Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 72
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 72
Espoo 2002 HUT-TCS-A72

SYMMETRY REDUCTION ALGORITHMS FOR DATA SYMMETRIES

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 72
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 72
Espoo 2002 HUT-TCS-A72

SYMMETRY REDUCTION ALGORITHMS FOR DATA SYMMETRIES

Tommi Junttila

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

ISBN 951-22-6010-7
ISSN 1457-7615

Picaset Oy
Helsinki 2002

ABSTRACT: The core problem in the symmetry reduction method for state space analysis is to decide whether two states are symmetric or to produce a symmetric representative state for a state. This report presents algorithms for the problem under data symmetries. The setting covers systems described in the Murφ language or in terms of high-level Petri nets. The first two algorithms are based on refining ordered partitions by using symmetry respecting invariants. The last algorithm exploits existing graph isomorphism algorithms that are then applied on characteristic graphs of states, i.e. graphs corresponding to the states in a symmetry respecting way. Some experimental results are also reported.

KEYWORDS: Symmetry, reachability analysis, Petri nets, the Murφ tool.

CONTENTS

1 INTRODUCTION

Model checking [Clarke et al. 1999] is an automatic technique for verifying concurrent systems such as communication protocols. The main obstacle for model checking is the so-called state space explosion problem [Valmari 1998]. It essentially means that a system may have exponentially many reachable states w.r.t. the size of the system description. A technique, among many others, to alleviate this problem is the so-called symmetry reduction method (see e.g. the articles in vol. 9, no. 1/2 of the *Formal Methods in System Design* journal). It exploits the symmetries (automorphisms) of the state space, the goal being to examine only one representative state in each set of mutually symmetric states (orbit). The core problem in the symmetry reduction method is: Given a set of already generated states and a newly generated one, does the set include a state that is symmetric to the newly generated one? There are basically two alternative ways to solve this problem. First, one can pairwisely compare the newly generated state with each already generated state for symmetry. Secondly, one can transform the newly generated state into a symmetric, canonical representative state and test whether it is in the set of already generated states. These two approaches can also be approximated by using a sound but incomplete symmetry check in the first one and by transforming the newly generated state into a symmetric, but not necessarily canonical, representative state. Of course, approximation may result in that more than one representative state from an orbit is examined.

In this report we develop algorithms for the above mentioned problem in the following setting. We assume systems in which states are assignments for a set of typed state variables. The type system for the state variables consists of a set of primitive types, upon which common high-level data structures such as lists, sets, structures, multi-sets, and association arrays are built. State space symmetries are then produced by permuting the elements in the domains of some primitive types. The class of systems studied in this report covers the Murφ description language [Ip and Dill 1996], and several classes of high-level Petri nets: Well-Formed Nets [Chiola et al. 1991], Extended Well-Formed Nets [Junttila 1999], and some commonly used subclasses of Colored Petri Nets [Jensen 1995]. Some experimental results are also presented.

2 SYSTEMS

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 [Junttila 1999] 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 paper.

First, a set $\mathcal{T}$ of *types* is assumed. Each type $T \in \mathcal{T}$ is associated with a non-empty *domain* $\mathscr{D}_T$. A *system* is a triple

$$\mathfrak{S} = \langle \mathcal{X}, \longrightarrow, \mathfrak{s}_0 \rangle$$

consisting of the following components.

– $\mathcal{X} = (\mathcal{X}_T)_{T \in \mathcal{T}}$ is a finite, pairwise disjoint family of typed *state variables*. A *state* $\mathfrak{s}$ is an assignment to the state variables such that $\mathfrak{s}(x) \in \mathscr{D}_T$ holds for each state variable $x \in \mathcal{X}_T \in \mathcal{X}$. The set of *all states* is denoted by $\mathcal{S}$.
– $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* describing the dynamic behavior of the system, i.e. how states evolve into others. We use $\mathfrak{s} \longrightarrow \mathfrak{s}'$ to denote that $\langle \mathfrak{s}, \mathfrak{s}' \rangle \in \longrightarrow$.
– $\mathfrak{s}_0 \in \mathcal{S}$ is the *initial state*.

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, (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 multi-set 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.

The *state space* of a system $\mathfrak{S}$ is the labeled transition system $\langle \mathcal{S}, \longrightarrow, \mathfrak{s}_0 \rangle$ consisting of all the possible states and transitions between them. The *reachability graph* of $\mathfrak{S}$ on the other hand describes which states can be reached when the system is started in the initial state $\mathfrak{s}_0$. Formally, it is the labeled transition system $\mathrm{RG} = \langle \vec{\mathcal{S}}, \longrightarrow_{\mathrm{RG}}, \mathfrak{s}_0 \rangle$, where $\vec{\mathcal{S}} \subseteq \mathcal{S}$ and $\longrightarrow_{\mathrm{RG}} \subseteq \vec{\mathcal{S}} \times \vec{\mathcal{S}}$ are inductively defined as follows.

1. $\mathfrak{s}_0 \in \vec{\mathcal{S}}$.
2. If $\mathfrak{s} \in \vec{\mathcal{S}}$ and $\mathfrak{s} \longrightarrow \mathfrak{s}'$, then $\mathfrak{s}' \in \vec{\mathcal{S}}$ and $\mathfrak{s} \longrightarrow_{\mathrm{RG}} \mathfrak{s}'$.
3. Nothing else is in $\vec{\mathcal{S}}$ or in $\longrightarrow_{\mathrm{RG}}$.

2.1 Type System

We now build a type system $\mathcal{T}$ for the state variables. First, we assume a set $\mathcal{T}_0$ of *primitive types*. Each primitive type $T \in \mathcal{T}_0$ is associated with a non-empty, countable *domain* $\mathscr{D}_T$. Based on primitive types, the set $\mathcal{T}$ of *types* is defined by the grammar

$$\begin{aligned} T \quad ::= \quad & T_0 \mid \mathsf{List}(T) \mid \mathsf{Struct}(T, \ldots, T) \mid \mathsf{Set}(T) \mid \mathsf{Multi\text{-}Set}(T) \mid \\ & \mathsf{AssocArray}(T, T) \mid \mathsf{Union}(T, \ldots, T) \end{aligned}$$

where T_0 ranges over $\mathcal{T}_0$. The types in $\mathcal{T} \setminus \mathcal{T}_0$ are called *structured types over* $\mathcal{T}_0$. The domains of structured types are defined inductively by the following rules:

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

where $\wp(A)$ denotes the powerset of the set A, $[A \rightarrow B]$ is the set of all functions from A to B, and $[A \rightsquigarrow B]$ denotes the set of all partial functions from A to B.[1,2] Note 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 us 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 list element $\langle T_2, \langle 3, 6\rangle\rangle$ is distinguished from the structure element $\langle T_1, \langle 3, 6\rangle\rangle$.

Example 2.1 Consider the EWF-Net shown in Fig. 1 that is a variant of the railroad net in [Genrich 1991] obtained by folding. For the railroad sections we have the primitive type Secs with the domain $\mathscr{D}_{\mathsf{Secs}} = \{s_0, \ldots, s_5\}$, and similarly for the trains the primitive type Trains with $\mathscr{D}_{\mathsf{Trains}} = \{t_a, t_b\}$. The net has two state variables: U of type $\mathsf{Multi\text{-}Set}(\mathsf{Struct}(\mathsf{Trains}, \mathsf{Secs}))$ and V of type $\mathsf{Multi\text{-}Set}(\mathsf{Secs})$. The initial state is

$$\mathfrak{s}_0 = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$$

and the transition relation is defined by the EWF-Net semantics, see [Junttila 1999]. The reachability graph of the net is shown in Fig. 2 (in which each state $\{U \mapsto v_1, V \mapsto v_2\}$ is denoted by "v_1, v_2"). ♣

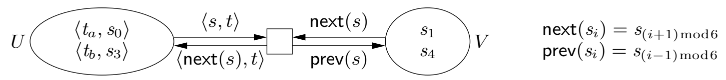


Figure 1: An EWFN for Genrich's railroad system

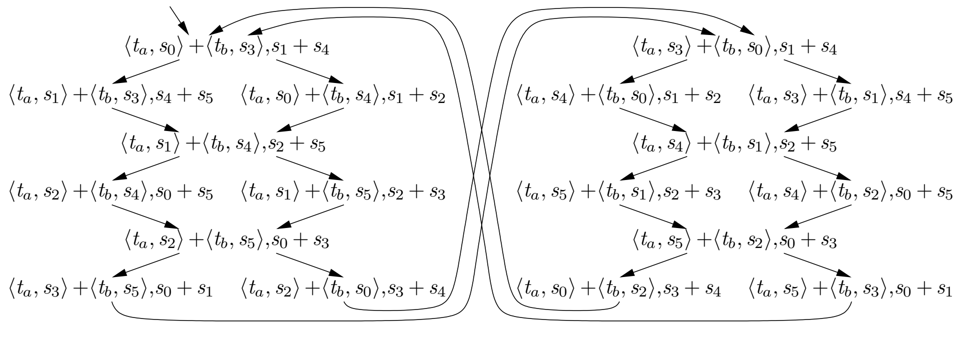


Figure 2: The reachability graph of the net in Fig. 1

3 DATA SYMMETRIES

A *state space symmetry* of a system $\mathfrak{S}$ is a permutation $\psi^{\mathcal{S}}$ on the set $\mathcal{S}$ of states preserving the transition relation, i.e. the *symmetry condition*

$$\mathfrak{s} \longrightarrow \mathfrak{s}' \Leftrightarrow \psi^{\mathcal{S}}(\mathfrak{s}) \longrightarrow \psi^{\mathcal{S}}(\mathfrak{s}')$$

[1] 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.

[2] We may use the formal sum notation to describe multi-sets, i.e. a multi-set $b \in [A \to \mathbb{N}]$ over a set A may be denoted by $\sum_{v \in A} b(v)'\, v$. Elements with multiplicity 0 as well as the unit multiplicities may be omitted, e.g. a multi-set $b = \{a \mapsto 0, b \mapsto 2, c \mapsto 1\}$ over the set $\{a, b, c\}$ may be written as $b = 2'\, b + c$.

must hold for all states $\mathfrak{s}, \mathfrak{s}' \in \mathcal{S}$. A *state space symmetry group* $\Psi^{\mathcal{S}}$ is a non-empty set of state space symmetries forming a group under the function composition operator $\circ$. Two states $\mathfrak{s}, \mathfrak{s}'$ are $\Psi^{\mathcal{S}}$*-symmetric*, denoted by $\mathfrak{s} \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}'$, if $\exists \psi^{\mathcal{S}} \in \Psi^{\mathcal{S}}, \psi^{\mathcal{S}}(\mathfrak{s}) = \mathfrak{s}'$. Since $\Psi^{\mathcal{S}}$ is a group, $\equiv_{\Psi^{\mathcal{S}}}$ is an equivalence relation on $\mathcal{S}$ and the equivalence class in which a state $\mathfrak{s}$ belongs is called the $\Psi^{\mathcal{S}}$-orbit of $\mathfrak{s}$.

A *quotient state space* of $\mathfrak{S}$ is a labeled transition system $\mathrm{QSS} = \langle \mathcal{S}, \longrightarrow_{\mathrm{Q}}, \mathfrak{s}'_0 \rangle$ such that (i) $\mathfrak{s}_0 \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}'_0$, (ii) $\mathfrak{s} \longrightarrow \mathfrak{s}'$ implies $\mathfrak{s} \longrightarrow_{\mathrm{Q}} \mathfrak{s}''$ for at least one $\mathfrak{s}''$ such that $\mathfrak{s}' \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}''$, and (iii) $\mathfrak{s} \longrightarrow_{\mathrm{Q}} \mathfrak{s}''$ implies $\mathfrak{s} \longrightarrow \mathfrak{s}'$ for at least one $\mathfrak{s}'$ such that $\mathfrak{s}' \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}''$. That is, in a quotient state space the transitions can be "redirected" to any state that is symmetric to the original successor state (and no other transitions may exist). Obviously, there can be many different quotient state spaces for $\mathfrak{S}$ under the group $\Psi^{\mathcal{S}}$. Furthermore, $\equiv_{\Psi^{\mathcal{S}}}$ is a strong bisimulation relation between any quotient state space and the original state space. A *quotient reachability graph* of $\mathfrak{S}$ is the reachable part of a quotient state space $\mathrm{QSS} = \langle \mathcal{S}, \longrightarrow_{\mathrm{Q}}, \mathfrak{s}'_0 \rangle$: $\mathrm{QRG} = \langle \tilde{\mathcal{S}}, \longrightarrow_{\mathrm{QRG}}, \mathfrak{s}'_0 \rangle$, where $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ and $\longrightarrow_{\mathrm{QRG}} \subseteq \tilde{\mathcal{S}} \times \tilde{\mathcal{S}}$ are inductively defined as follows.

1. $\mathfrak{s}'_0 \in \tilde{\mathcal{S}}$.
2. If $\mathfrak{s} \in \tilde{\mathcal{S}}$ and $\mathfrak{s} \longrightarrow_{\mathrm{Q}} \mathfrak{s}'$, then $\mathfrak{s}' \in \tilde{\mathcal{S}}$ and $\mathfrak{s} \longrightarrow_{\mathrm{QRG}} \mathfrak{s}'$.
3. Nothing else is in $\tilde{\mathcal{S}}$ or in $\longrightarrow_{\mathrm{QRG}}$.

Again, any quotient reachability graph is strongly bisimilar to the reachability graph. An algorithm that computes a quotient reachability graph is shown in Fig. 3. Considering the efficiency of the algorithm, the crucial point is line 8 where the successor state is chosen for the current state. In order to get as small as possible quotient reachability graphs, the successor state $\mathfrak{s}''$ should be chosen in a way that only one state in each set of mutually symmetric states is present in $\tilde{\mathcal{S}}$. There are basically two ways to achieve this:

1. For each state $\mathfrak{s}'''$ already in $\tilde{\mathcal{S}}$, check whether $\mathfrak{s}' \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}'''$. If this is the case, select $\mathfrak{s}'' = \mathfrak{s}'''$. Otherwise, select $\mathfrak{s}'' = \mathfrak{s}'$.
2. Define a *representative function*, i.e. a function $repr : \mathcal{S} \to \mathcal{S}$ such that $repr(\mathfrak{s}) \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}$ holds for all states $\mathfrak{s} \in \mathcal{S}$, and let the successor state $\mathfrak{s}''$ be $repr(\mathfrak{s}')$. If $repr$ fulfills the *canonicity condition* meaning that $\mathfrak{s}_1 \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}_2$ implies $repr(\mathfrak{s}_1) = repr(\mathfrak{s}_2)$, then the quotient reachability graph will have minimal number of states.

Since checking whether two states are symmetric and producing canonical representative states is, in general, computationally at least as hard as testing whether two graphs are isomorphic, see e.g. [Ip 1996; Junttila 1999], both approaches above can be approximated (i) by using a sound but incomplete symmetricity check in the first approach and (ii) by using a non-canonical representative function in the second approach. This potentially trades the computational cost of choosing a unique representative state per orbit for producing bigger quotient reachability graphs.

3.1 Domain Permutations

The particular class of symmetries studied in this report is produced by permuting the domains of the types. Formally, a *domain permutation for a*

Choose any $\mathfrak{s}_0'$ such that $\mathfrak{s}_0 \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}_0'$
Let $W = \{\mathfrak{s}_0'\}$
Let $\tilde{\mathcal{S}} = \{\mathfrak{s}_0'\}$
Let $\longrightarrow_{\mathrm{QRG}} = \emptyset$
While $W \neq \emptyset$ do
  Take any $\mathfrak{s} \in W$ and let $W = W \setminus \{\mathfrak{s}\}$
  For all $\mathfrak{s}'$ such that $\mathfrak{s} \longrightarrow \mathfrak{s}'$ do
    Choose any $\mathfrak{s}''$ such that $\mathfrak{s}' \equiv_{\Psi^{\mathcal{S}}} \mathfrak{s}''$
    Let $\longrightarrow_{\mathrm{QRG}} = \longrightarrow_{\mathrm{QRG}} \cup \langle \mathfrak{s}, \mathfrak{s}'' \rangle$
    If $\mathfrak{s}'' \notin \tilde{\mathcal{S}}$
      Let $W = W \cup \{\mathfrak{s}''\}$
      Let $\tilde{\mathcal{S}} = \tilde{\mathcal{S}} \cup \{\mathfrak{s}''\}$
Return $\mathrm{QRG} = \langle \tilde{\mathcal{S}}, \longrightarrow_{\mathrm{QRG}}, \mathfrak{s}_0' \rangle$

Figure 3: An algorithm for computing quotient reachability graphs

type T is a permutation ψ^T of its domain $\mathscr{D}_T$. A *domain permutation group for a type* T is permutation group Ψ^T on $\mathscr{D}_T$ (under the function composition operator $\circ$). A *domain permutation for a set* $\mathcal{T}'$ *of types* is a family $\psi^{\mathcal{T}'} = \left(\psi^T\right)_{T \in \mathcal{T}'}$ of domain permutations for the member types. A domain permutation group for a set $\mathcal{T}'$ of types is a non-empty set $\Psi^{\mathcal{T}'}$ of domain permutations for $\mathcal{T}'$ forming a group under the type-wise function composition operator $*$ defined by: $\left(\psi_1^T\right)_{T \in \mathcal{T}'} * \left(\psi_2^T\right)_{T \in \mathcal{T}'} = \left(\psi_3^T\right)_{T \in \mathcal{T}'}$, where $\psi_1^T \circ \psi_2^T = \psi_3^T$ for all $T \in \mathcal{T}'$.

