Symmetry Reduction in CSP Model Checking

Thomas Gibson-Robinson and Gavin Lowe

October 19, 2017

Abstract

We present an extension of FDR3, the model checker for the process algebra CSP, that exploits symmetry, to reduce the size of the state space searched. We define what it means for a process to be symmetric with respect to a group of permutations on the underlying alphabet. Our approach factors the state space of the search by symmetry equivalence, mapping each state to a representative of its equivalence class, thereby considering all symmetric states together. In contrast to previous approaches, we make only minor restrictions on the specification (that the initial state is symmetric). We show how to implement such a search using the powerful technique of supercombinators used in the implementation of FDR3: we identify conditions on a supercombinator for it to be symmetric, and explain how to apply a permutation to a state. We also prove a powerful syntactic result, identifying conditions under which a process will be symmetric in a particular type. Finally, we present a novel efficient technique for calculating representatives of equivalence classes, which normally finds unique representatives; our experiments suggest that this technique typically works faster than other techniques, and in particular scales better.

Contents

1 Introduction **4**
- 1.1 A running example . . . 6
- 1.2 Related work . . . 8

2 Background **10**
- 2.1 Labelled Transition Systems . . . 10
  - 2.1.1 Generalised labelled transition systems . . . 11
  - 2.1.2 Normalisation . . . 12
  - 2.1.3 The product automaton . . . 13
- 2.2 Permutations . . . 14

3 Symmetric GLTSs **15**

4 Refinement checking on symmetric GLTSs **16**
- 4.1 Symmetric normalised specifications . . . 16
- 4.2 Representative members . . . 18
- 4.3 The reduced product automata . . . 18
- 4.4 Refinement checking algorithms . . . 21
  - 4.4.1 The traces model . . . 21
  - 4.4.2 Stable failures model . . . 25
  - 4.4.3 Failures-divergences model . . . 26

5 Symmetry reduction on supercombinators **27**
- 5.1 Symmetries between supercombinators . . . 31

6 Symmetric datatypes **39**

7 Identifying symmetries in supercombinators **45**
- 7.1 Relating leaf components . . . 45
- 7.2 Checking recursive mappability . . . 46
- 7.3 Applying permutations to states . . . 48

8 Calculating representatives **50**
- 8.1 Component ordering algorithm . . . 51
- 8.2 Permutation generation . . . 54
- 8.3 Uniqueness of representations . . . 56
- 8.4 Relaxing assumptions . . . 63
- 8.5 On compression . . . 63
- 8.6 Implementation considerations and alternatives . . . 64
- 8.7 Comparisons . . . 65

9 Experiments **66**
9.1 Comparison with the sorted and segmented techniques 69

10 Conclusions **69**

A Proof of Proposition 45 **74**
A.1 Auxiliary definitions and results 75
A.2 Proof of Proposition 45 77

1 Introduction

FDR [GRABR15] is a powerful model checker for the process algebra CSP [Ros10]. FDR takes a list of CSP processes, written in machine-readable CSP (henceforth CSP_M); it can check if one process refines another according to the CSP denotational models (e.g. the traces, failures and failures-divergences models); or it can check other properties, including deadlock-freedom, livelock-freedom and determinism. FDR has been widely used both within industry and in academia for verifying systems [Law05, MS01, FW99]. It is also used as a verification back-end for several other tools including: Casper [Low98] which verifies security protocols; SVA [RH07] which can verify simple shared-variable programs; and several industrial tools (e.g. ModelWorks and ASD). The last few years have seen significant advances in FDR, leading to FDR3 [GRABR15], exploiting multi-core algorithms and using more efficient internal representations of processes, and also supporting large compute clusters; these advances have made a step change in the class of systems that can be analysed.

Many systems that one might want to model check contain symmetries. In this paper, we present an extension of FDR that exploits these symmetries: this gives considerable speed-ups in model checking (see Figure 5 on page 67); more importantly, we can now check much larger systems, including systems that, without symmetry reduction, would have well over *10²⁶* states, and so would be too large to check (on the same architecture) by a factor of more than *10¹⁶*.

Our main interest in symmetry reduction arises from our analysis of concurrent datatypes, particularly those based on linked-lists [Low17]. Here, each node in the linked list is modelled by a CSP process, say of the form `Node(me, datum, next)`, where `me` is the node's identity, `datum` is some piece of data, and `next` is the identity of the next node in the list or a special value `Null`. Threads that operate on these nodes are also modelled as CSP processes. One can then analyse a system with some number n of nodes, and some number t of threads. Clearly, a list of a particular length l can be formed in $n!/(n-l)!$ different ways by using different nodes; but all such states that correspond to the same sequence of `datum` values are symmetric. Further, different states can be symmetric in the type of the `datum`s: for example, a list holding the sequence $\langle A, B, A\rangle$ is symmetric to one holding $\langle B, C, B\rangle$, say. Finally, the system is symmetric in the type of the thread identities. We formalise our notion of symmetry in Section 3: we define what it means for a labelled transition system (LTS) to be symmetric with respect to a group G of permutations on the labels of transitions, and for a pair of states to be related under a permutation $\pi \in G$ (π-*bisimilar*).

By verifying the behaviour of the system from one state, we can deduce its correctness in all symmetric states. In Section 4 we present the idea behind the symmetry reduction. We map each state to a representative member of its (G-bisimilarity) equivalence class. FDR performs model checking by searching in the product automaton formed from the LTSs for the specification and implementation processes. We show how to perform a symmetry reduction on this product automaton, and how to exploit this in a model checking algorithm. Our approach assumes only that the initial states of the specification and implementation processes are symmetric with respect to some group G of permutations; this contrasts with most other approaches which assume that *every* state of the specification is G-symmetric.

In Section 5 we consider the way an LTS is represented internally in FDR, namely using a *supercombinator*, consisting of LTSs for component processes, with rules describing how component transitions are combined. We prove a result which identifies conditions on a supercombinator under which the corresponding LTS is symmetric in G.

In Section 6 we consider how to identify syntactically that a system is symmetric in particular types $T_1, \ldots, T_N$. We make certain assumptions about the CSP script, principally that the script uses no constants of the relevant types. We show that the set of values associated with each channel or datatype constructor is invariant under permutations on each of $T_1, \ldots, T_N$. Further, we show that, for any CSP_M expression e, any environment ρ giving values to free variables, and any permutation π, evaluating e in ρ and then applying π gives the same result as first applying π to the values in ρ and then evaluating e: we denote this $\pi(\mathsf{eval}\,\rho\,e) = \mathsf{eval}(\pi \circ \rho)\,e$. In particular, this means that in the initial environment ρ_1, $\pi(\mathsf{eval}\,\rho_1\,e) = \mathsf{eval}\,\rho_1\,e$ (since $\pi \circ \rho_1 = \rho_1$), and hence that the semantics of each process is symmetric under π. CSP_M includes, as a sub-language, a lazy functional language, roughly equivalent to Haskell without type classes, but with the addition of sets, mappings and associative concatenation ("dot"). This sub-language is very convenient for modelling complex data, but considerably complicates reasoning about the full language.

In Section 7 we build on this syntactic result and show how to identify symmetries within a supercombinator, and hence how to apply a particular permutation to a state of the supercombinator.

In Section 8 we describe a way to calculate representative members of equivalence classes. This is believed to be a difficult problem, in general [CEJS98]. Our technique does not always give unique representatives (although nearly always does), but allows representatives to be calculated efficiently. Our approach works well in practice.

In Section 9 we report the results of experiments using our extension. The

experiments show that the symmetry reduction provides considerable speed-ups in model checking; further, it allows us to analyse much larger systems than would otherwise have been possible. We also compare experimentally our technique for finding representative members of equivalence classes with two existing techniques; our results suggest that our approach is typically faster, and in particular scales better.

We conclude in Section 10.

Our main contributions, then, are: the adaptation of symmetry reduction to model checking based upon the powerful technique of supercombinators; the identification of general syntactic conditions under which a system will be symmetric; a general technique for finding representative members of equivalence classes for systems built from components; and the implementation of these techniques in an easy-to-use way, within an industrial-strength model checker.

1.1 A running example

We introduce here a running example, which we use to illustrate some of our techniques. The example is of a concurrent lock-based stack based on a linked list of nodes. The CSP model is presented in Figures 1 and 2. This particular model includes six nodes, four possible data values that can be stored, and three threads, but these parameters can easily be changed.

Each node is represented by a process that is initially free (process FreeNode(me)), but may be initialised by a thread to hold a datum and a reference to another node (process Node(me, datum, next)); subsequently the datum or next reference may be read, or the node freed.

A variable holding the top of the stack is also represented by a process (Top(top)), where the top may be read or set. Likewise, the lock is represented by a process (Lock) which may be alternately locked and unlocked.

Finally, each thread is represented by a process (Thread(me)). A thread may perform a push by obtaining the lock, reading the top, initialising a node appropriately to reference the previous top, setting the top to reference the new node, signalling completion, and releasing the lock. It may perform a pop by obtaining the lock and reading the top; if the top is Null, then it signals that the pop failed because the stack is empty, and releases the lock; otherwise it obtains the node referenced by the top node, updates the top to reference it, reads the datum from the previous top, signals completion, frees the node, and releases the lock.

The processes are combined in parallel, with all events hidden except those signalling completion of operations (process System). The specification is that of a stack (process Stack(s); s represents the contents of the stack): if

```
--------- Basic types
datatype NodeIDType = Null | N0 | N1 | N2 | N3 | N4 | N5 -- the type of nodes
NodeID = diff(NodeIDType, {Null})                        -- real nodes
datatype Data = A | B | C | D                            -- the type of data
datatype ThreadID = T0 | T1 | T2                         -- the type of thread IDs

--------- Nodes
channel initNode : ThreadID.NodeID.Data.NodeIDType -- initialise a node
channel getDatum : ThreadID.NodeID.Data -- read data from a node
channel getNext : ThreadID.NodeID.NodeIDType -- read next reference from a node
channel freeNode : ThreadID.NodeID -- free a node
-- A single node with identity me, initially free
FreeNode(me) = initNode?t!me?datum?next → Node(me, datum, next)
-- A node holding datum with reference to next
Node :: (NodeIDType, Data, NodeIDType) → Proc
Node(me, datum, next) =
  getDatum?t!me!datum → Node(me, datum, next)
  □ getNext?t!me!next → Node(me, datum, next)
  □ freeNode?t!me → FreeNode(me)

--------- Variable storing the value of the top node
channel getTop, setTop : ThreadID.NodeIDType -- get or set the top pointer
Top :: (NodeIDType) → Proc
Top(top) = getTop?t!top → Top(top) □ setTop?t?top1 → Top(top1)

--------- A lock process
channel lock, unlock : ThreadID
Lock = lock?t → unlock.t → Lock

--------- Threads
channel beginPush, push, pop : ThreadID.Data -- Signal start or end of operation
channel beginPop, popEmpty : ThreadID        -- Signal start or end of operation
Thread(me) =
  beginPush.me?d → lock.me → getTop.me?top → initNode.me?n!d!top →
  setTop.me!n → push.me.d → unlock.me → Thread(me)
  □
  beginPop.me → lock.me → getTop.me?top →
  if top = Null then popEmpty.me → unlock.me → Thread(me)
  else getNext.me.top?n → setTop.me!n → getDatum.me.top?d →
       pop.me!d → freeNode.me!top → unlock.me → Thread(me)
```

Figure 1: The running example (part 1).

```
--------- The system, specification and refinement check
Nodes = ||| n : NodeID • FreeNode(n)
Threads = ||| t : ThreadID • Thread(t)
System =
  let sync = diff(Events, {| pop, popEmpty, push |})
  within (Threads [| sync |] (Lock ||| Top(Null) ||| Nodes)) \ sync
-- The specification; Spec(s) represents a stack holding s
Spec(s) =
  (if s ≠ <> then pop?t!head(s) → Spec(tail(s)) else popEmpty?t → Spec(s))
  □ length(s) < card(NodeID) & push?t?d → Spec(<d>^s)
assert Spec(<>) ⊑_T System
```

Figure 2: The running example (continued).

the stack is non-empty, the top element can be popped, and otherwise a pop may fail; if the stack is non-full, an element can be pushed on. The refinement check tests whether the system refines the specification, i.e. whether every trace of the system is allowed by the specification.

FDR can verify the refinement check. However, it is slow, taking about thirty minutes on a 32 core machine, and exploring 7.8 billion states and 21.4 billion transitions. However, there is a lot of symmetry in the system: it is symmetric in the types NodeID of node identities, Data of data, and ThreadID of thread identities. Applying the symmetry-reduction technique of this paper to this system, for these three types, reduces the number of states to 99 thousand, and reduces the checking time to less than a second.

The symmetry-reduction techniques we describe in this paper reduce the number of states from 7.8 billion to 99 thousand, and reduce the checking time from over thirty minutes to less than a second.

1.2 Related work

There have been several previous works applying symmetry reduction to model checking. [MDC06] gives an excellent survey.

Clarke et al. [CEFJ96] consider symmetry in the context of symbolic temporal logic model checking. They use the representative technique: they adapt the transition relation so as to produce representative members of each equivalence class of states, thereby factoring the transition system with respect to equivalence. They then show that, subject to certain restrictions, the (CTL*) specification is satisfied in the reduced transition system iff it is satisfied in the original system; more precisely, they require that equivalent

states have the same set of labels that are contained in the specification. They show that finding unique representatives, in general, is at least as hard as the graph isomorphism problem. They therefore adapt their technique to use representatives that might not be unique. Their approach requires the user to define how to choose representatives.

Emerson and Sistla [ES96] also consider symmetry in the context of temporal logic model checking. Their focus is on systems containing many identical or isomorphic components. They show how symmetry of the model can be deduced from symmetry of the system's structure. As with [CEFJ96], they factor the transition system with respect to equivalence, and prove that the (CTL* or Mu-Calculus) specification is satisfied in the reduced transition system iff it is satisfied in the original system; they allow the group actions to also operate on the labels of the state, but require that the group actions preserve certain significant sub-formulas of the specification.

Sistla et al. [SGE00] describe a model checker, SMC, that builds on the ideas of [ES96]. In addition to factoring the transition system with respect to symmetric equivalence, they employ a second reduction strategy known as *state symmetry*: if there are several symmetric transitions from a particular state (so the successor states are symmetric), then only one such transition is expanded. (We leave the investigation of this reduction strategy within FDR as future work: it seems somewhat harder in our setting, because multiple processes synchronize on each transition.) They store previously seen states in a hash table using a hash function that respects symmetries (so symmetric states are placed in the same bucket). When a new state s is encountered, for every state s' that hashes to the same value, the algorithm tries to test whether s and s' are indeed symmetric (using an approximating algorithm that sometimes fails to identify symmetries).

Ip and Dill [ID96] investigate symmetry using the Murϕ model checker. They introduce the notion of a *scalarset*: effectively a type where all elements are treated equivalently, as with our symmetric types. They show that any Murϕ program using such scalarsets is symmetric in each scalarset. They then factor the transition system with respect to the symmetry relation, as with the previous two papers.

Bošnački et al. [BDH02] describe an extension to the Spin model checker to support symmetry, building on the techniques of [ID96]. They present several strategies for defining representative functions; we compare these with our own approach in Section 8.

Clarke et al. [CEJS98] show that the problem of finding unique representatives is at least as hard as the graph isomorphism problem, which is widely accepted as being difficult (although not known to be NP-complete).

The work closest to ours is that by Moffat et al. [MGR08], who investi-

gate the use of symmetry in CSP model checking. They introduce the notion of permutation bisimulations —informally, renaming transitions of an LTS according to some permutation on the events— which we adapt. They then factor the LTS according to the induced equivalence. They present *structured machines* —a restricted form of supercombinators— to represent CSP systems, and present some algebraic rules that can be used to deduce symmetries between components. They then present a model checking algorithm based on these ideas, restricting the specification process to one such that *every* state is symmetric (in contrast to our approach). However, they lack an efficient, general representative choosing algorithm.

2 Background

2.1 Labelled Transition Systems

In this section we review the relevant background material on labelled transition systems.

We assume a set of events Σ such that $\tau, \checkmark \notin \Sigma$. We write $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$ and $\Sigma^{\tau\checkmark} = \Sigma \cup \{\checkmark, \tau\}$.

Definition 1. A *labelled transition system (LTS)* is a tuple $L = (S, \Delta, init)$ where:

- S is a set of *states*;
- $\Delta \subseteq S \times \Sigma^{\tau\checkmark} \times S$ is a transition relation;
- $init \in S$ is the *initial state*.

If $(s, a, s') \in \Delta$, we write $s \xrightarrow{a} s'$ (we decorate the arrow with "L" if this is not implicit from the context). We write $s \xrightarrow{a}$ iff $\exists\, s' \cdot s \xrightarrow{a} s'$. We write $s \xmapsto{a_1 \ldots a_n} s'$ iff there exists $s_1, \ldots, s_n$ such that $s = s_1 \xrightarrow{a_1} s_2 \cdots s_n \xrightarrow{a_n} s'$. We write $s \xmapsto{\tau^*} s'$ iff s can perform zero or more τ-events to become s'.

We sometimes write $s \in L$ to mean $s \in S$.

We say that state s is *stable* if it can perform no τ-transitions:

$$stable(s) \quad \Leftrightarrow \quad s \overset{\tau}{\not\rightarrow} .$$

We say that s *stably accepts* X if it is stable and can perform precisely the events from X:

$$s \;\mathsf{acc}\; X \quad \Leftrightarrow \quad stable(s) \wedge X = \{a | s \xrightarrow{a}\}.$$

We say that s *stably refuses* X if it is stable and can perform no event from X:

$$s \ \mathsf{ref}\ X \quad \Leftrightarrow \quad \mathit{stable}(s) \land \forall\, x \in X \cdot s \stackrel{a}{\not\rightarrow} .$$

We say that s diverges, denoted $\mathsf{div}\, s$, if there is an infinite path of τ-transitions starting from s.

2.1.1 Generalised labelled transition systems

FDR can perform various compressions upon LTSs. These compressions often remove τ-transitions, and merge states. In order for the semantics to be preserved, the states of LTSs are labelled with additional information, to produce a *generalised labelled transition system*.

Definition 2. A *generalised labelled transition system (GLTS)* is a tuple $L = (S, \Delta, \mathit{init}, \mathsf{minaccs}, \mathsf{div})$, where $(S, \Delta, \mathit{init})$ is an LTS, and

- $\mathsf{minaccs} : S \rightarrow \mathbf{P}(\mathbf{P}\ \Sigma^{\checkmark})$ gives the *minimal acceptances* of a state: sets E of events that can be stably accepted, i.e. all the events of E can be accepted (and no more), and such that no proper subset can be stably accepted.
- $\mathsf{div} : S \rightarrow \mathit{Bool}$ indicates whether the process can immediately diverge from this state.

Note that each state of a GLTS formed by applying a compression might correspond to a *set* of states of the original LTS. Hence $\mathsf{minaccs}$ returns a *set* of acceptances, one for each constituent state.

In practice, when working in the traces model we can omit the $\mathsf{minaccs}$ and div components from a GLTS, and when working in the stable-failures model we can omit the div component.

Lemma 3. An LTS $(S, \Delta, \mathit{init})$ can be interpreted as a GLTS $(S, \Delta, \mathit{init}, \mathsf{minaccs}, \mathsf{div})$ where

- $\mathsf{minaccs}(s) = \{\}$ if $s \stackrel{\tau}{\rightarrow}$; and otherwise $\mathsf{minaccs}(s) = \{\{a | s \stackrel{a}{\rightarrow}\ \}\}$.
- $\mathsf{div}(s)$ iff there is an infinite path of τ-transitions starting at s.

Most of the rest of the results in this paper will deal with GLTSs; the above lemma will allow these results to also be applied to LTSs.

In the remainder of this paper we restrict to *connected* (G)LTSs, i.e. where every state is reachable from the initial state by zero or more transitions.

Let s be a state of a GLTS. We say that X is stably refused in s, denoted $s \mathsf{\ ref\ } X$, if s has some minimal acceptance that includes no event of X:

$$s \mathsf{\ ref\ } X \quad \Leftrightarrow \quad \exists A \in \mathsf{minaccs}(s) \cdot A \cap X = \{\}.$$

We can then define the traces, stable failures, divergences, and full failures of a state s of a GLTS.

$$\begin{array}{rcl} traces(s) & = & \{tr \setminus \{\tau\} | s \overset{tr}{\longmapsto}\}, \\ failures(s) & = & \{(tr \setminus \{\tau\}, X) | s \overset{tr}{\longmapsto} s' \wedge s' \mathsf{\ ref\ } X\}, \\ divs(s) & = & \{tr \setminus \{\tau\}^\frown tr' | s \overset{tr}{\longmapsto} s' \wedge \mathsf{div}(s') \wedge tr' \in \Sigma^{\checkmark *}\}, \\ failures_\bot(s) & = & failures(s) \cup \\ & & \{(tr^\frown tr', X) | tr \in divs(s) \wedge tr' \in \Sigma^{\checkmark *} \wedge X \subseteq \mathbf{P}\,\Sigma^\checkmark\}. \end{array}$$

If L is a GLTS, we will write $traces(L)$ for the traces of the initial state of L, and similarly for failures and divergences.

Let S and I be GLTSs, representing a specification and implementation respectively. We define refinement between S and I in the three main models of CSP as follows.

$$\begin{array}{rll} S \sqsubseteq_T I & \mathit{iff} & traces(S) \supseteq traces(I), \\ S \sqsubseteq_F I & \mathit{iff} & traces(S) \supseteq traces(I) \wedge failures(S) \supseteq failures(I), \\ S \sqsubseteq_{FD} I & \mathit{iff} & failures_\bot(S) \supseteq failures_\bot(I) \wedge divs(S) \supseteq divs(I). \end{array}$$

FDR translates CSP processes into GLTSs, and then tests for the above refinements.

2.1.2 Normalisation

When FDR performs a refinement check of the form $Spec \sqsubseteq Impl$, it starts by normalising the specification $Spec$ [Ros10, Section 16.1].

Definition 4. Given a GLTS $L = (S, \Delta, init, \mathsf{minaccs}, \mathsf{div})$, the *prenormal form* is a GLTS $N = (\mathbf{P}\,S - \{\{\}\}, \Delta_N, init_N, \mathsf{minaccs}_N, \mathsf{div}_N)$ defined as follows. Each state is a non-empty element of $\mathbf{P}\,S$. The initial state is $\{s | init \overset{\tau^*}{\longmapsto}_L s\}$, i.e. all states reachable from $init$ by zero or more τ-transitions. For each state $\hat{s} \in \mathbf{P}\,S - \{\{\}\}$:

- For each non-τ event a that can be performed by a member of $\hat{s}$, we include an a-transition to $\{s' | \exists\, s \in \hat{s} \cdot s \overset{a\tau^*}{\longmapsto}_L s'\}$, i.e. all states reached from s by an a-transition followed by zero or more τ-transitions.

- $\mathsf{minaccs}_N(\hat{s}) = mins(\bigcup\{\mathsf{minaccs}(s) | s \in \hat{s}\})$, where $mins$ returns the $\subseteq$-minimal elements of its argument.

- $\mathsf{div}_N(\hat{s}) \Leftrightarrow \exists\, s \in \hat{s} \cdot \mathsf{div}(s)$.

The *normal form* for L, denoted $norm(L)$, is calculated by taking the prenormal form for L, restricting to reachable states, and then factoring by strong bisimulation, taking into account the divergences and minimal acceptances information.

Given an LTS L, the normal form for L is calculated by first considering L as a GLTS, as in Lemma 3, and then applying the above construction.

We say that a GLTS is *normalised* if it is the normal form of some (G)LTS.

Note that the normal form for P has no τ transitions, no pair of transitions from the same state with the same label, and no strongly bisimilar states. The following lemma shows that in a normalised GLTS, the state reached after a particular trace is unique.

Lemma 5. If GLTS $P = (S, \Delta, init, \mathsf{minaccs}, \mathsf{div})$ is normalised, and $init \overset{tr}{\mapsto} p$ and $init \overset{tr}{\mapsto} p'$ then $p = p'$.

Lemma 6. Let $N = norm(P)$. Then the semantic representations (i.e. traces, failures and divergences) of P and N are equal.

In practice, when performing refinements in the traces model, the minimal acceptances and divergences components of the normal form can be omitted, since these play no part in the traces model. Likewise, when performing refinements in the stable failures model, the divergences can be omitted.

2.1.3 The product automaton

In order to check if $P \sqsubseteq Q$, FDR considers the *product automaton* of P and Q.

Definition 7. Let $P = (S_P, \Delta_P, init_P, \mathsf{minaccs}_P, \mathsf{div}_P)$ be a normalised GLTS, and $Q = (S_Q, \Delta_Q, init_Q, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ be a GLTS. The *product automaton* of P and Q is a tuple $(S, \Delta, init, \mathsf{minaccs}_P, \mathsf{div}_P, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ such that

- $S = S_P \times S_Q$;

- $((p, q), a, (p', q')) \in \Delta$ iff $q \overset{a}{\rightarrow}_Q q'$, and if $a \neq \tau$ then $p \overset{a}{\rightarrow} p'$ else $p = p'$.

- $init = (init_P, init_Q)$.

Note that $(S, \Delta, init)$ forms an LTS.

Lemma 8. Suppose M is the product automaton of P and Q. Then $(init_P, init_Q) \stackrel{tr}{\longmapsto}_M (p_1, q_1)$ iff $init_P \stackrel{tr \setminus \tau}{\longmapsto}_P p_1$ and $init_Q \stackrel{tr}{\longmapsto}_Q q_1$. Further, p_1 is unique (for a given choice of tr).

Proof. This follows immediately from the fact that Q is normalised. □

2.2 Permutations

Let X be a set. A *permutation* on X is a bijection $\pi : X \to X$. The set of all permutations of a set X forms a group under composition, which we denote $Sym(X)$. In the Introduction, we mentioned permutations of the (sub-)types NodeID, Data and ThreadID.

We denote the inverse of a permutation π by π^{-1}. We write $\pi \,;\, \pi'$ for the forward composition of π and π', and $\pi \circ \pi'$ for the backwards composition:

$$(\pi \,;\, \pi')(x) = (\pi' \circ \pi)(x) = \pi'(\pi(x)).$$

We let id_X denote the identity permutation on the set X. We write $(v\ v')$ for the permutation that swaps v and v'. If G is a group, then $G' \leq G$ denotes that G' is a subgroup of G.

Lemma 9. Let $G \leq Sym(X)$, and let $\equiv_G$ be a relation on X such that $x \equiv_G y$ iff $\exists\, \pi \in G$ such that $\pi(x) = y$. Then $\equiv_G$ is an equivalence relation.

Our main focus is on *event permutations*. However, we do not want to change the semantic events, $\checkmark$ and τ. Thus, we let $EvSym \leq Sym(\Sigma^{\tau\checkmark})$ denote the largest symmetry subgroup such that for all permutations $\pi \in EvSym$, $\pi(\tau) = \tau$ and $\pi(\checkmark) = \checkmark$. π is an *event permutation* iff $\pi \in EvSym$.

We will often consider systems that are symmetric in one or more disjoint datatypes, say $T_1, \ldots, T_n$; the system in the running example is symmetric in NodeID, Data and ThreadID. In this case, let $\pi_i \in Sym(T_i)$, for $i = 1, \ldots, n$; then consider $\pi = \bigcup_{i=1}^{n} \pi_i \in Sym(\bigcup_{i=1}^{n} T_i)$. We say that π is *type-preserving*, since it maps elements of each T_i onto T_i. Let $Sym(T_1, \ldots, T_n)$ be the group containing all such π.

