AND STINSON, D. R. 1999. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, Boca Raton, Florida, USA.

KRENTEL, M. W. 1988. The complexity of optimization problems. *Journal of Computer and System Sciences 36,* 3 (June), 490–509.

LERDA, F. AND VISSER, W. 2001. Addressing dynamic issues of program model checking. See Dwyer [2001], 80–102.

LORENTSEN, L. 2002. Coloured Petri nets and state space generation with the symmetry method. In *Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, K. Jensen, Ed. Number DAIMI PB–560 in technical reports of the Department of Computer Science, University of Aarhus. 121–138.

LORENTSEN, L. AND KRISTENSEN, L. M. 2001. Exploiting stabilizers and parallelism in state space generation with the symmetry method. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD 2001)*. IEEE Computer Society, 211–220.

MÄKELÄ, M. 2001a. Optimizing enabling tests and unfoldings of algebraic system nets. In *Application and Theory of Petri Nets 2001; Proceedings of the 22nd International Conference, ICATPN 2001*, J.-M. Colom and M. Koutny, Eds. Lecture Notes in Computer Science, vol. 2075. Springer, 283–302.

MÄKELÄ, M. 2001b. A reachability analyser for algebraic system nets. Research Report A69, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland. June.

MÄKELÄ, M. 2002. Maria: Modular reachability analyser for algebraic system nets. In *Application and Theory of Petri Nets 2002; Proceedings of the 23rd International Conference, ICATPN 2002*, J. Esparza and C. Lakos, Eds. Lecture Notes in Computer Science, vol. 2360. Springer, 434–444.

MANKU, G. S., HOJATI, R., AND BRAYTON, R. 1998. Structural symmetry and model checking. See Hu and Vardi [1998], 159–171.

MCKAY, B. D. 1981. Practical graph isomorphism. *Congressus Numerantium 30*, 45–87.

MCKAY, B. D. 1990. Nauty user's guide (version 1.5). Tech. Rep. TR-CS-90-02, Computer Science Department, Australian National University.

MILLER, G. L. 1979. Graph isomorphism, general remarks. *Journal of Computer and System Sciences 18,* 2 (Apr.), 128–142.

PANDEY, M. AND BRYANT, R. E. 1999. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18,* 7 (July), 918–935.

PAPADIMITRIOU, C. H. 1995. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, USA.

PELED, D. A. AND VARDI, M. Y., Eds. 2002. *Formal Techniques for Networked and Distributed Systems, FORTE 2002*. Lecture Notes in Computer Science, vol. 2529. Springer.

PETRUCCI, L. 1990. Combining Finkel's and Jensen's reduction techniques to build covering trees for coloured nets. *Petri Net Newsletter 36*, 32–36.

REISIG, W. AND ROZENBERG, G., Eds. 1998a. *Lectures on Petri Nets I: Basic Models*. Lecture Notes in Computer Science, vol. 1491. Springer.

REISIG, W. AND ROZENBERG, G., Eds. 1998b. *Lectures on Petri Nets II: Applications*. Lecture Notes in Computer Science, vol. 1492. Springer.

SCHMIDT, K. 2000a. How to calculate symmetries of Petri nets. *Acta Informatica 36,* 7, 545–590.

SCHMIDT, K. 2000b. Integrating low level symmetries into reachability analysis. In *Tools and Algorithms for the Construction and Anal-*

ysis of Systems; 6th International Conference, TACAS 2000, S. Graf and M. Schwartzbach, Eds. Lecture Notes in Computer Science, vol. 1785. Springer, 315–330.

SCHMIDT, K. 2000c. LoLA: A low level analyser. In *Application and Theory of Petri Nets 2000; Proceedings of the 21st International Conference, ICATPN 2000*, M. Nielsen and D. Simpson, Eds. Lecture Notes in Computer Science, vol. 1825. Springer, 465–474.

SISTLA, A. P. AND GODEFROID, P. 2001. Symmetry and reduced symmetry in model checking. In *Computer Aided Verification: 13th International Conference, CAV 2001*, G. Berry, H. Comon, and A. Finkel, Eds. Lecture Notes in Computer Science, vol. 2102. Springer, 91–103.

SISTLA, A. P., GYURIS, V., AND EMERSON, E. A. 2000. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology 9*, 2 (Apr.), 133–166.

STARKE, P. H. 1991. Reachability analysis of Petri nets using symmetries. *Systems Analysis Modelling Simulation 8*, 4/5, 293–303.

TIUSANEN, M. 1994. Symbolic, symmetry, and stubborn set searches. In *Application and Theory of Petri Nets 1994*, R. Valette, Ed. Lecture Notes in Computer Science, vol. 815. Springer, 511–530.

VALMARI, A. 1991. Stubborn sets of coloured Petri nets. In *XII International Conference on Application and Theory of Petri Nets*. Gjern, Denmark, 102–121.

VALMARI, A. 1998. The state explosion problem. See Reisig and Rozenberg [1998a], 429–528.

WANG, F. AND SCHMIDT, K. 2002. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. See Peled and Vardi [2002], 50–64.

WIRSING, M. 1990. Algebraic specification. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B: Formal models and semantics. Elsevier Science Publishers B.V., Chapter 13, 675–788.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A67 Timo Latvala
Model Checking Linear Temporal Logic Properties of Petri Nets with Fairness Constraints. January 2001.

HUT-TCS-A68 Javier Esparza, Keijo Heljanko
Implementing LTL Model Checking with Net Unfoldings. March 2001.

HUT-TCS-A69 Marko Mäkelä
A Reachability Analyser for Algebraic System Nets. June 2001.

HUT-TCS-A70 Petteri Kaski
Isomorph-Free Exhaustive Generation of Combinatorial Designs. December 2001.

HUT-TCS-A71 Keijo Heljanko
Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets. February 2002.

HUT-TCS-A72 Tommi Junttila
Symmetry Reduction Algorithms for Data Symmetries. May 2002.

HUT-TCS-A73 Toni Jussila
Bounded Model Checking for Verifying Concurrent Programs. August 2002.

HUT-TCS-A74 Sam Sandqvist
Aspects of Modelling and Simulation of Genetic Algorithms: A Formal Approach. September 2002.

HUT-TCS-A75 Tommi Junttila
New Canonical Representative Marking Algorithms for Place/Transition-Nets. October 2002.

HUT-TCS-A76 Timo Latvala
On Model Checking Safety Properties. December 2002.

HUT-TCS-A77 Satu Virtanen
Properties of Nonuniform Random Graph Models. May 2003.

HUT-TCS-A78 Petteri Kaski
A Census of Steiner Triple Systems and Some Related Combinatorial Objects. June 2003.

HUT-TCS-A79 Heikki Tauriainen
Nested Emptiness Search for Generalized Büchi Automata. July 2003.

HUT-TCS-A80 Tommi Junttila
On the Symmetry Reduction Method for Petri Nets and Similar Formalisms. September 2003.

ISBN 951-22-6744-6
ISSN 1457-7615