Assume that we have a domain permutation (group) for the set of primitive types. It is extended to operate on all types and states as follows. First, each domain permutation $\psi^{\mathcal{T}_0} = \left(\psi^T\right)_{T \in \mathcal{T}_0}$ for the set of primitive types is canonically extended to the domain permutation $\psi^{\mathcal{T}} = \left(\psi^T\right)_{T \in \mathcal{T}}$ for all types by the following inductive rules.

- $\psi^{\mathsf{List}(T)}(\langle v_1, \ldots, v_n \rangle) = \langle \psi^T(v_1), \ldots, \psi^T(v_n) \rangle$,
- $\psi^{\mathsf{Struct}(T_1,\ldots,T_n)}(\langle v_1, \ldots, v_n \rangle) = \langle \psi^{T_1}(v_1), \ldots, \psi^{T_n}(v_n) \rangle$,
- $\psi^{\mathsf{Set}(T)}(V) = \{\psi^T(v) \mid v \in V\}$,
- $\psi^{\mathsf{Multi\text{-}Set}(T)}(b) = \{\langle \psi^T(v), i \rangle \mid \langle v, i \rangle \in b\}$,
- $\psi^{\mathsf{AssocArray}(T_1,T_2)}(a) = \{\langle \psi^{T_1}(v_1), \psi^{T_2}(v_2) \rangle \mid \langle v_1, v_2 \rangle \in a\}$, and
- $\psi^{\mathsf{Union}(T_1,\ldots,T_n)}(\langle T_i, v \rangle) = \langle T_i, \psi^{T_i}(v) \rangle$.

Finally, each domain permutation $\psi^{\mathcal{T}} = \left(\psi^T\right)_{T \in \mathcal{T}}$ for all types is extended to operate on the set $\mathcal{S}$ of states by $\psi^{\mathcal{S}}(\mathfrak{s})(x) = \psi^T(\mathfrak{s}(x))$ for each state variable $x \in \mathcal{X}_T \in \mathcal{X}$. Thus each domain permutation $\psi^{\mathcal{T}_0}$ on primitive types induces a unique permutation $\psi^{\mathcal{S}}$ on $\mathcal{S}$. Furthermore, there is a unique group homomorphism from a domain permutation group $\Psi^{\mathcal{T}_0}$ to $\Psi^{\mathcal{S}} = \{\psi^{\mathcal{S}} \mid \psi^{\mathcal{T}_0} \in \Psi^{\mathcal{T}_0}\}$. Although $\Psi^{\mathcal{S}}$ is a permutation group on $\mathcal{S}$, it is not necessarily a state space symmetry group because the induced permutations $\psi^{\mathcal{S}}$ in it do not necessarily preserve the transition relation.

3.2 Allowed Domain Permutations

The set of primitive types, $\mathcal{T}_0$, is partitioned into three subclasses: *ordered*, *cyclic*, and *unordered* primitive types. (Unordered primitive types are called *scalar sets* in the Murφ terminology.) The difference between these classes is what kind of domain permutation groups they are associated to.

1. For each ordered primitive type T the allowed domain permutation group is the trivial group $\Theta^T = \{\mathbf{I}\}$, where $\mathbf{I}$ is the identity mapping.
2. For each cyclic primitive type T, the domain $\mathscr{D}_T = \{v_1, v_2, \ldots, v_n\}$ is assumed to be finite and associated with the cyclic successor function $succ_T$ such that $succ_T(v_i) = v_{(i \bmod n)+1}$. The group of allowed domain permutations for T is $\Theta^T = \{succ_T^1, \ldots, succ_T^n\}$ i.e. the cyclic permutation group generated by $succ_T$.[3]
3. For an unordered primitive type T, the domain $\mathscr{D}_T = \{v_1, v_2, \ldots, v_n\}$ is assumed to be finite and the allowed domain permutation group is the symmetric group $\Theta^T = \mathrm{Sym}(\mathscr{D}_T)$ consisting of all permutations of $\mathscr{D}_T$.

Cyclic and unordered primitive types are also called *permutable* primitive types and the set of such types of denoted by $\mathcal{T}_P$.

The *allowed domain permutation group* $\Theta^{\mathcal{T}_0}$ for the set of primitive types is the external direct product of the allowed domain permutation groups for the individual primitive types:

$$\Theta^{\mathcal{T}_0} = \bigotimes_{T \in \mathcal{T}_0} \Theta^T = \{\left(\theta^T\right)_{T \in \mathcal{T}_0} \mid \forall T \in \mathcal{T}_0, \theta^T \in \Theta^T\}.$$

It is then extended to the allowed domain permutation group $\Theta^{\mathcal{T}}$ for all types, and to the permutation group $\Theta^{\mathcal{S}}$ on the set $\mathcal{S}$ as described above. The fact that $\Theta^{\mathcal{S}}$ is a state space symmetry group for the formalisms mentioned earlier is guaranteed by imposing syntactic restrictions on system descriptions (in some cases the system description writer is also expected to take care that no symmetry-breaking constructions are used). For instance, it is not possible to do any arithmetic operations such as addition between two elements in the domain of an unordered primitive type. We may omit the superscripts and simply write θ for $\theta^{\mathcal{T}_0}$, $\theta^{\mathcal{T}}$, and $\theta^{\mathcal{S}}$ whenever the meaning is clear from the context. Similarly for Θ. In addition, note that any $\theta^{\mathcal{T}_0}$, $\theta^{\mathcal{T}}$, or $\theta^{\mathcal{S}}$ can be fully specified by giving the domain permutations for the permutable primitive types only. We say that two states are *symmetric* if they are Θ-symmetric.

Example 3.1 Continuing Ex. 2.1, assume that Secs is a cyclic primitive type and Trains is an unordered primitive type. Then the size of the allowed domain permutation group $\Theta^{\mathcal{T}_0}$ is $|\mathscr{D}_{\mathsf{Secs}}| \cdot |\mathscr{D}_{\mathsf{Trains}}|! = 6 \cdot 2! = 12$ and $\theta = \left(\theta^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_2 & s_3 & s_4 & s_5 & s_0 & s_1 \end{smallmatrix}\right), \theta^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right)$ is an allowed domain permutation mapping the initial state

$$\mathfrak{s}_0 = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_3\rangle, V \mapsto s_1 + s_4\}$$

into

$$\theta(\mathfrak{s}_0) = \{U \mapsto \langle t_a, s_5\rangle + \langle t_b, s_2\rangle, V \mapsto s_0 + s_3\}$$

[3]For a function $f : A \to A$, f^k means $\underbrace{f \circ \ldots \circ f}_{k \text{ times}}$.

In fact, the states in the reachability graph shown in Fig. 2 belong to two equivalence classes under Θ: the six states that are equivalent to the initial state $\mathfrak{s}_0$ and the twelve states that are equivalent to the state $\{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\}$. Figure 4 shows two quotient reachability graphs for the net. ♣

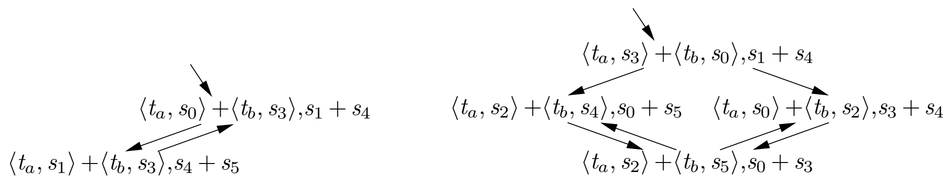


Figure 4: Two quotient reachability graphs

3.3 Stabilizers and Storing Subgroups

We now introduce the concept of stabilizers for the elements of types and for the states. In a nutshell, stabilizers are domain permutations that permute an object to itself.

A domain permutation $\psi = \left(\psi^T\right)_{T \in \mathcal{T}}$ *fixes* (or *stabilizes*) an element $v \in \mathscr{D}_T$ of a type T if $\psi^T(v) = v$. The *stabilizer (sub)group* of v in a domain permutation group Ψ is

$$\mathrm{Stab}(\Psi, v) = \{\psi \mid \psi \in \Psi \text{ and } \psi \text{ is a stabilizer of } v\}.$$

Similarly for states, a domain permutation $\psi = \left(\psi^T\right)_{T \in \mathcal{T}}$ is a *stabilizer* of a state $\mathfrak{s}$ if $\psi(\mathfrak{s}) = \mathfrak{s}$. Clearly this is equivalent to the requirement that $\psi^T(\mathfrak{s}(x)) = \mathfrak{s}(x)$ for each state variable $x \in \mathcal{X}_T \in \mathcal{X}$. Given a domain permutation group Ψ, the *stabilizer group* of a state $\mathfrak{s}$ in Ψ is

$$\mathrm{Stab}(\Psi, \mathfrak{s}) = \{\psi \in \Psi \mid \psi(\mathfrak{s}) = \mathfrak{s}\}.$$

Obviously, $\mathrm{Stab}(\Psi, \mathfrak{s}) = \bigcap_{x \in \mathcal{X}} \mathrm{Stab}(\Psi, \mathfrak{s}(x))$. Stabilizers can also be calculated iteratively: assuming that the state variables are $x_1, \ldots, x_n$, let $\Psi_1 = \mathrm{Stab}(\Psi, \mathfrak{s}(x_1))$, $\Psi_2 = \mathrm{Stab}(\Psi_1, \mathfrak{s}(x_2))$, ... , and $\Psi_n = \mathrm{Stab}(\Psi_{n-1}, \mathfrak{s}(x_n))$. Now $\Psi_n = \mathrm{Stab}(\Psi, \mathfrak{s})$.

Theorem 3.2 *Assume a domain permutation $\psi \in \Psi$ that maps a state $\mathfrak{s}_1$ to $\mathfrak{s}_2$ i.e. $\psi(\mathfrak{s}_1) = \mathfrak{s}_2$. Then*

1. $\mathrm{Stab}(\Psi, \mathfrak{s}_2) = \psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1}$*, where* $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1} = \{\psi * \psi' * \psi^{-1} \mid \psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)\}$*, and*
2. *the left coset* $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) = \{\psi * \psi' \mid \psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)\}$ *is exactly the set of all domain permutations in Ψ mapping $\mathfrak{s}_1$ to $\mathfrak{s}_2$.*

Consequently, (i) $|\mathrm{Stab}(\Psi, \mathfrak{s}_1)| = |\mathrm{Stab}(\Psi, \mathfrak{s}_2)|$*, (ii) there are* $|\mathrm{Stab}(\Psi, \mathfrak{s}_1)|$ *domain permutations mapping $\mathfrak{s}_1$ to $\mathfrak{s}_2$, and (iii) there are* $|\Psi| \,/\, |\mathrm{Stab}(\Psi, \mathfrak{s}_1)|$ *states that are Ψ-symmetric to $\mathfrak{s}_1$.*

The elements in the group $\mathrm{Stab}(\Theta, \mathfrak{s})$, where Θ is the group of allowed domain permutations, are of special importance and therefore they are called the *self-symmetries* of the state $\mathfrak{s}$ and $\mathrm{Stab}(\Theta, \mathfrak{s})$ is the *self-symmetry group* of $\mathfrak{s}$.

Example 3.3 Continuing Examples 2.1 and 3.1, 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 self-symmetry 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, see [Butler 1991]. In order to use those data structures, we only have 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 $\bigcup_{T \in \mathcal{T}_P} \mathscr{D}_T$.

4 VALUE TREES AND CHARACTERISTIC GRAPHS

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 as follows.

- For a primitive type T, the root node $T{::}v$ has no children.
- A root node $\mathsf{List}(T){::}\langle v_1, \dots, v_n\rangle$, 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 $\mathsf{Struct}(T_1, \dots, T_n){::}\langle v_1, \dots, v_n\rangle$, 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 $\mathsf{Multi\text{-}Set}(T){::}m$ 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 $\mathsf{Union}(T_1, \dots, T_n){::}\langle T_i, v_i\rangle$ has the value tree $\mathcal{VT}(T_i, v_i)$ as its only child, the edge to it having weight 1.

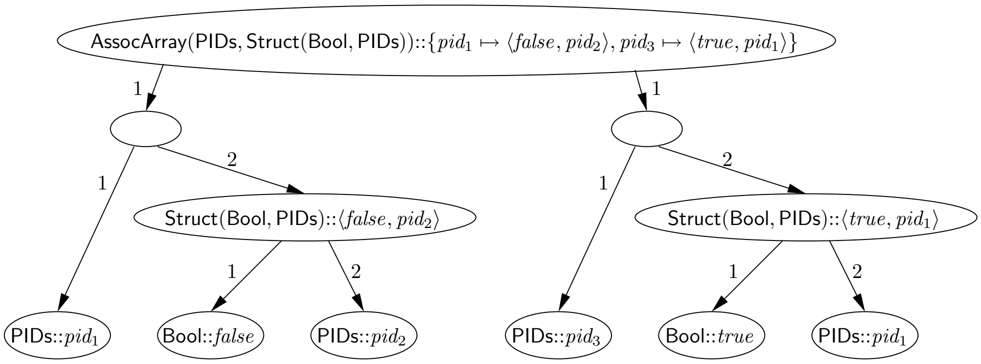


Figure 5: A value tree

Example 4.1 Fig. 5 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 4.2 *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. Now the *characteristic graph* of a state $\mathfrak{s}$ is a node 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 into 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.

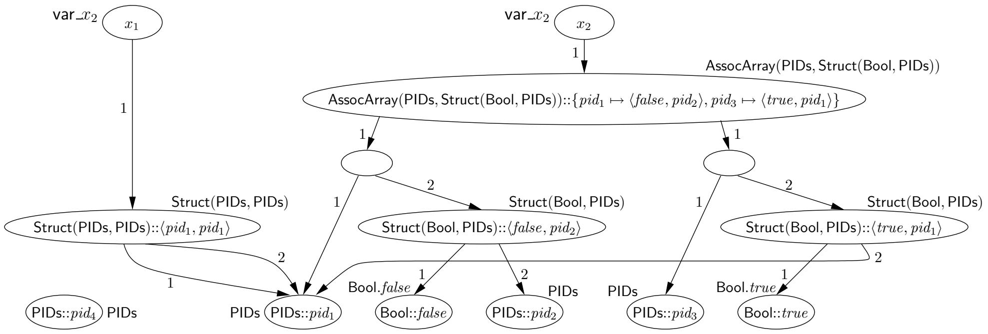


Figure 6: A characteristic graph

Example 4.3 Recall the previous example and assume that Bool is an ordered primitive type and PIDs is an unordered primitive type. Figure 6 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. ♣

An *isomorphism* from a node labeled and edge weighted directed graph to another such graph is a bijective mapping from nodes to nodes preserving node labels, edges, and edge weights. Two such graphs are *isomorphic* if there is an isomorphism between them. Since isomorphisms have to preserve node labels and edge weights, it is quite straightforward to see that characteristics graphs have the following properties:

Fact 4.4 *For all allowed domain permutations 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 $v \in \mathscr{D}_T$, $\theta^T(v) = v' \Leftrightarrow \gamma(T{::}v) = T{::}v'$.*

Fact 4.5 *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 iff the states are symmetric*. Furthermore, the self-symmetry group of a state can be easily extracted from the automorphism group of the characteristic graph of the state.