Given a permutation π on atomic values, π can be lifted to $\Sigma^{\tau\checkmark}$ by point-wise application; for example, $\pi(getDatum.t.n.d) = getDatum.\pi(t).\pi(n).\pi(d)$. Let $EvSym(T_1, \ldots, T_n) \leq EvSym$ be the group containing the result of lifting π, for $\pi \in Sym(T_1, \ldots, T_n)$. In the running example, we are interested in $EvSym$(NodeID, Data, ThreadID).

3 Symmetric GLTSs

For the remainder of this section, fix an event permutation group $G \leq EvSym$.

The following definition is adapted from [MGR08].

Definition 10 (Permutation bisimilarity)**.** Let

$$\begin{array}{rcl} L_1 & = & (S_1, \Delta_1, init_1, \mathsf{minaccs}_1, \mathsf{div}_1), \\ L_2 & = & (S_2, \Delta_2, init_2, \mathsf{minaccs}_2, \mathsf{div}_2) \end{array}$$

be GLTSs, and let $\pi \in G$ be an event permutation. We say that $\sim \subseteq S_1 \times S_2$ is a π*-bisimulation between* G_1 *and* G_2 iff whenever $(s_1, s_2) \in \sim$ and $a \in \Sigma^{\tau\checkmark}$:

- If $s_1 \xrightarrow{a} s_1'$ then $\exists\, s_2' \in S_2 \cdot s_2 \xrightarrow{\pi(a)} s_2' \wedge s_1' \sim s_2'$;
- If $s_2 \xrightarrow{a} s_2'$ then $\exists\, s_1' \in S_1 \cdot s_1 \xrightarrow{\pi^{-1}(a)} s_1' \wedge s_1' \sim s_2'$;
- $\mathsf{minaccs}_2(s_2) = \{\{\pi(a) | a \in A\} | A \in \mathsf{minaccs}_1(s_1)\}$; henceforth, we write the latter expression as $\pi(\mathsf{minaccs}_1(s_1))$;
- $\mathsf{div}_1(s_1) \Leftrightarrow \mathsf{div}_2(s_2)$.

We say that $s_1, s_2 \in S$ are π*-bisimilar*, denoted $s_1 \sim_\pi s_2$ iff there exists a π-bisimulation relation $\sim$ such that $s_1 \sim s_2$. We say that L_1 and L_2 are π*-bisimilar*, denoted $L_1 \sim_\pi L_2$, iff $init_1 \sim_\pi init_2$.

Note that the case $\pi = id$ corresponds to strong bisimulation taking acceptance and divergence labels into account.

The following lemmas follow immediately from the relevant definitions.

Lemma 11. Let π and π' be event permutations.

1. If $s_1 \sim_\pi s_2$ then $s_2 \sim_{\pi^{-1}} s_1$;
2. If $s_1 \sim_\pi s_2$ and $s_2 \sim_{\pi'} s_3$ then $s_1 \sim_{\pi;\pi'} s_3$.

Lemma 12. Suppose $s \sim_\pi s'$. Then

$$s \;\mathsf{ref}\; X \quad\Leftrightarrow\quad s' \;\mathsf{ref}\; \pi(X).$$

Definition 13. Let L be a GLTS and $\pi \in EvSym$ be an event permutation. We say that L is π*-symmetric* iff $L \sim_\pi L$. We say that L is G*-symmetric* iff for all $\pi \in G$, L is π-symmetric.

Note that every GLTS is $\{id\}$-symmetric.

Lemma 14. If L is π-symmetric then for every state s of L, there exists a state s' in L such that $s \sim_\pi s'$.

Proof. This follows via a simple induction and our assumption that the GLTS is connected. □

4 Refinement checking on symmetric GLTSs

In this section we present —at a fairly high level of abstraction— our refinement checking algorithm. In Section 4.1 we consider relevant properties of the specification, in particular that normalising a symmetric specification GLTS preserves symmetry. As noted in the Introduction, our basic approach is to map each state encountered in the search to a representative member of its G-equivalence class; we define such representatives in Section 4.2. In Section 4.3 we present the reduced product automaton, which is explored by the model checking algorithms, created by replacing each state by its representative; we then present relevant results about the reduced automaton. In Section 4.4, we translate refinement relationships (for each of the semantic models) into properties of the product automaton, and so present the model checking algorithms themselves.

4.1 Symmetric normalised specifications

Symmetry is preserved by normalisation.

Lemma 15. If $P = (S_P, \Delta_P, init_P, \mathsf{minaccs}_P, \mathsf{div}_P)$ is a G-symmetric GLTS, then $norm(P)$ is a G-symmetric normalised GLTS.

Proof. Let $N = (S_N, \Delta_N, init_N, \mathsf{minaccs}_N, \mathsf{div}_N)$ be the prenormal form for P. Let $\pi \in G$. We show that N is π-symmetric. Suppose $\sim$ is a π-bisimulation over S_P. Define a corresponding relation $\approx$ over S_N as follows:

$$\begin{array}{rcl} N_1 \approx N_2 & \Leftrightarrow & (\forall\, s_1 \in N_1 \cdot \exists\, s_2 \in N_2 \cdot s_1 \sim s_2) \wedge \\ & & (\forall\, s_2 \in N_2 \cdot \exists\, s_1 \in N_1 \cdot s_1 \sim s_2). \end{array}$$

We prove that $\approx$ is a π-bisimulation. Suppose $N_1 \approx N_2$.

- Suppose $N_1 \xrightarrow{a}_N N_1'$. Then there exists $s_1 \in N_1$ and $s_1' \in N_1'$ such that $s_1 \overset{a\tau^*}{\longmapsto}_P s_1'$; and conversely, for each $s_1' \in N_1'$, there is a corresponding $s_1 \in N_1$. Further, since $N_1 \approx N_2$, for each such s_1, there exists $s_2 \in N_2$

such that $s_1 \sim s_2$. But then, for each such s_1', there exists s_2' such that $s_2 \stackrel{\pi(a)\tau^*}{\longmapsto}_P s_2'$ and $s_1' \sim s_2'$. Let N_2' contain all such s_2', as s_1, s_1' and s_2 range over values as above; i.e.:

$$N_2' = \{s_2' | \exists\, s_1 \in N_1, s_1' \in N_1', s_2 \in N_2 \cdot \\ s_1 \stackrel{a\tau^*}{\longmapsto}_P s_1' \wedge s_1 \sim s_2 \wedge s_2 \stackrel{\pi(a)\tau^*}{\longmapsto}_P s_2' \wedge s_1' \sim s_2'\}.$$

Then $N_1' \approx N_2'$ by construction.

Let

$$N_2'' = \{s_2' | \exists\, s_2 \in N_2 \cdot s_2 \stackrel{\pi(a)\tau^*}{\longmapsto}_P s_2'\}.$$

Clearly $N_2' \subseteq N_2''$; and $N_2 \xrightarrow{\pi(a)}_N N_2''$ by the definition of the prenormal form. We prove $N_2'' \subseteq N_2'$. So suppose $s_2' \in N_2''$, $s_2 \in N_2$ and $s_2 \stackrel{\pi(a)\tau^*}{\longmapsto}_P s_2'$. Then there exists $s_1 \in N_1$ such that $s_1 \sim s_2$. But hence there exists s_1' such that $s_1 \stackrel{a\tau^*}{\longmapsto}_P s_1'$ and $s_1' \sim s_2'$. Then necessarily $s_1' \in N_1'$. But hence $s_2' \in N_2'$, by the construction of N_2', as required.

Hence $N_2' = N_2''$, and $N_2 \xrightarrow{\pi(a)}_N N_2'$, as required.

- The case $N_2 \xrightarrow{a}_N N_2'$ is very similar.
- Suppose $A \in \mathsf{minaccs}_N(N_1)$; then there exists $s_1 \in N_1$ such that $A \in \mathsf{minaccs}_P(s_1)$. But there exists $s_2 \in N_2$ such that $s_1 \sim s_2$; and so $\pi(A) \in \mathsf{minaccs}_P(s_2)$.

  To show that $\pi(A)$ is a *minimal* acceptance of N_2, suppose, for a contradiction, that some member s_2' of N_2 has a minimal acceptance $B \subset \pi(A)$. But there exists $s_1' \in N_1$ such that $s_1' \sim s_2'$. But then $\pi^{-1}(B) \in \mathsf{minaccs}_P(s_1')$, and $\pi^{-1}(B) \subset A$, contradicting the fact that A is a *minimal* acceptance of N_1.

  Hence $\pi(A) \in \mathsf{minaccs}_N(N_2)$, as required.

  The reverse condition is very similar.
- The condition for divergences is very similar.

Finally, consider $init_N = \{s | init_P \stackrel{\tau^*}{\longmapsto}_P s\}$. We show $init_N \approx init_N$. So let $s \in init_N$. Then $init_P \stackrel{\tau^*}{\longmapsto}_P s$. But since $init_P \sim init_P$, then exists s' such that $init_P \stackrel{\tau^*}{\longmapsto}_P s'$ and $s \sim s'$. But then $s' \in init_N$, as required. The converse holds similarly.

Hence the prenormal form N is G-symmetric.

Clearly neither factoring by strong bisimulation nor removing unreachable states breaks G-symmetry. Hence $norm(P)$ is G-symmetric. □

We will need to apply permutations to states of the specification. The following lemma justifies the soundness of this.

Lemma 16. Let P be a G-symmetric normalised GLTS. Then, for each $s \in P$ and $\pi \in G$, there exists a unique $s' \in P$ such that $s \sim_\pi s'$.

Proof. The existence of s' follows directly from Lemma 14. In order to show s' is unique, suppose $s \sim_\pi s''$. Then $s' \sim_{\pi;\pi^{-1}} s''$, i.e. $s' \sim_{id} s''$, that is, s and s'' are strongly bisimilar; so $s' = s''$ by definition of normalisation. □

Henceforth, we write $\pi(s)$ for the unique s', implied by the above lemma, such that $s \sim_\pi s'$.

4.2 Representative members

The following definition formalises the notion of a representative member of an equivalence class.

Definition 17. Let $L = (S, \Delta, init, \mathsf{minaccs}, \mathsf{div})$ be a G-symmetric GLTS. Write $s_1 \sim_G s_2$ iff there exists $\pi \in G$ such that $s_1 \sim_\pi s_2$. Note that $\sim_G$ is an equivalence relation.

We say that $rep : S \to S$ is a *G-representative function for L* if $s \sim_G rep(s)$ for every $s \in S$. We define $rep(S) = \{rep(s) | s \in S\}$, the set of representative states; we abuse notation and write $rep(L)$ for the same set.

We say that rep gives *unique representatives* if $\forall s, s' \in S \cdot s \sim_G s' \Rightarrow rep(s) = rep(s')$, i.e. rep selects a unique representative of each equivalence class.

For the rest of this section, we assume the existence of a G-representative function rep for Q. We will typically denote representative states by $\hat{s}$, $\hat{q}$, etc.

Ideally, we would like our representative functions to produce unique representatives, because this will give the greatest reduction in the state space. However, finding unique representatives is hard, in general [CEJS98]. Therefore, our approach will not assume this in general. Section 8 considers how to define a suitable efficient representative function that produces unique representatives in most cases.

4.3 The reduced product automata

We now show how to factor the product automaton using a G-representative function. Our subsequent model checking algorithms will search in this reduced product automaton.

Throughout this section, let $P = (S_P, \Delta_P, init_P, \mathsf{minaccs}_P, \mathsf{div}_P)$ be a normalised G-symmetric GLTS, $Q = (S_Q, \Delta_Q, init_Q, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ be a G-symmetric GLTS, and *rep* be a G-representative function on Q.

Definition 18. We lift *rep* to pairs of states from $S_P \times S_Q$ by defining

$$rep(p, q) = (\pi(p), rep(q)) \quad \text{where } \pi \text{ is such that } q \sim_\pi rep(q).$$

Below we use names like $(\hat{p}, \hat{q})$ for such representative pairs of states.

The *rep-reduced product automaton* of P and Q is a product automaton $(S_P \times rep(Q), \Delta, init, \mathsf{minaccs}_P, \mathsf{div}_P, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ such that

- $(\hat{p}, \hat{q}) \xrightarrow{a} rep(p', q')$ in Δ if $\hat{q} \in rep(Q)$, and $(\hat{p}, \hat{q}) \xrightarrow{a} (p', q')$ in the standard product automaton of P and Q (i.e. $\hat{q} \xrightarrow{a}_Q q'$, and if $a \neq \tau$ then $\hat{p} \xrightarrow{a}_P p'$ else $\hat{p} = p'$).
- $init = rep(init_P, init_Q)$.

Throughout the rest of this section, let S be the standard product automaton of P and Q, and R the reduced product automaton of P and Q.

The following lemma shows how steps of the standard product automaton are matched by steps of the reduced product automaton, and vice versa.

Lemma 19.

1. If $(p_1, q_1) \xrightarrow{a}_S (p_2, q_2)$ and $q_1 \sim_\pi \hat{q}_1 \in rep(Q)$ then

$$\exists\, \hat{q}_2 \in rep(Q), \pi' \in G \cdot (\pi(p_1), \hat{q}_1) \xrightarrow{\pi(a)}_R (\pi'(p_2), \hat{q}_2) \wedge q_2 \sim_{\pi'} \hat{q}_2.$$

2. If $(\hat{p}_1, \hat{q}_1) \xrightarrow{a}_R (\hat{p}_2, \hat{q}_2)$, $q_1 \sim_\pi \hat{q}_1$ and $\hat{p}_1 = \pi(p_1)$ then

$$\exists\, p_2, q_2, \pi' \cdot (p_1, q_1) \xrightarrow{\pi^{-1}(a)}_S (p_2, q_2) \wedge q_2 \sim_{\pi'} \hat{q}_2 \wedge \hat{p}_2 = \pi'(p_2).$$

Proof.

1. Since $(p_1, q_1) \xrightarrow{a}_S (p_2, q_2)$ we have that $q_1 \xrightarrow{a}_Q q_2$. Then since $q_1 \sim_\pi \hat{q}_1$, there exists $q' \in S_Q$ such that $\hat{q}_1 \xrightarrow{\pi(a)}_Q q' \wedge q_2 \sim_\pi q'$. Let $\hat{q}_2 = rep(q')$ and π'' be such that $q' \sim_{\pi''} \hat{q}_2$. Then, $q_2 \sim_{\pi'} \hat{q}_2$ where $\pi' = \pi \,;\, \pi''$.

   If $a \neq \tau$, then $p_1 \xrightarrow{a}_P p_2$ by the definition of S. Hence, $\pi(p_1) \xrightarrow{\pi(a)}_P \pi(p_2)$. So by the definition of R, $(\pi(p_1), \hat{q}_1) \xrightarrow{\pi(a)}_R (\pi''(\pi(p_2), \hat{q}_2)$. But $\pi''(\pi(p_2)) = \pi'(p_2)$, as required.

   The case $a = \tau$ is similar, except $p_1 = p_2$ and $\pi(p_1) = \pi(p_2)$.

2. From the definition of R, there exists q_2' such that $\hat{q}_1 \xrightarrow{a}_Q q_2'$ and $\hat{q}_2 = rep(q_2')$. Then by the definition of $\sim_\pi$, there exists q_2 such that $q_1 \xrightarrow{\pi^{-1}(a)}_Q q_2$ and $q_2 \sim_\pi q_2'$. Let π'' be such that $q_2' \sim_{\pi''} \hat{q}_2$. Then $q_2 \sim_{\pi'} \hat{q}_2$ where $\pi' = \pi \,;\, \pi''$.

   If $a \neq \tau$, then by the definition of R, there exists p_2' such that $\hat{p}_1 \xrightarrow{a}_P p_2'$ and $\hat{p}_2 = \pi''(p_2')$. Then, since $\hat{p}_1 = \pi(p_1)$, there exists p_2 such that $p_1 \xrightarrow{\pi^{-1}(a)}_P p_2$ and $p_2' = \pi(p_2)$. Hence $\hat{p}_2 = \pi'(p_2)$. Then from the definition of S, $(p_1, q_1) \xrightarrow{\pi^{-1}(a)}_S (p_2, q_2)$.

   The case $a = \tau$ is similar, except $\hat{p}_1 = p_2'$ and $p_1 = p_2$.

□

The following lemma relates traces of the standard and reduced product automata.

Lemma 20.

1. Suppose the standard product automaton has transitions as follows:

$$(p_0, q_0) \xrightarrow{a_0}_S (p_1, q_1) \xrightarrow{a_1}_S \ldots \xrightarrow{a_{n-1}}_S (p_n, q_n) \land q_0 \sim_{\pi_0} \hat{q}_0 \in rep(Q).$$

   Then the reduced product automaton has transitions as follows:

$$\begin{aligned} &\exists\, \hat{q}_1, \ldots, \hat{q}_n \in rep(Q), \pi_1, \ldots, \pi_n \in G \cdot \\ &\quad (\pi_0(p_0), \hat{q}_0) \xrightarrow{\pi_0(a_0)}_R (\pi_1(p_1), \hat{q}_1) \xrightarrow{\pi_1(a_1)}_R \ldots \\ &\quad\quad \xrightarrow{\pi_{n-1}(a_{n-1})}_R (\pi_n(p_n), \hat{q}_n) \land \\ &\quad \forall\, i \in \{0, \ldots, n\} \cdot q_i \sim_{\pi_i} \hat{q}_i. \end{aligned}$$

2. Suppose the reduced product automaton has transitions as follows:

$$(\hat{p}_0, \hat{q}_0) \xrightarrow{a_0}_R (\hat{p}_1, \hat{q}_1) \xrightarrow{a_1}_R \ldots \xrightarrow{a_{n-1}}_R (\hat{p}_n, \hat{q}_n) \land q_0 \sim_{\pi_0} \hat{q}_0 \land \hat{p}_0 = \pi_0(p_0).$$

   Then the standard product automaton has transitions as follows:

$$\begin{aligned} &\exists\, p_1, \ldots, p_n \in S_P, q_1, \ldots, q_n \in S_Q, \pi_1, \ldots, \pi_n \in G \cdot \\ &\quad (p_0, q_0) \xrightarrow{\pi_0^{-1}(a_0)}_S (p_1, q_1) \xrightarrow{\pi_1^{-1}(a_1)}_S \ldots \xrightarrow{\pi_{n-1}^{-1}(a_{n-1})}_S (p_n, q_n) \land \\ &\quad \forall\, i \in \{0, \ldots, n\} \cdot q_i \sim_{\pi_i} \hat{q}_i \land \hat{p}_i = \pi_i(p_i). \end{aligned}$$

Proof. This is a straightforward induction, making use of Lemma 19. □

Lemma 21. Let tr be a trace in $\Sigma^{\tau\surd *}$.

1. Suppose in the standard product automaton:

$$(p, q) \overset{tr}{\longmapsto}_S (p', q') \wedge q \sim_\pi \hat{q} \in rep(Q).$$

   Then in the reduced product automaton:

$$\begin{array}{l} \exists\, \hat{q}' \in rep(Q), \pi' \in G, tr' \in \Sigma^{\tau\checkmark *} \cdot \\ \quad (\pi(p), \hat{q}) \overset{tr'}{\longmapsto}_R (\pi'(p'), \hat{q}') \wedge q' \sim_{\pi'} \hat{q}'. \end{array}$$

2. Suppose in the reduced product automaton:

$$(\hat{p}, \hat{q}) \overset{tr}{\longmapsto}_R (\hat{p}', \hat{q}') \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

   Then in the standard product automaton:

$$\begin{array}{l} \exists\, p' \in S_P, q' \in S_Q, \pi' \in G, tr' \in \Sigma^{\tau\checkmark *} \cdot \\ \quad (p, q) \overset{tr'}{\longmapsto}_S (p', q') \wedge q' \sim_{\pi'} \hat{q}' \wedge \hat{p}' = \pi'(p'). \end{array}$$

Proof. This follows easily from the previous lemma. □

4.4 Refinement checking algorithms

We now present model checking algorithms for each of the three models.

Throughout this section, let $P = (S_P, \Delta_P, init_P, \mathsf{minaccs}_P, \mathsf{div}_P)$ be a normalised G-symmetric GLTS, $Q = (S_Q, \Delta_Q, init_Q, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ be a G-symmetric GLTS, *rep* be a G-representative function on Q, S be the standard product automaton of P and Q, and R the reduced product automaton of P and Q.

4.4.1 The traces model

The following proposition shows how trace refinements are exhibited in the reduced product automaton.

Proposition 22. $init_P \sqsubseteq_T init_Q$ iff

$$\begin{array}{l} \nexists tr \in \Sigma^{\tau\checkmark *}, a \in \Sigma^{\checkmark}, \hat{p} \in S_P, \hat{q} \in S_Q \cdot \\ \quad rep(init_P, init_q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \hat{q} \overset{a}{\rightarrow}_Q \ \wedge \hat{p} \overset{a}{\not\rightarrow}_P . \end{array}$$

Proof. ($\Rightarrow$) We prove the contrapositive. Suppose

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \hat{q} \overset{a}{\rightarrow}_Q \wedge \hat{p} \overset{a}{\not\rightarrow}_P .$$

Then by Lemma 21, there exist a trace tr', states p and q, and $\pi \in G$ such that

$$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p)$$

Also, since $\hat{q} \xrightarrow{a}_Q$ and $\hat{p} \not\xrightarrow{a}_P$, we have $q \xrightarrow{\pi^{-1}(a)}_Q \wedge p \not\xrightarrow{\pi^{-1}(a)}_P$. Hence $tr' \frown \langle \pi^{-1}(a) \rangle \in traces(init_Q)$, but (by the uniqueness of the state p reached after tr', Lemma 5) $tr' \frown \langle \pi^{-1}(a) \rangle \notin traces(init_P)$. Hence $init_P \not\sqsubseteq_T init_Q$.

($\Leftarrow$) We prove the contrapositive. Suppose $init_P \not\sqsubseteq_T init_Q$. Then there exist states p and q, $tr' \in \Sigma^{\tau\checkmark *}$, $b \in \Sigma^{\checkmark}$ such that

$$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q \xrightarrow{b}_Q \wedge p \not\xrightarrow{b}_P .$$

Then by Lemma 21, there exist a trace $tr \in \Sigma^{\tau\checkmark *}$, states $\hat{p}$ and $\hat{q}$, and $\pi \in G$ such that

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \hat{p} = \pi(p) \wedge q \sim_\pi \hat{q}.$$

Further,

$$\hat{q} \xrightarrow{\pi(b)}_Q \wedge \hat{p} \not\xrightarrow{\pi(b)}_P .$$

Taking $a = \pi(b)$ we have the result. □

The above proposition justifies the model checking algorithm in Figure 3. The algorithm searches the reduced product automaton for a state $(\hat{p}, \hat{q})$ such that $\hat{q} \xrightarrow{a}_Q \wedge \hat{p} \not\xrightarrow{a}_P$. We do not explicitly build the product automaton: instead we explore it on-the-fly, based on the specification and implementation GLTSs. The algorithm maintains a set *seen* of all states seen so far, and a set *pending* of states that still need to be expanded; FDR implements *pending* as a queue, so as to perform a breadth-first search. Note that the algorithm is identical to the standard algorithm, except for the application of *rep* to obtain the representative member. Hence the highly optimised refinement algorithms of [GRABR15] can be used.

In practice, there is no need to include the acceptances and divergences components of the GLTSs.

When FDR detects that a refinement assertion does not hold, it presents the user with an informative counterexample that explains why it does not hold. For example, if $P \sqsubseteq_T Q$ fails, then the counterexample is of the form $tr \frown \langle a \rangle$ where $tr \in traces(P) \cap traces(Q)$, but $tr \frown \langle a \rangle \in traces(Q) \setminus traces(P)$.

Whenever FDR finds a new state, it records which state it was reached from. This allows FDR to construct the path followed through the reduced product automaton, i.e. a sequence of states $(\hat{p}_0, \hat{q}_0), (\hat{p}_1, \hat{q}_1), \dots (\hat{p}_n, \hat{q}_n)$.

Input:

- The normalised GLTS $P = (S_P, \Delta_P, init_P, \mathsf{minaccs}_P, \mathsf{div}_P)$ for the specification;
- The GLTS $Q = (S_Q, \Delta_Q, init_Q, \mathsf{minaccs}_Q, \mathsf{div}_Q)$ for the implementation;
- A representative function *rep*.

Algorithm:

$seen = \{rep(init_P, init_Q)\}$; $pending = \{rep(init_P, init_Q)\}$
while $pending \neq \{\}$ do
  pick $(\hat{p}, \hat{q}) \in pending$; $pending = pending \setminus \{(\hat{p}, \hat{q})\}$
  if $\exists\, a \in \Sigma^{\checkmark} \cdot \hat{q} \xrightarrow{a}_Q \;\wedge\; \hat{p} \not\xrightarrow{a}_P$ then report non-refinement; exit
  else
    for each a, p, q such that $\hat{q} \xrightarrow{a}_Q q \wedge$ if $a \neq \tau$ then $\hat{p} \xrightarrow{a}_P p$ else $p = \hat{p}$ do
      $(\hat{p}', \hat{q}') = rep(p, q)$
      if $(\hat{p}', \hat{q}') \notin seen$ then
        $pending = pending \cup \{(\hat{p}', \hat{q}')\}$; $seen = seen \cup \{(\hat{p}', \hat{q}')\}$
    end
end
report success

Figure 3: Traces-refinement model checking algorithm on the reduced product automaton.

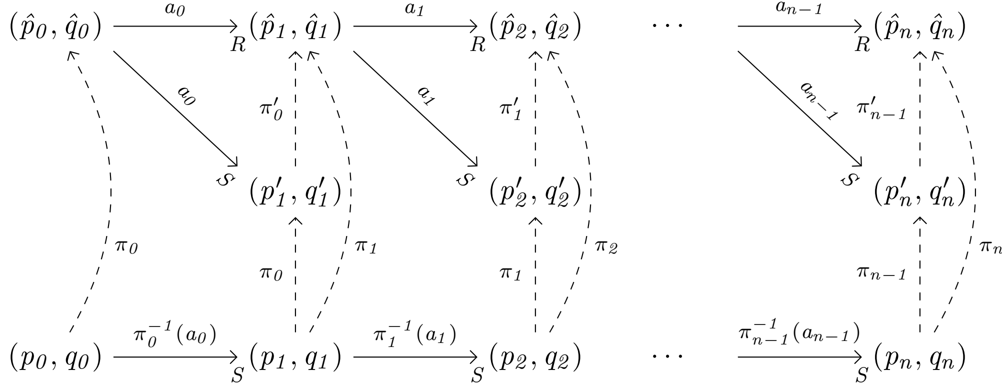


Figure 4: An explanation of counterexample unwinding: the path in the reduced product automaton is in the top row, and the path found in the standard product automaton is in the bottom row. Transitions are represented by solid arrows, and permutation bisimulations are represented by dashed arrows.

However, it does not record the labels of the transitions, nor the permutations used to produce representatives, in order to reduce memory usage. It needs to find the corresponding path through the standard product automaton, together with the labels of transitions. The construction is illustrated in Figure 4. Below we write $(p, q) \sim_\pi (p', q')$ for $\pi(p) = p' \wedge q \sim_\pi q'$.

The path in the standard product construction starts at the initial state $(p_0, q_0) = (init_P, init_Q)$. FDR can calculate the permuation π_0 such that $(p_0, q_0) \sim_{\pi_0} (\hat{p}_0, \hat{q}_0) = rep(p_0, q_0)$.