Example 4.6 Recall Examples 2.1 and 3.1. Figure 7 shows the 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, we have omitted some node names and labels that should be obvious). ♣

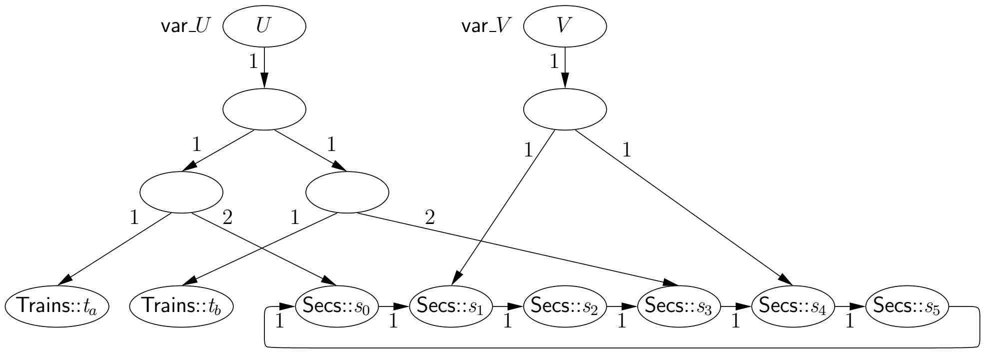


Figure 7: A characteristic graph

5 BASIC ALGORITHM BASED ON PARTITION REFINEMENT

We now present a relatively simple representative algorithm based on the following basic idea. Consider a state and two elements in the domain of a permutable primitive type. If the other element appears in the value of a state variable but the other does not, we may think that the elements are somehow different in the state. Distinguishing between the elements of permutable primitive types according to this or some other more complex criteria, we obtain an ordered partition of the elements. The main thing to take care during the partitioning process is to see that for permuted states, the similarly permuted elements are distinguished. In other words, the partitioning criteria should be invariant with respect to the symmetries of the system. After obtaining an ordered partition for the permutable primitive types, we select an allowed domain permutation according to the partition and return the state permuted with it as the representative state. The partitioning phase of the algorithm resembles the preprocessing step used in many graph auto-/isomorphism algorithms, where a partition of the vertices of a graph is obtained by applying some vertex invariants, see e.g. [Kreher and Stinson 1999; McKay 1981].

5.1 Ordered Partitions

An *ordered partition* of a non-empty set A is a list $[C_1, \dots, C_n]$ such that $\{C_1, \dots, C_n\}$ 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* (or blocks) of the partition. An ordered partition is *discrete* if all its cells are singleton sets and *unit* if it contains only one cell (the set A). Define the function $incell$ from the ordered partitions of A and the elements of A to the natural numbers by $incell([C_1, \dots, C_n], x) = i \Leftrightarrow x \in C_i$.

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 of p_1 is a subset of a cell of p_2. An ordered partition p_1 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 $p_2 = [C_{2,1}, \dots, C_{2,n}]$, then any p_1 such that $p_1 \preceq p_2$ is of form $[C_{1,1,1}, \dots, C_{1,1,d_1}, \dots C_{1,n,1}, \dots, C_{1,n,d_n}]$, where $\bigcup_{1 \leq j \leq d_i} C_{1,i,j} = C_{2,i}$ for $1 \leq i \leq n$. For example, $[\{b\}, \{a\}, \{c\}] \leq [\{a\}, \{b, c\}]$, $[\{b\}, \{a\}, \{c\}] \not\preceq$

$[\{a\}, \{b, c\}]$, 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 $\gamma([C_1, \dots, C_n]) = [\gamma(C_1), \dots, \gamma(C_n)]$. Clearly, $incell(p, x) = incell(\gamma(p), \gamma(x))$ for all ordered partitions p of A and all $x \in A$. Furthermore, if $\gamma(p_1) = p_2$, $p_1 \preceq p_3$, and $p_2 \preceq p_3$, then $\gamma(p_3) = p_3$.

5.2 Basic Algorithm

We now present the basic representative state algorithm. The idea is that, given a state for which we wish to compute a representative,

1. we first assign the state in a symmetry respecting way a partitioning of the domains of the permutable primitive types,
2. then select an allowed domain permutation based on the partitioning, and
3. finally return the state permuted with the selected domain permutation as the representative.

First, we define an *ordered permutable primitive type partition* to be a family $\mathfrak{p} = \left(\mathfrak{p}^T\right)_{T \in \mathcal{T}_P}$, 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 $\psi = \left(\psi^T\right)_{T \in \mathcal{T}_P}$ acts on a partition $\mathfrak{p} = \left(\mathfrak{p}^T\right)_{T \in \mathcal{T}_P}$ by $\psi(\mathfrak{p}) = \left(\psi^T(\mathfrak{p}^T)\right)_{T \in \mathcal{T}_P}$ and $\left(\mathfrak{p}_1^T\right)_{T \in \mathcal{T}_P} \preceq \left(\mathfrak{p}_2^T\right)_{T \in \mathcal{T}_P}$ if $\mathfrak{p}_1^T \preceq \mathfrak{p}_2^T$ for all $T \in \mathcal{T}_P$. As we will deal exclusively with ordered partitions, we usually omit the prefix "ordered" and simply speak of partitions. For convenience, we may also omit the prefix "permutable primitive type" whenever no confusion can arise.

Given a state $\mathfrak{s}$, we associate it with a partition by using a function that respects the group of allowed domain permutations (i.e. symmetries of the system).

Definition 5.1 (Invariant Partition Generators) *A function $\mathcal{G} : \mathcal{S} \to \mathfrak{P}$ that maps each state to a permutable primitive type partition is an* invariant partition generator *if*

$$\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s}))$$

holds for all allowed domain permutations $\theta \in \Theta$.

That is, for permuted states the partition assigned by $\mathcal{G}$ should be similarly permuted. We will develop a way to produce such functions in Sec. 5.3.

Next, we select an allowed domain permutation according to the partition produced by an (fixed) invariant partition generator. The set of allowed domain permutations from which we have to select is given by the following compatibility definition.

Definition 5.2 (Compatibility) *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 $\mathscr{D}_T = \{v_1, \ldots, v_n\}$ 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_1.*
- *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.

To sum up, assuming an arbitrary but *fixed* invariant partition generator $\mathcal{G}$, the procedure for generating (not necessarily canonical) representative states is described in Alg. 1.

Algorithm 1 Representative algorithm 1

Input: A state $\mathfrak{s}$
Output: A representative state that is symmetric to $\mathfrak{s}$
Require: An invariant partition generator $\mathcal{G}$
1: Compute the partition $\mathfrak{p} = \mathcal{G}(\mathfrak{s})$
2: Choose any allowed domain permutation θ that is compatible with $\mathfrak{p}$
3: Return $\theta(\mathfrak{s})$ as the representative state

The following lemma and corollary show that this algorithm preserves the possibility for perfect reduction in the sense that, for all symmetric states, it is possible to choose the same representative state. That is, there is nothing inherent in the algorithm that would force the set of representative states for a set of mutually symmetric states to contain more than one state. Of course, choosing the right allowed domain permutations in line 2 of Alg. 1 may require an exponential amount of work or very good luck, but nevertheless it is possible to do so.

Lemma 5.3 *Let $\hat{\theta}$ be an allowed domain permutation compatible with a partition $\mathfrak{p}$. Then for each allowed domain permutation θ it holds that the allowed domain permutation $\hat{\theta} * \theta^{-1}$ is compatible with the partition $\theta(\mathfrak{p})$.*

Corollary 5.4 *Assume an invariant partition generator $\mathcal{G}$, and take two symmetric states, $\mathfrak{s}_1$ and $\mathfrak{s}_2$. Let $\hat{\theta}_1$ be an allowed domain permutation compatible with the partition $\mathcal{G}(\mathfrak{s}_1)$. Then there is an allowed domain permutation $\hat{\theta}_2$ compatible with the partition $\mathcal{G}(\mathfrak{s}_2)$ such that $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\mathfrak{s}_2)$.*

Example 5.5 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 Fig. 1 (recall Ex. 2.1 and Ex. 3.1). Assume an invariant partition generator $\mathcal{G}$ that produces the partition

$$\mathcal{G}(\mathfrak{s}) = \left(\mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Secs}} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

for $\mathfrak{s}$. 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 domain permutations compatible with the partition are

$$\begin{aligned}
\theta_1 &= \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \\
\theta_2 &= \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \\
\theta_3 &= \left(\theta_3^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_3^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\
\theta_4 &= \left(\theta_4^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_4^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).
\end{aligned}$$

The corresponding possible representative states for $\mathfrak{s}$ are:

$$\begin{aligned}
\theta_1(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \mathfrak{s}, \\
\theta_2(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\}, \\
\theta_3(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_5\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_3\}, \text{ and} \\
\theta_4(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_5\rangle, V \mapsto s_2 + s_3\}.
\end{aligned}$$

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 invariant partition generator $\mathcal{G}$ must assign the partition $\theta(\mathcal{G}(\mathfrak{s}))$ to $\mathfrak{s}'$, i.e.

$$\mathcal{G}(\mathfrak{s}') = \theta(\mathcal{G}(\mathfrak{s})) = \left(\mathfrak{p}_{\mathfrak{s}',4}^{\mathsf{Secs}} = [\{s_3, s_5\}, \{s_1, s_2\}, \{s_0, s_4\}], \mathfrak{p}_{\mathfrak{s}',4}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

The four possible domain permutations compatible with the partition are

$$\begin{aligned}
\theta_{1'} &= \left(\theta_{1'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta_{1'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_2 * \theta^{-1}, \\
\theta_{2'} &= \left(\theta_{2'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta_{2'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_1 * \theta^{-1}, \\
\theta_{3'} &= \left(\theta_{3'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{3'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_4 * \theta^{-1}, \text{ and} \\
\theta_{4'} &= \left(\theta_{4'}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{4'}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_3 * \theta^{-1}.
\end{aligned}$$

The corresponding possible representative states for $\mathfrak{s}'$ are:

$$\begin{aligned}
\theta_{1'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\} = \theta_2(\mathfrak{s}), \\
\theta_{2'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \theta_1(\mathfrak{s}), \\
\theta_{3'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_5\rangle, V \mapsto s_2 + s_3\} = \theta_4(\mathfrak{s}), \text{ and} \\
\theta_{4'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_5\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_3\} = \theta_3(\mathfrak{s}).
\end{aligned}$$

Thus the sets of possible representative states for $\mathfrak{s}$ and $\mathfrak{s}'$ are the same. This was expected because of Lemma 5.3, Corollary 5.4, and the fact that $\mathfrak{s}$ and $\mathfrak{s}'$ are symmetric. ♣

Producing Canonical Representative States

Assuming that the set $\mathcal{S}$ of states is totally ordered, algorithm Alg. 1 can be extended to produce canonical representative states. For a state $\mathfrak{s}$, simply compute the partition $\mathcal{G}(\mathfrak{s})$ and then check all the allowed domain permutations θ that are compatible with $\mathcal{G}(\mathfrak{s})$ and return the smallest state $\theta(\mathfrak{s})$ found as the representative state. By Cor. 5.4, this procedure produces canonical representative states.

5.3 Partition Refiners and Invariants

We now focus on building invariant partition generators. First, we introduce partition refiners. 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 5.6 (Partition Refiners) 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*.*

Partition refiners can be composed:

Lemma 5.7 *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.*

This implies that any finite sequence $\mathcal{R}_n \star \mathcal{R}_{n-1} \star \cdots \star \mathcal{R}_1$ of partition refiners, read by $\mathcal{R}_n \star (\mathcal{R}_{n-1} \star (\cdots (\mathcal{R}_2 \star \mathcal{R}_1)))$ or $\mathcal{R}_n(\mathfrak{s}, \mathcal{R}_{n-1}(\mathfrak{s}, \cdots (\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p})) \ldots))$, is also a partition refiner. That is, we first refine the argument partition with the first refiner, then refine the resulting partition with the second refiner, and so on. When a partition refiner is applied to the unit partition, the result is an invariant partition generator.

Theorem 5.8 *For a partition refiner* $\mathcal{R}$*, the function* $\mathcal{G}_{\mathcal{R}}(\mathfrak{s}) = \mathcal{R}(\mathfrak{s}, \mathfrak{p}_0)$*, where* $\mathfrak{p}_0 = \left(\mathfrak{p}_0^T = [\mathscr{D}_T]\right)_{T \in \mathcal{T}_P}$ *is the unit partition, is an invariant partition generator.*

Now the task of building invariant 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. Formally, we define the following:

Definition 5.9 (Invariants) *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* $<$*.*

We say that 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 5.10 (Type Invariants) 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 5.11 *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.*

Example 5.12 Define 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'", by the following rules:

1. If T' is primitive type, then $\sharp_T^{T'}(v, v') = 1$ if $T = T'$ and $v = v'$, and 0 otherwise.
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. $\sharp_T^{\mathsf{Multi\text{-}Set}(T')}(v, m) = \sum_{v' \in \mathscr{D}_{T'}} m(v') \cdot \sharp_T^{T'}(v, v')$
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 $\mathfrak{s}$. ♣

We will introduce more invariants later, including some partition dependent ones, too. Given an invariant for a permutable primitive type T and a partition $\mathfrak{p}$, we may refine the partition 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 5.13 *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) *$incell(\mathfrak{p}_{\text{ref}}^T, v) = incell(\mathfrak{p}_{\text{ref}}^T, v')$ iff $incell(\mathfrak{p}^T, v) = incell(\mathfrak{p}^T, v')$ and $I(v, \mathfrak{s}, \mathfrak{p}) = I(v', \mathfrak{s}, \mathfrak{p})$, and*
   (b) *if $incell(\mathfrak{p}_{\text{ref}}^T, v) < incell(\mathfrak{p}_{\text{ref}}^T, v')$, then 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 5.14 *The function $\mathcal{R}_I$ is a partition refiner.*

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 $\mathcal{R}_I(\mathfrak{s}, \mathfrak{p})$, we say that $\mathfrak{p}$ is *refined according to* I. Given a sequence $I_1.I_2.\ldots.I_n$ of invariants (for arbitrary primitive types), we say that a partition $\mathfrak{p}$ is *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, we obtain an invariant partition generator by (i) defining a sequence $I_1.I_2.\ldots.I_n$ of invariants, and (ii) refining the unit partition according to the sequence (by Lemma 5.14 and Thm. 5.8).

Example 5.15 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 Fig. 1 (cf Ex. 5.5). Initially, the partition is the unit partition

$$\mathfrak{p}_{\mathfrak{s},0} = \left(\mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Secs}} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right) .$$

We now apply the invariant sequence $I_{\sharp\mathsf{Trains}\text{ in }U} . I_{\sharp\mathsf{Trains}\text{ in }V} . I_{\sharp\mathsf{Secs}\text{ in }U} . I_{\sharp\mathsf{Secs}\text{ in }V}$ to the partition (recall Ex. 5.12). Refining the partition for Trains according to the invariant $I_{\sharp\mathsf{Trains}\text{ in }U}$ leads to

$$\mathfrak{p}_{\mathfrak{s},1} = \left(\mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Secs}} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right) ,$$

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 $I_{\sharp\mathsf{Trains}\text{ in }V}$ changes nothing. Refining the partition for Secs according to the invariant $I_{\sharp\mathsf{Secs}\text{ in }U}$ leads to

$$\mathfrak{p}_{\mathfrak{s},3} = \left(\mathfrak{p}_{\mathfrak{s},3}^{\mathsf{Secs}} = [\{s_0, s_2, s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}_{\mathfrak{s},3}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right) ,$$

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 $I_{\sharp\mathsf{Secs}\text{ in }V}$ gives us

$$\mathfrak{p}_{\mathfrak{s},4} = \left(\mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Secs}} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}_{\mathfrak{s},4}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right) .$$

Applying the same sequence of invariants to the state $\mathfrak{s}' = \theta(\mathfrak{s}) = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_4\rangle, V \mapsto s_1 + s_2\}$, where θ is as defined in Ex. 5.5, gives the partition

$$\mathfrak{p}_{\mathfrak{s}',4} = \theta(\mathfrak{p}_{\mathfrak{s},4}) = \left(\mathfrak{p}_{\mathfrak{s}',4}^{\mathsf{Secs}} = [\{s_3, s_5\}, \{s_1, s_2\}, \{s_0, s_4\}], \mathfrak{p}_{\mathfrak{s}',4}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right) .$$

♣

5.4 Some Useful Invariants

A Successor Based Invariant for Cyclic Primitive Types

We are now ready to introduce our first partition *dependent* invariant. 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

$$I_{T,\mathrm{succ}}(v, \mathfrak{s}, \mathfrak{p}) = \mathit{incell}(\mathfrak{p}^T, \mathit{succ}_T(v))$$

i.e. the function that returns the cell number of the successor element of the element v in the partition $\mathfrak{p}$. That is, $I_{T,\mathrm{succ}}$ distinguishes between two elements if their successors are already distinguished in the current partition $\mathfrak{p}$.

Lemma 5.16 *The function $I_{T,\mathrm{succ}}$ is an invariant.*

Therefore, if we have refined the initial partition according to an invariant sequence, we may further refine the resulting partition by using the invariant $I_{T,\mathrm{succ}}$. The resulting partition may again be further refined by the same invariant until no refinement happens i.e. we reach a fixed point (in other words, the sequence of length $|\mathscr{D}_T|$ of invariant $I_{T,\mathrm{succ}}$ is applied). Note that, while $I_{T,\mathrm{succ}}$ is partition dependent, it does not depend on the state argument i.e. is *state independent*.

Example 5.17 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

$$\mathfrak{p}_{\mathfrak{s},4} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},4} = [\{s_0, s_2\}, \{s_4, s_5\}, \{s_1, s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},4} = [\{t_a, t_b\}]\right)$$

for it given in Ex. 5.15. Evaluating the invariant $I_{\mathsf{Secs},\mathrm{succ}}$ in the partition gives

$$\begin{array}{ll} I_{\mathsf{Secs},\mathrm{succ}}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 3, & I_{\mathsf{Secs},\mathrm{succ}}(s_1, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 1, \\ I_{\mathsf{Secs},\mathrm{succ}}(s_2, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 3, & I_{\mathsf{Secs},\mathrm{succ}}(s_3, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 2, \\ I_{\mathsf{Secs},\mathrm{succ}}(s_4, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 2, \text{ and} & I_{\mathsf{Secs},\mathrm{succ}}(s_5, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},4}) = 1. \end{array}$$

Refining according to this results in the partition

$$\mathfrak{p}_{\mathfrak{s},5} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},5} = [\{s_0, s_2\}, \{s_5\}, \{s_4\}, \{s_1\}, \{s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},5} = [\{t_a, t_b\}]\right).$$

Further evaluating the invariant $I_{\mathsf{Secs},\mathrm{succ}}$ in this partition gives

$$\begin{array}{ll} I_{\mathsf{Secs},\mathrm{succ}}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 4, & I_{\mathsf{Secs},\mathrm{succ}}(s_1, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 1, \\ I_{\mathsf{Secs},\mathrm{succ}}(s_2, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 5, & I_{\mathsf{Secs},\mathrm{succ}}(s_3, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 3, \\ I_{\mathsf{Secs},\mathrm{succ}}(s_4, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 2, \text{ and} & I_{\mathsf{Secs},\mathrm{succ}}(s_5, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},5}) = 1. \end{array}$$

Refining according to this results in the partition

$$\mathfrak{p}_{\mathfrak{s},6} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s},6} = [\{s_0\}, \{s_2\}, \{s_5\}, \{s_4\}, \{s_1\}, \{s_3\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s},6} = [\{t_a, t_b\}]\right).$$

Now there are only two domain permutations compatible with the partition:

$$\begin{array}{rcl} \theta_1 & = & \left(\theta^{\mathsf{Secs}}_1 = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_1 = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\ \theta_2 & = & \left(\theta^{\mathsf{Secs}}_2 = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_2 = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right). \end{array}$$

The corresponding possible representative states for $\mathfrak{s}$ are:

$$\begin{array}{rcl} \theta_1(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \mathfrak{s}, \text{ and} \\ \theta_2(\mathfrak{s}) & = & \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\}. \end{array}$$

Applying the same successor invariants to the state

$$\mathfrak{s}' = \{U \mapsto \langle t_a, s_0\rangle + \langle t_b, s_4\rangle, V \mapsto s_1 + s_2\}$$

in the partition

$$\mathfrak{p}_{\mathfrak{s}',4} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s}',4} = [\{s_3, s_5\}, \{s_1, s_2\}, \{s_0, s_4\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s}',4} = [\{t_a, t_b\}]\right)$$

results in

$$\mathfrak{p}_{\mathfrak{s}',6} = \left(\mathfrak{p}^{\mathsf{Secs}}_{\mathfrak{s}',6} = [\{s_3\},\{s_5\},\{s_2\},\{s_1\},\{s_4\},\{s_0\}], \mathfrak{p}^{\mathsf{Trains}}_{\mathfrak{s}',6} = [\{t_a,t_b\}]\right).$$

Now the two domain permutations compatible with the partition are:

$$\begin{aligned}\theta_{1'} &= \left(\theta^{\mathsf{Secs}}_{1'} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2\end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{1'} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b\end{smallmatrix}\right)\right), \text{ and} \\ \theta_{2'} &= \left(\theta^{\mathsf{Secs}}_{2'} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2\end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{2'} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a\end{smallmatrix}\right)\right).\end{aligned}$$

The corresponding possible representative states for $\mathfrak{s}'$ are:

$$\begin{aligned}\theta_{1'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_3\rangle + \langle t_b, s_1\rangle, V \mapsto s_4 + s_5\} = \theta_2(\mathfrak{s}), \text{ and} \\ \theta_{2'}(\mathfrak{s}') &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, V \mapsto s_4 + s_5\} = \theta_1(\mathfrak{s}) = \mathfrak{s}.\end{aligned}$$

♣

Ordered Structured Types

Consider a structured type T' composed only of primitive types, lists, structures and unions. Now the value tree (recall Sec. 4) for any $v' \in \mathscr{D}_{T'}$ is ordered in the sense that the children of each node can be totally ordered by the arc 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 Fig. 8 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. 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 earlier.

The above procedure does not work for structured types composed of sets, multi-sets or association arrays. This is because the value tree is not ordered in the above sense and therefore we cannot assign the nodes in it a unique numbering. 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 a normal array of size 3 (with possibly undefined elements), and the elements

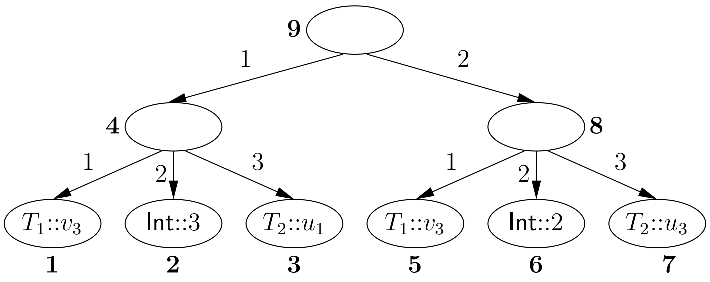


Figure 8: An ordered value tree

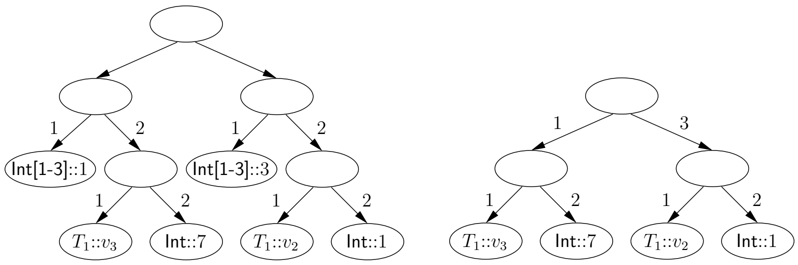


Figure 9: Mapping an unordered value tree into an ordered one

in it are totally ordered. In this kind of case the value tree can be modified to be ordered, as shown in Fig. 9 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 multi-set types which are not handled by the above procedure. Yet the above procedure can be applied to multi-sets over the "easy" structured types in some important special cases: if a multi-set contains only one element or all the elements in the multi-set 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 $\mathsf{Multi\text{-}Set}(T)$, where T is a permutable primitive type. Define the partition independent invariant $I_{\text{multiplicity}} : \mathscr{D}_T \times \mathscr{D}_{\mathsf{Multi\text{-}Set}(T)} \times \mathfrak{P} \to \mathbb{N}$ 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 $\theta_2^{\mathsf{Multi\text{-}Set}(T)}(m) = \theta_3^{\mathsf{Multi\text{-}Set}(T)}(m)$ 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 some sense canonizes the multi-set value m.

Hash-Like Invariants

The invariants we have been using so far have been quite simple. We could introduce more complicated special invariants, but there are too many of them to cover all imaginable cases. For instance, assuming a state variable x of type $\mathsf{Multi\text{-}Set}(\mathsf{Struct}(\mathsf{Int}, T))$, where Int with $\mathscr{D}_{\mathsf{Int}} = \{0, 1, 2, \ldots\}$ is an ordered primitive type, the function $I_{T,x,\langle 3,v\rangle}$ for the type T defined by $I_{T,x,\langle 3,v\rangle}(v, \mathfrak{s}, \mathfrak{p}) = \mathfrak{s}(x)(\langle 3, v\rangle)$, 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. We now show how to calculate a general purpose invariant that depends on the structure of a state in a degree larger than the earlier 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, we define a function

$$\mathfrak{g}_T(v, T', v', \mathfrak{p})$$

over four arguments: 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 we compute the "hash value", while the second and third arguments describe the object in which we perform this computation. 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 into which the third argument belongs.

Firstly, we assume an *associative and commutative* binary operation $\oplus$ on $\mathbb{Z}$. Furthermore, for a 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' with $\mathscr{D}_{T'} = \{v_1, \ldots, v_n\}$,

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = \begin{cases} h_{T'}(\mathit{incell}(\mathfrak{p}^{T'}, v')) & \text{if } T \neq T' \\ h_{T',2}(k, \mathit{incell}(\mathfrak{p}^{T'}, v')) & \text{if } T = T' \text{ and } v' \text{ is the } k\text{-successor of } v. \end{cases}$$

3. For an unordered primitive type T',

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = \begin{cases} h_{T',2}(\mathit{incell}(\mathfrak{p}^{T'}, v'), 0) & \text{if } T \neq T' \text{ or } T = T' \wedge v \neq v' \\ h_{T',2}(\mathit{incell}(\mathfrak{p}^{T'}, v'), 1) & \text{if } T = T' \text{ and } v = v'. \end{cases}$$

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 multi-set type $T' = \mathsf{Multi\text{-}Set}(T_1)$,

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = h_{T'}\left(\bigoplus_{v'' \in \mathscr{D}_{T_1}, v'(v'') \geq 1} h_{T',2}(v'(v''), \mathfrak{g}_T(v, T_1, v'', \mathfrak{p}))\right).$$

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 5.18 *If $\theta = \left(\theta^T\right)_{T \in \mathcal{T}}$ is an allowed domain permutation, then*

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})).$$

Corollary 5.19 *For a permutable primitive type T and for a type T',*

$$I_{T,\text{hash in } T'}(v, v', \mathfrak{p}) = \mathfrak{g}_T(v, T', v', \mathfrak{p})$$

is a type invariant. Similarly, if x is a state variable of type T', then

$$I_{T,\text{hash in } x}(v, \mathfrak{s}, \mathfrak{p}) = \mathfrak{g}_T(v, T', \mathfrak{s}(x), \mathfrak{p})$$

is an invariant.

Example 5.20 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 Fig. 1, recall Examples 5.5, 5.15 and 5.17. Let $\oplus$ be the integer addition operation, and let

$h_{\mathsf{Trains},2}(1,0) = 374,$	$h_{\mathsf{Trains},2}(2,0) = 1374,$
$h_{\mathsf{Trains},2}(1,1) = 242 \cdot 374,$	$h_{\mathsf{Trains},2}(2,1) = 242 \cdot 1374,$
$h_{\mathsf{Secs},2}(k,1) = (k+1) \cdot 837,$	$h_{\mathsf{Secs},2}(k,2) = (k+1) \cdot 274,$
$h_{\mathsf{Secs},2}(k,3) = (k+1) \cdot 97,$	$h_{\mathsf{Secs},2}(k,4) = (k+1) \cdot 4732,$
$h_{\mathsf{Secs},2}(k,5) = (k+1) \cdot 194,$	$h_{\mathsf{Secs},2}(k,6) = (k+1) \cdot 958,$
$h_{\mathsf{Multi\text{-}Set(Struct(Trains,Secs)}}(x) = x,$	$h_{\mathsf{Multi\text{-}Set(Struct(Trains,Secs)),2}}(x,y) = x \cdot y,$
$h_{\mathsf{Struct(Trains,Secs),2}}(x,y) = x \cdot \lfloor \frac{y}{2} \rfloor.$	

Initially, the partition is

$$\mathfrak{p}_{\mathfrak{s},0} = \left(\mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Secs}} = [\{s_0, s_1, s_2, s_3, s_4, s_5\}], \mathfrak{p}_{\mathfrak{s},0}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

When we evaluate the invariant $I_{\mathsf{Secs},\text{hash in } U}$ in the partition, we get

$$\begin{aligned}
I_{\mathsf{Secs},\text{hash in } U}(s_0, \mathfrak{s}, \mathfrak{p}_{\mathfrak{s},0}) &= \\
\mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Multi\text{-}Set(Struct(Trains, Secs))}, \langle t_a, s_1\rangle + \langle t_b, s_3\rangle, \mathfrak{p}_{\mathfrak{s},0}) &= \\
1 \cdot \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Struct(Trains, Secs)}, \langle t_a, s_1\rangle, \mathfrak{p}_{\mathfrak{s},0}) & \\
+1 \cdot \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Struct(Trains, Secs)}, \langle t_b, s_3\rangle, \mathfrak{p}_{\mathfrak{s},0}) &= \\
1 \cdot (\mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Trains}, t_a, \mathfrak{p}_{\mathfrak{s},0}) \cdot \lfloor \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Secs}, s_1, \mathfrak{p}_{\mathfrak{s},0})/2 \rfloor) & \\
+1 \cdot (\mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Trains}, t_b, \mathfrak{p}_{\mathfrak{s},0}) \cdot \lfloor \mathfrak{g}_{\mathsf{Secs}}(s_0, \mathsf{Secs}, s_3, \mathfrak{p}_{\mathfrak{s},0})/2 \rfloor) &= \\
1 \cdot (h_{\mathsf{Trains}}(1,0) \cdot \lfloor h_{\mathsf{Secs},2}(1,1)/2 \rfloor) + 1 \cdot (h_{\mathsf{Trains}}(1,0) \cdot \lfloor h_{\mathsf{Secs},2}(3,1)/2 \rfloor) &= \\
1 \cdot (374 \cdot \lfloor (2 \cdot 837)/2 \rfloor) + 1 \cdot (374 \cdot \lfloor (4 \cdot 837)/2 \rfloor) &= \\
1 \cdot (374 \cdot 837) + 1 \cdot (374 \cdot 1674) &= \\
939114, &
\end{aligned}$$

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 into

$$\mathfrak{p}_{\mathfrak{s},1} = \left(\mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Secs}} = [\{s_1\},\{s_3\},\{s_0\},\{s_5\},\{s_2\},\{s_4\}], \mathfrak{p}_{\mathfrak{s},1}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right).$$

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 into

$$\mathfrak{p}_{\mathfrak{s},2} = \left(\mathfrak{p}_{\mathfrak{s},2}^{\mathsf{Secs}} = [\{s_1\},\{s_3\},\{s_0\},\{s_5\},\{s_2\},\{s_4\}], \mathfrak{p}_{\mathfrak{s},2}^{\mathsf{Trains}} = [\{t_b\},\{t_a\}]\right).$$

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 very optimal since they are quite similar. Although they suffice for demonstrative purposes, in a real implementation some bit twisting operations should be applied instead in order to reduce the possibility of value collision. The main thing to take care of is that the operation $\oplus$ is commutative and associative. The h-functions may, for instance, employ pseudo-random numbers to obtain relative independence from each other.

5.5 Limitations of Invariant Partition Generators

We now study some fundamental, inherent limitations involved in the use of invariant partition generators. Directly from the definition of invariant partition generators we observe the following. Let $\mathfrak{s}$ and $\mathfrak{s}'$ be two symmetric states and let $\mathcal{G}$ be an invariant partition generator. If θ is an allowed domain permutation mapping $\mathfrak{s}$ to $\mathfrak{s}'$, then θ maps the cells in the partition $\mathcal{G}(\mathfrak{s})$ to the corresponding cells in $\mathcal{G}(\mathfrak{s}')$ since $\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s})) \Rightarrow \mathcal{G}(\mathfrak{s}') = \theta(\mathcal{G}(\mathfrak{s}))$. From this fact we are able to obtain a lower limit for the sizes of cells in partitions produced by any invariant partition generator.

Fact 5.21 *Let $\theta = \left(\theta^T\right)_{T\in\mathcal{T}_P}$ be a self-symmetry of a state $\mathfrak{s}$, i.e. $\theta(\mathfrak{s}) = \mathfrak{s}$ or, equivalently, $\theta \in \mathrm{Stab}(\Theta, \mathfrak{s})$. Then $\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s}))$ implies $\mathcal{G}(\mathfrak{s}) = \theta(\mathcal{G}(\mathfrak{s}))$ for any invariant partition generator $\mathcal{G}$. Thus each self-symmetry of $\mathfrak{s}$ respects the cells in $\mathcal{G}(\mathfrak{s})$, meaning that if $v \in \mathscr{D}_T$ belongs to the cell C_i^T in the partition $\mathcal{G}(\mathfrak{s})$, then $\theta^T(v)$ belongs to the cell C_i^T, too.*

Optimal invariant 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 we could always compute such functions efficiently, we 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 us that a graph has non-trivial automorphisms. For this task we know no polynomial-time algorithms.