Suppose, inductively, we have a state (p_i, q_i) of the standard product automaton, and a permutation π_i such that $(p_i, q_i) \sim_{\pi_i} (\hat{p}_i, \hat{q}_i)$. FDR needs to computes a b, (p_{i+1}, q_{i+1}) and π_{i+1} such that $(p_i, q_i) \xrightarrow{b} (p_{i+1}, q_{i+1})$ and $(p_{i+1}, q_{i+1}) \sim_{\pi_{i+1}} (\hat{p}_{i+1}, \hat{q}_{i+1})$. Consider the transition $(\hat{p}_i, \hat{q}_i) \rightarrow_R (\hat{p}_{i+1}, \hat{q}_{i+1})$ in the reduced product automaton. FDR searches over the transitions of $(\hat{p}_i, \hat{q}_i)$ in the standard product automaton to find a transition $(\hat{p}_i, \hat{q}_i) \xrightarrow{a_i}_S (p'_{i+1}, q'_{i+1})$ such that $rep(p'_{i+1}, q'_{i+1}) = (\hat{p}_{i+1}, \hat{q}_{i+1})$. Then $(\hat{p}_i, \hat{q}_i) \xrightarrow{a_i}_R (\hat{p}_{i+1}, \hat{q}_{i+1})$ by construction of the reduced automaton. Also, since $(p_i, q_i) \sim_{\pi_i} (\hat{p}_i, \hat{q}_i)$, we have $(p_i, q_i) \xrightarrow{\pi_i^{-1}(a_i)}_S (p_{i+1}, q_{i+1})$ for some (p_{i+1}, q_{i+1}) such that $(p_{i+1}, q_{i+1}) \sim_{\pi_i} (p'_{i+1}, q'_{i+1})$. Let π'_i be such that $(p'_{i+1}, q'_{i+1}) \sim_{\pi'_i} (\hat{p}_{i+1}, \hat{q}_{i+1})$. Then letting $\pi_{i+1} = \pi_i \mathbin{;} \pi'_i$, we have $(p_{i+1}, q_{i+1}) \sim_{\pi_{i+1}} (\hat{p}_{i+1}, \hat{q}_{i+1})$, as required.

Continuing in this way, we can construct the path corresponding to the counterexample through the standard product automaton.

4.4.2 Stable failures model

We now consider refinement in the stable failures model. The following proposition shows how stable-failures refinements are exhibited in the reduced product automaton.

Proposition 23. $init_P \sqsubseteq_F init_Q$ iff

$$\begin{array}{l} \nexists tr \in \Sigma^{\tau\checkmark *}, \hat{p} \in S_P, \hat{q} \in S_Q \cdot \\ \quad rep(init_P, init_q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \\ \quad (\ \exists\, a \in \Sigma^{\checkmark} \cdot \hat{q} \overset{a}{\rightarrow}_Q \ \wedge \hat{p} \overset{a}{\not\rightarrow}_P \\ \quad\ \ \vee \\ \quad\ \ \exists\, X \in \mathbf{P}\,\Sigma^{\checkmark} \cdot \hat{q}\ \mathsf{ref}_Q\ X \wedge \neg\hat{p}\ \mathsf{ref}_P\ X\ ). \end{array}$$

Proof. ($\Rightarrow$) We prove the contrapositive. Suppose

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}).$$

If $\hat{q} \overset{a}{\rightarrow}_Q \wedge \hat{p} \overset{a}{\not\rightarrow}_P$, then the proof is as for Proposition 22. So suppose

$$\hat{q}\ \mathsf{ref}_Q\ X \wedge \neg\hat{p}\ \mathsf{ref}_P\ X.$$

Then by Lemma 21, there exist a trace tr', states p and q, and $\pi \in G$ such that

$$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

Now $\hat{q}\ \mathsf{ref}_Q\ X$, so $q\ \mathsf{ref}_Q\ \pi^{-1}(X)$ by Lemma 12. And similarly $\neg\hat{p}\ \mathsf{ref}_P\ X$ so $\neg p\ \mathsf{ref}_P\ \pi^{-1}(X)$. Hence

$$(tr' \setminus \{\tau\}, \pi^{-1}(X)) \quad \in \quad failures(init_Q) \setminus failures(init_P),$$

(by the uniqueness of the state of P reached after tr'). Hence $init_P \not\sqsubseteq_F init_Q$.

($\Leftarrow$) We prove the contrapositive. Suppose $init_P \not\sqsubseteq_F init_Q$. If this corresponds to there being a trace of $init_Q$ that is not a trace of $init_P$, then the proof is as in Proposition 22. So suppose

$$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q\ \mathsf{ref}_Q\ Y \wedge \neg p\ \mathsf{ref}_P\ Y.$$

Then by Lemma 21, there exist a trace tr, states $\hat{p}$ and $\hat{q}$, and $\pi \in G$ such that

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

But then $\hat{q}\ \mathsf{ref}_Q\ \pi(Y) \wedge \neg\hat{p}\ \mathsf{ref}_P\ \pi(Y)$ by Lemma 12. Letting $X = \pi(Y)$ we have the result. $\square$

It is straightforward to adapt the model checking algorithm from Figure 3 to the stable failures model. The only change necessary is to replace the condition leading to a non-refinement by

$$(\exists\, a \in \Sigma^{\checkmark} \cdot \hat{q} \xrightarrow{a}_Q \;\wedge \hat{p} \not\xrightarrow{a}_P\,) \vee (\exists\, X \in \mathbf{P}\,\Sigma^{\checkmark} \cdot \hat{q}\ \mathsf{ref}_Q\ X \wedge \neg \hat{p}\ \mathsf{ref}_P\ X).$$

Note that the latter disjunct is equivalent to

$$\exists\, A \in \mathsf{minaccs}_Q(\hat{q}) \cdot \forall\, A' \in \mathsf{minaccs}_P(\hat{p}) \cdot A' \not\subseteq A.$$

4.4.3 Failures-divergences model

We now consider refinement in the failures-divergences model.

Proposition 24. $init_P \sqsubseteq_{FD} init_Q$ iff

$$\begin{array}{l} \nexists tr \in \Sigma^{\tau\checkmark *}, \hat{p} \in S_P, \hat{q} \in S_Q \cdot \\ \quad rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \neg\, \mathsf{div}_P\, \hat{p} \wedge \\ \quad ((\exists\, X \in \mathbf{P}\, \Sigma^{\checkmark} \cdot \hat{q}\ \mathsf{ref}_Q\ X \wedge \neg \hat{p}\ \mathsf{ref}_P\ X) \vee \mathsf{div}_Q\ \hat{q}). \end{array}$$

Proof. (⇒) We prove the contrapositive. Let tr be the shortest trace that makes the right-hand side false. So

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge \neg\, \mathsf{div}_P\, \hat{p}.$$

So necessarily $init_P$ does not diverge after tr or any prefix of it; and necessarily $init_Q$ does not diverge on any proper prefix of tr, by the presumed minimality of tr.

- If $\exists\, X \in \mathbf{P}\, \Sigma^{\checkmark} \cdot \hat{q}\ \mathsf{ref}_Q\ X \wedge \neg \hat{p}\ \mathsf{ref}_P\ X$, the proof is as for Proposition 23.

- If $\mathsf{div}_Q\ \hat{q}$, then by Lemma 21, there exist a trace tr', states p and q, and $\pi \in G$ such that

  $$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

  But then $\mathsf{div}_Q\ q \wedge \neg\, \mathsf{div}_P\ p$. Hence

  $$tr' \setminus \{\tau\} \quad \in \quad divs(init_Q) \setminus divs(init_P),$$

  (by the uniqueness of the state of P reached after tr'). Hence $init_P \not\sqsubseteq_{FD}$ $init_Q$.

(⇐) We again prove the contrapositive. Suppose $init_P \not\sqsubseteq_{FD} init_Q$.

- Suppose $divs(init_Q) \not\subseteq divs(init_P)$. Then there exists a trace $tr' \in \Sigma^{\tau\checkmark *}$ and states p and q such that

$$(init_P, init_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge \mathsf{div}_Q\, q \wedge \neg\, \mathsf{div}_P\, p.$$

By Lemma 21, there exist a trace tr, states $\hat{p}$ and $\hat{q}$, and $\pi \in G$ such that

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

But then $\mathsf{div}_Q\, \hat{q} \wedge \neg\, \mathsf{div}_P\, \hat{p}$, as required.

- Suppose $divs(init_Q) \subseteq divs(init_P)$ but

$$failures_\perp(init_Q) \quad \not\subseteq \quad failures_\perp(init_P).$$

Then there exist $tr' \in \Sigma^{\tau\checkmark *}$, refusal $Y \in \mathbf{P}\, \Sigma^\checkmark$, and states p and q such that

$$(init_P, initst_Q) \overset{tr'}{\longmapsto}_S (p, q) \wedge q\ \mathsf{ref}\ Y \wedge \neg\, \mathsf{div}_P\, p \wedge \neg p\ \mathsf{ref}\ Y.$$

Then by Lemma 21, there exist a trace tr, states $\hat{p}$ and $\hat{q}$, and $\pi \in G$ such that

$$rep(init_P, init_Q) \overset{tr}{\longmapsto}_R (\hat{p}, \hat{q}) \wedge q \sim_\pi \hat{q} \wedge \hat{p} = \pi(p).$$

But then $\hat{q}\ \mathsf{ref}\ \pi(Y) \wedge \neg\, \mathsf{div}_P\, \hat{p} \wedge \neg \hat{p}\ \mathsf{ref}\ \pi(Y)$. Letting $X = \pi(Y)$, we have the result.

□

It is again straightforward to adapt the model checking algorithm from Figure 3 to the failures-divergences model. The condition leading to a non-refinement is changed to

$$\neg\, \mathsf{div}_P\, \hat{p} \wedge ((\exists\, X \in \mathbf{P}\, \Sigma^\checkmark \cdot \hat{q}\ \mathsf{ref}_Q\ X \wedge \neg \hat{p}\ \mathsf{ref}_P\ X) \vee \mathsf{div}_Q\, \hat{q}).$$

5 Symmetry reduction on supercombinators

FDR uses an implicit representation of a GLTS, called a *supercombinator* [GRABR15]. It consists of some component GLTSs, along with rules that describe how transitions of the components should be combined to give transitions of the whole system. In the running example, the component GLTSs would comprise an LTS (interpreted as a GLTS, as in Lemma 3) for each of the threads, each of the nodes, the lock, and the top variable (strictly

speaking, this is just one possible choice, since FDR uses various heuristics to optimise supercombinators; but it is the most natural choice).

In this section we formally define supercombinators, and the induced GLTS. We then prove a result that identifies circumstances under which two supercombinators induces π-bisimilar GLTSs, where π is an event permutation.

The rules of a supercombinator are partitioned into *formats*: in a state, rules from one format are active (the running example uses a single format, essentially because the construction of the system is static). Each rule combines transitions of a subset of the components and determines the event the supercombinator performs. Rules may also *reset* components to their initial state, and may change the format.

Let $-$ be a value, not in $\Sigma^{\tau\checkmark}$; we use it to denote a process performing no event. Let $\Sigma^{-} = \Sigma^{\tau\checkmark} \cup \{-\}$.

Definition 25. A *supercombinator* is a 5-tuple $(\mathcal{L}, F, \mathcal{R}, on, f_0)$ where:

- $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ is a sequence of *component GLTSs*.
- F is a finite set of *formats*.
- $on : F \to \mathbf{P}\{1, \ldots, n\}$ indicates, for each format, which of the component GLTSs are *on*.
- $\mathcal{R}$ is a function from formats to sets of *supercombinator rules*. For each $f \in F$, $\mathcal{R}(f)$ is a finite set of supercombinator rules (e, a, r, f') where:
  - $e \in (\Sigma^{-})^n$ specifies the action each on-component must perform, where $-$ indicates that it performs none; if $e(i) \neq -$ then $i \in on(f)$.
  - $a \in \Sigma$ is the event the supercombinator performs.
  - $r \subseteq \{1, \ldots, n\}$ are the indices of the L_i that are reset.
  - $f' \in F$ is the subsequent format.
- $f_0 \in F$ is the *initial format*.

In FDR, a component GLTS can be implemented in one of three ways:

- As a low-level LTS, explicitly listing the transitions for each state, and interpreted as a GLTS, as in Lemma 3. This is the default, and will be the case when no compression operators are involved.

- As a low-level GLTS, explicitly listing the transitions, minimal acceptances and divergence information for each state. This results from application of certain compression functions. In particular, this GLTS might have been obtained by compressing another supercombinator: in this case, each state of the low-level GLTS will correspond to a state of the nested supercombinator.

- As a lazy enumerated GLTS, calculating transitions as needed. This results from compression functions such as lazyenumerate, chase and prioritise. The lazy enumerated GLTS might be formed by wrapping a nested supercombinator; in this case, each state of the GLTS will again correspond to a state of the nested supercombinator.

Given a supercombinator, a corresponding GLTS can be constructed.

Definition 26. Let $\mathcal{S} = (\langle L_1, \ldots, L_n\rangle, F, \mathcal{R}, on, f_0)$ be a supercombinator where $L_i = (S_i, \Delta_i, init_i, \mathsf{minaccs}_i, \mathsf{div}_i)$. The GLTS *induced by* $\mathcal{S}$ is the GLTS $(S, \Delta, init, \mathsf{minaccs}, \mathsf{div})$ such that:

- States are tuples consisting of the state of each component, plus the identifier of the format: $S \subseteq S_1 \times \cdots S_n \times F$.

- The initial state is the tuple containing the initial states of each of the components, along with the initial format: $init = (init_1, \ldots, init_n, f_0)$.

- The transitions correspond to the supercombinator rules firing. Let $\sigma = (s_1, \ldots, s_n, f)$, and $\sigma' = (s'_1, \ldots, s'_n, f')$. Then $(\sigma, a, \sigma') \in \Delta$ iff there exists $((b_1, \ldots, b_n), a, r, f') \in \mathcal{R}(f)$ such that for each component i, there exists a state s''_i such that

  1. If $b_i = -$ then $s''_i = s_i$; and if $b_i \neq -$ then $s_i \xrightarrow{b_i}_i s''_i$; i.e. component i performs b_i, or does nothing if $b_i = -$;
  2. If $i \notin r$ then $s'_i = s''_i$; and if $i \in r$ then $s'_i = init_i$; i.e. the components in r are reset to their initial states.

- For each state $\sigma = (s_1, ..., s_n, f)$ with $on(f) = \{i_1, ..., i_k\}$:

  $$\mathsf{minaccs}(\sigma) = mins\{join_f(X_{i_1}, \ldots, X_{i_k}) | X_{i_1} \in \mathsf{minaccs}_{i_1}(s_{i_1}), \ldots, X_{i_k} \in \mathsf{minaccs}_{i_k}(s_{i_k})\},$$

  where

  $$join_f(X_{i_1}, \ldots, X_{i_k}) = \{a | \exists (e, a, r, f') \in \mathcal{R}(f) \cdot \forall j \in \{1, \ldots, k\} \cdot e(i_j) \neq _ \Rightarrow e(i_j) \in X_{i_j}\},$$

  and *mins* returns the $\subseteq$-minimal elements of its argument.

- For each state σ, $\mathsf{div}(\sigma)$ is true iff either:
  - from σ, $\mathcal{S}$ can perform a finite sequence of τ-transitions to some state $\sigma' = (s_1, ..., s_n, f)$ such that some on component can diverge, i.e. $\exists\, i \in on(f) \cdot \mathsf{div}_i(s_i)$; or
  - from σ, $\mathcal{S}$ can perform an infinite sequence of τ-transitions.

It is straightforward to define a supercombinator corresponding to each CSP operator and recursion, and to compose supercombinators hierarchically so as to define a single supercombinator for a system. The following examples illustrate the ideas.

Example 27. Let $T = \{t_1, \ldots, t_n\}$, and consider

$$\sqcap_{t \in T}(P(t) \setminus X(t)).$$

The natural supercombinator would be $(\langle L_1, \ldots, L_n\rangle, \{f_0, \ldots, f_n\}, \mathcal{R}, on, f_0)$, where:

- L_i is the LTS for $P(t_i)$.
- f_0 is the initial format, and for $i > 0$, f_i is the format corresponding to the nondeterministic choice choosing t_i; so $on(f_0) = \{\}$ and $on(f_i) = \{i\}$ for $i > 0$.
- The rules $\mathcal{R}$ are as follows. We write "$-^n$" for the tuple $(-, \ldots, -)$ of size n, and "e_i^a" for the tuple with a in position i and $-$ elsewhere; then we have

$$\begin{array}{rcl} \mathcal{R}(f_0) & = & \{(-^n, \tau, \{\}, f_i) | i \in \{1, \ldots, n\}\}, \\ \mathcal{R}(f_i) & = & \{(e_i^a, a, \{\}, f_i) | a \in \Sigma^{\tau\checkmark} \setminus X(t_i)\} \cup \\ & & \{(e_i^a, \tau, \{\}, f_i) | a \in X(t_i)\}, \qquad i = 1, \ldots, n. \end{array}$$

  The rule $\mathcal{R}(f_0)$ captures that the system can perform a τ (with no component changing state), and evolve into the state captured by format f_i. The rule $\mathcal{R}(f_i)$ captures that if the ith component can perform a transition labelled with a, then the system can perform a transition labelled with either a or τ (depending on whether $a \in X(t_i)$), and remain in the same format. In each case, no component is reset.

Example 28. Let P be a process that can perform $\checkmark$ (indicating termination), and let $Q = P\ ;\ Q$. Then the natural supercombinator for Q would

have a single component LTS L_1, corresponding to P, and a single format f_0 such that $on(f_0) = \{1\}$, and

$$\mathcal{R}(f_0) = \{((a), a, \{\}, f_0) | a \in \Sigma^\tau\} \cup \{((\surd), \tau, \{1\}, f_0)\}$$

The last rule captures that if L_1 performs $\surd$, the $\surd$ is made internal (i.e. τ), and L_1 is reset to its initial state.

5.1 Symmetries between supercombinators

We now consider symmetries between supercombinators, and how these correspond to symmetries between the corresponding GLTSs. We are mainly interested in showing that the supercombinator corresponding to the implementation process in a refinement check is symmetric, i.e. π-bisimilar to itself for every event permutation π in some group G. However, when several components of a supercombinator are implemented as nested supercombinators, we will sometimes want to show that that one nested supercombinator is π-bisimilar to another, for a particular event permutation π.

We start by relating the component GLTSs of two supercombinators.

Definition 29. Consider two collections of GLTSs, $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ and $\mathcal{L}' = \langle L'_1, \ldots, L'_n \rangle$. Let π be an event permutation, and let α be a bijection from $\{1, \ldots, n\}$ to itself. We say that $\mathcal{L}$ is *π-mappable to $\mathcal{L}'$ using component bijection α* if for every i, $L_i \sim_\pi L'_{\alpha(i)}$. This shows how to map L_i onto a π-bisimilar GLTS $L'_{\alpha(i)}$. (We sometimes write α as α_π, to emphasise the event permutation π.)

Example 30. Consider the system $\parallel t \in T \cdot [A(t)] P(t)$. Let $T = \{t_1, \ldots, t_n\}$ and let the GLTSs for $P(t_1), \ldots, P(t_n)$ be $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ (respectively). Suppose, also, that these processes use data from some polymorphic type V.

Suppose $\pi \in EvSym(T, V)$, i.e. π is an event permutation formed by lifting permutations on T and V. For each $i \in \{1, \ldots, n\}$, define $\alpha_\pi(i) = j$ such that $\pi(t_i) = t_j$; note that such a j exists, and that α_π is a bijection. We will show later (Proposition 45) —subject to some reasonable assumptions, in particular that P contains no constants of type T or V— that $L_i \sim_\pi L_{\alpha_\pi(i)}$; i.e. the GLTSs for $P(t_i)$ and $P(\pi(t_i))$ are π-bisimilar. Hence $\mathcal{L}$ is π-mappable to itself.

Example 31. Consider the processes

$$\begin{aligned} P &= |||\, t : T \cdot compress(Q(t)), \\ Q(t) &= |||\, t' : T \cdot R(t, t'), \end{aligned}$$

where *compress* is some compression function. FDR will normally implement each $compress(Q(t))$ as a component of the top-level supercombinator for P; each such component is produced by compressing the GLTS of the supercombinator corresponding to $Q(t)$; each component GLTS of that nested supercombinator will correspond to $R(t, t')$ for some $t' \in T$.

Let π be a bijection on T, lifted to events by point-wise application. Suppose $\mathcal{S}$ and $\mathcal{S}'$ are the supercombinators corresponding to $Q(t)$ and $Q(\pi(t))$, respectively, and suppose $\mathcal{L} = \langle L_1, \ldots, L_n \rangle$ and $\mathcal{L}' = \langle L'_1, \ldots, L'_n \rangle$ are the components of those supercombinators. We show that $\mathcal{L}$ is π-mappable to $\mathcal{L}'$. Define $\alpha = \alpha_\pi$ such that if L_i corresponds to $R(t, t')$ then $L'_{\alpha(i)}$ corresponds to $R(\pi(t), \pi(t'))$; note that α is well-defined and a bijection. Under reasonable assumptions (as in Example 30) we will have $L_i \sim_\pi L'_{\alpha(i)}$, as required.

We now consider how to relate the rules of two supercombinators. Given a permutation π on $\Sigma^{\tau\checkmark}$, we extend it to Σ^- by defining $\pi(-) = -$. The following definition captures when two formats (maybe in different supercombinators) act in a similar way, but on different component GLTSs, and with events renamed under π. We will use this to identify when the supercombinators induce π-bisimilar GLTSs.

Definition 32. Let π be an event permutation. Let

$$\begin{array}{rcl} \mathcal{S} & = & (\langle L_1, \ldots, L_n \rangle, F, \mathcal{R}, on, f_0), \\ \mathcal{S}' & = & (\langle L'_1, \ldots, L'_n \rangle, F', \mathcal{R}', on', f'_0) \end{array}$$

be supercombinators whose component GLTSs are π-mappable with component bijection α. Then a relation $\sim^F$ over $F \times F'$ is a *format π-bisimulation using α* if whenever $f \sim^F f$:

- If $(e, a, r, f_1) \in \mathcal{R}(f)$ then there is a rule $(e', \pi(a), \alpha(r), f'_1) \in \mathcal{R}'(f')$ such that $\forall\, i \in \{1, \ldots, n\} \cdot e'(\alpha(i)) = \pi(e(i))$ and $f_1 \sim^F f'_1$.
- If $(e', a, r, f'_1) \in \mathcal{R}'(f')$ then there is a rule $(e, \pi^{-1}(a), \alpha^{-1}(r), f_1) \in \mathcal{R}(f)$ such that $\forall\, i \in \{1, \ldots, n\} \cdot e'(\alpha(i)) = \pi(e(i))$ and $f_1 \sim^F f'_1$.
- $\alpha(on(f)) = on'(f')$.

This says that f acts on each GLTS L_i in the same way as f' acts on $L'_{\alpha(i)}$, but with the latter's events renamed under π.

Definition 33. Consider two supercombinators $\mathcal{S} = (\mathcal{L}, F, \mathcal{R}, on, f_0)$ and $\mathcal{S}' = (\mathcal{L}', F', \mathcal{R}', on', f'_0)$. Then *$\mathcal{S}$ is π-mappable to $\mathcal{S}'$ using component bijection α* if

1. $\mathcal{L}$ and $\mathcal{L}'$ are π-mappable using α; and

2. There is a format π-bisimulation $\sim_\pi^F$ over $F \times F'$ using α such that $f_0 \sim_\pi^F f_0'$.

Example 34. Let $T = \{t_1, \ldots, t_n\}$, and consider

$$\bigsqcap_{t \in T} (P(t) \setminus X(t)).$$

The natural supercombinator $\mathcal{S}$ for this system was described in Example 27. Let π be a permutation on T. We show that $\mathcal{S}$ is π-mappable to itself.

As in Example 30, under reasonable assumptions, the GLTSs are π-mappable to themselves under the component bijection α such that $\alpha(i) = j$ when $\pi(t_i) = t_j$. Further, we will show later (Proposition 45) that, under reasonable assumptions, $\pi(X(t_i)) = X(\pi(t_i))$.

Define

$$\sim_\pi^F \quad = \quad \{(f_0, f_0)\} \cup \{(f_i, f_{\alpha(i)}) | i \in \{1, \ldots, n\}\}.$$

Clearly $f_0 \sim_\pi^F f_0$. We show that $\sim_\pi^F$ is a format π-bisimulation. The relevant conditions on the rules are clearly satisfied by the pair (f_0, f_0). For $i > 0$, note that

$$(e_i^a, b, \{\}, f_i) \in \mathcal{R}(f_i) \quad \Leftrightarrow \quad (e_{\alpha(i)}^{\pi(a)}, \pi(b), \{\}, f_{\alpha(i)}) \in \mathcal{R}(\alpha(f_i)),$$

for each a, b, since

- If $a \in X(t_i)$ then $\pi(a) \in \pi(X(t_i)) = X(\pi(t_i)) = X(t_{\alpha(i)})$, and the two rules have $b = \pi(b) = \tau$;

- If $a \notin X(t_i)$, then similarly $\pi(a) \notin X(t_{\alpha(i)})$, and the two rules have $b = a$, and $\pi(b) = \pi(a)$, respectively.

Further, these rules correspond, as required by Definition 32; in particular $e_{\alpha(i)}^{\pi(a)}(\alpha(j)) = \pi(e_i^a(j))$ (since both sides equal $\pi(a)$ if $i = j$, and equal $-$ otherwise).

Finally, $\alpha(on(0)) = \{\} = on(0)$, and for $i > 0$, $\alpha(on(f_i)) = \{\alpha(i)\} = on(f_{\alpha(i)})$.

Hence $\mathcal{S}$ is π-mappable to itself, for each $\pi \in EvSym(T)$.

We now show that π-mappable supercombinators induce π-bisimilar GLTSs.

Lemma 35. Let π be an event permutation, and let

$$\begin{array}{rcl} \mathcal{S} & = & (\langle L_1, \ldots, L_n \rangle, F, \mathcal{R}, on, f_0) \\ \mathcal{S}' & = & (\langle L'_1, \ldots, L'_n \rangle, F', \mathcal{R}', on', f'_0) \end{array}$$

be π-mappable supercombinators with component bijection α and format π-bisimulation $\sim^F_\pi$. Consider the relation $\approx_\pi$ defined over states of the induced GLTSs by

$$(s_1, \ldots, s_n, f) \approx_\pi (s'_1, \ldots, s'_n, f') \text{ iff } (\forall\, i \in \{1, \ldots, n\} \cdot s_i \sim_\pi s'_{\alpha(i)}) \wedge f \sim^F_\pi f'.$$

Then $\approx_\pi$ is a π-bisimulation. Further, the initial states of the two GLTSs are related by $\approx_\pi$.

Proof. Suppose $\sigma = (s_1, \ldots, s_n, f) \approx_\pi \sigma' = (s'_1, \ldots, s'_n, f')$, so $s_i \sim_\pi s'_{\alpha(i)}$ for $i = 1, \ldots, n$ and $f \sim^F_\pi f'$.

Suppose $\sigma \xrightarrow{a} \sigma''$ in $\mathcal{S}$. We show $\sigma' \xrightarrow{\pi(a)} \sigma'''$ in $\mathcal{S}'$, for some σ''' such that $\sigma'' \approx_\pi \sigma'''$. Suppose $\sigma'' = (s''_1, \ldots, s''_n, f'_1)$, and suppose the transition $\sigma \xrightarrow{a} \sigma''$ comes from rule $(e, a, r, f'_1) \in \mathcal{R}(f)$. Consider each index i.