6 IMPROVEMENTS BASED ON SEARCH TREES

Recall the Algorithm 1 for producing representative states. Assuming a fixed invariant partition generator $\mathcal{G}$, given a state $\mathfrak{s}$, we produce the partition $\mathcal{G}(\mathfrak{s})$ and take arbitrarily an allowed domain permutation θ that is compatible with it and return $\theta(\mathfrak{s})$ as the representative. In the case the partition $\mathcal{G}(\mathfrak{s})$ 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 $\mathcal{G}(\mathfrak{s})$ 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 in any way the $n-1$ choices we must make for the rest of the elements. Nor does it affect the choices that have to be made for other non-singleton cells. We now present 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.

First, we assume a fixed invariant partition generator $\mathcal{G}$ and a fixed partition refiner $\mathcal{R}$.

Definition 6.1 (Search Trees) *The* search tree *of a state $\mathfrak{s}$ and a partition $\mathfrak{p} = \left(\mathfrak{p}^T\right)_{T \in \mathcal{T}_P}$ is a tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ defined by the following inductive rules.*

1. *If each partition $\mathfrak{p}^T$ in $\mathfrak{p}$ is discrete, then the tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ 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 $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ then consists of the root node $\mathfrak{p}$ which has as its children the trees $\mathcal{T}(\mathfrak{s}, \mathcal{R}(\mathfrak{s}, \mathfrak{p}_j))$, 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 T is*

$$\mathfrak{p}_j^T = [C_1^T, \ldots, C_{i-1}^T, \{v_{i,j}\}, C_i^T \setminus \{v_{i,j}\}, C_{i+1}^T, \ldots].$$

In other words, for each element in the first non-discrete cell C_i^T, we split the cell in two parts by distinguishing the element into its own cell and refine the resulting partition with the refinement invariants. The child $\mathcal{T}(\mathfrak{s}, \mathcal{R}(\mathfrak{s}, \mathfrak{p}_j))$ above is called the $v_{i,j}$-child of the node $\mathfrak{p}$ and the edge from $\mathfrak{p}$ to it is labeled with $T.v_{i,j}$. We use $\mathfrak{p} \xrightarrow{T.v} \mathfrak{p}'$ to denote that $\mathfrak{p}'$ is a v-child of $\mathfrak{p}$.

The search tree $\mathcal{T}(\mathfrak{s})$ *of a state* $\mathfrak{s}$ *is the search tree* $\mathcal{T}(\mathfrak{s},\mathcal{G}(\mathfrak{s}))$.

We now modify our representative algorithm Alg. 1 as follows. Given a state $\mathfrak{s}$, we *travel along one, arbitrary path in the search tree* $\mathcal{T}(\mathfrak{s})$ *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 Alg. 2.

Algorithm 2 Representative algorithm 2

Input: A state $\mathfrak{s}$
Output: A representative state that is symmetric to $\mathfrak{s}$
1: Build the partition $\mathfrak{p} = \mathcal{G}(\mathfrak{s})$
2: Choose any path in the search tree $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ 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 6.2 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 Fig. 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 invariant partition generator) gives us 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 invariant 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

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

is a self-symmetry of $\mathfrak{s}$ (recall Fact 5.21). Assuming a 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 domain permutations compatible with the partition are

$$\begin{aligned}
\theta_1 &= \left(\theta_1^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_1^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \\
\theta_2 &= \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right), \\
\theta_3 &= \left(\theta_3^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_3^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right), \text{ and} \\
\theta_4 &= \left(\theta_4^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_4^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right).
\end{aligned}$$

The corresponding two possible representative states for $\mathfrak{s}$ are:

$$\begin{aligned}
\theta_1(\mathfrak{s}) = \theta_4(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\} \text{ and} \\
\theta_2(\mathfrak{s}) = \theta_3(\mathfrak{s}) &= \{U \mapsto \langle t_a, s_1\rangle + \langle t_b, s_4\rangle, V \mapsto s_2 + s_5\}.
\end{aligned}$$

Assume the partition refiner $\mathcal{R}$ that is induced by the invariant sequence that first contains 6 $I_{\mathsf{Secs,succ}}$ invariants and after those enough hash-like invariants described in Sec. 5.4. The search tree $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ 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 into

$$\mathfrak{p}_{1,1} = \left(\mathfrak{p}_{1,1}^{\mathsf{Secs}} = [\{s_2\}, \{s_5\}, \{s_1, s_4\}, \{s_0, s_3\}], \mathfrak{p}_{1,1}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right)$$

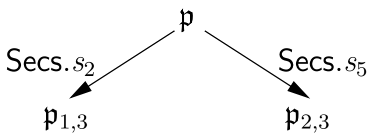


Figure 10: A search tree

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 6 invariants $I_{\mathsf{Secs},\mathrm{succ}}$ gives us

$$\mathfrak{p}_{1,2} = \left(\mathfrak{p}_{1,2}^{\mathsf{Secs}} = [\{s_2\}, \{s_5\}, \{s_1\}, \{s_4\}, \{s_0\}, \{s_3\}], \mathfrak{p}_{1,2}^{\mathsf{Trains}} = [\{t_a, t_b\}]\right)$$

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 $I_{\mathsf{Secs},\mathrm{hash\ in}\ U}$ or $I_{\mathsf{Secs},\mathrm{hash\ in}\ V}$ improves nothing since the partitions for Secs are already discrete. However, refining the partitions with the $I_{\mathsf{Trains},\mathrm{hash\ in}\ U}$ invariant, by using the functions of Ex. 5.20, 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 $\mathcal{T}(\mathfrak{s})$, shown in Fig. 10, and the allowed domain permutations compatible with them are

$$\begin{aligned} \theta_{1,3} &= \left(\theta_{1,3}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_4 & s_5 & s_0 & s_1 & s_2 & s_3 \end{smallmatrix}\right), \theta_{1,3}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) = \theta_1, \text{ and} \\ \theta_{2,3} &= \left(\theta_{2,3}^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_{2,3}^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right) = \theta_4. \end{aligned}$$

The corresponding representative state for $\mathfrak{s}$ is:

$$\theta_{1,3}(\mathfrak{s}) = \theta_{2,3}(\mathfrak{s}) = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}.$$

♣

6.1 Properties of Search Trees

We now list some properties of search trees.

Theorem 6.3 *For each allowed domain permutation θ, a partition $\mathfrak{p}_{child}$ is a v-child of the root node of the search tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ iff the partition $\theta(\mathfrak{p}_{child})$ is a $\theta^T(v)$-child of the root node of the search tree $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$.*

Corollary 6.4 *For all allowed domain permutations 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 $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ iff $\theta(\mathfrak{p}) \xrightarrow{T_1.\theta^{T_1}(v_1)} \theta(\mathfrak{p}_1) \cdots \xrightarrow{T_n.\theta^{T_n}(v_n)} \theta(\mathfrak{p}_n)$ is a path in the search tree $\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$.*

Corollary 6.5 *For all allowed domain permutations θ, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ iff the partition $\theta(\mathfrak{p}')$ is a node in the search tree $\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))$.*

Since $\theta(\mathcal{G}(\mathfrak{s})) = \mathcal{G}(\theta(\mathfrak{s}))$ for the invariant partition generator $\mathcal{G}$, the above results generalize to search trees for states, for instance:

Corollary 6.6 *For all allowed domain permutations θ, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{T}(\mathfrak{s})$ iff the partition $\theta(\mathfrak{p}')$ is a node in the search tree $\mathcal{T}(\theta(\mathfrak{s}))$.*

Corollary 6.7 *For all self-symmetries $\theta \in \mathrm{Stab}(\Theta,\mathfrak{s})$ of a state $\mathfrak{s}$, a partition $\mathfrak{p}'$ is a node in the search tree $\mathcal{T}(\mathfrak{s})$ iff 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, we have some additional properties. First of all, all the nodes in a search tree are mutually distinct partitions. Furthermore, each descendant of a node is a cell order preserving refinement of the node. 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 $\mathfrak{p}_1 \preceq \mathfrak{p}_2$. Therefore, *the number of possible representative states for Alg. 2 is at most that for Alg. 1* (when the same invariant partition generator is used). In addition, by Cor. 6.7 it holds that the number of leaf nodes in the search tree $\mathcal{T}(\mathfrak{s})$ 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 $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ defines the set $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$ of states by: if $\mathfrak{p}_{\mathrm{leaf}}$ is a leaf node in $\mathcal{T}(\mathfrak{s},\mathfrak{p})$, then and only then the corresponding state $\hat{\theta}_{\mathfrak{p}_{\mathrm{leaf}}}(\mathfrak{s})$ is in $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$. Define $\mathcal{S}_{\mathcal{T}(\mathfrak{s})} = \mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathcal{G}(\mathfrak{p}))}$.

Lemma 6.8 *For any allowed domain permutation θ, $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))}$ and consequently $\mathcal{S}_{\mathcal{T}(\mathfrak{s})} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}))}$*

This implies that the sets of possible representative states returned by Alg. 2 for symmetric states are the same, i.e. Alg. 2 preserves the possibility for perfect reduction.

6.2 Producing Canonical Representative States

Although Alg. 2 is better than Alg. 1, it does not necessarily produce canonical representative markings. In order to accomplish this, we assume a total order $<$ on the set $\mathcal{S}$ of states. Given a state $\mathfrak{s}$, we can now select the smallest state w.r.t. the order $<$ in the set $\mathcal{S}_{\mathcal{T}(\mathfrak{s})}$ as the representative state. This can be done by performing a depth-first search in the search tree $\mathcal{T}(\mathfrak{s})$. This procedure produces canonical representative states because $\mathcal{S}_{\mathcal{T}(\mathfrak{s})} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}))}$ for any allowed domain permutation θ as stated by Lemma 6.8. The problem is that the search tree can have exponentially many nodes, at least it has $|\mathrm{Stab}(\Theta,\mathfrak{s})|$ nodes by Corollary 6.7. Fortunately, we can prune the search tree with some techniques adapted from the graph isomorphism algorithms, see e.g. [McKay 1981; Kreher and Stinson 1999].

Pruning by Image Restriction. Assume that $\mathfrak{s}_{best}$ is the smallest state in $\mathcal{S}_{\mathcal{T}(\mathfrak{s})}$ found so far during the the search tree traversal. If the current partition whose children are not yet traversed is $\mathfrak{p}$ and we can deduce that all the states in $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$ must be larger than $\mathfrak{s}_{best}$, then we can backtrack i.e. skip the sub-tree $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ of $\mathcal{T}(\mathfrak{s})$. Deducing that all the states in $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$ 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 $\mathfrak{p}$. Thus if a state is in $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$, 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 $\mathscr{D}_T = \{v_1, \ldots, v_n\}$, then θ must map an element in C_1^T to v_1. Thus the possible images of the elements of permutable primitive types are restricted by $\mathfrak{p}$ and we may be able to deduce that all states in $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$ must be larger than $\mathfrak{s}_{best}$. Of course, this deduction step depends on the selected total order $<$ on the states.

Pruning with Self-Symmetries. Consider the root node $\mathfrak{p}$ of a sub-tree $\mathcal{T}(\mathfrak{s},\mathfrak{p})$ in the search tree $\mathcal{T}(\mathfrak{s})$. Assume that it has two children, e.g. $\mathfrak{p} \xrightarrow{v} \mathfrak{p}_1$ and $\mathfrak{p} \xrightarrow{v'} \mathfrak{p}_2$. If there is a self-symmetry θ of $\mathfrak{s}$ that (i) respects $\mathfrak{p}$, i.e. $\theta(\mathfrak{p}) = \mathfrak{p}$, and (ii) maps v to v', then $\theta(\mathfrak{p}_1) = \mathfrak{p}_2$. Now $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p}_1)} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}_1))} = \mathcal{S}_{\mathcal{T}(\mathfrak{s},\theta(\mathfrak{p}_1))} = \mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p}_2)}$, meaning that the possible representative states in the sub-trees $\mathcal{T}(\mathfrak{s},\mathfrak{p}_1)$ and $\mathcal{T}(\mathfrak{s},\mathfrak{p}_2)$ are the same. Therefore, if we have already traversed the sub-tree $\mathcal{T}(\mathfrak{s},\mathfrak{p}_1)$, we do not have to traverse the sub-tree $\mathcal{T}(\mathfrak{s},\mathfrak{p}_2)$ in our quest for the smallest state.

The question now is, how do we obtain the self-symmetries of a state? It turns out that they can be found during the search tree traversal. Assume that we have already visited a leaf node $\mathfrak{p}_1$ in the search tree. If we are currently visiting a leaf node $\mathfrak{p}_2$ 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 self-symmetry 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 w.r.t. $<$ encountered so far. As for every leaf node $\mathfrak{p}$ in the search tree and for every self-symmetry 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 self-symmetry is encountered at least once during the search tree traversal by using this strategy.

Finding the self-symmetries it not enough: recall that in order to prune the child $\mathfrak{p} \xrightarrow{v'} \mathfrak{p}_2$ of a node $\mathfrak{p}$ and only traverse the child $\mathfrak{p} \xrightarrow{v} \mathfrak{p}_1$, we must have a self-symmetry 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 self-symmetries 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 self-symmetries generated by the self-symmetries found so far. Assume that the current search node $\mathfrak{p}$, whose v, v'-children we would like to prune, is reached from the root node of the search tree via a path $\mathcal{G}(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_n.v_n} \mathfrak{p}$. If there is a self-symmetry θ that fixes all the elements $v_1, \ldots, v_n$, then θ maps $\mathfrak{p}$ to itself. Thus we must find whether there is a self-symmetry stored into set of self-symmetries found so far that fixes $v_1, \ldots, v_n$ and maps v to v'. This can be accomplished by using an operation called base change on the Schreier-Sims representation of the self-symmetries 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), the exponent in the polynomial is usually relatively high, such as 5, depending on the algorithm.

The other approach does not store the self-symmetries at all. Assume that we have already visited a leaf node $\mathfrak{p}_{n,1}$ by traversing a path $\mathcal{G}(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_i.v_i} \mathfrak{p}_i \xrightarrow{T_{i+1}.v_{i+1,1}} \mathfrak{p}_{i+1,1} \cdots \xrightarrow{T_{n,1}.v_{n,1}} \mathfrak{p}_{n,1}$ and that we already have traversed the whole sub-tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$. Suppose now that we are currently visiting a leaf node $\mathfrak{p}_{n,2}$ via a path $\mathcal{G}(\mathfrak{s}) \xrightarrow{T_1.v_1} \mathfrak{p}_1 \cdots \xrightarrow{T_i.v_i} \mathfrak{p}_i \xrightarrow{T_{i+1}.v_{i+1,2}} \mathfrak{p}_{i+1,2} \cdots \xrightarrow{T_{n,2}.v_{n,2}} \mathfrak{p}_{n,2}$, 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 self-symmetry 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 sub-trees $\mathcal{T}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$ and $\mathcal{T}(\mathfrak{s}, \mathfrak{p}_{i+1,2})$ have the same possible representative states. Therefore, we can immediately skip the rest of the sub-tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p}_{i+1,2})$ as $\mathcal{T}(\mathfrak{s}, \mathfrak{p}_{i+1,1})$ 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'-sub-trees of $\mathfrak{p}_j$ are the same.

6.3 A Relative Hardness Measure for States

We now present a hardness measure for states *relative to* the selected invariant partition generator $\mathcal{G}$ and partition refiner $\mathcal{R}$. The set of states, $\mathcal{S}$, can be divided into three classes:

- A state $\mathfrak{s}$ is *trivial* if the search tree $\mathcal{T}(\mathfrak{s})$ contains only one node, i.e. $\mathcal{G}(\mathfrak{s})$ 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 $\mathcal{T}(\mathfrak{s})$ it holds that there is a self-symmetry $\theta \in \mathrm{Stab}(\Theta, \mathfrak{s})$ 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 6.9 *If a state $\mathfrak{s}$ is trivial/easy/hard, then $\theta(\mathfrak{s})$ is also trivial/easy/hard for any allowed domain permutation θ.*

We say that an algorithm *produces 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

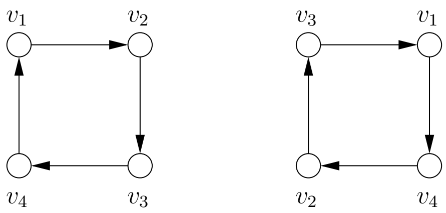


Figure 11: Two isomorphic graphs

symmetric then the algorithm will produce the same representative state for them.

Theorem 6.10 *If a state $\mathfrak{s}$ is trivial, then Algorithm 1 produces a canonical representative for it.*

Theorem 6.11 *If a state $\mathfrak{s}$ is trivial or easy, then Algorithm 2 produces a canonical representative for it.*

6.4 A Sidetrack on Testing Symmetricity of two States

Let us consider the problem of determining whether two states, say $\mathfrak{s}$ and $\mathfrak{s}'$, are symmetric. 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 $\theta(\mathfrak{s}) = \mathfrak{s}'$. We now show how this approach can be improved by using the techniques introduced earlier in this section.

Assuming an invariant partition generator $\mathcal{G}$, we have directly from the definition of invariant partition generators 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 $\mathcal{G}(\mathfrak{s})$ to the partition $\mathcal{G}(\mathfrak{s}')$. 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 $\mathcal{G}(\mathfrak{s})$ to the partition $\mathcal{G}(\mathfrak{s}')$. 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 $\mathcal{G}(\mathfrak{s})$ and $\mathcal{G}(\mathfrak{s}')$ differ in the number of cells or in the size of the corresponding cells, we can directly conclude that there are no allowed domain permutations mapping $\mathcal{G}(\mathfrak{s})$ to $\mathcal{G}(\mathfrak{s}')$ and thus $\mathfrak{s}$ and $\mathfrak{s}'$ are not symmetric. A similar approach is 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 6.12 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 Fig. 11. The states are symmetric 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* invariant partition generator $\mathcal{G}$ 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 $\mathcal{G}(\mathfrak{s})$ and $\mathcal{G}(\mathfrak{s}')$. This follows from Fact 5.21 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 $\mathcal{G}(\mathfrak{s})$ to the (same) partition $\mathfrak{p}^{\mathsf{Vertices}}$ in $\mathcal{G}(\mathfrak{s}')$. However, by Thm. 3.2 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, where we will have $n!$ 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 symmetry testing approach described above. ♣

The above symmetricity test can be improved by using search trees. Assume that we are given two states, $\mathfrak{s}_1$ and $\mathfrak{s}_2$. Take any path $\mathcal{G}(\mathfrak{s}_1) \xrightarrow{T_1, v_{1,1}} \mathfrak{p}_{1,1} \ldots \xrightarrow{T_n, v_{1,n}} \mathfrak{p}_{1,n}$ in the search tree $\mathcal{T}(\mathfrak{s}_1)$ 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 symmetric, then there is an allowed domain permutation θ mapping $\mathfrak{s}_1$ to $\mathfrak{s}_2$ and consequently (by Cor. 6.6) a leaf node $\theta(\mathfrak{p}_{1,n})$ in the search tree $\mathcal{T}(\mathfrak{s}_2, \mathcal{G}(\mathfrak{s}_2))$. Then by Lemma 5.3, $\hat{\theta}_1 * \theta^{-1}$ 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 $\mathcal{T}(\mathfrak{s}_2, \mathcal{G}(\mathfrak{s}_2))$ 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 symmetric. Therefore, in order to check whether $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are symmetric, do a backtracking search in the search tree $\mathcal{T}(\mathfrak{s}_2)$ 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 symmetric iff 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 Cor. 6.4 there is a path $\theta(\mathcal{G}(\mathfrak{s}_1)) \xrightarrow{T_1, \theta^{T_1}(v_{1,1})} \theta(\mathfrak{p}_{1,1}) \ldots \xrightarrow{T_1, \theta^{T_1}(v_{1,n})} \theta(\mathfrak{p}_{1,n})$ in the search tree $\mathcal{T}(\mathfrak{s}_2)$ and by Lemma 5.3 $\hat{\theta}_1 * \theta^{-1}$ 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 $\mathcal{G}(\mathfrak{s}_2) \xrightarrow{T_1, v_{2,1}} \mathfrak{p}_{2,1} \ldots \xrightarrow{T_1, v_{2,k}} \mathfrak{p}_{2,k}$, $k < n$, is currently being traversed in the search tree $\mathcal{T}(\mathfrak{s}_2)$, 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 randomly trying the paths in the search tree $\mathcal{T}(\mathfrak{s}_2)$.

Example 6.13 (Ex. 6.12 continued) Applying any reasonably efficient invariants, such as those described in Sec. 5.4, in the partition refiner $\mathcal{R}$ will make the search tree for the state $\mathfrak{s}$ to look something like this:

$$[\{v_1, v_2, v_3, v_4\}]$$

$$[\{v_1\}, \{v_3\}, \{v_4\}, \{v_2\}] \quad [\{v_2\}, \{v_4\}, \{v_1\}, \{v_3\}] \quad [\{v_3\}, \{v_1\}, \{v_2\}, \{v_4\}] \quad [\{v_4\}, \{v_2\}, \{v_3\}, \{v_1\}]$$

and the search tree for the state $\mathfrak{s}'$ is thus:

$$[\{v_1, v_2, v_3, v_4\}]$$

$$[\{v_1\},\{v_2\},\{v_3\},\{v_4\}] \quad [\{v_2\},\{v_1\},\{v_4\},\{v_3\}] \quad [\{v_3\},\{v_4\},\{v_2\},\{v_1\}] \quad [\{v_4\},\{v_3\},\{v_1\},\{v_2\}]$$

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 symmetricity test approach.

7 HANDLING VERY LARGE AND INFINITE SCALAR SETS

So far we have implicitly assumed that the domains of unordered primitive types (scalar sets) are finite. However, it would be convenient to have unordered primitive types with infinite domains. For instance, modeling unbounded resources such as process identifiers would require the domain to be infinite. Without restrictions, infinite domains cause problems 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 redefining the invariant partitioning process. Likewise, having a cell with infinitely many elements not as the last cell would invalidate the Def. 5.2 of compatible 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 $\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s}))$ implies $\mathcal{G}(\mathfrak{s}) = \theta(\mathcal{G}(\mathfrak{s}))$, meaning that the elements v and v' must belong to the same cell in the partition assigned to $\mathfrak{s}$ by any invariant partition generator $\mathcal{G}$. Therefore, invariant 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 we may conclude that we can in effect *ignore the elements of unordered primitive types that do not appear in the state in question.* An algorithmic view of this is to first apply the following invariant for each unordered primitive type T during the computation of the invariant partition generator.

Definition 7.1 *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 into 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 under the assumption made above). We then ignore the latter cell. Because we have 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" into the first n elements in the domain by any allowed domain permutation compatible with the partition produced this way.

Another view of the same thing 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 symmetric iff $\mathfrak{s}'_1 = \theta_1(\mathfrak{s}_1)$ and $\mathfrak{s}'_2 = \theta_2(\mathfrak{s}_2)$ are symmetric 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 $\theta_2(\mathfrak{s}_2)$. We 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 symmetric under the allowed domain permutation group for the reduced domains iff 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 $\theta_{1'}(\theta_1(\mathfrak{s}_1)) = \theta_{2'}(\theta_1(\mathfrak{s}_2))$ when $\theta_{1'}$ and $\theta_{2'}$ are interpreted as if they were allowed domain permutations for the unreduced domains. Thus the (canonical) representatives computer under the reduced domains can be directly used as (canonical) representatives of the original states.

8 ALGORITHMS BASED ON CHARACTERISTIC GRAPHS

Recall Sec. 4 describing characteristic graphs for states. Assuming that we have an algorithm deciding whether two node labeled, edge weighted directed graphs are isomorphic, the obvious algorithm for deciding whether two states are symmetric under the group Θ of all allowed domain permutations is 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 then check whether $\mathcal{G}_{\mathfrak{s}}$ and $\mathcal{G}_{\mathfrak{s}'}$ are isomorphic. (In the case our graph isomorphism algorithm only supports a weaker form of graphs, say node labeled undirected graphs, we have to transform the characteristic graph into that graph class by replacing edges with additional, appropriately labeled vertices.)

We now show how to obtain a canonical representative function for states, provided that we have a canonizer for graphs. Formally, a canonizer for graphs is a function $\mathcal{K}$ from graphs to graphs such that (i) for all graphs G, G and $\mathcal{K}(G)$ are isomorphic, and (ii) if two graphs G and G' are isomorphic, then $\mathcal{K}(G) = \mathcal{K}(G')$. Furthermore, we assume a canonizer that 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 form $\mathcal{K}(G)$ 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. For instance, the *nauty* tool [McKay 1990] includes a canonizer that also gives a mapping κ.

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 $\mathcal{G}_{\mathfrak{s}} = \langle V, \ldots \rangle$. 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 $\theta = \left(\theta^T\right)_{T \in \mathcal{T}_P}$ that is *compatible* with κ, meaning that the following rules are fulfilled.

- For each cyclic primitive type T with $\mathscr{D}_T = \{v_1, \ldots, v_n\}$, 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_1.
- 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 iff v has the ith smallest value $\kappa(T{::}v)$.

We denote the state $\theta(\mathfrak{s})$ by $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$. The whole algorithm is shown in Alg. 3. The next theorem establishes the correctness of the algorithm.

Algorithm 3 Representative algorithm 3

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

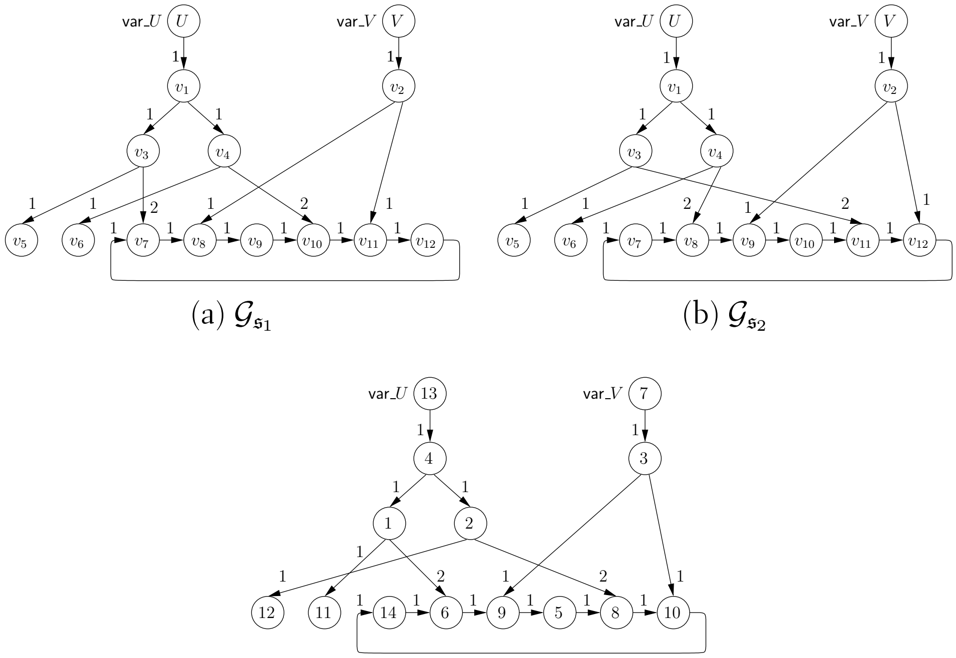
Theorem 8.1 *The function $\mathcal{K}_{\mathcal{S}}$ is a canonical representative function.*

Example 8.2 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 symmetric 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

$$\theta_2 = \left(\theta_2^{\mathsf{Secs}} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_0 \end{smallmatrix}\right), \theta_2^{\mathsf{Trains}} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right)$$



(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 12: Two characteristic graphs and their common canonical version

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 Figs. 12(a) and 12(b), respectively. In the figures, we use the following common abbreviations for vertex names: $v_5 = \mathsf{Trains}::t_a$, $v_6 = \mathsf{Trains}::t_b$, $v_7 = \mathsf{Secs}::s_0$, $v_8 = \mathsf{Secs}::s_1$, $v_9 = \mathsf{Secs}::s_2$, $v_{10} = \mathsf{Secs}::s_3$, $v_{11} = \mathsf{Secs}::s_4$ 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 Fig. 12(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 $\mathcal{K}(\mathcal{G}_{\mathfrak{s}_2})$:

$$\begin{aligned}
\kappa_{2,1} &= \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 1 & 2 & 11 & 12 & 5 & 8 & 10 & 14 & 6 & 9 \end{smallmatrix}\right) \text{ and} \\
\kappa_{2,2} &= \left(\begin{smallmatrix} U & V & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\ 13 & 7 & 4 & 3 & 2 & 1 & 12 & 11 & 14 & 6 & 9 & 5 & 8 & 10 \end{smallmatrix}\right).
\end{aligned}$$

The two allowed domain permutations compatible with these isomorphisms are

$$\begin{aligned}
\theta_{\mathfrak{s}_2,1} &= \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_2,1} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_2,1} = \left(\begin{smallmatrix} t_a & t_b \\ t_a & t_b \end{smallmatrix}\right)\right) \text{ and} \\
\theta_{\mathfrak{s}_2,2} &= \left(\theta^{\mathsf{Secs}}_{\mathfrak{s}_2,2} = \left(\begin{smallmatrix} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 \\ s_3 & s_4 & s_5 & s_0 & s_1 & s_2 \end{smallmatrix}\right), \theta^{\mathsf{Trains}}_{\mathfrak{s}_2,2} = \left(\begin{smallmatrix} t_a & t_b \\ t_b & t_a \end{smallmatrix}\right)\right),
\end{aligned}$$

respectively. The canonical representative state for $\mathfrak{s}_2$ is thus

$$\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_2) = \theta_{\mathfrak{s}_2,1}(\mathfrak{s}_2) = \theta_{\mathfrak{s}_2,2}(\mathfrak{s}_2) = \{U \mapsto \langle t_a, s_4\rangle + \langle t_b, s_1\rangle, V \mapsto s_2 + s_5\}$$

which equals to $\mathcal{K}_{\mathcal{S}}(\mathfrak{s}_1)$. ♣

Note that the algorithms Alg. 1 and Alg. 3 could be combined as follows. Given a state, first compute the partition $\mathcal{G}(\mathfrak{s})$ for it by using a fixed invariant partition generator $\mathcal{G}$. If $\mathcal{G}(\mathfrak{s})$ 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 $\mathcal{G}(\mathfrak{s})$. If $\left(\mathfrak{p}^T\right)_{T\in\mathcal{T}_P} = \mathcal{G}(\mathfrak{s})$ 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 line 2 of algorithm Alg. 3.

9 SOME EXPERIMENTAL RESULTS

We have implemented the algorithms proposed in this report in the version 3.1 of the Murφ tool. In this section we present some experimental results on the example systems 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 this text 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 Alg. 1 in this work is basically the last Murφ algorithm except that we use more powerful invariants for building the partition. In more detail, the invariant partition generator in algorithms Alg. 1 and Alg. 2 is obtained as follows. First, we apply the invariants for ordered structured types described in Sec. 5.4 on state variables if possible. Then, for the other state variables, we apply the hash-like invariants described in Sec. 5.4 until no refinement occurs. In algorithm Alg. 2, the applied partition refiner is produced by refining with the hash-like invariants.