- Suppose $e(i) \neq -$.
  - If $i \notin r$, then $s_i \xrightarrow{e(i)} s''_i$ in L_i. But $s_i \sim_\pi s'_{\alpha(i)}$, so $s'_{\alpha(i)} \xrightarrow{\pi(e(i))} s'''_{\alpha(i)}$ in $L'_{\alpha(i)}$, for some $s'''_{\alpha(i)}$ such that $s''_i \sim_\pi s'''_{\alpha(i)}$. Further, $\alpha(i) \notin \alpha(r)$.
  - If $i \in r$ then $s_i \xrightarrow{e(i)}$ in L_i, and $s''_i = init_i$. As in the previous case, $s'_{\alpha(i)} \xrightarrow{\pi(e(i))}$ in $L'_{\alpha(i)}$. Let $s'''_{\alpha(i)} = init_{\alpha(i)}$; so $s''_i \sim_\pi s'''_{\alpha(i)}$. Also, $\alpha(i) \in \alpha(r)$.
- Suppose $e(i) = -$.
  - If $i \notin r$, then $s_i = s''_i$. Let $s'''_{\alpha(i)} = s'_{\alpha(i)}$, so $s''_i \sim_\pi s'''_{\alpha(i)}$. Further, $\alpha(i) \notin \alpha(r)$.
  - If $i \in r$, then $s''_i = init_i$. Let $s'''_{\alpha(i)} = init_{\alpha(i)}$. Then $s''_i \sim_\pi s'''_{\alpha(i)}$. Further, $\alpha(i) \in \alpha(r)$.

But by Definition 32, there is a rule $(e', \pi(a), \alpha(r), f'_2) \in \mathcal{R}(f')$ where $e'(\alpha(i)) = \pi(e(i))$, for each i, and $f'_1 \sim^F_\pi f'_2$. Let $\sigma''' = (s'''_1, \ldots, s'''_n, f'_2)$. This rule can be used to deduce that $\sigma' \xrightarrow{\pi(a)} \sigma'''$ in $\mathcal{S}'$, as required. But $\sigma'' \approx_\pi \sigma'''$, by construction.

The reverse-direction clause in the definition of π-bisimulation can be proven in an identical way.

We now consider the minimal acceptances. Suppose $on(f) = \{i_1, \ldots, i_k\}$. Suppose $X_{i_j} \in \mathsf{minaccs}(s_{i_j})$ for $j = 1, \ldots, k$, and consider

$$A = join_f(X_{i_1}, \ldots, X_{i_k}),$$

which is an acceptance (not necessarily minimal) of σ. Then $on(f') = \{\alpha(i_1), \ldots, \alpha(i_k)\}$; and $\pi(X_{i_j}) \in \mathsf{minaccs}(s'_{\alpha(i_j)})$, for $j = 1, \ldots, k$, since $s_{i_j} \sim_\pi s'_{\alpha(i_j)}$. We show that $\pi(A) = join_{f'}(\pi(X_{i_1}), \ldots, \pi(X_{i_k}))$, which implies $\pi(A) \in \mathsf{minaccs}(\sigma')$. Consider $a \in A$, corresponding to the rule $(e, a, r, f'') \in \mathcal{R}(f)$; so $\forall j \in \{1, \ldots, k\} \cdot e(i_j) \neq - \Rightarrow e(i_j) \in X_{i_j}$. Then there is a corresponding rule $(e', \pi(a), \alpha(r), f''') \in \mathcal{R}(f')$ such that $e'(\alpha(i)) = \pi(e(i))$, for all i. But then, for $j \in \{1, \ldots, k\}$:

$$\begin{aligned} e'(\alpha(i_j)) \neq - &\Rightarrow e(i_j) \neq - \\ &\Rightarrow e(i_j) \in X_{i_j} \\ &\Rightarrow \pi(e'(\alpha(i_j))) = \pi(e(i_j)) \in \pi(X_{i_j}). \end{aligned}$$

So $\pi(a) \in join_{f'}(\pi(X_{i_1}), \ldots, \pi(X_{i_k}))$. The reverse inclusion is very similar. Hence we have shown that for every acceptance A of σ, $\pi(A)$ is an acceptance of σ'. The reverse direction is very similar. Then clearly the *minimal* acceptances agree: $\pi(\mathsf{minaccs}(\sigma)) = \mathsf{minaccs}(\sigma')$.

We now consider divergences. Suppose $\mathsf{div}(\sigma)$. We perform a case analysis.

- Suppose $\sigma \overset{\tau^*}{\longmapsto} (s''_1, \ldots, s''_n, f'')$ in $\mathcal{S}$, and for some $i \in on(f'')$, $\mathsf{div}(s_i)$. But then, using the above result for transitions, $\sigma' \overset{\tau^*}{\longmapsto} (s'''_1, \ldots, s''_n, f''')$ in $\mathcal{S}'$, with $(s''_1, \ldots, s''_n, f'') \approx_\pi (s'''_1, \ldots, s''_n, f''')$. But then $\alpha(i) \in \alpha(on(f'')) = on(f''')$, and $\mathsf{div}(s_{\alpha(i)})$. Hence $\mathsf{div}(\sigma')$.

- Suppose σ can perform an infinite sequence of τ-transitions in $\mathcal{S}$. Then, using the above result for transitions, σ' can perform an infinite sequence of τ-transitions in $\mathcal{S}'$. Hence $\mathsf{div}(\sigma')$.

The reverse direction is very similar.

Finally, $\approx_\pi$ relates the two initial states, since $L_i \sim_\pi L_{\alpha(i)}$ and $f_0 \sim_\pi^F f'_0$, by assumption. □

The proposition below follows easily from the above lemma.

Proposition 36.

1. Suppose supercombinator $\mathcal{S}$ is π-mappable to supercombinator $\mathcal{S}'$. Then the induced GLTSs are π-bisimilar.

2. Suppose $\mathcal{S}$ is a supercombinator that is π-mappable to itself for every $\pi \in G$. Then the induced GLTS is G-symmetric.

Recall that a supercombinator might contain components that are implemented using nested supercombinators. Together, these supercombinators form a tree (but only to a finite depth). When a component is not implemented as a nested supercombinator, we call it a *leaf*.

In order to statically prove that two supercombinators, possibly with nested supercombinators, induce π-bisimilar GLTSs, we require a stronger condition, that descends through nested supercombinators, showing corresponding components are suitably related.

Definition 37. Consider two supercombinators $\mathcal{S} = (\langle L_1, \ldots, L_n \rangle, F, \mathcal{R}, on, f_0)$ and $\mathcal{S}' = (\langle L'_1, \ldots, L'_n \rangle, F', \mathcal{R}', on', f'_0)$. Let π be an event permutation, and let α be a bijection on $\{1, \ldots, n\}$. Then $\mathcal{S}$ *is recursively* π*-mappable to* $\mathcal{S}'$ *using component bijection* α if

1. For every $i \in \{1, \ldots, n\}$,

    (a) L_i and $L'_{\alpha(i)}$ are leaf GLTSs, and $L_i \sim_\pi L'_{\alpha(i)}$; or

    (b) L_i and $L'_{\alpha(i)}$ are nested supercombinators, and L_i is recursively π-mappable to $L'_{\alpha(i)}$ using some component bijection α_i.

2. There is a format π-bisimulation $\sim_\pi^F$ over $F \times F'$ using α such that $f_0 \sim_\pi^F f'_0$.

In Section 7 we will explain how to statically identify π-bisimilar leaf components. The above definition will then allow us to identify that two supercombinators and recursively π-mappable. The proposition below then shows that these supercombinators are π-bisimilar.

Proposition 38.

1. Suppose supercombinator $\mathcal{S}$ is recursively π-mappable to supercombinator $\mathcal{S}'$. Then the induced GLTSs are π-bisimilar.

2. Suppose $\mathcal{S}$ is a supercombinator that is recursively π-mappable to itself for every $\pi \in G$. Then the induced GLTS is G-symmetric.

Proof. The proof of part 1 is by induction on the depth of the tree of nested supercombinators. Given $\mathcal{S}$ and $\mathcal{S}'$, the inductive hypothesis says that any corresponding components that are implemented as nested supercombinators have induced GLTSs that are π-bisimilar. Hence $\mathcal{S}$ and $\mathcal{S}'$ are π-mappable, and so, by Proposition 36, the induced GLTSs are π-bisimilar.

Part 2 then follows immediately. □

We now show that the property of supercombinators being recursively mappable is compositional in the obvious way. We start by showing how format bisimulations compose. Below we sometimes decorate the component bijections α with the corresponding event permutation and/or their source and target supercombinators.

Lemma 39. Consider three supercombinators $\mathcal{S} = (\mathcal{L}, F, \mathcal{R}, on, f_0)$, $\mathcal{S}' = (\mathcal{L}', F', \mathcal{R}', on', f_0')$, and $\mathcal{S}'' = (\mathcal{L}'', F'', \mathcal{R}'', on'', f_0'')$. Suppose $\sim^F_\pi$ is a format π-bisimulation between $\mathcal{S}$ and $\mathcal{S}'$ using component bijection $\alpha^{\mathcal{S},\mathcal{S}'}_\pi$, and $\sim^F_{\pi'}$ is a format π'-bisimulation between $\mathcal{S}'$ and $\mathcal{S}''$ using component bijection $\alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'}$. Then $\sim^F_\pi \,;\, \sim^F_{\pi'}$ is a format $(\pi \,;\, \pi')$-bisimulation between $\mathcal{S}$ and $\mathcal{S}''$ using component bijection $\alpha^{\mathcal{S},\mathcal{S}'}_\pi \,;\, \alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'}$.

Proof. Suppose $f\,(\sim^F_\pi \,;\, \sim^F_{\pi'})\,f''$. Then there is a format f' such that $f \sim^F_\pi f'$ and $f' \sim^F_{\pi'} f''$. We check the conditions for being a format $(\pi;\pi')$-bisimulation.

Suppose

$$(e, a, r, f_1) \in \mathcal{R}(f).$$

Then, since $f \sim^F_\pi f'$, there is a rule

$$(e', \pi(a), \alpha_\pi(r), f_1') \in \mathcal{R}'(f')$$

such that

$$\forall\, i \in \{1, \ldots, n\} \cdot e'(\alpha_\pi(i)) = \pi(e(i)) \quad \text{and} \quad f_1 \sim^F_\pi f_1'.$$

But then, since $f' \sim^F_{\pi'} f''$, there is a rule

$$(e'', (\pi \,;\, \pi')(a), (\alpha_\pi \,;\, \alpha_{\pi'})(r), f_1'') \in \mathcal{R}''(f'')$$

such that

$$\forall\, i \in \{1, \ldots, n\} \cdot e''(\alpha_{\pi'}(i)) = \pi'(e'(i)) \quad \text{and} \quad f_1' \sim^F_{\pi'} f_1''.$$

Hence,

$$\forall\, i \in \{1, \ldots, n\} \cdot \\ e''((\alpha_\pi \,;\, \alpha_{\pi'})(i)) = e''(\alpha_{\pi'}(\alpha_\pi(i))) = \pi'(e'(\alpha_\pi(i))) = (\pi \,;\, \pi')(e(i)).$$

And $f_1\ (\sim^F_\pi\ ;\ \sim^F_{\pi'})\ f''_1$, as required.

The reverse condition is very similar.

Finally, $\alpha_\pi(on(f)) = on'(f')$ and $\alpha_{\pi'}(on'(f')) = on''(f'')$, so

$$(\alpha_\pi\ ;\ \alpha_{\pi'})(on(f)) = on''(f'').$$

□

Lemma 40. Consider three supercombinators $\mathcal{S} = (\mathcal{L}, F, \mathcal{R}, on, f_0)$, $\mathcal{S}' = (\mathcal{L}', F', \mathcal{R}', on', f'_0)$, and $\mathcal{S}'' = (\mathcal{L}'', F'', \mathcal{R}'', on'', f''_0)$. Suppose $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$ using component bijection $\alpha^{\mathcal{S},\mathcal{S}'}_\pi$, and $\mathcal{S}'$ is recursively π'-mappable to $\mathcal{S}''$ using component bijection $\alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'}$. Then $\mathcal{S}$ is recursively $(\pi\ ;\ \pi')$-mappable to $\mathcal{S}''$ using component bijection $\alpha^{\mathcal{S},\mathcal{S}'}_\pi\ ;\ \alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'}$.

Proof. The proof is by induction on the depth of supercombinator nesting. We prove the result, following the structure of Definition 37.

1. Suppose $\mathcal{L} = \langle L_1, \ldots, L_n\rangle$, $\mathcal{L}' = \langle L'_1, \ldots, L'_n\rangle$ and $\mathcal{L}'' = \langle L''_1, \ldots, L''_n\rangle$, respectively. Consider L_i, $L'_{\alpha_\pi(i)}$ and $L''_{\alpha_{\pi'}(\alpha_\pi(i))}$. There are two possibilities:
   (a) All three are leaf GLTSs, $L_i \sim_\pi L'_{\alpha_\pi(i)}$, and $L'_{\alpha_\pi(i)} \sim_{\pi'} L''_{\alpha_{\pi'}(\alpha_\pi(i))}$. Hence $L_i \sim_{\pi;\pi'} L''_{\alpha_{\pi'}(\alpha_\pi(i))}$.
   (b) All three are nested supercombinators, L_i is recursively π-mappable to $L'_{\alpha_\pi(i)}$, and $L'_{\alpha_\pi(i)}$ is recursively π'-mappable to $L''_{\alpha_{\pi'}(\alpha_\pi(i))}$. Then by the inductive hypothesis, L_i is recursively $(\pi\ ;\ \pi')$-mappable to $L''_{\alpha_{\pi'}(\alpha_\pi(i))}$.
2. By assumption, there is a format π-bisimulation $\sim^F_\pi$ between $\mathcal{S}$ and $\mathcal{S}'$ using α_π such that $f_0 \sim_\pi f'_0$, and there is a format π'-bisimulation $\sim^F_{\pi'}$ between $\mathcal{S}'$ and $\mathcal{S}''$ using $\alpha_{\pi'}$ such that $f'_0 \sim_{\pi'} f''_0$. By Lemma 39, $\sim^F_\pi\ ;\ \sim^F_{\pi'}$ is a format $(\pi;\pi')$-bisimulation between $\mathcal{S}$ and $\mathcal{S}''$ using $\alpha_\pi;\alpha_{\pi'}$. And $f_0\ (\sim^F_\pi\ ;\ \sim^F_{\pi'})\ f''_0$.

□

The way that the component bijections compose prompts the following definition.

Definition 41. We say that a collection of component bijections α is *homomorphic* if

$$\alpha^{\mathcal{S},\mathcal{S}''}_{\pi;\pi'} \quad = \quad \alpha^{\mathcal{S},\mathcal{S}'}_\pi\ ;\ \alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'},$$

whenever $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$ using $\alpha^{\mathcal{S},\mathcal{S}'}_\pi$ and $\mathcal{S}'$ is recursively π'-mappable to $\mathcal{S}''$ using $\alpha^{\mathcal{S}',\mathcal{S}''}_{\pi'}$.

6 Symmetric datatypes

We now restrict our attention to systems that are symmetric in one or more types. More precisely, we consider disjoint sets $T_1, \ldots, T_N$, where

$$T_i \;=\; \{A_{i,1}, \ldots, A_{i,N_i}\}.$$

Further, each T_i is a subset of a datatype $\hat{T}_i$, with a definition of the form

$$\textbf{datatype}\,\hat{T}_i = A_{i,1} | \ldots | A_{i,N_i} | \ldots,$$

so each $A_{i,j}$ is an atomic value. Here, the final "$| \ldots$" may contain other values in which the system is not symmetric, but may be empty. We call the T_i *distinguished subtypes*, and the $\hat{T}_i$ *distinguished supertypes*. In the running example, the system is symmetric in the subtype NodeID of "real" node identities, but not in the containing supertype NodeIDType, which includes the special value Null. Likewise it is symmetric in the type Data and the type ThreadID.

Let $\mathcal{T}$ be the union of the distinguished subtypes:

$$\mathcal{T} \;=\; \bigcup_{i=1}^{N} T_i.$$

For the rest of this section, let $\pi \in Sym(\mathcal{T})$ be a *type-preserving* permutation on $\mathcal{T}$, i.e. for each i, π maps values of type T_i to T_i.

We assume a well-typed script. Below we will show that, subject to certain assumptions —mainly that the CSP script contains no constants from $\mathcal{T}$— the value of every expression is symmetric in those types. We need, first, to sketch the semantics of machine-readable CSP.

We begin by defining the set of values associated with expressions in a script. The set *Value* contains the union of the following.

- Basic values: integers, booleans, and characters.
- Datatype constructors and channel names.
- Dotted tuples $v_1 . \ldots . v_n$: in such values, we assume each component v_i is not a dotted term; i.e. we treat the dot operator as associative, and "flatten" nested dots. We sometimes write ε for the empty dotted tuple, i.e. the unit of dot.
- Sequences $\langle v_1, \ldots, v_n \rangle$, sets $\{v_1, \ldots, v_n\}$, tuples $(v_1, \ldots, v_n)$ and mappings $\{v_1 \mapsto v_1', \ldots, v_n \mapsto v_n'\}$, where each v_i, v_i' is a value.

- Functions, considered as sets of maplets $(v \mapsto v')$; we sometimes represent functions using lambda abstractions.

- Augmented LTSs and GLTSs (defined below), representing processes.

- Distinguished values *error* representing an error, and $\perp$ representing a non-terminating computation.

Further, when we evaluate expressions, we will do so in an environment that stores the values of identifiers, and also stores information concerning channel and datatype declaration. We therefore define *EValue* to be the union of *Value*, and values of the following forms.

- For each channel declaration **channel** $c : e_1 . \ldots . e_n$, a channel value *channel* S, where S is the set of values that can be passed on c, i.e. all values of the form $v_1 . \ldots . v_n$ associated with $e_1 . \ldots . e_n$. In the case $n = 0$, we use the value $channel\{\varepsilon\}$.

- For each datatype declaration

  $$\textbf{datatype}\, D = A_1.e_{1,1}.\ldots.e_{1,m_1} | \ldots | A_n.e_{n,1}.\ldots.e_{n,m_n}$$

  datatype constructor values *dtcons* S_i corresponding to A_i, for $i = 1, \ldots, n$, where S_i is the set of values of the form $v_1 . \ldots . v_{m_i}$ associated with $e_{i,1} . \ldots . e_{i,m_i}$. In the case $m_i = 0$, we use the value $dtcons\{\varepsilon\}$.

(Note that the sets of values may be infinite, because datatype declarations may be recursive. FDR therefore uses an efficient representation. However, there is no need to model that in our mathematical description.)

We use an *environment* mapping identifiers (variables) to values:

$$Env = Var \nrightarrow EValue.$$

We write ρ, ρ', etc., for environments.

The type *Expr* represents expressions. The semantics of expressions is defined using a function $\mathsf{eval} : Env \rightarrow Expr \nrightarrow Value$ such that $\mathsf{eval}\, \rho\, e$ gives the value of expression e in environment ρ.

For later convenience, we define the semantics of processes in terms of *augmented GLTSs*, where, informally, states are labelled with the corresponding syntactic expression and environment. In particular, each component GLTS within a supercombinator will have states labelled in this way. This will allow us to apply a permutation π to a state of a component GLTS by applying π to the environment; Lemma 35 then shows how to apply π to the state of the supercombinator.

We define a *simple label* to be a pair (P, ρ) where P is a syntactic expression[1] and ρ is an environment. In order to apply a permutation π to such a state, we will apply π to each variable in the environment. However, in some cases we need a more complex representation of an environment than a simple mapping as described above, in particular to deal with two variables with the same name. For example, consider

$$P \mid\mid\mid P \quad \text{where} \quad P = in?x \to out!x \to STOP.$$

Here, after two initial communications on *in*, both component processes have a variable x, possibly holding different values. We need to distinguish these two variables.

In the interests of efficiency, we perform an alpha-renaming on variables of the script, so that no two distinct variables in the script have the same name. However, this does not solve the issue with P, above, since both instances of x correspond to the same instance in the script. Instead, we label the state after two *in* events with a label such as

$$(out!x \to STOP, \{x \mapsto T_1\}) \mid\mid\mid (out!x \to STOP, \{x \mapsto T_2\})$$

giving the states of both components.

We use similar compound labels for other binary operators. However, for a parallel composition, it is also necessary to store the environment in which the alphabets are evaluated. Likewise, we include a compound label for the unary operators of hiding and renaming, storing the environment in which the hidden set or renaming relation is evaluated.

For symmetric replicated operators, each label contains a multiset of sub-labels, one for each component; for example, a replicated interleaving giving n component processes could have a label of the form $\Big|\Big|\Big|\{(P_1, \rho_1), \ldots, (P_n, \rho_n)\}$. For non-symmetric replicated operators, each label contains a sequence of sub-labels. For replicated generalised parallel, we also store the environment in which the synchronization set is evaluated. For replicated alphabetised parallel, we pair each component label with the environment in which the alphabet for that component is evaluated.

FDR can apply various compressions to GLTSs, such that each state of the resulting GLTS represents a *set* of states of the corresponding CSP process. In such a case, we therefore label each state with a *set* of labels, together with the name of the compression function applied.

We therefore define a type of labels with the following syntax (we write

[1] In the implementation, each syntactic expression is represented by a distinct integer.

"**M**" for a multiset type constructor).

$$
\begin{array}{lll}
\mathit{Lbl} ::= & (\mathit{Exp}, \mathit{Env}) & \text{simple labels} \\
 & |\mathit{Lbl} \oplus \mathit{Lbl} & \text{binary operators, } \oplus = \Box, |||, \triangleright, \triangle, ; \\
 & |\mathit{Lbl} \oplus_{\mathit{Env}} \mathit{Lbl} & \text{parallel operators } \oplus = [|A|], [A \parallel B], [l \leftrightarrow r] \\
 & |\mathit{Lbl} \sharp_{\mathit{Env}} & \text{hiding and renaming, } \sharp = \setminus X, [[R]] \\
 & | \bigoplus \mathbf{M} \mathit{Lbl} & \text{symmetric replicated operators, } \bigoplus = \Box, ||| \\
 & | \bigoplus \mathit{Lbl}^* & \text{non-symmetric replicated operators,} \\
 & & \bigoplus = \triangleright,\ \triangle, ; \\
 & | \big\|_{A\ \mathit{Env}} \ \mathbf{M} \mathit{Lbl} & \text{replicated generalised parallel} \\
 & | \big\| [A]\, \mathbf{M}(\mathit{Lbl}, \mathit{Env}) & \text{replicated alphabetised parallel} \\
 & |[r \leftrightarrow l]\, \mathit{Lbl}^*\, \mathit{Env}^* & \text{replicated link parallel} \\
 & |c\, (\mathbf{P}\, \mathit{Lbl}) & \text{compressions; } c \text{ is a compression function.}
\end{array}
$$

In the interests of efficiency, we simplify labels as far as possible: we use compound labels only when necessary.

Processes will be represented by an augmented GLTS.

Definition 42. An *augmented GLTS* is a GLTS where each state is given a label of type *Lbl*.

The semantics of a state with a simple label (Q, ρ) is equal to $\mathsf{eval}\, \rho\, Q$. The semantics of states with other labels are analogous. For brevity, we will sometimes identify a state with its label.

We define a standard partial order $\sqsubseteq$ over *EValue*, with bottom element $\bot$, and such that all the datatype constructors are continuous.

Let π be a type-preserving permutation on $\mathcal{T}$. We extend π to other values in the obvious way.

- For values x not depending on $\mathcal{T}$ (including ε, $\bot$ and *error*) we have $\pi(x) = x$.

- We lift π to dotted tuples —including events— by $\pi(v_1 . \ldots . v_n) = \pi(v_1) . \ldots . \pi(v_n)$. This is well defined provided all channel and type declarations are closed under π (see below).

- We lift π to tuples, sets, sequences and maps by point-wise application.

- We lift π to functions, considered as sets of maplets, by $\pi(f) = \{\pi(x) \mapsto \pi(y) | x \mapsto y \in f\}$; equivalently, in terms of a lambda-abstraction, $\pi(f) = \lambda\, z \cdot \pi(f(\pi^{-1}(z)))$.

- For channel and datatype constructor values we define $\pi(channel\ S) = channel(\pi(S))$, and $(dtcons\ S) = dtcons(\pi(S))$.
- For convenience, we lift π to environments by functional composition, i.e. $\pi(\rho) = \pi \circ \rho$.
- We lift π to labels by point-wise application, so $\pi(Q, \rho) = (Q, \pi \circ \rho)$.
- We lift π to augmented LTSs by application of π to the events of the transitions, and to the labels of states.

  We similarly lift π to augmented GLTSs by application to the events of the transitions, to the labels of states, and also by application of π to the minimal acceptances. (Note that this is consistent with Lemma 3.)

  Note that if L is an augmented (G)LTS then $L \sim_\pi \pi(L)$.

Note that this lifting of π forms a bijection on *EValue*. Recall that we write $Sym(T_1, \ldots, T_N)$ for the group of all type-preserving permutations on $T_1, \ldots, T_N$. We write $VSym(T_1, \ldots, T_N)$ for the result of lifting all elements of $Sym(T_1, \ldots, T_N)$ to *EValue*. Recall that we write $EvSym(T_1, \ldots, T_N)$ for the subgroup of *EvSym* formed by lifting all elements of $Sym(T_1, \ldots, T_N)$ to Σ; this is equivalent to restricting elements of $VSym(T_1, \ldots, T_N)$ to Σ. We sometimes write $Sym(\mathcal{T})$, etc., for brevity.

The following definition captures our main assumption about the CSP script.

Definition 43. A CSP script is *constant-free* for $\mathcal{T}$ if

1. The only constants from $\mathcal{T}$ that appear are within the definition of the distinguished types constituting $\mathcal{T}$ itself.
2. The script makes no use of the built-in functions `seq`, `mapToList`, `mtransclose` or `show`.
3. The script makes no use of the compression functions `deter`, `chase` or `chase_nocache`.

Clearly, processes that use constants from $\mathcal{T}$ might not be symmetric: for example, $c!A \rightarrow STOP$, where $A \in \mathcal{T}$.

The functions listed in item 2 can be used to introduce constants from $\mathcal{T}$, and so can break symmetry. For example, `seq(S)` converts `S` into a sequence (in an implementation-dependent way), so `x = head(seq(T))` (where T is a distinguished subtype) effectively sets `x` to be a constant from `T`. In practice, FDR allows these functions to be used (a) on arguments of a concrete

(i.e. non-polymorphic) type not involving $\mathcal{T}$, or (b) within a subexpression of the form $\mathsf{error}(e)$.

The compression functions in item 3 prune an LTS by removing transitions according to certain rules (but in an implementation-dependent way). Thus they can also break symmetry. For example, each of them could convert the LTS corresponding to $\bigsqcap_{x:T} c!x \rightarrow STOP$ into the LTS corresponding to $c!A \rightarrow STOP$, for an arbitrary $A \in T$.

The following definition captures that the set of values associated with a channel or a datatype constructor is closed under π.

Definition 44. We say that an environment ρ *respects* π if the values associated with each channel name or datatype constructor is closed under π, and each element of the distinguished type is appropriately defined. i.e.:

- If $\rho(c) = \mathit{channel}\, S$ then $\pi(S) = S$;
- If $\rho(A) = \mathit{dtcons}\, S$ then $\pi(S) = S$;
- Either for each $A \in \mathcal{T}$, $\rho(A) = \perp$, or for each A in $\mathcal{T}$, $\rho(A) = \mathit{dtcons}\{\varepsilon\}$ (the former disjunct holds only in the initial environment).

Channel and datatype declarations may appear only as top-level declarations. Below, we will show that the environment ρ_1 formed from these declarations respects π. Hence each subsequent environment also respects π.

The following is the main result of this section.

Proposition 45. Suppose a script is constant-free for $\mathcal{T}$, and let π be a type-preserving permutation on $\mathcal{T}$.

1. Let ρ_1 be the environment resulting from binding the top-level declarations. Then ρ_1 respects π. This means that each subsequent environment also respects π.
2. For every expression e in the script, and every environment ρ that respects π,

$$\pi(\mathsf{eval}\,\rho\, e) \quad = \quad \mathsf{eval}(\pi \circ \rho) e.$$

The proof is in Appendix A.

7 Identifying symmetries in supercombinators

We now consider how we may apply an event permutation $\pi \in \mathit{EvSym}(\mathcal{T})$ to a state of the supercombinator $\mathcal{S}_{impl}$ for the implementation, as is required during model checking. In order to do this, we require that $\mathcal{S}_{impl}$ is recursively π-mappable to itself, for every $\pi \in \mathit{EvSym}(\mathcal{T})$. We explain how we check this in Section 7.2. We also describe pre-calculations that help in subsequent calculations of component bijections and format bisimulations. We explain how to actually apply an event permutation to a state of a supercombinator in Section 7.3. Before that, in Section 7.1, we explain how to apply a permutation to a state of a leaf component to obtain a state of (possibly) another leaf component.

Throughout this section we assume a constant-free script, and an error-free refinement check on that script.

When we construct the supercombinator $\mathcal{S}_{impl}$ for the implementation, we store the initial control state P and environment ρ for each component GLTS. If a component is a nested supercombinator, we likewise store the initial control state and environment for each of its components, and so on recursively.

7.1 Relating leaf components

In this section we describe how, given leaf components L and L' such that $L' = \pi(L)$, to apply π to a state of L to obtain a state of L'.

Lemma 46. Suppose L and L' are leaf components with $L' = \pi(L)$. Then for each state s of L, there is a state s' of L' such that $s \sim_\pi s'$. We write $\pi(s)$ for this state s'.

Proof. (sketch.) If L and L' are uncompressed leaf components, and s has label (Q, ρ), then taking s' to be the state with label $(Q, \pi \circ \rho)$ satisfies the conditions of the lemma.

If L and L' are compressed leaf components, then the state s'' with label $(Q, \pi \circ \rho)$ might not exist in L', because the compression has merged it with another state s'. However, s' will be strongly bisimilar to s'', and so satisfy the conditions of the lemma. □

We can apply the above lemma as follows. Internally to FDR, each component state is represented by an integer index, with the label of each state stored separately. For each component, we pre-calculate a mapping from labels to the corresponding state index. To find the state $s' = \pi(s)$ from

Lemma 46, we obtain the label (Q, ρ) of s, calculate the corresponding label $(Q, \pi \circ \rho)$, and find the index of the corresponding state using the above mapping. If we are dealing with a compressed component, we then find the state that this has been merged with.

7.2 Checking recursive mappability

We explain here how we verify that the supercombinator for the implementation process is recursively π-mappable to itself for every $\pi \in \mathit{EvSym}(\mathcal{T})$; by Proposition 38, this will mean that it is $\mathit{EvSym}(\mathcal{T})$-symmetric. Assuming a constant-free script, FDR will nearly always produce such a supercombinator. However, FDR uses various heuristics to decide how to construct supercombinators, which makes it infeasible to directly verify this. We are also aware of a corner-case where this is not (currently) the case (cf. footnote 2). We therefore verify it directly for each supercombinator generated. If this turns out not to be the case, our approach fails, and we give up (but we have never known this to happen on a non-contrived example).

We also describe some pre-calculations we make concerning component bijections and format bisimulations, for use when exploring the product automaton. Note that we want to avoid pre-calculating and storing *all* such component bijections and format bisimulations, since there are simply too many; instead, we calculate enough information for them to be calculated efficiently subsequently.

We attempt to prove that $\mathcal{S}_{impl}$ is recursively π-mappable to itself, for every permutation π of the distinguished types. However, by Lemma 40, it suffices to consider just permutations π from a set of generators of the full symmetry group. Note, though, that if $\mathcal{S}_{impl}$ contains nested supercombinators, we will need to show that one component supercombinator $\mathcal{S}$ is recursively π-mappable to another $\mathcal{S}'$, so we consider this more general problem.

So consider two supercombinators

$$\begin{array}{rcll} \mathcal{S} & = & (\mathcal{L}, F, \mathcal{R}, on, f_0), & \text{with } \mathcal{L} = \langle L_1, \ldots, L_n \rangle, \\ \mathcal{S}' & = & (\mathcal{L}', F', \mathcal{R}', on', f_0'), & \text{with } \mathcal{L}' = \langle L_1', \ldots, L_n' \rangle, \end{array}$$

and consider the problem of showing that $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$. We split this into two parts: showing that the components are π-mappable; and constructing a format π-bisimulation.

Suppose each component L_i has label (P_i, ρ_i) and each L_i' has label (P_i', ρ_i'), so $L_i = \mathsf{eval}\, \rho_i\, P_i$ and $L_i' = \mathsf{eval}\, \rho_i'\, P_i'$. Let

$$\mathcal{P} = \{(P_1, \rho_1), \ldots, (P_n, \rho_n)\} \quad \text{and} \quad \mathcal{P}' = \{(P_1', \rho_1'), \ldots, (P_n', \rho_n')\}.$$

We try to build a bijection α_π from $\{1,\ldots,n\}$ to itself such that $P'_{\alpha_\pi(i)} = P_i$ and $\rho'_{\alpha_\pi(i)} = \pi \circ \rho_i$ for each i. We say that $\mathcal{P}$ is π-mappable to $\mathcal{P}'$ if this holds. Proposition 45 will then tell us that $L'_{\alpha_\pi(i)} = \pi(L_i)$, so $L_i \sim_\pi L'_{\alpha_\pi(i)}$, for each i.

Note, however, that the labels in $\mathcal{P}$ or $\mathcal{P}'$ might not be distinct: we might be dealing with multisets rather than sets. In order to deal with such cases, we add a fresh dummy variable inst to each environment. Suppose there are k copies of a particular label (P,ρ) in $\mathcal{P}$. We extend each of these environments by mapping inst to distinct integers in the range $\{1,\ldots,k\}$, thereby distinguishing these labels. We do likewise with $\mathcal{P}'$. It is clear that if $\mathcal{P}$ is indeed π-mappable to $\mathcal{P}'$ then $\mathcal{P}$ will contain the same number k of copies of (P,ρ) as $\mathcal{P}'$ contains of $(P,\pi\circ\rho)$; hence the extensions of ρ and ρ' will use the same values $\{1,\ldots,k\}$ for inst. We will define α_π to relate indices of environments that have the same value for inst.

We calculate (and store for later use) a mapping $m_{\mathcal{P}'}$ from labels in $\mathcal{P}'$ to the corresponding index:

$$m_{\mathcal{P}'} \;=\; \{(P'_i,\rho'_i) \mapsto i \mid i \in \{1,\ldots,n\}\}.$$

Note that $m_{\mathcal{P}'}$ is indeed a mapping, because of the use of the inst variables. We then test whether $\mathcal{P}$ is indeed π-mappable to $\mathcal{P}'$. For each $i \in \{1,\ldots,n\}$, we calculate $(P_i,\pi\circ\rho_i)$ and check that it is in the domain of $m_{\mathcal{P}'}$; if so, we define $\alpha_\pi(i)$ to be the index it maps to, so $P'_{\alpha_\pi(i)} = P_i$ and $\rho'_{\alpha_\pi(i)} = \pi\circ\rho_i$. If this check fails, then the supercombinator produced by FDR is not symmetric and our approach fails. If the check succeeds for every i, then $\mathcal{P}$ is π-mappable to $\mathcal{P}'$. In this case, Proposition 45 tells us that $L_i \sim_\pi L'_{\alpha_\pi(i)}$ for each i, as required.[2].

Following Definition 37, we then check that either (a) L_i and $L'_{\alpha_\pi(i)}$ are both leaf GLTSs, or (b) both are nested supercombinators; in the latter case, we then check (recursively) that L_i is recursively π-mappable to $L'_{\alpha_\pi(i)}$.

We now consider format π-bisimulations. We can calculate the maximal format π-bisimulation between $\mathcal{S}$ and $\mathcal{S}'$ using a straightforward adaptation of the algorithm for calculating a strong bisimulation. Given a relation $\sim^F \subseteq$

[2]There are advantages in also including, in the domain of $m'_{\mathcal{P}}$, the position in the abstract syntax tree of the relevant subprocesses. For example, consider $|||_{t:T} P(t)$ where $P(t) = \mathit{compress}(Q(t) \; ||| \; (Q(t) \setminus X))$, for some processes $Q(t)$, set X, and compression function *compress*. Suppose each $P(t)$ is implemented as a nested supercombinator. Our current approach might define α_π, from the components of the supercombinator for $P(t)$ to those for $P(\pi(t))$, so that the instance of $Q(t)$ without the hiding maps onto the instance of $Q(\pi(t))$ with the hiding. This would be incorrect since the supercombinator rules treat them differently. We leave this for future work.

$F \times F'$ over formats, define $\mathcal{F}(\sim^F)$ to contain all pairs (f, f') satisfying the defining conditions for a format π-bisimulation, i.e.

- if $(e, a, r, f_1) \in \mathcal{R}(f)$ then there is a rule $(e', \pi(a), \alpha(r), f_1') \in \mathcal{R}'(f')$ such that $\forall\, i \in \{1, \ldots, n\} \cdot e'(\alpha(i)) = \pi(e(i))$ and $f_1 \sim^F f_1'$;
- if $(e', a, r, f_1') \in \mathcal{R}'(f')$ then there is a rule $(e, \pi^{-1}(a), \alpha^{-1}(r), f_1) \in \mathcal{R}(f)$ such that $\forall\, i \in \{1, \ldots, n\} \cdot e'(\alpha(i)) = \pi(e(i))$ and $f_1 \sim^F f_1'$;
- $\alpha(on(f)) = on'(f')$.

Then we calculate the greatest fixed point of $\mathcal{F}$: let $\sim^F_{\pi,0} = F \times F'$ be the universal relation over formats; calculate $\mathcal{F}(\sim^F_{\pi,0}), \mathcal{F}^2(\sim^F_{\pi,0}), \ldots$, until a fixed point $\sim^F_\pi$ is reached. We then check that the initial formats are related, i.e. $f_0 \sim^F_\pi f_0'$. If this succeeds, then $\mathcal{S}$ and $\mathcal{S}'$ are recursively π-mappable. (We store the format bisimulation found, for later use.)

Recall, that we apply the above procedure to show that the supercombinator for the implementation, $\mathcal{S}_{impl}$, is recursively π-mappable to itself, for every π in a set of generators of the full symmetry group. By Lemma 40, this tells us that $\mathcal{S}_{impl}$ is recursively π-mappable to itself for every event permutation π.

This means that for each event permutation π there is a bijection α_π on the components of $\mathcal{S}_{impl}$ giving corresponding components. Inductively, for every event permutation π, and for every nested supercombinator $\mathcal{S}$ (nested at an arbitrary depth), there is a nested supercombinator $\mathcal{S}'$ such that $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$. If π can be written in terms of generators as $\pi = \pi_1 ; \ldots ; \pi_n$ then there are supercombinators $\mathcal{S}_0 = \mathcal{S}, \mathcal{S}_1, \ldots, \mathcal{S}_n = \mathcal{S}'$ such that for each i, $\mathcal{S}_{i-1}$ is recursively π_i-mappable to $\mathcal{S}_i$ using some component bijection $\alpha^{\mathcal{S}_{i-1}, \mathcal{S}_i}_{\pi_i}$. Then the component bijection between the components of $\mathcal{S}$ and $\mathcal{S}'$ equals

$$\alpha^{\mathcal{S}, \mathcal{S}'}_\pi = \alpha^{\mathcal{S}_0, \mathcal{S}_1}_{\pi_1} ; \ldots ; \alpha^{\mathcal{S}_{n-1}, \mathcal{S}_n}_{\pi_n},$$

again by Lemma 40. Note that this corresponds to the homomorphism property (cf. Definition 41). In the next section we therefore seek to extend the definition of α, from that defined above (i.e. on generators π, between supercombinators shown to be recursively π-mappable), so that it is homomorphic.

7.3 Applying permutations to states

Suppose $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$. We now explain how to apply permutation π to a state of $\mathcal{S}$ to produce a state of $\mathcal{S}'$.

We start with the bijection $\alpha_\pi^{\mathcal{S},\mathcal{S}'}$ between the components of $\mathcal{S}$ and $\mathcal{S}'$. As noted above, it suffices for it to agree with the values for generators from the previous subsection and for it to be homomorphic. We extend that previous definition in the obvious way.

Let the initial labels of the components of $\mathcal{S}$ and $\mathcal{S}'$ be $\mathcal{P} = \{(P_1, \rho_1), \ldots, (P_n, \rho_n)\}$ and $\mathcal{P}' = \{(P'_1, \rho'_1), \ldots, (P'_n, \rho'_n)\}$. Let $m_{\mathcal{P}'}$ be the mapping from the labels in $\mathcal{P}'$ to the corresponding indices, as in the previous subsection. Then we can calculate

$$\alpha_\pi^{\mathcal{S},\mathcal{S}'}(i) \quad = \quad m_{\mathcal{P}'}(P_i, \pi \circ \rho_i).$$

Then $P'_{\alpha_\pi^{\mathcal{S},\mathcal{S}'}(i)} = P_i$ and $\rho'_{\alpha_\pi^{\mathcal{S},\mathcal{S}'}(i)} = \pi \circ \rho_i$, as required. This clearly agrees with the α_π from the previous subsection for generators π. We show that the α mappings are homomorphic.

Lemma 47. For the above definition of α:

$$\alpha_\pi^{\mathcal{S},\mathcal{S}'} \,;\, \alpha_{\pi'}^{\mathcal{S}',\mathcal{S}''} \quad = \quad \alpha_{\pi;\pi'}^{\mathcal{S},\mathcal{S}''}.$$

Proof. Consider labels (P, ρ_i), (P, ρ'_j) and (P, ρ''_k) corresponding to the states of the ith, jth and kth components of $\mathcal{S}$, $\mathcal{S}'$ and $\mathcal{S}''$, respectively. Suppose $\pi \circ \rho_i = \rho'_j$ and $\pi' \circ \rho'_j = \rho''_k$. So, from the above definition, $\alpha_\pi^{\mathcal{S},\mathcal{S}'}(i) = j$ and $\alpha_{\pi'}^{\mathcal{S}',\mathcal{S}''}(j) = k$. Hence $(\alpha_\pi^{\mathcal{S},\mathcal{S}'} \,;\, \alpha_{\pi'}^{\mathcal{S}',\mathcal{S}''})(i) = k$. But $(\pi \,;\, \pi') \circ \rho_i = \rho''_k$, so, from the above definition, $\alpha_{\pi;\pi'}^{\mathcal{S},\mathcal{S}''}(i) = k$. This gives the result. □

We now consider format bisimulations. Recall, from the previous subsection, that we pre-calculate and store format bisimulations for the generators of the permutation groups. From these, we can efficiently calculate arbitrary format bisimulations during the exploration of the state space. If π can be written in terms of generators as $\pi = \pi_1 \,;\, \ldots \,;\, \pi_n$ then there are supercombinators $\mathcal{S}_0 = \mathcal{S}, \mathcal{S}_1, \ldots \mathcal{S}_n = \mathcal{S}'$ such that for each i, $\mathcal{S}_{i-1}$ is recursively π_i-mappable to $\mathcal{S}_i$ using some component bijection $\alpha_{\pi_i}^{\mathcal{S}_{i-1},\mathcal{S}_i}$ and format bisimulation $\sim_\pi^{F,\mathcal{S}_{i-1},\mathcal{S}_i}$. Then, by Lemma 39, the format bisimulation between $\mathcal{S}$ and $\mathcal{S}'$ is

$$\sim_\pi^{F,\mathcal{S},\mathcal{S}'} \quad = \quad \sim_{\pi_1}^{F,\mathcal{S}_0,\mathcal{S}_1} ; \ldots ; \sim_{\pi_n}^{F,\mathcal{S}_{n-1},\mathcal{S}_n}.$$

(In fact, in the interests of efficiency, we pre-calculate and store format bisimulations for all two-cycles, rather than a minimal set of generators.)

The following proposition shows how, given a state σ of a supercombinator and a permutation π, to *calculate* a state, which we denote $\pi(\sigma)$, such that $\sigma \sim_\pi \pi(\sigma)$.

Proposition 48. Let $\mathcal{S}$ and $\mathcal{S}'$ be supercombinators with components $\langle L_1, \ldots, L_n \rangle$ and $\langle L'_1, \ldots, L'_n \rangle$, and let π be an event permutation. Suppose $\mathcal{S}$ is recursively π-mappable to $\mathcal{S}'$ using component bijection α and format π-bisimulation $\sim^F_\pi$. Consider the state

$$\sigma = (s_1, \ldots, s_n, f).$$

Define $\pi(\sigma)$ to be the state $(s'_1, \ldots, s'_n, f')$, where

- if L'_j is a leaf component, then $s'_j = \pi(s_{\alpha^{-1}(j)})$, constructed as described in Section 7.1;
- if L'_j is a nested supercombinator, then $s'_j = \pi(s_{\alpha^{-1}(j)})$, defined recursively;
- $f \sim^F_\pi f'$.

Then $\sigma \sim_\pi \pi(\sigma)$.

Proof. The proof is by induction on the depth of the tree of nested supercombinators.

If L'_j is an explicit augmented GLTS, then so is $L_{\alpha^{-1}(j)}$, and $L_{\alpha^{-1}(j)} \sim_\pi L'_j$, by the definition of recursive π-mappable. Then the existence of s'_j follows from Lemma 46, and $s_{\alpha^{-1}(j)} \sim_\pi s'_j$.

If L'_j is a nested supercombinator, then so is $L_{\alpha^{-1}(j)}$, and $L_{\alpha^{-1}(j)}$ is recursively π-mappable to L'_j. Then by the inductive hypothesis, L'_j has a state $s'_j = \pi(s_{\alpha^{-1}(j)})$ such that $s_{\alpha^{-1}(j)} \sim_\pi s'_j$.

In each case, $s_i \sim_\pi s'_{\alpha(i)}$, for each i. Hence $\sigma \sim_\pi \pi(\sigma)$ by Lemma 35. □

8 Calculating representatives

In this section, we describe an algorithm for calculating representatives (cf. Definition 17). Recall that we do not require unique representatives; however, we will aim for this to be the case in most examples, so as to give a better reduction in the state space.

Finding unique representatives is believed to be difficult in general. Clarke et al. [CEJS98] show that it is at least as hard as the graph isomorphism problem, which is widely accepted as being difficult (although not widely believed to be NP-complete).

We concentrate, below, on variables whose type is precisely that of some distinguished supertype $\hat{T}_i$; we call these *distinguished variables*. We ignore variables that hold tuples, sequences, sets, etc., that depend upon $\hat{T}_i$. We

describe in Section 8.4 techniques for extending our algorithms to include these (although we do not currently do so).

Recall that a state of a supercombinator is a tuple $(s_1, \ldots, s_n, f)$ where each s_i is a state of a component GLTS with a label of type Lbl, and f is a format. If the supercombinator has a nested supercombinator as a component, we include the states of that nested supercombinator; this is equivalent to "flattening" the top-level supercombinator state. We then pick an ordering $\langle s_{i_1}, \ldots, s_{i_n} \rangle$ of the component GLTS states, so that, as far as possible, two π-bisimilar supercombinator states map onto π-related orderings[3]. From this, we will later extract a particular ordering for each distinguished type, and hence a permutation; again, this will be done so that, as far as possible, two π-bisimilar states map onto the same final state.

Some parts of the algorithms below are left under-specified: the implementer may choose to implement them in one of several ways. We flag these as *implementation-dependent*, but suggest a particular implementation. The implementation-dependent parts we suggest, if applied consistently, will mostly ensure that two π-bisimilar states are treated equivalently. However, there will be a couple of places where this is not possible, and the algorithm will have to make an *arbitrary* decision (e.g. a random choice).

8.1 Component ordering algorithm

We describe the component ordering algorithm, which takes a multiset of LTS states S (the states of component LTSs of a supercombinator) and returns an ordering of S.

Given a label of a component state, we can extract a corresponding control state. For a simple label (P, ρ), the control state is simply P. For a label $l \oplus l'$, the control state is $P \oplus P'$ where P and P' are the control states of l and l', respectively. Other cases are very similar.

We assume some way of ordering the variables of a component state. For states with simple labels, we can just order the variables lexicographically. For most non-simple labels, we can just take the variables of the components in component order. For labels that contain a set or multiset of sub-labels, we order the sub-labels in each (multi)set by their control states, the values of non-distinguished variables, and the values of distinguished variables (lexicographically), and then extract the variables in order. We write $s(v_i)$ for the value of the ith distinguished variable in state s.

We build a multigraph model of the state: each node of the graph is a

[3]We use the word "ordering" rather than "permutation" here, to avoid over-loading the latter word.

component state; there is an edge labelled (i, k) from s_1 to s_2 if $s_1(v_i) = s_2(v_k)$, where i and k range over indices of distinguished variables of s_1 and s_2 with the same type. In the running example, considering just `Node` processes, there would be a $(3, 1)$ edge from n_1 to n_2 if n_1's `next` field points to n_2 (i.e. equals n_2's `me` field); this corresponds to how linked lists are typically depicted. Step 2 of the algorithm below is an adaptation of the naive refinement algorithm of [Bab95, Section 6.4] to this model.

Definition 49. The *component ordering algorithm* takes a multiset of LTS states S (the states of component LTSs of a supercombinator) and returns an ordering of S.

The algorithm maintain a sequence of multisets of component states $\langle S_1, \ldots, S_m \rangle$, such that $S_1, \ldots, S_m$ partition S; we call this an *ordered partition*. For each j, all states in S_j will have the same control state, and hence the same number of variables of distinguished types; further, all states in S_j will be from the same symmetric multiset. We refine the ordered partition in steps, until we obtain a sequence of singleton multisets.

1. Start by partitioning states into their symmetric multisets, by their control states, and the values of variables of non-distinguished types. Order these multisets in some implementation-dependent way (e.g. lexicographically on symmetric multisets, control states and the values of non-distinguished variables); where a variable may take either a value of a distinguished subtype or a different value from the containing supertype, order the states in some implementation-dependent way (e.g. states with distinguished values first).

2. Given an ordered partition $\langle S_1, \ldots, S_m \rangle$, we split each S_j based on matches between distinguished variables of each state s and distinguished variables of states in each element of the partition. Define $n_{i,j,k}(s)$ to be the number of (i, k) edges from s to elements of S_j:

   $$n_{i,j,k}(s) \quad = \quad \#\{s' \in S_j | s'(v_k) = s(v_i)\},$$

   where i ranges over indices of distinguished variables of s, j ranges over $\{1, \ldots, m\}$, and k ranges over indices of distinguished variables of S_j with the same type as v_i.

   Then let $n(s)$ be the tuple formed from the $n_{i,j,k}(s)$ (in lexicographic order of indices). If $n(s_1) = n(s_2)$ then the two states have the same relationship to states in other partitions.

   For each j, partition and order S_j according to the states' n-values. This produces a refined ordered partition.

Repeat this step until no more progress is made.

3. If a multiset in the partition is not a singleton, then, in some implementation-dependent way, pick a non-singleton multiset, and split it into two or more multisets, ordered in some way. For example, pick the first non-singleton multiset, say S_j, pick an arbitrary element $s \in S_j$, and split S_j into $\langle \{s\}, S_j \setminus \{s\} \rangle$. Then return to step 2.

   Otherwise all multisets are singletons, so return the corresponding sequence of states.

Example 50. We apply the above algorithm to the node processes of the running example, using the suggested implementation-dependent approaches.

We introduce suggestive notation: we write $Node(m, d, n)$ for the state $(node, \{\mathsf{me} \mapsto m, \mathsf{datum} \mapsto d, \mathsf{next} \mapsto n\})$, where $node$ is the control state corresponding to the right-hand side of the definition of Node; we write $FreeNode(m)$, similarly.

Consider the multiset of processes

$$\{ \; Node(N_0, B, N_4),\; Node(N_1, B, N_0),\; FreeNode(N_2),\\ Node(N_3, B, N_1),\; Node(N_4, B, Null),\; FreeNode(N_5) \; \}.$$

This represents a linked list of four nodes (N_3, N_1, N_0, N_4), all containing B, together with two free nodes (N_2, N_5).

Step 1 then partitions the processes as follows, assuming the control state for $FreeNode$ is less than that for $Node$:

$$\langle \; \{FreeNode(N_2),\; FreeNode(N_5)\},\\ \{Node(N_0, B, N_4),\; Node(N_1, B, N_0),\; Node(N_3, B, N_1)\},\\ \{Node(N_4, B, Null)\} \; \rangle.$$

For step 2, the n-value for $Node(N_0, B, N_4)$ is $\langle 0, 1, 1, 0\, ; 3, 1\, ; 0, 0, 1, 1 \rangle$ (semicolons used to improve readability). The first four entries correspond to the value N_0, counting the number of matches against variables in the first multiset, the me and next variables in the second multiset, and the me variables in the third multiset, respectively; the next two entries correspond to the value B; the last four entries correspond to N_4. The n-values for $Node(N_1, B, N_0)$ and $Node(N_3, B, N_1)$ are $\langle 0, 1, 1, 0\; ; 3, 1\; ; 0, 1, 1, 0 \rangle$ and $\langle 0, 1, 0, 0\; ; 3, 1\; ; 0, 1, 1, 0 \rangle$, respectively. However, the n-values for the two $FreeNode$ processes are each $\langle 1, 0, 0, 0 \rangle$. Partitioning according to these n-values gives the following ordered partition:

$$\langle \; \{FreeNode(N_2),\; FreeNode(N_5)\},\\ \{Node(N_3, B, N_1)\},\; \{Node(N_0, B, N_4)\},\; \{Node(N_1, B, N_0)\},\\ \{Node(N_4, B, Null)\} \; \rangle.$$

Repeating step 2 makes no further progress. (Note that if we had started with a longer linked list, we would have needed more iterations of step 2: step 1 splits off the final node; the first iteration of step 2 splits off the first and penultimate nodes; the second iteration would split off the second and antepenultimate nodes; and so on.)

Finally, step 3 splits the non-singleton multiset arbitrarily, say producing

$$\langle \{FreeNode(N_2)\},\ \{FreeNode(N_5)\},\ \{Node(N_3, B, N_1)\},\ \{Node(N_0, B, N_4)\},\ \{Node(N_1, B, N_0)\},\ \{Node(N_4, B, Null)\} \rangle.$$

All multisets are now singleton, so the process is complete.

Note that if we had started with any other system representing a linked list of four nodes, all containing the same value, we would have ended up with an equivalent ordering, starting with the free nodes, ordered arbitrarily, followed by the first, third, second and fourth nodes of the list, in that order.

8.2 Permutation generation

We now describe how to go from an ordering on component states to an ordering of each distinguished type.

Definition 51. Let T be a distinguished subtype. The *type ordering algorithm* for T takes an ordering of component states, and returns an ordering of T, as follows: it lists the values of type T in the order they appear in the ordering of component states, removes duplicates, and (if necessary) appends remaining values of T in an arbitrary order.