In addition to the example systems in the Murφ distribution, we use the following graph enumeration systems inspired by the proof of Thm. 3.4 in [Ip 1996]. Figure 13 shows a Murφ program called graphs5.m. It 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, the (optimal) symmetry reduced reachability 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 system called digraphs5.m, enumerating all the directed graphs of 5 vertices.

```
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 13: A graph enumeration system

Table 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 Alg. 3 are canonical representative functions and thus the state columns for these algorithms give the size of the optimally reduced reachability graph.

As the Murφ examples (from adash to n_peterson) show, algorithm Alg. 1 is quite fast and produces almost optimally 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 Sec. 5.4. 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 Alg. 2 produces optimally reduced reachability graphs for these instances although it is not a canonical representative algorithm. Furthermore, it is not significantly slower than algorithm Alg. 1.

In the graph enumeration problems (graph$n$ and digraph$n$), the algorithms Alg. 1 and 2 perform very well, producing reachability graphs that are reasonably close to the optimal ones. Again, especially the algorithm Alg. 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 Alg. 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.

We have also implemented the algorithm Alg. 3 by using the *nauty* tool [McKay 1990] as the graph canonizer. The bad experimental results shown in Table 1 are probably due to the fact that the characteristics graphs of states are 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, we had to add some nodes into 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 2768 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 Alg. 3 might look different if we had a graph canonizer designed for directed, edge weighted, and sparse graphs.

10 SOME RELATED WORK

The algorithms in the Murφ tool were already discussed in the previous section. The main difference between Alg. 1 and the Murφ algorithms is that we use more powerful invariants, especially the hash-like invariants in Sec. 5.4 are novel. Furthermore, we also handle cyclic primitive types (non-reflexive ring symmetries in the Murφ terminology) in a unified way. The two other algorithm presented in this report are new.

The approach taken in [Sistla et al. 2000] is discussed in Sec. 6.4.

In [Huber et al. 1985; Jensen 1995] a very elementary version of the partition refinement process is given and applied to checking whether two markings of a colored Petri net are symmetric. Especially, no other structured types than of form $\mathsf{Multi\text{-}Set}(T)$, where T is an unordered primitive type, are handled.

Recently, an approach based on computational group theory was 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 permutations in Θ are tested. However, whenever $|\mathrm{Stab}(\Theta, \mathfrak{s})|$ is very small, no such large savings are obtained; especially

system name	Murφ alg. 1		Murφ alg. 2		Murφ alg. 3		Murφ alg. 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 Alg. 1		algorithm Alg. 2		algorithm Alg. 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 1: Some experimental results

in systems in which most of the reachable states have no self-symmetries 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 symmetric 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 symmetric 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 we do not know any polynomial time algorithm). 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, we could probably compute the stabilizer group $\mathrm{Stab}(\Theta, \mathfrak{s})$ and the lexicographically smallest state symmetric 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 symmetric to $\mathfrak{s}$ and $\Theta_n = \mathrm{Stab}(\Theta, \mathfrak{s})$. Now a variant 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 (WFNs).

11 CONCLUSIONS

In this report we have presented symmetry reduction algorithms under data type symmetries. The first two algorithms resemble the preprocessing and search phases, respectively, of common graph isomorphism checking algorithms. The first algorithm, given a state, produces an ordered partition of the elements of the permutable primitive types, selects one permutation based on the partition and returns the permuted state as the representative state. The difference between the existing algorithms in the Murφ tool [Ip 1996] and the first algorithm is that more powerful invariants are used to obtain the ordered partition. The second algorithm improves the first one by selecting an arbitrary path in the search tree whose root is the partition produced in the first algorithm and then returns the state permuted with the permutation associated with the leaf partition as the representative state. The third algorithm presented exploits the characteristic graphs of states, i.e. graphs that are isomorphic iff the corresponding states are symmetric. The existing graph isomorphism/canonicalization algorithms are then used to canonicalize the characteristic graph of a state, resulting in the canonical representative state for the state. Some experimental results were also presented, showing that the algorithms are competitive against the previous ones implemented in the Murφ tool.

Acknowledgements

The author wishes to thank the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Academy of Finland (projects no. 47754 and no. 43963) for their financial support and Petteri Kaski for his comments on this work.

References

BUTLER, G. 1991. *Fundamental Algorithms for Permutation Groups*. Lecture Notes in Computer Science, vol. 559. Springer-Verlag, Berlin, Germany.

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.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts.

GENRICH, H. J. 1991. Predicate/transition nets. See Jensen and Rozenberg [1991], 3–43.

HUBER, P., JENSEN, A. M., JEPSEN, L. O., AND JENSEN, K. 1985. Towards reachability trees for high-level Petri nets. Tech. Rep. DAIMI PB 174, Datalogisk Afdeling, Matematisk Institut, Aarhus Universitet. May.

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. 1995. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer.

JENSEN, K. AND ROZENBERG, G., Eds. 1991. *High-level Petri Nets; Theory and Application*. Springer.

JUNTTILA, T. 1999. 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.

KREHER, D. L. AND STINSON, D. R. 1999. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, Boca Raton, Florida, USA.

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.

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.

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.

VALMARI, A. 1998. The state explosion problem. In *Lectures on Petri Nets I: Basic Models,* W. Reisig and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 1491. Springer, 429–528.

A PROOFS

Proof of Theorem 3.2

Theorem 3.2 *Assume a domain permutation $\psi \in \Psi$ that maps a state $\mathfrak{s}_1$ to $\mathfrak{s}_2$ i.e. $\psi(\mathfrak{s}_1) = \mathfrak{s}_2$. Then*

1. $\mathrm{Stab}(\Psi, \mathfrak{s}_2) = \psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1}$, *where* $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1} = \{\psi * \psi' * \psi^{-1} \mid \psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)\}$, *and*

2. *the left coset* $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) = \{\psi * \psi' \mid \psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)\}$ *is exactly the set of all domain permutations in* Ψ *mapping* $\mathfrak{s}_1$ *to* $\mathfrak{s}_2$.

Consequently, (i) $|\mathrm{Stab}(\Psi, \mathfrak{s}_1)| = |\mathrm{Stab}(\Psi, \mathfrak{s}_2)|$, *(ii) there are* $|\mathrm{Stab}(\Psi, \mathfrak{s}_1)|$ *domain permutations mapping* $\mathfrak{s}_1$ *to* $\mathfrak{s}_2$, *and (iii) there are* $|\Psi| \,/\, |\mathrm{Stab}(\Psi, \mathfrak{s}_1)|$ *states that are* Ψ*-symmetric to* $\mathfrak{s}_1$.

Proof. Part 1. If $\psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)$, then $(\psi * \psi' * \psi^{-1})(\mathfrak{s}_2) = (\psi * \psi')(\mathfrak{s}_1) = \psi(\mathfrak{s}_1) = \mathfrak{s}_2$ implying that $\psi * \psi' * \psi^{-1} \in \mathrm{Stab}(\Psi, \mathfrak{s}_2)$ and $\mathrm{Stab}(\Psi, \mathfrak{s}_2) \supseteq \psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1}$. On the other hand, if $\psi'' \in \mathrm{Stab}(\Psi, \mathfrak{s}_2)$, then $\psi^{-1} * \psi'' * \psi \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)$ and $\psi'' \in \psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1}$ implying that $\mathrm{Stab}(\Psi, \mathfrak{s}_2) \subseteq \psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1) * \psi^{-1}$.

Part 2. Each domain permutation in the left coset $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1)$ maps $\mathfrak{s}_1$ to $\mathfrak{s}_2$ since $(\psi * \psi')(\mathfrak{s}_1) = \psi(\mathfrak{s}_1) = \mathfrak{s}_2$ whenever $\psi' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)$. On the other hand, assume that $\psi''(\mathfrak{s}_1) = \mathfrak{s}_2$ for a domain permutation $\psi'' \in \Psi$. Thus $(\psi^{-1} * \psi'')(\mathfrak{s}_1) = \psi^{-1}(\mathfrak{s}_2) = \mathfrak{s}_1$ implying that $\psi^{-1} * \psi'' \in \mathrm{Stab}(\Psi, \mathfrak{s}_1)$. But now $\psi * (\psi^{-1} * \psi'') = \psi''$ belongs to the left coset $\psi * \mathrm{Stab}(\Psi, \mathfrak{s}_1)$. □

Proof of Lemma 5.3

Lemma 5.3 *Let* $\hat{\theta}$ *be an allowed domain permutation compatible with a partition* $\mathfrak{p}$. *Then for each allowed domain permutation* θ *it holds that the allowed domain permutation* $\hat{\theta} * \theta^{-1}$ *is compatible with the partition* $\theta(\mathfrak{p})$.

Proof. Let $\hat{\theta} = \left(\hat{\theta}^T\right)_{T \in \mathcal{T}_P}$, $\mathfrak{p} = \left(\mathfrak{p}^T\right)_{T \in \mathcal{T}_P}$, and $\mathfrak{p}^T = [C_1^T, \ldots, C_{c_T}^T]$.

For a cyclic primitive type T, assume that $\hat{\theta}^T$ maps a $v_i \in C_1^T$ to v_1, i.e. $\hat{\theta}^T(v_i) = v_1$. Observe that $\hat{\theta}^T = \hat{\theta}^T \circ \theta^{T^{-1}} \circ \theta^T$ and therefore $(\hat{\theta}^T \circ \theta^{T^{-1}})(\theta^T(v_i)) = v_1$. But now $\theta^T(v_i)$ is in the first cell 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 $\mathit{incell}(\theta^T(\mathfrak{p}^T), v_i) < \mathit{incell}(\theta^T(\mathfrak{p}^T), v_j)$. Thus $\mathit{incell}(\mathfrak{p}^T, \theta^{T^{-1}}(v_i)) < \mathit{incell}(\mathfrak{p}^T, \theta^{T^{-1}}(v_j))$ 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. □

Proof of Corollary 5.4

Corollary 5.4 *Assume an invariant partition generator* $\mathcal{G}$, *and take two symmetric states,* $\mathfrak{s}_1$ *and* $\mathfrak{s}_2$. *Let* $\hat{\theta}_1$ *be an allowed domain permutation compatible with the partition* $\mathcal{G}(\mathfrak{s}_1)$. *Then there is an allowed domain permutation* $\hat{\theta}_2$ *compatible with the partition* $\mathcal{G}(\mathfrak{s}_2)$ *such that* $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\mathfrak{s}_2)$.

Proof. First, there is an allowed domain permutation $\hat{\theta}_2$ such that $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\mathfrak{s}_2)$. This is because $\mathfrak{s}_1$ and $\mathfrak{s}_2$ are symmetric and therefore there is an allowed domain permutation θ such that $\theta(\mathfrak{s}_1) = \mathfrak{s}_2$. Thus the requirement $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\mathfrak{s}_2)$ is equivalent to $\hat{\theta}_1(\mathfrak{s}_1) = \hat{\theta}_2(\theta(\mathfrak{s}_1)) = (\hat{\theta}_2 * \theta)(\mathfrak{s}_1)$. Obviously, $\hat{\theta}_2 = \hat{\theta}_1 * \theta^{-1}$ is a solution to this. Second, since $\mathcal{G}$ is an invariant partition generator, $\mathcal{G}(\mathfrak{s}_2) = \mathcal{G}(\theta(\mathfrak{s}_1)) = \theta(\mathcal{G}(\mathfrak{s}_1))$. By Lemma 5.3, $\hat{\theta}_2 = \hat{\theta}_1 * \theta^{-1}$ is compatible with $\mathcal{G}(\mathfrak{s}_2) = \theta(\mathcal{G}(\mathfrak{s}_1))$ because $\hat{\theta}_1$ is compatible with $\mathcal{G}(\mathfrak{s}_1)$. □

Proof of Lemma 5.7

Lemma 5.7 *The composition $\mathcal{R}_1 \star \mathcal{R}_2$ of two partition refiners $\mathcal{R}_1$ and $\mathcal{R}_2$, defined by $(\mathcal{R}_1 \star \mathcal{R}_2)(\mathfrak{s}, \mathfrak{p}) = \mathcal{R}_2(\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p}))$, is a partition refiner.*

Proof. Clearly $\mathfrak{p}_{\mathrm{ref}} = (\mathcal{R}_1 \star \mathcal{R}_2)(\mathfrak{s}, \mathfrak{p}) = \mathcal{R}_2(\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p}))$ is a cell order preserving refinement of $\mathfrak{p}$ since $\mathcal{R}_2(\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p})) \preceq \mathcal{R}_1(\mathfrak{s}, \mathfrak{p}) \preceq \mathfrak{p}$. On the other hand, $\theta((\mathcal{R}_1 \star \mathcal{R}_2)(\mathfrak{s}, \mathfrak{p})) = \theta(\mathcal{R}_2(\mathfrak{s}, \mathcal{R}_1(\mathfrak{s}, \mathfrak{p}))) = \mathcal{R}_2(\theta(\mathfrak{s}), \theta(\mathcal{R}_1(\mathfrak{s}, \mathfrak{p}))) = \mathcal{R}_2(\theta(\mathfrak{s}), \mathcal{R}_1(\theta(\mathfrak{s}), \theta(\mathfrak{p}))) = (\mathcal{R}_1 \star \mathcal{R}_2)(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$. □

Proof of Theorem 5.8

Theorem 5.8 *For a partition refiner $\mathcal{R}$, the function $\mathcal{G}_{\mathcal{R}}(\mathfrak{s}) = \mathcal{R}(\mathfrak{s}, \mathfrak{p}_0)$, where $\mathfrak{p}_0 = \left(\mathfrak{p}_0^T = [\mathscr{D}_T]\right)_{T \in \mathcal{T}_P}$, is an invariant partition generator.*

Proof. Observe that $\theta(\mathfrak{p}_0) = \mathfrak{p}_0$ for any allowed domain permutation θ. Thus $\mathcal{G}_{\mathcal{R}}(\theta(\mathfrak{s})) = \mathcal{R}(\theta(\mathfrak{s}), \mathfrak{p}_0) = \mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}_0)) = \theta(\mathcal{R}(\mathfrak{s}, \mathfrak{p}_0)) = \theta(\mathcal{G}_{\mathcal{R}}(\mathfrak{s}))$. □

Proof of Lemma 5.11

Lemma 5.11 *If I is a type invariant for a 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 $\theta \in \Theta$, $I_x(\theta^T(v), \theta(\mathfrak{s}), \theta(\mathfrak{p})) = I(\theta^T(v), (\theta(\mathfrak{s}))(x), \theta(\mathfrak{p})) = I(\theta^T(v), \theta^{T'}(\mathfrak{s}(x)), \theta(\mathfrak{p})) = I(v, \mathfrak{s}(x), \mathfrak{p}) = I_x(v, \mathfrak{s}(x), \mathfrak{p})$. □

Proof of Lemma 5.14

Lemma 5.14 *The function $\mathcal{R}_I$ is a partition refiner.*

Proof. The fact that $\mathcal{R}_I(\mathfrak{s}, \mathfrak{p}) \preceq \mathfrak{p}$ follows directly from the items 2(a) and 2(b) of the definition.

Take any state $\mathfrak{s}$, any partition $\mathfrak{p} = \left(\mathfrak{p}^T\right)_{T \in \mathcal{T}_P}$, and any allowed domain permutation $\theta = \left(\theta^T\right)_{T \in \mathcal{T}_P}$. We must show that $\theta(\mathcal{R}(\mathfrak{s}, \mathfrak{p})) = \mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$.

First, consider any primitive type $T' \neq T$. Since the invariant I is for the type T, (i) the partition for a T' in the partition $\mathcal{R}(\mathfrak{s}, \mathfrak{p})$ is $\mathfrak{p}^T$, and (ii) the partition for a T' in the partition $\mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$ is $\theta(\mathfrak{p}^T)$.

Now consider the partitions for the type T in $\mathcal{R}(\mathfrak{s}, \mathfrak{p})$ and in $\mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$. Take any two elements $v, v' \in \mathscr{D}_T$. Now it suffices to notice the following:

1. v and v' belong to the same cell in the partition $\mathfrak{p}^T$ iff $\theta^T(v)$ and $\theta^T(v')$ belong to the same cell in the partition $\theta(\mathfrak{p}^T)$,
2. v belongs to an earlier cell than v' in $\mathfrak{p}^T$ iff $\theta^T(v)$ belongs to an earlier cell than $\theta^T(v')$ in $\theta(\mathfrak{p}^T)$,
3. $I(v, \mathfrak{s}, \mathfrak{p}) = I(v', \mathfrak{s}, \mathfrak{p})$ iff $I(\theta^T(v), \theta(\mathfrak{s}), \theta(\mathfrak{p})) = I(\theta^T(v'), \theta(\mathfrak{s}), \theta(\mathfrak{p}))$, and
4. $I(v, \mathfrak{s}, \mathfrak{p}) < I(v', \mathfrak{s}, \mathfrak{p})$ iff $I(\theta^T(v), \theta(\mathfrak{s}), \theta(\mathfrak{p})) < I(\theta^T(v'), \theta(\mathfrak{s}), \theta(\mathfrak{p}))$.

Thus C_i^T is the ith cell in the partition for the type T in $\mathcal{R}(\mathfrak{s}, \mathfrak{p})$ iff $\theta^T(C_i^T)$ is the ith cell in the partition for the type T in $\mathcal{R}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$. □

Proof of Lemma 5.16

Lemma 5.16 *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 an allowed domain permutation $\theta = \left(\theta^T\right)_{T \in \mathcal{T}_P}$ in which $\theta^T = succ_T^k$ for a $1 \leq k \leq |\mathscr{D}_T|$, $succ_T^k(succ_T(v)) = succ_T(succ_T^k(v)) = succ_T(\theta^T(v))$ belongs to the ith cell in the partition $\theta(\mathfrak{p}^T)$. □

Proof of Lemma 5.18

Lemma 5.18 *If $\theta = \left(\theta^T\right)_{T \in \mathcal{T}}$ is an allowed domain permutation, then*

$$\mathfrak{g}_T(v, T', v', \mathfrak{p}) = \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})).$$

Proof. By induction on the structure of T'.

Induction base. Assume an allowed domain permutation $\theta = \left(\theta^T\right)_{T \in \mathcal{T}}$.

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 with $\mathscr{D}_{T'} = \{v_1, \ldots, v_n\}$.
   (a) If $T \neq T'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T'}(incell(\theta(\mathfrak{p}^{T'}), \theta^{T'}(v'))) \\ &= h_{T'}(incell(\mathfrak{p}^{T'}, v')) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}). \end{aligned}$$

   (b) If $T = T'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(k, incell(\theta(\mathfrak{p}^{T'}), \theta^{T'}(v'))) \\ &= h_{T',2}(k, incell(\mathfrak{p}^{T'}, v')) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}) \end{aligned}$$

   because v' is the k-successor of v iff $\theta^T(v')$ 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

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(incell(\theta(\mathfrak{p}^{T'}), \theta^{T'}(v')), 0) \\ &= h_{T',2}(incell(\mathfrak{p}^{T'}, v'), 0) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}). \end{aligned}$$

   Note that in the case $T = T'$, $v \neq v'$ iff $\theta^T(v) \neq \theta^T(v')$.

(b) If $T = T'$ and $v = v'$, then

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= h_{T',2}(incell(\theta(\mathfrak{p}^{T'}), \theta^{T'}(v')), 1) \\ &= h_{T',2}(incell(\mathfrak{p}^{T'}, v'), 1) \\ &= \mathfrak{g}_T(v, T', v', \mathfrak{p}). \end{aligned}$$

Again, if $T = T'$, then $v = v'$ iff $\theta^T(v) = \theta^T(v')$.

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 $T' = \mathsf{Struct}(T_1, \ldots, T_n)$,

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(\langle v_1, \ldots, v_n\rangle), \theta(\mathfrak{p})) &= \\ \mathfrak{g}_T(\theta^T(v), T', \langle \theta^{T_1}(v_1), \ldots, \theta^{T_n}(v_n)\rangle, \theta(\mathfrak{p})) &= \\ h_{T',n}(\mathfrak{g}_T(\theta^T(v), T_1, \theta^{T_1}(v_1), \theta(\mathfrak{p})), ..., \mathfrak{g}_T(\theta^T(v), T_n, \theta^{T_n}(v_n), \theta(\mathfrak{p}))) &= \\ h_{T',n}(\mathfrak{g}_T(v, T_1, v_1, \mathfrak{p}), \ldots, \mathfrak{g}_T(v, T_n, v_n, \mathfrak{p})) &= \\ \mathfrak{g}_T(v, T', \langle v_1, \ldots, v_n\rangle, \mathfrak{p}). & \end{aligned}$$

- For a set type $T' = \mathsf{Set}(T_1)$,

$$\begin{aligned} \mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(v'), \theta(\mathfrak{p})) &= \\ h_{T'}\left(\bigoplus_{v'' \in \theta^{T'}(v')} \mathfrak{g}_T(\theta^T(v), T_1, v'', \theta(\mathfrak{p}))\right) &= \\ h_{T'}\left(\bigoplus_{\theta^{T_1}(v'') \in \theta^{T'}(v')} \mathfrak{g}_T(\theta^T(v), T_1, \theta^{T_1}(v''), \theta(\mathfrak{p}))\right) &= \\ h_{T'}\left(\bigoplus_{\theta^{T_1}(v'') \in \theta^{T'}(v')} \mathfrak{g}_T(v, T_1, v'', \mathfrak{p})\right) &= \\ h_{T'}\left(\bigoplus_{v'' \in v'} \mathfrak{g}_T(v, T_1, v'', \mathfrak{p})\right) &= \\ \mathfrak{g}_T(v, T', v', \mathfrak{p}) & \end{aligned}$$

  by using the commutativity and associativity of $\oplus$, and by noticing that for all $v' \in \mathscr{D}_{\mathsf{Set}(T_1)}$ and all $v'' \in \mathscr{D}_{T_1}$, $v'' \in v'$ iff $\theta^{T_1}(v'') \in \theta^{\mathsf{Set}(T_1)}(v')$.
- Assume that $T' = \mathsf{Multi\text{-}Set}(T_1)$. Now an element $v'' \in \mathscr{D}_{T_1}$ has multiplicity n in a multi-set $v' \in \mathscr{D}_{\mathsf{Multi\text{-}Set}(T_1)}$ iff 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 $\langle v_1, v_2\rangle \in v'$ iff $\langle \theta^{T_1}(v_1), \theta^{T_2}(v_2)\rangle \in \theta^{T'}(v')$. The rest of the proof is similar to the case $T' = \mathsf{Set}(T_1)$.
- For an union type $T' = \mathsf{Union}(T_1, \ldots, T_n)$,

$$\begin{aligned}
\mathfrak{g}_T(\theta^T(v), T', \theta^{T'}(\langle T_i, v'\rangle), \theta(\mathfrak{p})) &= \mathfrak{g}_T(\theta^T(v), T', \langle T_i, \theta^{T_i}(v')\rangle, \theta(\mathfrak{p}))\\
&= h_{T'}(\mathfrak{g}_T(\theta^T(v), T_i, \theta^{T_i}(v'), \theta(\mathfrak{p})))\\
&= h_{T'}(\mathfrak{g}_T(v, T_i, v', \mathfrak{p}))\\
&= \mathfrak{g}_T(v, T', \langle T_i, v'\rangle, \mathfrak{p}).
\end{aligned}$$

□

Proof of Theorem 6.3

Theorem 6.3 *For each allowed domain permutation θ, a partition $\mathfrak{p}_{child}$ is a v-child of the root node of the search tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ iff the partition $\theta(\mathfrak{p}_{child})$ is a $\theta^T(v)$-child of the root node of the search tree $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$.*

Proof. If $\mathfrak{p}$ is discrete, then $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ has no children. But now $\theta(\mathfrak{p})$ is also discrete and $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$ has no children.

Clearly, $\mathfrak{p}^T = [C_1^T, C_2^T, \ldots]$ is the first non-discrete partition in $\mathfrak{p}$ iff $\theta(\mathfrak{p}^T) = [\theta^T(C_1^T), \theta^T(C_2^T), \ldots]$ 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 $\mathfrak{p}^T$ iff $\theta^T(C_i^T) = \{\theta^T(v_{i,1}), \theta^T(v_{i,2}), \ldots\}$ is the first non-singleton cell in $\theta(\mathfrak{p}^T)$. Now the root node $\mathfrak{p}$ of the tree $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ has as its children the nodes $\mathcal{R}(\mathfrak{s}, \mathfrak{p}_j)$, where for each $1 \le j \le |C_i^T|$ 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 $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$ has as its children the nodes $\mathcal{R}(\theta(\mathfrak{s}), \mathfrak{p}_{j'})$, where for each $1 \le j \le |\theta^T(C_i^T)| = |C_i^T|$ 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 $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$ 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 $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ iff $\theta(\mathfrak{p}_{\text{child}})$ is a $\theta^T(v_{i,j})$-child of $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$. □

Proof of Lemma 6.8

Lemma 6.8 *For any allowed domain permutation θ, $\mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))}$ and consequently $\mathcal{S}_{\mathcal{T}(\mathfrak{s})} = \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}))}$*

Proof. For all allowed domain permutations θ, (i) by Cor. 6.5 a partition $\mathfrak{p}_{\text{leaf}}$ is a leaf node in $\mathcal{T}(\mathfrak{s}, \mathfrak{p})$ iff $\theta(\mathfrak{p}_{\text{leaf}})$ is a leaf node in $\mathcal{T}(\theta(\mathfrak{s}), \theta(\mathfrak{p}))$, (ii) $\hat{\theta}$ is compatible with $\mathfrak{p}_{\text{leaf}}$ iff $\hat{\theta} * \theta^{-1}$ is compatible with the partition $\theta(\mathfrak{p}_{\text{leaf}})$ (by Lemma 5.3 and the fact that $\mathfrak{p}_{\text{leaf}}$ is discrete), and (iii) thus $(\hat{\theta} * \theta^{-1})(\theta(\mathfrak{s})) = \hat{\theta}(\mathfrak{s}) \in \mathcal{S}_{\mathcal{T}(\theta(\mathfrak{s}),\theta(\mathfrak{p}))}$ iff $\hat{\theta}(\mathfrak{s}) \in \mathcal{S}_{\mathcal{T}(\mathfrak{s},\mathfrak{p})}$. □

Proof of Lemma 6.9

Lemma 6.9 *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 $\mathcal{T}(\mathfrak{s}, \mathcal{G}(\mathfrak{s}))$ contains only one node, i.e. $\mathcal{G}(\mathfrak{s})$ is a discrete partition. But now for any allowed domain permutation θ, it must be that $\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s}))$, which is a discrete partition, and thus the search tree $\mathcal{T}(\theta(\mathfrak{s}), \mathcal{G}(\theta(\mathfrak{s})))$ 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. Now the search tree $\mathcal{T}(\theta(\mathfrak{s}), \mathcal{G}(\theta(\mathfrak{s})))$ 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 $\mathcal{T}(\theta(\mathfrak{s}), \mathcal{G}(\theta(\mathfrak{s})))$. By Corollary 6.6, $\theta^{-1}(\mathfrak{p}_{1'})$ and $\theta^{-1}(\mathfrak{p}_{2'})$ are leaf nodes in the search tree $\mathcal{T}(\mathfrak{s}, \mathcal{G}(\mathfrak{s}))$. Since $\mathfrak{s}$ is easy, there is a self-symmetry $\theta_{\text{self-symm}}$ of $\mathfrak{s}$ such that $\theta_{\text{self-symm}}(\theta^{-1}(\mathfrak{p}_{1'})) = \theta^{-1}(\mathfrak{p}_{2'})$. Now $\theta * \theta_{\text{self-symm}} * \theta^{-1}$ is a self-symmetry of $\theta(\mathfrak{s})$ and $(\theta * \theta_{\text{self-symm}} * \theta^{-1})(\mathfrak{p}_{1'}) = \theta(\theta_{\text{self-symm}}(\theta^{-1}(\mathfrak{p}_{1'}))) = \theta(\theta^{-1}(\mathfrak{p}_{2'})) = \mathfrak{p}_{2'}$. 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. □

Proof of Theorem 6.10

Theorem 6.10 *If a state $\mathfrak{s}$ is trivial, then Algorithm 1 produces a canonical representative for it.*

Proof. Since $\mathfrak{s}$ is trivial, the partition $\mathcal{G}(\mathfrak{s})$ is a discrete partition, there is a unique allowed domain permutation $\hat{\theta}$ compatible with $\mathcal{G}(\mathfrak{s})$, and the representative state is $\hat{\theta}(\mathfrak{s})$. For any allowed domain permutation θ, $\theta(\mathfrak{s})$ is also trivial, the partition $\mathcal{G}(\theta(\mathfrak{s})) = \theta(\mathcal{G}(\mathfrak{s}))$ is discrete, $\hat{\theta} * \theta^{-1}$ is compatible with $\theta(\mathcal{G}(\mathfrak{s}))$ by Lemma 5.3 and the representative state for $\theta(\mathfrak{s})$ is $(\hat{\theta} * \theta^{-1})(\theta(\mathfrak{s})) = \hat{\theta}(\mathfrak{s})$. □

Proof of Theorem 6.11

Theorem 6.11 *If a state $\mathfrak{s}$ is trivial or easy, then Algorithm 2 produces a canonical representative for it.*

Proof. The case in which $\mathfrak{s}$ is trivial follows directly from Thm. 6.10. Now assume that $\mathfrak{s}$ is easy.

We first have to show that choosing any path in the search tree $\mathcal{T}(\mathfrak{s}, \mathcal{G}(\mathfrak{s}))$ 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 $\mathcal{T}(\mathfrak{s}, \mathcal{G}(\mathfrak{s}))$. Since $\mathfrak{s}$ is easy, there is a self-symmetry θ 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 5.3. But now

$(\hat{\theta} * \theta^{-1})(\mathfrak{s}) = \hat{\theta}(\theta^{-1}(\mathfrak{s})) = \hat{\theta}(\mathfrak{s})$ since θ^{-1} is a self-symmetry of $\mathfrak{s}$ because θ is.

Finally, we have to show that for any allowed domain permutation θ, the representative state for the easy state $\theta(\mathfrak{s})$ is the same as the representative state for $\mathfrak{s}$. Take any leaf node $\mathfrak{p}_{1'}$ in the search tree $\mathcal{T}(\theta(\mathfrak{s}), \mathcal{G}(\theta(\mathfrak{s})))$ and the unique allowed domain permutation $\hat{\theta}_{1'}$ compatible with $\mathfrak{p}_{1'}$. By Corollary 6.6, $\theta^{-1}(\mathfrak{p}_{'1})$ is a leaf node in the search tree $\mathcal{T}(\mathfrak{s}, \mathcal{G}(\mathfrak{s}))$. By Lemma 5.3, $\hat{\theta}_{1'} * (\theta^{-1})^{-1} = \hat{\theta}_{1'} * \theta$ is the unique allowed domain permutation compatible with $\theta^{-1}(\mathfrak{p}_{'1})$. But now $(\hat{\theta}_{1'} * \theta)(\mathfrak{s}) = \hat{\theta}_{1'}(\theta(\mathfrak{s}))$ and the representative states for $\mathfrak{s}$ and $\theta(\mathfrak{s})$ coincide. □

Proof of Theorem 8.1

Theorem 8.1 *The function $\mathcal{K}_{\mathcal{S}}$ is a canonical representative function.*

Proof. Obviously, for any state $\mathfrak{s}$, $\mathfrak{s}$ and $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$ are symmetric since $\mathcal{K}_{\mathcal{S}}(\mathfrak{s})$ is obtained from $\mathfrak{s}$ by using an allowed domain permutation.

Assume two symmetric states $\mathfrak{s}_1$ and $\mathfrak{s}_2$. We now have to prove 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 symmetric),
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.

Our goal is to show that $\theta_{\kappa_1}(\mathfrak{s}_1) = \theta_{\kappa_2}(\mathfrak{s}_2)$. For this it suffices to show that $\theta_{\kappa_2}{}^{-1} * \theta_{\kappa_1} = \left(\theta^{T\ -1}_{\kappa_2} \circ \theta^T_{\kappa_1}\right)_{T \in \mathcal{T}_P}$ 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 $\mathcal{G}_{\mathfrak{s}_2}$. By Fact 4.5, 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 all permutable primitive types T and all $v \in \mathscr{D}_T$, $(\kappa_2{}^{-1} \circ \kappa_1)(T{::}v) = T{::}v' \Leftrightarrow \theta^T(v) = v'$. 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 $\kappa_1(\{T{::}v \mid v \in \mathscr{D}_T\})$ of the nodes in the characteristic graph $\mathcal{G}_{\mathfrak{s}_1}$ corresponding to the elements of the type must equal to the image $\kappa_2(\{T{::}v \mid v \in \mathscr{D}_T\})$ of the nodes in the characteristic graph $\mathcal{G}_{\mathfrak{s}_2}$ since the isomorphisms κ_1 and κ_2 must respect node types. We now have the following two cases.

1. Let T be a cyclic primitive type with $\mathscr{D}_T = \{v_1, \ldots, v_n\}$. 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, $\theta^T_{\kappa_1}(v') = v_1 = \theta^T_{\kappa_2}(v'')$ and $\theta^{T\ -1}_{\kappa_2} \circ \theta^T_{\kappa_1}$ 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 $\theta^{T\ -1}_{\kappa_2} \circ \theta^T_{\kappa_1}$.
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 $({\kappa_2}^{-1} \circ \kappa_1)(T{::}v') = T{::}v''$.

$\square$

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A59	Tommi Junttila	Computational Complexity of the Place/Transition-Net Symmetry Reduction Method. April 2000.
HUT-TCS-A60	Javier Esparza, Keijo Heljanko	A New Unfolding Approach to LTL Model Checking. April 2000.
HUT-TCS-A61	Tuomas Aura, Carl Ellison	Privacy and accountability in certificate systems. April 2000.
HUT-TCS-A62	Kari J. Nurmela, Patric R. J. Östergård	Covering a Square with up to 30 Equal Circles. June 2000.
HUT-TCS-A63	Nisse Husberg, Tomi Janhunen, Ilkka Niemelä (Eds.)	Leksa Notes in Computer Science. October 2000.
HUT-TCS-A64	Tuomas Aura	Authorization and availability - aspects of open network security. November 2000.
HUT-TCS-A65	Harri Haanpää	Computational Methods for Ramsey Numbers. November 2000.
HUT-TCS-A66	Heikki Tauriainen	Automated Testing of Büchi Automata Translators for Linear Temporal Logic. December 2000.
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.

ISBN 951-22-6010-7
ISSN 1457-7615