Example 52. Recall that in Example 50 we obtained the ordering of states

$$\langle\ FreeNode(N_2),\ FreeNode(N_5), Node(N_3, B, N_1),\ Node(N_0, B, N_4),\ Node(N_1, B, N_0), Node(N_4, B, Null)\ \rangle.$$

Applying the type ordering algorithm can give

$$\langle N_2, N_5, N_3, N_1, N_0, N_4\rangle \quad \text{and} \quad \langle B, A, C, D\rangle,$$

where in the latter case the ordering of all except B is arbitrary. (This particular example contains a large amount of arbitrariness because the initial state contains so few values from *Data*.)

An alternative approach, which can be more efficient, is to identify when certain variables act as *identities* for processes. For example, the `me` variables

act as identity variables for the node processes. These variables take the same value throughout the process's execution, and the values of the type are in a one-one relationship with the processes. In this case, we can order the type by extracting the values of just those variables from the ordering of states. In the running example, this would give the ordering $\langle N_2, N_5, N_3, N_0, N_1, N_4 \rangle$ over NodeID.

The following algorithms produce a corresponding permutation π on the distinguished subtypes, and hence a representative.

Definition 53. The *permutation generating algorithm* proceeds as follows, given an ordering $\vec{s}$ of the component LTS states. For each distinguished subtype $T = \{A_1, \dots, A_N\}$, run the T-ordering algorithm to obtain an ordering $\langle A_{i_1}, \dots, A_{i_N} \rangle$ of T; then define the permutation function $\pi_T = \{A_{i_1} \mapsto A_1, \dots, A_{i_N} \mapsto A_N\}$. Let π be the union of the individual π_T.

The *representative generating algorithm*, given state σ, runs the component ordering algorithm on σ to obtain a component ordering $\vec{s}$, then runs the permutation generating algorithm on $\vec{s}$ to obtain a permutation π, and then returns $\pi(\sigma)$ (calculated as in Proposition 48).

Example 54. Based on the orderings from Example 52, we obtain the permutation functions

$$\begin{aligned} \pi_{NodeID} &= \{N_2 \mapsto N_0, N_5 \mapsto N_1, N_3 \mapsto N_2, \\ &\quad N_0 \mapsto N_3, N_1 \mapsto N_4, N_4 \mapsto N_5\}, \\ \pi_{Data} &= \{B \mapsto A, A \mapsto B, C \mapsto C, D \mapsto D\}. \end{aligned}$$

Let $\pi = \pi_{NodeID} \cup \pi_{Data}$. Applying π to the original vector of processes from Example 50, as in Proposition 48, we obtain the vector of processes

$$\begin{aligned} &\langle\ FreeNode(N_0),\ FreeNode(N_1),\ Node(N_2, A, N_4), \\ &\quad Node(N_3, A, N_5),\ Node(N_4, A, N_3),\ Node(N_5, A, Null)\ \rangle. \end{aligned}$$

In the generation of π, we made two arbitrary decisions. First, in step 3 of Example 50, we chose an arbitrary order for the two *FreeNode*s. Second, in Example 52, we chose an arbitrary order for all elements of Data except B. However, note that if we had made these decisions in any other way, we would have obtained the *same* final vector of processes, essentially because the initial vector is symmetric in $\{N_2, N_5\}$ and in $\{A, C, D\}$.

Note further, that if we had started with *any* vector of processes that was symmetric to the chosen initial vector —that is, representing a linked list of four nodes, all containing the same datum— then we would have ended up with the same final vector of processes. That is, the above vector is a *unique* representative for its equivalence class.

The following lemma relates permutations obtained from related vectors of processes.

Lemma 55. If the permutation generating algorithm applied to $\vec{s}$ gives π_1, then the permutation generation algorithm applied to $\pi(\vec{s})$ gives a function that agrees with $\pi_1 \circ \pi^{-1}$ on all distinguished values that appear in $\pi(\vec{s})$.

Proof. Suppose the T-ordering algorithm applied to $\vec{s}$ produces $\langle A_{i_1}, \ldots, A_{i_N} \rangle$, where $A_{i_1}, \ldots, A_{i_k}$ appear in $\vec{s}$, and the other values are ordered arbitrarily. Then $\pi(A_{i_1}), \ldots, \pi(A_{i_k})$ appear in the corresponding order in $\pi(\vec{s})$, so the T-ordering algorithm applied to $\pi(\vec{s})$ produces $\langle \pi(A_{i_1}), \ldots, \pi(A_{i_k}) \rangle$ followed by an arbitrary ordering of $\pi(A_{i_{k+1}}), \ldots, \pi(A_{i_N})$.

Then the permutation generating algorithm applied to $\vec{s}$ gives

$$\pi_1 = \{A_{i_1} \mapsto A_1, \ldots, A_{i_N} \mapsto A_N\}.$$

And the permutation generating algorithm applied to $\pi(\vec{s})$ gives a function that includes

$$\{\pi(A_{i_1}) \mapsto A_1, \ldots, \pi(A_{i_k}) \mapsto A_k\},$$

and maps $\pi(A_{i_{k+1}}), \ldots, \pi(A_{i_N})$ to an arbitrary permutation of $A_{k+1}, \ldots, A_N$. But the above partial function equals

$$\{\pi(A_{i_1}) \mapsto \pi_1(A_{i_1}), \ldots, \pi(A_{i_k}) \mapsto \pi_1(A_{i_k})\},$$

which agrees with $\pi_1 \circ \pi^{-1}$ on the distinguished values $\pi(A_{i_1}), \ldots, \pi(A_{i_k})$ that appear in $\pi(\vec{s})$. □

8.3 Uniqueness of representations

We now consider circumstances under which the above algorithms, with the specific implementation-dependent parts suggested, give unique representatives. The following example shows that this is not always the case.

Example 56. Now consider the set of processes

$$\{\ Node(N_0, A, N_1),\ Node(N_1, A, N_2),\ Node(N_2, B, N_3),\ Node(N_3, B, N_0)\ \}.$$

For simplicity, suppose NodeID $= \{N_0, N_1, N_2, N_3\}$ and Data $= \{A, B\}$. The above vector represents a circular linked list; such a state could not arise in our running example, but could in different linked-list algorithms.

Applying steps 1 and 2 of the component ordering algorithm fails to split the processes. We now perform a case analysis on the way that step 3 chooses to split the set.

- Suppose, first, that step 3 splits the partition into

$$\langle\ \{Node(N_0, A, N_1)\},\\ \{Node(N_1, A, N_2),\ Node(N_2, B, N_3),\ Node(N_3, B, N_0)\}\ \rangle.$$

  Another iteration of step 2 produces

$$\langle\ \{Node(N_0, A, N_1)\},\\ \{Node(N_2, B, N_3)\},\ \{Node(N_3, B, N_0)\},\ \{Node(N_1, A, N_3)\}\ \rangle.$$

  Applying the identity-based ordering algorithm to NodeID, and the default ordering algorithm to Data gives permutation

$$\pi\ =\ \{N_0 \mapsto N_0, N_1 \mapsto N_3, N_2 \mapsto N_1, N_3 \mapsto N_2,\ A \mapsto A, B \mapsto B\},$$

  and hence the final vector of processes

$$\langle\ Node(N_0, A, N_3),\ Node(N_1, B, N_2),\\ Node(N_2, B, N_0),\ Node(N_3, A, N_1)\ \rangle.$$

- Suppose, instead, that step 3 splits the partition into

$$\langle\ \{Node(N_1, A, N_2)\},\\ \{Node(N_0, A, N_1),\ Node(N_2, B, N_3),\ Node(N_3, B, N_0)\}\ \rangle.$$

  Another iteration of step 2 produces

$$\langle\ \{Node(N_1, A, N_2)\},\\ \{Node(N_3, B, N_0)\},\ \{Node(N_0, A, N_1)\},\ \{Node(N_2, B, N_3)\}\ \rangle.$$

  This gives the following permutation and vector of processes:

$$\pi = \{N_0 \mapsto N_2, N_1 \mapsto N_0, N_2 \mapsto N_3, N_3 \mapsto N_1\ A \mapsto A, B \mapsto B\},\\ \langle\ Node(N_0, A, N_3),\ Node(N_1, B, N_2),\\ Node(N_2, A, N_0),\ Node(N_3, B, N_1)\ \rangle.$$

- If step 3 splits the partition in any other way, one of the above two final vectors is obtained.

Similarly, any other cycle of four nodes, holding two different values arranged in adjacent pairs will produce one of these three final vectors. However, the choice of how to split at step 3 might be made differently on different such cycles. Thus our approach might not give *unique* representatives in this case. However, it does give a good reduction, from 24 cases to 2. Note, also, that the approach *does* give unique representatives for rings that contain different numbers of As and Bs, or where the data are in an alternating sequence such at $\langle A, B, A, B\rangle$ round the ring.

Note that the above example is somewhat unrealistic. In a more realistic setting, there would be an external pointer into the ring (comparable to Top in the running example), which would be enough to break the symmetry, and so avoid having to make arbitrary decisions.

The permutation generating algorithm is potentially nondeterministic, because of the arbitrary choices in step 3 of the component ordering algorithm and in the type ordering algorithm. The latter nondeterminism is clearly irrelevant, since it only affects the result of the final permutation on values that do not appear, and so does not affect the final state obtained. (Note that, while we would expect any implementation to be deterministic, we are interested here in nondeterminism of the specification.) We show that in the case that the arbitrary choices in the component ordering algorithm do not cause nondeterminism of the outcome —i.e., for each input state, the final state is independent of how those choices are made— the algorithm produces unique representatives.

Lemma 57. Suppose that the representative generating algorithm when run on input state σ always produces final state σ'. Then if the algorithm is run on input state $\pi(\sigma)$, it again always produce final state σ'.

Proof. Consider first the component ordering algorithm. It is easy to see that this algorithm treats $\pi(\sigma)$ and σ equivalently. More precisely, if the algorithm is run in parallel on these two states, for each intermediate ordered partition $\langle \pi(S_1), \ldots, \pi(S_m) \rangle$ obtained from $\pi(\sigma)$, it is also possible for $\langle S_1, \ldots, S_m \rangle$ to be obtained from σ.

- Step 1 partitions the component states equivalently, because corresponding component states in $\pi(\sigma)$ and σ have the same control states and values of non-distinguished variables, and are in the same symmetric sets.

- Step 2 refines the partitions equivalently, since the n-values of corresponding states agree.

- If step 3 chooses a particular state $\pi(s)$ to split off from $\pi(S_j)$, then it can also choose to split off state s from S_j.

Hence if the component ordering algorithm applied to $\pi(\sigma)$ can produce the ordering $\pi(\vec{s})$, then when applied to σ it can produce $\vec{s}$.

Now suppose the permutation generating function applied to $\vec{s}$ produces π_1. Then by Lemma 55, the permutation generating function applied to $\pi(\vec{s})$ gives a function that agrees with $\pi_1 \circ \pi^{-1}$ on all distinguished values that appear in $\pi(\sigma)$. Applying these permutations to the initial states,

we obtain $\pi_1(\sigma)$ in each case. By assumption, this value must equal σ', as required. □

Corollary 58. Suppose the representative generating algorithm is deterministic on all inputs. Then it returns unique representatives.

We now investigate conditions under which the representative generating algorithm is deterministic. We need a couple of technical lemmas.

Definition 59. We say that a supercombinator has *unique identifiers* if, for each non-singleton symmetric set, all processes have an identity variable, and no two processes in the same symmetric set have the same value for that identity variable.

Our running example satisfies this condition: the two non-singleton symmetric sets correspond to the Node and Thread processes; these have identities corresponding to NodeID and ThreadID, with distinct identities for different processes.

Lemma 60. Suppose a supercombinator has unique identifiers. Consider a state $\langle s_1, \ldots, s_n, f\rangle$ of the supercombinator, and suppose s_i and s_j are in the same symmetric set, and $s_j = \pi(s_i)$. Then $j = \alpha_\pi(i)$.

Proof. If the symmetric set is a singleton, the result is trivial. Otherwise, let id_i and id_j be the values of the identity variable in s_i and s_j, so $id_j = \pi(id_i)$. Then the identities in the initial states of these components are the same, and so are similarly related. Note that component $\alpha_\pi(i)$ has identity $\pi(id_i)$, by definition of α_π. No other component has that identity, by assumption. Hence $j = \alpha_\pi(i)$. □

Lemma 61. Suppose a supercombinator has unique identifiers. Consider states

$$\begin{array}{rcl} \sigma & = & \langle s_1, \ldots, s_n, f\rangle, \\ \sigma' & = & \langle s'_1, \ldots, s'_n, f'\rangle, \end{array}$$

and suppose $f \sim_\pi^F f'$ (using the format bisimulation of Section 7.3), and that for each symmetric set $\{i_1, \ldots, i_k\}$ we have

$$\{\pi(s_{i_1}), \ldots, \pi(s_{i_k})\} \quad = \quad \{s'_{i_1}, \ldots, s'_{i_k}\}.$$

Then $\sigma' = \pi(\sigma)$.

Proof. Suppose $\pi(s_{i_i}) = s'_{i_j}$ with i_i and i_j in the same symmetric set. Then by Lemma 60, $i_j = \alpha_\pi(i_i)$. So $s'_{\alpha_\pi(i)} = \pi(s_i)$, for each i. The result then follows from Proposition 48. □

The following definition and proposition identify a condition under which splitting a set at step 3 of the component ordering algorithm does not introduce nondeterminism.

Definition 62. Given an ordered partition $\langle S_1, \ldots, S_m \rangle$, we say that the component S_j is *fully symmetric* if for all $s, s' \in S_j$, there exists a permutation π_1 such that $\pi_1(s) = s'$ and for each i, $\pi_1(S_i) = S_i$.

In Example 50, the set $\{FreeNode(N_2), FreeNode(N_5)\}$ split by step 3 is fully symmetric. However, in Example 56, the set $\{Node(N_0, A, N_1),$ $Node(N_1, A, N_2), Node(N_2, B, N_3), Node(N_3, B, N_0)\}$ split by step 3 is not symmetric: for example, if π_1 is such that $\pi_1(Node(N_0, A, N_1)) =$ $Node(N_1, A, N_2)$, then $\pi_1(Node(N_3, B, N_0))$ is of the form $Node(n, B, N_1)$, for some n, which is not a member of the set.

Proposition 63. Suppose that a supercombinator has unique identifiers, and that for each format f and permutation π, $f \sim_\pi^F f$ (using the format bisimulation of Section 7.3). Suppose further that whenever we apply step 3 of the component ordering algorithm to split a set S_j, that set is fully symmetric.

Then if we apply the above representative generating algorithm to some state σ, the result is independent of the precise value split off by step 3; i.e. the algorithm is deterministic.

Hence the algorithm returns unique representatives.

Proof. Consider the effect of running the component ordering algorithm, starting from state

$$\sigma = (s_1, \ldots, s_n, f).$$

Consider two ordered partitions $\langle S_1, \ldots, S_m \rangle$ and $\langle S'_1, \ldots, S'_m \rangle$ that could be reached after the same number of steps, maybe corresponding to splitting off different elements at step 3 (it is clear that these partitions have the same number of elements). We show by induction that there is some permutation π such that $S'_i = \pi(S_i)$ for each i. It is clear that this is established by step 1 of the algorithm (with π the identity permutation), and maintained by each iteration of step 2 (since the n-values will agree). Consider instances of step 3, splitting off s from S_j and $\pi(s')$ from $\pi(S_j)$ where $s' \in S_j$; these produce

$$\langle S_1, \ldots, S_{j-1}, \{s\}, S_j - \{s\}, S_{j+1}, \ldots, S_m \rangle$$

and

$$\langle \pi(S_1), \ldots, \pi(S_{j-1}), \{\pi(s')\}, \pi(S_j) - \{\pi(s')\}, \pi(S_{j+1}), \ldots, \pi(S_m) \rangle.$$

Let π_1 be as in Definition 62, so $\pi_1(s) = s'$ and $\pi_1(S_i) = S_i$ for each i. Let $\pi' = \pi \circ \pi_1$. We show the resulting ordered partitions are related by π':

- $\pi(S_i) = \pi(\pi_1(S_i)) = \pi'(S_i)$ for each i;
- $\{\pi(s')\} = \{\pi(\pi_1(s))\} = \pi'(\{s\})$;
- $\pi(S_j) - \{\pi(s')\} = \pi(\pi_1(S_j)) - \{\pi(\pi_1(s))\} = \pi'(S_j - \{s\})$.

Hence any two final sequences of states produced by the component ordering algorithm will be of the form

$$\vec{s} = \langle s'_1, \ldots, s'_n \rangle \quad \text{and} \quad \pi(\vec{s}) = \langle \pi(s'_1), \ldots, \pi(s'_n) \rangle$$

for some permutation π. But each pair of corresponding states s'_i and $\pi(s'_i)$ are from the same symmetric state (since we start by partitioning by symmetric states). Thus, for each symmetric set $\{i_1, \ldots, i_k\}$, we have $\{\pi(s_{i_1}), \ldots, \pi(s_{i_k})\} = \{s_{i_1}, \ldots, s_{i_k}\}$. Hence, by Lemma 61 (with $\sigma' = \sigma$), $\pi(\sigma) = \sigma$.

Now, if the permutation generating function applied to the component ordering $\vec{s}$ gives π_1, then the permutation generating function applied to $\pi(\vec{s})$ gives $\pi_1 \circ \pi^{-1}$, by Lemma 55. Hence applying the two resulting permutations to the initial state σ we obtain $\pi_1(\sigma)$ and $(\pi_1 \circ \pi^{-1})(\sigma) = (\pi_1 \circ \pi^{-1})(\pi(\sigma)) = \pi_1(\sigma)$, i.e. the same value in each case, as required.

Hence the algorithm produces unique representatives, by Corollary 58. □

Lemma 64. Consider a system representing a reference-linked data structure, using Node processes parametrised as in the running example, i.e. every node is of the form Node(me, datum, next) or FreeNode(me) where me is an identity variable (and no other processes). Suppose further than in every state, the nodes are arranged in a single linked list, possibly with some free nodes: in other words, each node has at most one predecessor, so the next references form a single list, rather than a tree or forest. Then the algorithm produces unique representatives.

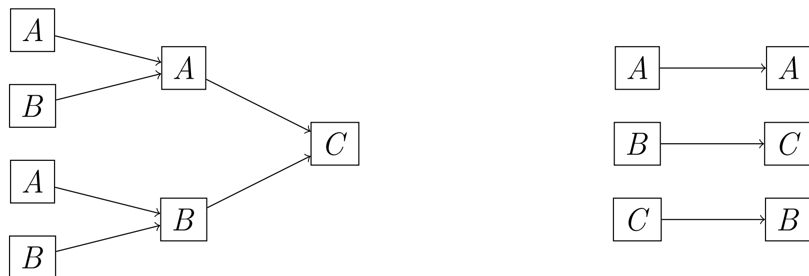
Proof. As in Example 50, each iteration of step 2 of the component ordering algorithm strips off two more nodes from the list. This continues until the only non-singleton set (if any) contains the free nodes. The system is then fully symmetric (since the identities of the free nodes appear nowhere else, and the free nodes contain no data; in fact, the result still holds if the free nodes hold a single piece of data). The result then follows from Proposition 63. □

The premise of the above lemma —that there are no processes other than the node processes— is unrealistic. However, in most circumstances

the result is still applicable. Suppose, in addition, we have some processes that represent threads operating on the linked list. For simplicity, suppose that step 3 of the component ordering algorithm is used to split sets of nodes before it is used to split sets of thread processes. In most cases, each thread process will have a reference to at least one node, which will then allow such sets to be split by an application of step 2. A common exception will be when several threads are in their initial state; but in this case the set of such threads will be fully symmetric, and so Proposition 63 can be applied to deduce that unique representatives are obtained. A somewhat contrived example where unique representatives are not obtained is with the set of processes $\{Thread'(T_1, A, A), Thread'(T_2, A, B), Thread'(T_3, B, A), Thread'(T_4, B, B)\}$, for some state $Thread'$, and where A and B are data values that are not stored in any node: in this case, different representatives are obtained depending on whether step 3 splits off a process holding two copies of the same data value or two distinct values.

The following example justifies the requirement that the nodes form a single list.

Example 65. Consider a reference-linked collection of seven nodes, arranged as on the left, below.



Steps 1 and 2 of the component ordering algorithm will split the nodes into three sets, corresponding to the three columns in the picture. If the set corresponding to the first column is split, then different representatives will be obtained, depending on whether the node split off has the same datum as its successor, or not.

Now consider the arrangement of six nodes on the right. This similarly does not give unique representatives.

Lemma 66. Consider a system representing a reference-linked data structure, where we now allow a node to have several references (held in distinct variables). Suppose that in addition there are some other distinguished (non-indexed) processes that may hold a reference to a node (like the Top process in the running example). Suppose that every non-free node can be reached

from one of these distinguished variables by following references. Then the algorithm produces unique representatives.

Proof. Consider running the component-ordering algorithm. The nodes referenced by the distinguished variables will be stripped off by the first iteration of step 2 (and placed into singleton sets). The remaining non-free nodes can inductively be stripped off by subsequent iterations, since all are reachable. The free nodes can be dealt with as in Lemma 64. □

8.4 Relaxing assumptions

We now describe techniques for relaxing our assumption about values from $\mathcal{T}$ being held only in simple variables. We explain (implementation-dependent) ways to treat non-simple variables involving $\mathcal{T}$ within the component ordering algorithm, in particular in forming the initial partition in step 1, and in calculating n values in step 2.

- If a variable holds an k-tuple $(v_1, \ldots, v_k)$, it can (for the purpose of the component ordering algorithm) instead be considered as k distinct variables, holding $v_1, \ldots, v_k$, respectively.
- If a variable holds a sequence of values, we can, in step 1, partition states based on the length of the sequence. We can then treat a sequence of length k as k distinct variables, as with k-tuples.
- Dotted sequences can be treated similarly to sequences.
- Sets are slightly harder. One approach, when the elements of the set contain a mix of $\mathcal{T}$ and non-$\mathcal{T}$ parts, is to use the non-$\mathcal{T}$ parts within the partitioning in step 1. Further, in the calculation of the n values, sets can be used in various ways; for example, the clause "$s'(v_k) = s(v_i)$" could be replaced by "$s'(v_k) \in s(v_i)$" when v_k is of type $\hat{T}$ and v_i is of type $\mathbf{P}\,\hat{T}$ (for a distinguished supertype $\hat{T}$).

We do not currently support these extensions: indeed, it is not clear that it would be useful to do so, since doing so would slow the algorithm down, and our experience is that other fields are normally sufficient to break symmetries. We intend to carry out experiments to assess these extensions.

8.5 On compression

As mentioned earlier, FDR uses various compressions. For example, it will automatically compress each leaf GLTS, factoring it by strong bisimulation.

Compression and symmetry reduction work well together: their combination produces smaller state spaces than either technique on its own. However, they do not combine perfectly, as we now explain.

Suppose an uncompressed GLTS L contains two strongly bisimilar states $s_1 = (Q_1, \rho_1)$ and $s_2 = (Q_2, \rho_2)$. Then FDR will pick one of them, say s_1, as the representative of its bisimilarity equivalence class, to include in the compressed GLTS. Now consider the uncompressed GLTS $L' = \pi(L)$. This has strongly bisimilar states $s'_1 = (Q_1, \pi \circ \rho_1)$ and $s'_2 = (Q_2, \pi \circ \rho_2)$. If FDR were to pick s'_1 as the representative of its bisimilarity equivalence class, the compression and symmetry reduction would combine well: given two π-bisimilar states σ and σ' that contain s_1 and s'_1, respectively, the above results show that in many settings, the same representative will be chosen. However, suppose, instead, FDR picks s'_2 as the representative of its bisimilarity equivalence class. Now, when we calculate the representative of the state σ'' that contains s'_2 instead of s'_1, we might well obtain a different representative: the algorithm may be working with a completely different control state and variable binding.

The effect of this is that slightly more states are explored than if the compression and symmetry combined perfectly; however, the effect is rather small, normally less than 1%. Further, the number of states can vary slightly from one run to another: FDR may choose different representatives for a particular bisimilarity equivalence class on different runs.

8.6 Implementation considerations and alternatives

Calculating the n-values is a potentially expensive part of the implementation. We outline our approach. We start by calculating, for each value v of a distinguished type, the set

$$B(v) \quad = \quad \{(s, i) | s \in S,\ i \text{ is an index of a variable of } s \text{ with } s(v_i) = v\}.$$

We call $B(v)$ a *bucket*. All the buckets can be efficiently calculated by calculating a vector of $(s, i, s(v_i))$ tuples, and then sorting and partitioning by the third component. This can be done in time $O(N \log N)$ where N is the total number of variables in the states.

Then to calculate the n-values, we iterate over each bucket $B(v)$, and for each $(s, i), (s', k) \in B(v)$, increment $n_{i,j,k}(s)$, where j is the index in the partition of s'. This takes time $O(\sum_v (\#B(v))^2)$. In practice, the buckets tend to be fairly small.

We intend to investigate alternatives to our definition of the n-values, which might not give such a large reduction in the state space, but that can

be calculated more quickly, and hence give an overall reduction in checking time. Consider

$$n'_{i,j,k}(s) \quad = \quad \text{if } \exists\, s' \in S_j \cdot s'(v_k) = s(v_i) \text{ then } \mathit{1} \text{ else } \mathit{0}.$$

For each s and s', the vector of values of $s'(v_k) = s(v_i)$ (as i and k vary) can be pre-calculated and stored as a bit map. Calculating $n'(s)$ can then be performed as a sequence of bit-wise operations. In the typical case that each bit map fits into a single word (i.e. the number of pairs of variables of the same type is at most 32) this can be done in $O(n)$ operations.

Now consider

$$n''_j(s) \quad = \quad \#\{s' \in S_j \,|\, \exists\, i, k \cdot s'(v_k) = s(v_i)\}.$$

For each s and s', the value of $\exists\, i, k \cdot s'(v_k) = s(v_i)$ can be pre-calculated and stored. The value of $n''(s)$ can then be calculated in $O(n)$ operations. Curiously, for a linked list with no external references, the algorithm will not necessarily give unique representatives, since it will fail to distinguish a linked list from its reversal; however, a more realistic example would have an external reference to the first node, breaking this symmetry.

8.7 Comparisons

In [BDH02], Bošnački et al. define several strategies for producing representative functions. We discuss the two main ones here.

- The **sorted** strategy sorts component states by their non-distinguished parts (i.e. effectively step 1 of Definition 49). If two components have the same non-distinguished parts, they will be ordered arbitrarily; this gives non-unique representatives. We believe that this will work poorly in our setting, because we often have components with the same non-distinguished parts.

- The **segmented** strategy considers all permutations of the distinguished variables that would sort the component states by non-distinguished parts; it then picks the one that produces the lexicographically smallest state. Thus this gives unique representatives. We believe that this would again work poorly in our setting, because there are too many such permutations to consider.

We perform an experimental comparison between these algorithms and our own in the next section.

Sistla et al. [SGE00] employ an algorithm rather similar to ours, although their aim is to test whether two given states s and s' are symmetric rather than to find a representative. They use the naive refinement algorithm of [Bab95] (cf. the first two steps of Definition 49) on each of s and s' until a fixed point is reached. They then try to generate a permutation π to match the states, pairing components in corresponding multisets; for non-singleton multisets produced by the first phase, the relevant parts of the permutation are generated randomly. They then test whether π does indeed relate the two states.

This may falsely report that two states are not symmetric. In particular, this can happen in cases where our approach finds unique representatives. For example, suppose s and s' each corresponds to two linked lists, each of length two. The first phase will, for each state, partition the nodes into those that are the first and second nodes of their lists. Then the second phase will report the states to be symmetric only if the randomly generated permutation happens to pair off the correct states. By contrast, step 3 of our component ordering algorithm will split one of the multisets in an implementation-dependent way; but subsequently step 2 will split the other multiset in a compatible way.

Babai and Kučera [BK79] show that just three steps of the naive refinement algorithm applied to a random graph with n nodes fails to produce a canonical form with probability $(o(1))^n$. This result is not directly applicable to our setting, since we do not deal with random graphs; however, it does suggest that the approach works well.

9 Experiments

We have implemented the techniques described earlier within FDR4. Figure 5 gives results of experiments run to assess the benefits of symmetry reduction.[4] The experiments were performed on a 32-core server (two 2.1GHz Intel(R) Xeon(R) E5-2683 CPUs with hyperthreading enabled, with 256GB of RAM). The example are as follows.

- ListStack represents the linked-list-based implementation of a stack using locking, from the Introduction. The parameters represent the number of nodes in the linked list, the number of threads, and the number of data values.

[4]The interaction with compression (Section 8.5) means that the state count can vary slightly from run to run. However, the variation is normally small, at most 1%. We report typical figures.

Example	Parameters	Without sym. red. States	Time	With sym. red. States	Time
ListStack	(6,4,3)	5952M	528s	108.9K	0.41
	(7,4,2)	3812M	366s	37.57K	0.29s
	(8,4,2)	o/m	—	75.33K	0.41s
	(8,4,4)	—	—	3971K	8.6s
	(12,4,2)	—	—	1208K	6.2s
	(19,4,2)	—	—	154.6M	1130s
TreeBroadcast	(5,2)	303.0M	33s	4362K	3.2s
	(6,2)	o/m	–	3547M	3710s
LockFreeQueue	(4,3,3)	5465M	2060s	6654K	24s
	(5,2,3)	677.6M	205s	614.7K	1.3s
	(5,3,3)	o/m	—	139.1M	692s
	(6,2,2)	1679M	575s	644.9K	1.6s
	(6,3,2)	—	—	173.7M	823s
	(7,2,2)	—	—	3310K	10s
	(11,2,2)	—	—	1412M	8540s
CoarseGrainedListSet	(4,3)	70.33M	9.8s	771.8K	1.3s
	(5,3)	1666M	243s	4140K	7.9s
	(6,3)	—	—	19.15M	31s
	(7,3)	—	—	107.5M	183s
FineGrainedListSet	(3,3)	45.48M	6.9s	1406K	1.9s
	(4,3)	2274M	352s	18.12M	30s
	(5,3)	—	—	173.4M	303s
	(6,3)	—	—	1296M	2530s
ArrayQueue	(2,2,4,2)	8675K	0.94s	82.68K	0.42s
	(3,2,6,3)	o/m	—	102.3M	202s

Figure 5: Experimental results, without and with symmetry reduction, giving the number of states explored and time taken (excluding compilation time). "o/m" indicates that the check ran out of memory; "—" indicates a test not run, since we expected it to run out of memory.

- TreeBroadcast is a model of a network routing algorithm. A collection of nodes, with one distinguished sender, grow a spanning tree of the network, and then use it to broadcast messages. The parameters are the number of nodes excluding the sender, and the number of data values.

- LockFreeQueue is the model from [Low17] of the lock-free queue from [MS96]. The parameters are as for ListStack.

- CoarseGrainedListSet is the model of a linked-list-based implementation of a set from [Che15]; the implementation, based on [HS12, Section 9.4], orders nodes by a hash of their datums and uses coarse-grained synchronization. The parameters are the numbers of nodes and threads (this system is not symmetric in the type of data, because of the use of the hash function; however, we use the same number of data values as nodes in each case).

- FineGrainedListSet is the model of a fine-grained linked-list-based implementation of a set from [Che15]; the implementation, based on [HS12, Section 9.5], associates a lock with each node. The parameters are as for CoarseGrainedListSet.

- ArrayQueue is the model of an array-based queue from [Jan15], based on [CG05]. The parameters are the numbers of threads, data values, sequence numbers on "head" and "tail" references into the array, and sequence numbers on data in the array.

In the latter four cases, the refinement checks tested whether the datatypes were linearizable [HW90]: whether the operations seem to take place one at a time, each between the time at which it is called and when it returns.

Some models needed to be adapted slightly to make them symmetric. For example, LockFreeQueue used a particular node as an initial dummy header node, and a particular data value for it; in order to avoid using constants (to satisfy Definition 43) we adapted the script to choose these initial values nondeterministically (figures in Figure 5 without symmetry reduction are for the initial script, which was optimised for that case).

Speed-ups are considerable, often two or three orders of magnitude. More importantly, we can now check much larger systems than previously. For the LinkedList example without symmetry reduction, it is easy to see that the number of states with parameters $(n, t, 2)$ grows at least proportional to $n! \times 2^n$; hence the case for (19,4,2) would have at least 3.8×10^{26} states

(extrapolating from the (7,4,2) case), which would be too large to check by a factor of more than 10^{16}.

9.1 Comparison with the sorted and segmented techniques

Figure 6 gives results of an experimental comparison between our technique for finding representatives (Section 8), and the sorted and segmented techniques from [BDH02] (Section 8.7), which we have also implemented within FDR.

Recall that the sorted and segmented techniques can perform symmetry reduction only for types that index a family of processes. They can't, therefore, be used to perform symmetry reduction over the type of data values. In order to allow for comparison, we have therefore not performed reduction over this type (contrast with Figure 5).

These experiments suggest that the segmented approach is considerably slower than our own approach, often by two orders of magnitude: recall that this approach considers *all* permutations of the datatype: the cost of doing so is just too great. It is noticeable that the state counts are often very similar to our own. The segmented approach finds unique representatives[5], so this suggests that we normally find unique representatives.

The sorted approach works better than the segmented approach. On small examples, it tends to explore more states than our approach, but take less time: it is a simpler algorithm than ours, so takes less time on each state. However, with larger examples —where speed-ups are more important— our approach tends to be faster: our approach seems to scale better, giving proportionately larger reduction in states in these cases. Further, our technique completes on several examples where the segmented approach runs out of memory. Finally, on examples that make significant use of a symmetric type of data (ListStack, LockFreeQueue and ArrayQueue), our approach where we perform reduction on that type (Figure 5) is faster that the segmented case, except on some very small examples.

10 Conclusions

In this paper we have presented an extension to FDR4 that exploits symmetry in the system. The basic idea is to factor the transition system with

[5]The interaction with compressions (Section 8.5) means that this isn't quite true: indeed sometimes the state count for this approach is actually slightly higher than for our own approach.

Example	Parameters	Our technique States	Our technique Time	Sorted States	Sorted Time	Segmented States	Segmented Time
ListStack	(6,4,3)	650.9K	1.4s	1146K	0.47s	650.1K	299s
	(7,4,2)	75.12K	0.40s	100.6K	0.18s	75.11K	205s
	(8,4,2)	150.6K	0.58s	201.7K	0.26s	150.6K	3310s
	(8,4,4)	94.39M	169s	206.0M	50s	t/o	—
	(10,4,2)	1507M	1760s	3295M	985s	t/o	—
	(12,4,2)	o/m	—	o/m	—	t/o	—
TreeBroadcast	(5,2)	4508K	3.0s	5063K	2.5s	4503K	2.9s
	(6,2)	3595M	3580s	4023M	3230s	3619M	3560s
LockFreeQueue	(4,3,3)	39.25M	114s	134.3M	223s	39.15K	1030s
	(5,2,3)	3643K	7.1s	10.63M	7.5s	2890K	117s
	(5,3,3)	829.5M	3820s	2921M	7530s	t/o	—
	(6,2,2)	1290K	2.8s	6488K	5.1s	1238K	240s
	(6,3,2)	347.2M	1600s	2453M	6130s	t/o	—
	(7,2,2)	6619K	17s	42.07M	32s	4943K	7700s
	(11,2,2)	2824M	16500s	o/m	—	t/o	—
CoarseGrainedListSet	(4,3)	771.8K	1.3s	771.8K	0.85s	771.8K	5.5s
	(5,3)	4139K	7.9s	4139K	5.0s	4139K	37s
	(6,3)	19.15M	31s	19.15M	19s	19.15M	334s
	(7,3)	107.5M	183s	107.5M	113s	107.5M	3750s
FineGrainedListSet	(3,3)	1406K	1.9s	1440K	1.0s	1428K	46s
	(4,3)	18.12M	30s	7951K	19s	7863K	2720s
	(5,3)	173.4M	303s	304.2M	700s	t/o	—
	(6,3)	1296M	2530s	o/m	—	—	—
ArrayQueue	(2,2,4,2)	4349K	2.5s	4389K	1.2s	4343K	1.9s
	(3,2,6,3)	o/m	—	o/m	—	o/m	—

Figure 6: Experimental results, comparing with the sorted and segmented strategies. Conventions are as for Figure 5; “t/o” indicates a timeout after ten hours (36000s).

respect to permutation bisimilarity, picking a representative member of each equivalence class. We have presented refinement-checking algorithms based on this technique, and shown how to extract informative counterexamples when the refinement does not hold.

We have shown how to apply this within the powerful and general supercombinator framework used by FDR: we have shown how to verify that a supercombinator induces a symmetric GLTS, and how to apply a permutation to a state of that GLTS; the same techniques could be applied in other process algebraic settings.

We have presented a general syntactic result showing that a process is symmetric with respect to a datatype, subject to fairly general assumptions, principally that the script contains no constants of that type.

We have presented a novel technique for calculating representatives of equivalence classes, and given evidence that it often finds *unique* representatives. Finally we have carried out experiments that demonstrate the efficacy of our approach. In particular, the results show that our technique for finding representatives works better in practice than previous techniques, particularly for larger examples; further, our technique allows for symmetry reduction over a type that does not index a family of processes, such as the type of data.

It would be possible to further improve the performance of the symmetry implementation in FDR by making the CSP_M evaluator aware of the symmetries. For example, suppose $P(x)$ denotes a symmetric process over the type T: the current FDR evaluator will evaluate $P(x)$ for each member x of T, even though it would be sufficient to evaluate $P(x)$ for a single x. Extending the evaluator to exploit such symmetries is left as future research.

We believe that our approach is widely applicable. Whenever a system is structurally symmetric, the corresponding GLTS is also symmetric. Thus it can be applied to a wide class of concurrent algorithms and distributed systems. Further, when a system uses data with no distinguished values, then the induced GLTS is symmetric in the type of that data.

Acknowledgements We would like to thank Bill Roscoe for useful comments. Research into FDR3 and FDR4 has been partially sponsored by DARPA under agreement number FA8750-12-2-0247, and EPSRC under agreement number EP/N022777.

References

[Bab95] László Babai. Automorphism groups, isomorphism, reconstruction. In R. L. Graham, M. Grötschel, and L. Lováz, editors, *Handbook of Combinatorics*, volume II, chapter 27, pages 1447–1540. North Holland, 1995.

[BDH02] Dragan Bošnački, Dennis Dams, and Leszek Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4:92–106, 2002.

[BK79] László Babai and Ludik Kučera. Canonical labelling of graphs in linear average time. In *Proc. 20th IEEE Symposium on Foundations of Computer Science*, pages 39–46, 1979.

[CEFJ96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9:77–104, 1996.

[CEJS98] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV '98)*, pages 147–158, 1998.

[CG05] Robert Colvin and Linsay Groves. Formal verification of an array-based nonblocking queue. In *Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 507–516, 2005.

[Che15] Ke Chen. Analysing concurrent datatypes in CSP. Master's thesis, University of Oxford, 2015.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9:105–131, 1996.

[FW99] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In *Proceedings of Integrated Formal Methods (IFM '99)*, pages 315–334, 1999.

[GMR+03] Michael Goldsmith, Nick Moffat, A. W. Roscoe, Tim Whitworth, and Irfan Zakiuddin. Watchdog transformations for property-oriented model-checking. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*, 2003.

[GRABR15] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 2015.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, revised first edition edition, 2012.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[ID96] C. Norris Ip and David L Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[Jan15] Ruben Janssen. Verification of concurrent datatypes using CSP. Master's thesis, University of Oxford, 2015.

[Law05] J. Lawrence. Practical applications of CSP and FDR to software design. In *Communicating Sequential Processes: The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2005.

[Low98] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1–2):53–84, 1998.

[Low17] Gavin Lowe. Analysing lock-free linearizable datatypes using CSP. In *Concurrency, Security and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *Lecture Notes in Computer Science*, pages 162–184. Springer, 2017.

[MDC06] Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3), 2006.

[MGR08] Nick Moffat, Michael Goldsmith, and Bill Roscoe. A representative function approach to symmetry exploitation for CSP refinement checking. In *International Conference on Formal Engineering Methods*, volume 5256 of *Lecture Notes in Computer Science*, 2008.

[MS96] Maged Michael and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[MS01] A. Mota and A. Sampaio. Model-checking CSP-Z: Strategy, tool support and industrial application. *Science of Computer Programming*, 40(1):59–96, 2001.

[RH07] A. W. Roscoe and D. Hopkins. SVA: A tool for analysing shared-variable programs. In *Proceedings of Automatic Verification of Critical Systems (AVoCS)*, pages 177–183, 2007.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[SGE00] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A symmetry-based model checker for verification of safety and linveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):133–166, 2000.

[Uni15] University of Oxford. *FDR Documentation*, 2015. `http://www.cs.ox.ac.uk/projects/fdr/manual/index.html`.

A Proof of Proposition 45

In this appendix, we prove Proposition 45. We start by giving more detail about the syntax of CSP_M, and how it is evaluated. The type *Expr* represents expressions. The type *Pat* represents patterns. *Decl* is the type of declarations of one of the following forms:

- $pat = e$, where pat is a pattern and e is an expression;
- Function definitions using pattern matching of the form

$$f(\vec{pat}_{1,1}) \ldots (\vec{pat}_{1,k}) = e_1$$
$$\ldots$$
$$f(\vec{pat}_{n,1}) \ldots (\vec{pat}_{n,k}) = e_n.$$

  Where each $\vec{pat}_{i,j}$ is a vector of patterns, and for each $j \in \{1, \ldots, k\}$, the vectors $\vec{pat}_{1,j}, \ldots, \vec{pat}_{n,j}$ all have the same length.

- **channel** $c : e$, where c is a name, and e is an expression which should evaluate to a set, or may be omitted;

- **datatype** $D = A_1.e_1 | \ldots | A_n.e_n$, where $D, A_1, \ldots, A_n$ are names, and $e_1, \ldots, e_n$ are expressions which should evaluate to sets, or may be omitted.

Stmts is the type of statements used in, for example, list or set comprehensions and indexed operators. Each statement is of one of the following forms:

- A generator $pat \leftarrow s$, where pat is a pattern and s is (depending on context) either a sequence or a set; for each element of s matched by pat, in turn, the free variables of pat get bound appropriately.
- A predicates $pred$, which filters out certain cases.

The semantics is defined using the following functions:

- $\mathsf{eval} : Env \rightarrow Expr \nrightarrow Value$ such that $\mathsf{eval}\,\rho\,e$ gives the value of expression e in environment ρ.
- $\mathsf{matches} : Env \rightarrow Pat \rightarrow Value \rightarrow Bool$ such that $\mathsf{matches}\,\rho\,pat\,v$ tests whether pattern pat can be matches by value v.
- $\mathsf{bind}_0, \mathsf{bind} : Env \rightarrow Pat \rightarrow Value \nrightarrow Env$ such that $\mathsf{bind}_0\,\rho\,pat\,v$ gives the updates to the environment by binding pattern pat to value v, and is defined when $\mathsf{matches}\,\rho\,pat\,v$; and $\mathsf{bind}\,\rho\,pat\,v$ gives the result of those updates, i.e.

$$\mathsf{bind}\,\rho\,pat\,v \quad = \quad \rho \oplus \mathsf{bind}_0\,\rho\,pat\,v.$$

- $\mathsf{bindDecls} : Env \rightarrow Decl^* \nrightarrow Env$ such that $\mathsf{bindDecl}\,\rho\,decls$, updates ρ corresponding to the list of declarations $decls$, assuming that all names bound by the declarations are distinct, and that each pattern matches the corresponding expression.
- $\mathsf{evalStmts} : Env \rightarrow Stmt^* \rightarrow Env^*$ such that $\mathsf{evalStmts}\,\rho\,stmts$ gives the sequence of environments resulting from evaluating the statements $stmts$. (In some circumstances, such as set comprehensions, this sequence of environments is converted into a set; we elide this conversion below.)

A.1 Auxiliary definitions and results

The following definition applies only to values that can be pattern matched against, so excluding functions, maps, LTSs, etc.

Definition 67. A value v is *complete* in environment ρ if

- v is a basic value, tuple, set, or sequence;
- v is of the form $A.w$, $\rho(A)$ is of the form *dtcons* S, and $w \in S$; i.e. v is a complete value of some datatype;
- v is of the form $c.w$, $\rho(c)$ is of the form *channel* S, and $w \in S$; i.e. v is an event.

(We drop the reference to ρ where it is implicit.)

A value v is *completable* in ρ if either it is complete, or it has some extension $v.v'$ that is complete. Necessarily no proper prefix of v is complete in this case.

Lemma 68. let π be a type-preserving permutation on $\mathcal{T}$. Suppose ρ respects π. Then

- v is complete in ρ iff $\pi(v)$ is complete in ρ;
- v is completable in ρ iff $\pi(v)$ is completable in ρ.

Proof. Suppose, first, that $v = A.w$ is a complete value of some datatype, but not from the distinguished subtype T. Then $\rho(A)$ is of the form *dtcons* S where $w \in S$. But then by the assumption that ρ respects π, $\pi(w) \in S$. So then $\pi(v) = A.\pi(w)$ is a complete value in ρ.

Now suppose $v = A$, a member of $\mathcal{T}$. Then either $\rho(A) = \rho(\pi(A)) = dtcons\{\varepsilon\}$, and both v and $\pi(v)$ are complete, or $\rho(A) = \rho(\pi(A)) = \perp$, and neither is complete.

Other cases are similar or trivial.

The result for completability follows easily. □

Definition 69. A dotted term v is *well-formed* in environment ρ if it can be written in the form $v_1. \ldots .v_n.w$ where $n \geq 0$, each of $v_1, \ldots, v_n$ is a complete value, and w is a completable value.

Lemma 70. Suppose ρ respects π. Then v is well-formed iff $\pi(v)$ is well-formed.

Proof. Suppose v is well-formed. Then it is of the form $v_1. \ldots .v_n.w$ where $n \geq 0$, each of $v_1, \ldots, v_n$ is complete, and w is completable. Then $\pi(v) = \pi(v_1). \ldots .\pi(v_n).\pi(w)$. But by Lemma 68, each of $\pi(v_1), \ldots, \pi(v_n)$ is complete, and $\pi(w)$ is completable. Hence $\pi(v)$ is well-formed. The converse is similar. □

A.2 Proof of Proposition 45

We prove Proposition 45 by proving the following, stronger, result, which includes subsidiary results for the different syntactic categories.

Proposition 71. Suppose a script is constant-free for $\mathcal{T}$, let π be a type-preserving permutation on $\mathcal{T}$, and suppose ρ respects π.

1. Let *pat* be a pattern, and v a value; then

$$\mathsf{matches}\,\rho\,pat\;v \quad \Leftrightarrow \quad \mathsf{matches}(\pi \circ \rho)\rho\,pat\;(\pi(v)).$$

2. Let *pat* be a pattern, and v a value; then

$$\pi \circ (\mathsf{bind}_{\mathit{0}}\;\rho\,pat\;v) \quad = \quad \mathsf{bind}_{\mathit{0}}(\pi \circ \rho)\;pat\;(\pi(v)),$$

   and hence

$$\pi \circ (\mathsf{bind}\,\rho\,pat\;v) \quad = \quad \mathsf{bind}(\pi \circ \rho)\;pat\;(\pi(v)).$$

3. Suppose *decls* is a list of declarations of values, channels and datatypes, with disjoint names. Then

$$\pi \circ \mathsf{bindDecls}\,\rho\;decls \quad = \quad \mathsf{bindDecls}(\pi \circ \rho)\;decls.$$

   Hence, letting ρ_{1} be the environment formed from evaluating the top-level declarations, we have that $\pi(\rho_{\mathit{1}}(x)) = \rho_{\mathit{1}}(x)$ for each name x bound at the top level.

   In particular, this shows that the set of values associated with each channel or datatype name is closed under π; i.e. ρ_{1} respects π.

4. For every expression e in the script, and every environment ρ

$$\pi(\mathsf{eval}\,\rho\,e) \quad = \quad \mathsf{eval}(\pi \circ \rho)e.$$

5. For every sequence *stmts* of statements (generators or qualifiers), and every environment ρ

$$\pi(\mathsf{evalStmts}\,\rho\,stmts) \quad = \quad \mathsf{evalStmts}(\pi \circ \rho)stmts.$$

Proof. We prove the clauses by a simultaneous induction over the syntax. (In this proof, we assume familiarity with CSP_M; see [Uni15]. We omit explicit definitions of the semantic functions; the reader should be able to deduce them easily from the relevant parts of the proof.)

1. We prove

$$\mathsf{matches}\,\rho\,pat\ v \quad\Leftrightarrow\quad \mathsf{matches}(\pi\circ\rho)\,pat\,(\pi(v))$$

by performing a case analysis over the pattern pat.

- Case x, a variable. This matches all values, so both sides are $true$.
- A pattern of the form $pat = pat_1 @@ pat_2$. Then (using the inductive hypothesis):

$$\begin{array}{ll} & \mathsf{matches}\,\rho\,pat\ v \\ \Leftrightarrow & \mathsf{matches}\,\rho\,pat_1\ v \wedge \mathsf{matches}\,\rho\,pat_2\ v \\ \Leftrightarrow & \mathsf{matches}(\pi\circ\rho)\,pat_1\,(\pi(v)) \wedge \mathsf{matches}(\pi\circ\rho)\,pat_2\,(\pi(v)) \\ \Leftrightarrow & \mathsf{matches}(\pi\circ\rho)\rho\,pat\,(\pi(v)). \end{array}$$

- A pattern of the form $pat = A$ where A is a constructor of a datatype (i.e., $\rho(A)$ and $(\pi\circ\rho)(A)$ are $dtcons$ values). Necessarily A is not an element of $\mathcal{T}$ (since the pattern contains no constant of type $\mathcal{T}$). Then $\mathsf{matches}\,pat\ v$ iff $v = A$. But this holds iff $\mathsf{matches}\,pat\,(\pi(v))$, since $\pi(A) = A$.
- A pattern of the form $pat = A.pat'$ where A is a constructor of a datatype, and pat' is a non-empty pattern. Again A is not an element of $\mathcal{T}$. Then $\mathsf{matches}\,\rho\,pat\ v$ iff v is of the form $A.v'$ and $\mathsf{matches}\,\rho\,pat'\ v'$. By the inductive hypothesis, the latter clause holds iff $\mathsf{matches}(\pi\circ\rho)\,pat'\,(\pi(v'))$. But then $\mathsf{matches}(\pi\circ\rho)\,pat\,\pi(v)$, since $\pi(v) = A.\pi(v')$.
- A pattern of the form $pat = c$ or $pat = c.pat'$ where c is a channel name. These cases are very similar to the two previous cases.
- A pattern of the form $pat = x.pat_1$ where x is a variable, and pat_1 is a pattern. Consider, first, a dotted value $v = v_0.v_1$, where v_0 is the shortest prefix of v that is a completed value, and v_1 is non-empty (so x will be matched by v_0). Note that $\pi(v_0)$ is also a completed value, by Lemma 68. Then

$$\begin{array}{ll} & \mathsf{matches}\,\rho\,pat\ v \\ \Leftrightarrow & \mathsf{matches}\,\rho\,pat_1\ v_1 \\ \Leftrightarrow & \qquad \big(\text{inductive hypothesis}\big) \\ & \mathsf{matches}(\pi\circ\rho)\,pat_1\,(\pi(v_1)) \\ \Leftrightarrow & \qquad \big(\pi(v_0) \text{ is a completed value}\big) \\ & \mathsf{matches}(\pi\circ\rho)\,(x.pat_1)\,(\pi(v_0).\pi(v_1)) \\ \Leftrightarrow & \mathsf{matches}(\pi\circ\rho)\,pat\,(\pi(v)). \end{array}$$

If v is not of the above form, then neither $\mathsf{matches}\,\rho\, pat\; v$ nor $\mathsf{matches}(\pi \circ \rho)\, pat\, (\pi(v))$ holds.

- A pattern of the form $pat = _.pat_1$ where pat_1 is a pattern. This case is very similar to the previous.
- A pattern of the form $pat = (pat_1@@pat_2).pat_3$. Suppose $pat_1@@pat_2$ matches dotted tuples of minimal size $n > 0$. (Note that the sub-patterns may end with variables or wildcards that allow them to match arbitrarily long dotted tuples; here we are defining n to be the minimal length dotted tuple that can be matched; FDR calculates n in a straightforward way.) If v is not a dotted tuple of size at least $n + 1$ then pat matches neither v nor $\pi(v)$. Let $v = v_1.v_2$ where v_1 has size n. Then

$$\begin{array}{ll} & \mathsf{matches}\,\rho\, pat\; v \\ \Leftrightarrow & \mathsf{matches}\,\rho\, pat_1\; v_1 \wedge \mathsf{matches}\,\rho\, pat_2\; v_1 \wedge \mathsf{matches}\,\rho\, pat_3\; v_2 \\ \Leftrightarrow & \mathsf{matches}(\pi \circ \rho)\, pat_1\, (\pi(v_1)) \wedge \mathsf{matches}(\pi \circ \rho)\, pat_2\, (\pi(v_1)) \wedge \\ & \mathsf{matches}(\pi \circ \rho)\, pat_3\, (\pi(v_2)) \\ \Leftrightarrow & \mathsf{matches}(\pi \circ \rho)\, pat\, (\pi(v)), \end{array}$$

noting that $\pi(v) = \pi(v_1).\pi(v_2)$ and $\pi(v_1)$ has size n.

- A pattern of the form $pat = pat_0.pat_1$, other than the previous five cases; i.e. pat_0 is a basic value, tuple, set or sequence pattern, so pat_0 matches just the first term of a dotted tuple. Consider a dotted value $v = v_0.v_1$, where v_0 is not a dotted value. Then

$$\begin{array}{ll} & \mathsf{matches}\,\rho\, pat\; v \\ \Leftrightarrow & \mathsf{matches}\,\rho\, pat_0\; v_0 \wedge \mathsf{matches}\,\rho\, pat_1\; v_1 \\ \Leftrightarrow & \quad \big(\text{inductive hypothesis}\big) \\ \Leftrightarrow & \mathsf{matches}(\pi \circ \rho)\, pat_0\, (\pi(v_0)) \wedge (\pi \circ \rho)\ \mathsf{matches}\, pat_1\, (\pi(v_1)) \\ \Leftrightarrow & \mathsf{matches}(\pi \circ \rho)\, (pat_0.pat_1)\, (\pi(v_0).\pi(v_1)) \\ \Leftrightarrow & \mathsf{matches}(\pi \circ \rho)\, pat\, (\pi(v)). \end{array}$$

If v is not a dotted value, then neither $\mathsf{matches}\,\rho\, pat\; v$ nor $\mathsf{matches}\,(\pi \circ \rho)\, pat\, (\pi(v))$ holds.

- All other cases are similar.

2. We prove $\pi \circ \mathsf{bind}_0\, \rho\, pat\; v = \mathsf{bind}_0(\pi \circ \rho)\, pat\, (\pi(v))$ by performing a case analysis over the pattern pat. In all cases, by part 1, $\mathsf{bind}_0\, \rho\, pat\; v$ is defined iff $\mathsf{matches}\,\rho\, pat\; v$ iff $\mathsf{matches}(\pi \circ \rho)\, pat\, (\pi(v))$ iff $\mathsf{bind}_0(\pi \circ \rho)$ $pat\, (\pi(v))$ is defined; we assume these hold in the following.

- Case x, a variable.

$$
\begin{aligned}
& \pi \circ \mathsf{bind}_0\, \rho\, x\, v \\
= \; & \pi \circ \{x \mapsto v\} \\
= \; & \{x \mapsto \pi(v)\} \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, x\, (\pi(v)).
\end{aligned}
$$

- A pattern of the form $pat = A$ where A is a constructor of a datatype. Again, A is not a member of $\mathcal{T}$, by assumption. Necessarily, $v = A$, and so $\pi(v) = A$. Then

$$
\pi \circ \mathsf{bind}_0\, \rho\, pat\, v = \{\} = \mathsf{bind}_0 (\pi \circ \rho)\, pat\, (\pi(v)).
$$

- A pattern of the form $pat = A.pat_1$ where A is a constructor of a datatype and pat_1 is nonempty. Again, A is not a member of $\mathcal{T}$. Necessarily, v is of the form $A.v_1$ such that $\mathsf{matches}\, \rho\, pat_1\, v_1$, so $\pi(v) = A.\pi(v_1)$. Then

$$
\begin{aligned}
& \pi \circ \mathsf{bind}_0\, \rho\, pat\, v \\
= \; & \pi \circ \mathsf{bind}_0\, \rho\, pat_1\, v_1 \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, pat_1\, (\pi(v_1)) \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, pat\, (\pi(v)).
\end{aligned}
$$

- A pattern of the form $pat = pat_0.pat_1$, where pat_0 is not a dotted term, a variable or wildcard, a datatype constructor or channel name, nor a double pattern; hence pat_0 matches just the first term of a dotted tuple. Necessarily, the variables bound by pat_0 and pat_1 are disjoint. Consider a dotted value $v = v_0.v_1$ where v_0 is not a dotted term; so $\mathsf{matches}\, \rho\, pat_0\, v_0$ and $\mathsf{matches}\, \rho\, pat_1\, v_1$. Then

$$
\begin{aligned}
& \pi \circ \mathsf{bind}_0\, \rho\, pat\, v \\
= \; & \pi \circ (\mathsf{bind}_0\, \rho\, pat_0\, v_0 \cup \mathsf{bind}_0\, \rho\, pat_1\, v_1) \\
= \; & (\pi \circ \mathsf{bind}_0\, \rho\, pat_0\, v_0) \cup (\pi \circ \mathsf{bind}_0\, \rho\, pat_1\, v_1) \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, pat_0\, (\pi(v_0)) \cup \mathsf{bind}_0 (\pi \circ \rho)\, pat_1\, (\pi(v_1)) \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, (pat_0.pat_1)\, (\pi(v_0).\pi(v_1)) \\
= \; & \mathsf{bind}_0 (\pi \circ \rho)\, pat\, v.
\end{aligned}
$$

- A pattern of the form $pat = x.pat_1$ where x is a variable, and pat_1 is a pattern. This is very similar to the previous case, except x matches the shortest prefix of v that is a completed value.

- A pattern of the form $(pat_0 @@ pat_1).pat_2$, where $pat_0 @@ pat_1$ matches dotted tuples of minimal size $n > 0$. This is very similar to the previous case, taking v_0 to be the first n terms of v.

Other cases are similar.

Then

$$\begin{aligned} & \pi \circ \mathsf{bind}\, \rho\, pat\ v \\ = \ & \pi \circ (\rho \oplus \mathsf{bind}_0\, \rho\, pat\ v) \\ = \ & (\pi \circ \rho) \oplus (\pi \circ \mathsf{bind}_0\, \rho\, pat\ v) \\ = \ & (\pi \circ \rho) \oplus \mathsf{bind}_0 (\pi \circ \rho)\, pat\ v) \\ = \ & \mathsf{bind}(\pi \circ \rho)\, pat\ v. \end{aligned}$$

3. We now prove

$$\pi \circ \mathsf{bindDecls}\, \rho\, decls \quad = \quad \mathsf{bindDecls}(\pi \circ \rho)\, decls.$$

We use a subsidiary function $\mathsf{bindDecl} : Env \rightarrow Decl \nrightarrow Env$ such that $\mathsf{bindDecl}\, \rho\, decl$ gives the update to the environment corresponding to a *single* declaration $decl$ (assuming that pattern matching succeeds). We assume that ρ respects π, and show that

$$\pi \circ \mathsf{bindDecl}\, \rho\, decl \quad = \quad \mathsf{bindDecl}(\pi \circ \rho)\, decl,$$

by a case analysis on $decl$.

- Case $pat = e$. Then

$$\begin{aligned} & \pi \circ \mathsf{bindDecl}\, \rho\, (pat = e) \\ = \ & \pi \circ (\text{let } v = \mathsf{eval}\, \rho\, e \text{ in} \\ & \quad\quad \text{if } \mathsf{matches}\, \rho\, pat\ v \text{ then } \mathsf{bind}_0\, \rho\, pat\ v \text{ else } error) \\ = \ & \text{let } v = \mathsf{eval}\, \rho\, e \text{ in} \\ & \text{if } \mathsf{matches}\, \rho\, pat\ v \text{ then } \pi \circ \mathsf{bind}_0\, \rho\, pat\ v \text{ else } \pi(error) \\ = \ & \quad\quad \big(\text{parts 1 and 2; } \pi(error) = error\big) \\ & \text{let } v = \mathsf{eval}\, \rho\, e \text{ in} \\ & \text{if } \mathsf{matches}(\pi \circ \rho)\, pat\ (\pi(v)) \text{ then } \mathsf{bind}_0(\pi \circ \rho)\, pat\ (\pi(v)) \\ & \text{else } error \\ = \ & \quad\quad \big(\text{letting } w = \pi(v)\big) \\ & \text{let } w = \pi(\mathsf{eval}\, \rho\, e) \text{ in} \\ & \text{if } \mathsf{matches}(\pi \circ \rho)\, pat\ w \text{ then } \mathsf{bind}_0(\pi \circ \rho)\, pat\ w \text{ else } error \\ = \ & \quad\quad \big(\text{part 4}\big) \\ & \text{let } w = \mathsf{eval}(\pi \circ \rho)\, e \text{ in} \\ & \text{if } \mathsf{matches}(\pi \circ \rho)\, pat\ w \text{ then } \mathsf{bind}_0(\pi \circ \rho)\, pat\ w \text{ else } error \\ = \ & \mathsf{bindDecl}(\pi \circ \rho)\, (pat = e). \end{aligned}$$

- Case of a function definition. Consider, first, a function of a single argument with $m > 0$ sub-arguments. The definition can use $n > 0$ clauses, each clause using patterns for the sub-arguments. We write $\vec{pat}_i$ for the tuple of m patterns used in the ith clause; necessarily, for each i, the variables of the patterns in $\vec{pat}_i$ are disjoint. Let *decl* be the declaration

$$f(\vec{pat}_1) = e_1$$
$$\ldots$$
$$f(\vec{pat}_n) = e_n.$$

We write $\vec{v}$ for a tuple of m values (supplied as actual parameters). We extend matches and bind to tuples in the obvious way:

$$\begin{array}{l} \mathsf{matches}\,\rho\,(pat_1, \ldots, pat_m)\,(v_1, \ldots, v_m) = \\ \qquad \mathsf{matches}\,\rho\,pat_1\ v_1\ \wedge \ldots \wedge\ \mathsf{matches}\,\rho\,pat_m\ v_m, \\ \mathsf{bind}\,\rho\,(pat_1, \ldots, pat_m)\,(v_1, \ldots, v_m) = \\ \qquad \mathsf{bind}(\ldots(\mathsf{bind}(\mathsf{bind}\,\rho\,pat_1\ v_1)\,pat_2\ v_2)\ldots)\,pat_m\ v_m. \end{array}$$

It is clear that parts 1 and 2 imply corresponding results for these

generalised versions. Then

$$
\begin{array}{ll}
& \pi \circ \mathsf{bindDecl}\,\rho\,\mathit{decl} \\
= & \pi \circ \{f \mapsto \lambda\,\vec{v} \cdot \text{if}\,\mathsf{matches}\,\rho\,\vec{pat}_{\mathit{1}}\ \vec{v} \\
& \qquad\qquad\quad \text{then}\ \mathsf{eval}(\mathsf{bind}\,\rho\,\vec{pat}_{\mathit{1}}\ \vec{v})\ e_{\mathit{1}} \\
& \qquad\qquad\quad \text{else}\ \ldots \\
& \qquad\qquad\quad \text{else if}\,\mathsf{matches}\,\rho\,\vec{pat}_n\ \vec{v} \\
& \qquad\qquad\quad \text{then}\ \mathsf{eval}(\mathsf{bind}\,\rho\,\vec{pat}_n\ \vec{v})\ e_n \\
& \qquad\qquad\quad \text{else}\ \mathit{error}\} \\
= & \qquad \big(\text{letting}\ \vec{w} = \pi(\vec{v})\big) \\
& \{f \mapsto \lambda\,\vec{w} \cdot \text{if}\,\mathsf{matches}\,\rho\,\vec{pat}_{\mathit{1}}\ (\pi^{-\mathit{1}}(\vec{w})) \\
& \qquad\qquad \text{then}\ \pi(\mathsf{eval}(\mathsf{bind}\,\rho\,\vec{pat}_{\mathit{1}}\ (\pi^{-\mathit{1}}(\vec{w})))\ e_{\mathit{1}}) \\
& \qquad\qquad \text{else}\ \ldots \\
& \qquad\qquad \text{else if}\,\mathsf{matches}\,\rho\,\vec{pat}_n\ (\pi^{-\mathit{1}}(\vec{w})) \\
& \qquad\qquad \text{then}\ \pi(\mathsf{eval}(\mathsf{bind}\,\rho\,\vec{pat}_n\ (\pi^{-\mathit{1}}(\vec{w})))\ e_n) \\
& \qquad\qquad \text{else}\ \pi(\mathit{error})\} \\
= & \qquad \big(\text{part 4}\big) \\
& \{f \mapsto \lambda\,\vec{w} \cdot \text{if}\,\mathsf{matches}\,\rho\,\vec{pat}_{\mathit{1}}\ (\pi^{-\mathit{1}}(\vec{w})) \\
& \qquad\qquad \text{then}\ \mathsf{eval}(\pi \circ \mathsf{bind}\,\rho\,\vec{pat}_{\mathit{1}}\ (\pi^{-\mathit{1}}(\vec{w})))\ e_{\mathit{1}} \\
& \qquad\qquad \text{else}\ \ldots \\
& \qquad\qquad \text{else if}\,\mathsf{matches}\,\rho\,\vec{pat}_n\ (\pi^{-\mathit{1}}(\vec{w})) \\
& \qquad\qquad \text{then}\ \mathsf{eval}(\pi \circ \mathsf{bind}\,\rho\,\vec{pat}_n\ (\pi^{-\mathit{1}}(\vec{w})))\ e_n \\
& \qquad\qquad \text{else}\ \mathit{error}\} \\
= & \qquad \big(\text{parts 1 and 2, generalised to tuples}\big) \\
& \{f \mapsto \lambda\,\vec{w} \cdot \text{if}\,\mathsf{matches}(\pi \circ \rho)\ \vec{pat}_{\mathit{1}}\ \vec{w} \\
& \qquad\qquad \text{then}\ \mathsf{eval}(\mathsf{bind}(\pi \circ \rho)\ \vec{pat}_{\mathit{1}}\ \vec{w})\ e_{\mathit{1}} \\
& \qquad\qquad \text{else}\ \ldots \\
& \qquad\qquad \text{else if}\,\mathsf{matches}(\pi \circ \rho)\ \vec{pat}_n\ \vec{w} \\
& \qquad\qquad \text{then}\ \mathsf{eval}(\mathsf{bind}(\pi \circ \rho)\ \vec{pat}_n\ \vec{w})\ e_n \\
& \qquad\qquad \text{else}\ \mathit{error}\} \\
= & \mathsf{bindDecl}(\pi \circ \rho)\ \mathit{decl}.
\end{array}
$$

The extension to a function with k curried arguments —where each argument has several sub-arguments, presented as patterns— is straightforward but notationally messy.

- Case of a channel declaration, **channel** $c : e_1.\dots.e_n$.

$$
\begin{array}{ll}
 & \pi \circ \mathsf{bindDecl}\,\rho\,(\textbf{channel}\ c : e_1.\dots.e_n) \\
= & \pi \circ \{c \mapsto dtcons\{v_1.\dots.v_n | v_1 \in \mathsf{eval}\,\rho\,e_1, \dots, v_n \in \mathsf{eval}\,\rho\,e_n\}\} \\
= & \{c \mapsto dtcons\{\pi(v_1).\dots.\pi(v_n)| \\
 & \qquad\qquad v_1 \in \mathsf{eval}\,\rho\,e_1, \dots, v_n \in \mathsf{eval}\,\rho\,e_n\}\} \\
= & \qquad \big(\text{letting } w_i = \pi(v_i)\big) \\
 & \{c \mapsto dtcons\{w_1.\dots.w_n| \\
 & \qquad\qquad w_1 \in \pi(\mathsf{eval}\,\rho\,e_1), \dots, w_n \in \pi(\mathsf{eval}\,\rho\,e_n)\}\} \\
= & \qquad \big(\text{item } 4\big) \\
 & \{c \mapsto dtcons\{w_1.\dots.w_n| \\
 & \qquad\qquad w_1 \in \mathsf{eval}(\pi \circ \rho)\,e_1, \dots, w_n \in \mathsf{eval}(\pi \circ \rho)\,e_n\}\} \\
= & \mathsf{bindDecl}(\pi \circ \rho)\,(\textbf{channel}\ c : e_1.\dots.e_n).
\end{array}
$$

Note, in particular, that $v_1.\dots.v_n$ is a complete value iff $w_1.\dots.w_n$ is a complete value, by Lemma 68.

- Case of a datatype declaration. Let $decl$ be the declaration

$$\textbf{datatype}\ D = A_1.e_{1,1}.\dots.e_{1,m_1}|\dots|A_n.e_{n,1}.\dots.e_{n,m_n}\ .$$

Suppose, first, that this is not the declaration of a distinguished supertype $\hat{T}_i$. Then

$$
\begin{array}{ll}
 & \pi \circ \mathsf{bindDecl}\,\rho\,decl \\
= & \text{let } S_i = \{v_{i,1}.\dots.v_{i,m_i} | v_{i,j} \in \mathsf{eval}\,\rho\,e_{i,j},\ j = 1, \dots, m_i\}, \\
 & \qquad\qquad \text{for } i = 1, \dots, n \text{ in} \\
 & \pi \circ (\{A_i \mapsto dtcons\,S_i | i \in \{1, \dots, n\}\} \\
 & \quad \cup \{D \mapsto \{A_i.v_i | i \in \{1, \dots, n\}, v_i \in S_i\}\}) \\
= & \text{let } S_i = \{v_{i,1}.\dots.v_{i,m_i} | \pi(v_{i,j}) \in \pi(\mathsf{eval}\,\rho\,e_{i,j}),\ j = 1, \dots, m_i\}, \\
 & \qquad\qquad \text{for } i = 1, \dots, n \text{ in} \\
 & \{A_i \mapsto dtcons\,\pi(S_i) | i \in \{1, \dots, n\}\} \\
 & \cup \{D \mapsto \{\pi(A_i).\pi(v_i) | i \in \{1, \dots, n\}, v_i \in S_i\}\} \\
= & \qquad \begin{pmatrix} \pi(A_i) = A_i;\ \text{letting } S_i' = \pi(S_i), \\ w_i = \pi(v_i),\ \ w_{i,j} = \pi(v_{i,j}) \end{pmatrix} \\
 & \text{let } S_i' = \{w_{i,1}.\dots.w_{i,m_i} | w_{i,j} \in \pi(\mathsf{eval}\,\rho\,e_{i,j}),\ j = 1, \dots, m_i\}, \\
 & \qquad\qquad \text{for } i = 1, \dots, n \text{ in} \\
 & \{A_i \mapsto dtcons\,S_i' | i \in \{1, \dots, n\}\} \\
 & \cup \{D \mapsto \{A_i.w_i | i \in \{1, \dots, n\}, w_i \in S_i'\}\} \\
= & \qquad \big(\text{part } 4\big) \\
 & \text{let } S_i' = \{w_{i,1}.\dots.w_{i,m_i} | w_{i,j} \in \mathsf{eval}(\pi \circ \rho)\,e_{i,j},\ j = 1, \dots, m_i\}, \\
 & \qquad\qquad \text{for } i = 1, \dots, n \text{ in} \\
 & \{A_i \mapsto dtcons\,S_i' | i \in \{1, \dots, n\}\} \\
 & \cup \{D \mapsto \{A_i.w_i | i \in \{1, \dots, n\}, w_i \in S_i'\}\} \\
= & \mathsf{bindDecl}(\pi \circ \rho)\,decl.
\end{array}
$$

Note, in particular, that $A_i.v_i$ is a complete value iff $A_i.w_i$ is a complete value, by Lemma 68.

Now suppose this is the declaration of a distinguished super-type $\hat{T}_i$. Without loss of generality, suppose the elements of the distinguished sub-type T_i are the first N branches of *decl*, i.e. $T_i = \{A_1, \ldots, A_N\}$, with $N \leq n$. The only part of the above calculation that changes is the binding for D, which becomes

$$\begin{array}{ll} & D \mapsto \{\pi(A_1), \ldots, \pi(A_N)\} \cup \\ & \quad \{\pi(A_i).\pi(v_i) | i \in \{N+1, \ldots, n\}, v_i \in S_i\} \\ = & \quad \big(\pi \text{ is a permutation on } T_i\text{; as above}\big) \\ & D \mapsto \{A_1, \ldots, A_N\} \cup \\ & \quad \{A_i.w_i | i \in \{N+1, \ldots, n\}, w_i \in S'_i\} \end{array}$$

which is the relevant part of $\mathsf{bindDecl}(\pi \circ \rho)\mathit{decl}$.

Now consider a set of declarations *decls*, which necessarily bind disjoint variables. These may be mutually recursive, so their semantics is defined as a fixed point. One step of the iteration corresponding to that fixed point is given by the following function:

$$F(\rho) \;=\; \rho \oplus \bigcup \{\mathsf{bindDecl}\,\rho\,\mathit{decl} | \mathit{decl} \in \mathit{decls}\}.$$

Given an environment ρ and a set of declarations *decls*, we write $\rho^\perp$ for the extension of ρ that maps the variables bound by *decls* to $\perp$. Then the semantics of *decls* is defined as the least fixed point of F:

$$\mathsf{bindDecls}\,\rho\,\mathit{decls} \;=\; \bigsqcup_{n \in \mathbf{N}} F^n(\rho^\perp).$$

We have (still assuming that ρ respects π).

$$\begin{array}{ll} & \pi \circ F(\rho) \\ = & (\pi \circ \rho) \oplus \bigcup \{\pi \circ \mathsf{bindDecl}\,\rho\,\mathit{decl} | \mathit{decl} \in \mathit{decls}\} \\ = & \quad \big(\text{above result}\big) \\ & (\pi \circ \rho) \oplus \bigcup \{\mathsf{bindDecl}(\pi \circ \rho)\,\mathit{decl} | \mathit{decl} \in \mathit{decls}\} \\ = & F(\pi \circ \rho). \end{array}$$

Say that environment ρ is π-*invariant* if $\pi \circ \rho = \rho$. Recall that datatype and channel declarations occur only at the top level. We show that each of the (partial) environments ρ in which these declarations are evaluated is π-invariant and respects π, as is the resulting environment. We then discharge our assumption that *all* declarations are evaluated in an environment that respects π.

- Consider the initial empty environment $\rho_0 = \{\}$, and its extension $\rho_0^\perp$ that maps variables bound at the top level to $\perp$. Clearly $\pi \circ \rho_0^\perp = \rho_0^\perp$ so $\rho_0^\perp$ is π-invariant. And $\rho_0^\perp$ respects π, trivially.
- Suppose ρ is π-invariant and respects π, and let $\rho' = F(\rho)$. Then $\pi \circ \rho' = \pi \circ F(\rho) = F(\pi \circ \rho) = F(\rho) = \rho'$, using the above calculation. Hence ρ' is π-invariant.

  Suppose $\rho'(A) = dtcons\, S$. Then, since ρ' is π-invariant, $(\pi \circ \rho')(A) = dtcons\, S$. But $(\pi \circ \rho')(A) = dtcons(\pi(S))$ so $S = \pi(S)$. A similar result holds for channel values. Further, ρ' maps each element of $\mathcal{T}$ to $dtcons\{\varepsilon\}$. Hence ρ' respects π.
- A simple induction than shows that $F^n(\rho_0^\perp)$ is π-invariant and respects π.

  Let $\rho_1 = \bigsqcup_{n \in \mathbf{N}} F^n(\rho_0^\perp)$ be the environment resulting from evaluating the top-level declarations. Then, by continuity,

  $$\pi \circ \rho_1 = \bigsqcup_{n \in \mathbf{N}} \pi \circ F^n(\rho_0^\perp) = \bigsqcup_{n \in \mathbf{N}} F^n(\rho_0^\perp) = \rho_1.$$

  So ρ_1 is π-invariant.

  Arguing as in the previous bullet point, ρ_1 respects π.
- Hence the environment resulting from the top-level declarations respects π. Subsequent environments do not change datatype or channel bindings, and so also respect π.

Continuing, using the above result $\pi \circ F(\rho) = F(\pi \circ \rho)$, a simple induction shows that

$$\pi \circ F^n(\rho) \quad = \quad F^n(\pi \circ \rho), \qquad \text{for } n \in \mathbf{N}.$$

So

$$\begin{array}{ll} & \pi \circ \mathsf{bindDecls}\, \rho\, decls \\ = & \pi \circ \bigsqcup_{n \in \mathbf{N}} F^n(\rho^\perp) \\ = & \qquad \big(\text{continuity}\big) \\ & \bigsqcup_{n \in \mathbf{N}} \pi \circ F^n(\rho^\perp) \\ = & \bigsqcup_{n \in \mathbf{N}} F^n(\pi \circ \rho^\perp) \\ = & \bigsqcup_{n \in \mathbf{N}} F^n((\pi \circ \rho)^\perp) \\ = & \mathsf{bindDecls}(\pi \circ \rho)\, decls. \end{array}$$

4. We now show $\pi(\mathsf{eval}\, \rho\, e) = \mathsf{eval}(\rho \circ \pi) e$.

   We start with a small result which will be useful when dealing with functions. Suppose we have a CSP$_M$ function f, whose denotation is the

(mathematical) function f (i.e. $\mathsf{eval}\,\rho\,\mathsf{f} = f$, independent of ρ). Suppose further that $f(\pi(x)) = \pi(f(x))$. Then we can show $\pi(\mathsf{eval}\ \rho\,\mathsf{f}) = \mathsf{eval}(\pi \circ \rho)\,\mathsf{f}$, as follows.

$$
\begin{array}{cl}
 & \pi(\mathsf{eval}\ \rho\,\mathsf{f}) \\
= & \pi(\lambda\, x \cdot f(x)) \\
= & \lambda\, y \cdot \pi(f(\pi^{-1}(y))) \\
= & \qquad \text{(assumption)} \\
 & \lambda\, y \cdot f(y) \\
= & \mathsf{eval}(\pi \circ \rho)\,\mathsf{f}.
\end{array}
$$

We now prove $\pi(\mathsf{eval}\,\rho\, e) = \mathsf{eval}(\rho \circ \pi) e$, via a case analysis over e. We start with non-process expressions.

- Case A, the name of a datatype constructor (i.e. such that $\rho(A)$ is a *dtcons* value, and hence $(\pi \circ \rho)(A)$ is also a *dtcons* value). Necessarily A is not in $\mathcal{T}$, so $\pi(A) = A$. Then $\pi(\mathsf{eval}\,\rho\, A) = \pi(A) = A = \mathsf{eval}(\pi \circ \rho)\, A$. Names of channels are similar.
- Case x, a variable. Then $\pi(\mathsf{eval}\,\rho\, x) = \pi(\rho(x)) = \mathsf{eval}(\pi \circ \rho)\, x$.
- Case literals (integers, booleans, characters). These cases are trivial.
- Case $e_1 . e_2$. Assuming, for the moment, that all values are well-formed:

$$
\begin{array}{cl}
 & \pi(\mathsf{eval}\,\rho\,(e_1 . e_2)) \\
= & \pi(\mathsf{eval}\,\rho\, e_1 . \mathsf{eval}\,\rho\, e_2) \\
= & \pi(\mathsf{eval}\,\rho\, e_1) . \pi(\mathsf{eval}\,\rho\, e_2) \\
= & \mathsf{eval}(\pi \circ \rho)\; e_1 . \mathsf{eval}(\pi \circ \rho)\; e_2 \\
= & \mathsf{eval}(\pi \circ \rho)\; (e_1 . e_2).
\end{array}
$$

  The left-hand side is well-formed iff the right-hand side is well-formed, by Lemma 70; otherwise, both sides give *error*.
- Tuples and function application are similar.
- **if** statements are similar, using the fact that boolean values are invariant under π.
- Suppose $\oplus$ is a binary operator such that

$$\pi(v \oplus v') \;\; = \;\; \pi(v) \oplus \pi(v').$$

  Then (below, the first and last occurrences of $\oplus$ are pieces of syntax, and the other occurrences are the corresponding semantic

operation):

$$\begin{aligned}\pi(\mathsf{eval}\,\rho\,(e_1 \oplus e_2)) &= \pi(\mathsf{eval}\,\rho\,e_1 \oplus \mathsf{eval}\,\rho\,e_2)\\ &= \pi(\mathsf{eval}\,\rho\,e_1) \oplus \pi(\mathsf{eval}\,\rho\,e_2)\\ &= \mathsf{eval}(\pi \circ \rho)\,e_1 \oplus \mathsf{eval}(\pi \circ \rho)\,e_2\\ &= \mathsf{eval}(\pi \circ \rho)(e_1 \oplus e_2).\end{aligned}$$

We show that the above conditions holds for all the binary operators over non-process values.

- For operators over integers (e.g. $+$) or booleans (e.g. and), the result holds trivially.
- For equality and inequality the result holds since π is a bijection.
- Order relations "<=", "<", ">" and ">=" can be applied to a number of different types. Note that these operators cannot be applied directly to elements of $\mathcal{T}$. The cases of integer or character arguments are trivial. For tuples the operators represent lexicographic ordering; the result holds by a straightforward induction. For sequences, sets and maps, the operators represent prefix, subset and submap, respectively; the result holds since, for example, v <= v' implies $\pi(v)$ <= $\pi(v')$ in each case.
- For list concatenation (^) the result clearly holds.

- The cases for the following built-in functions are similar: ! (boolean negation); the set functions card, diff, empty, inter, Inter, member, Seq, Set, union and Union; the sequence functions # (length), elem, head, length, null, set and tail; the map functions emptyMap, mapDelete, mapFromList, mapLookup, mapMember, mapUpdate, mapUpdateMultiple, and Map; and the relation functions relational_image, relational_inverse_image and transpose.

- Case $\lambda\, pat \cdot e$, where pat is a pattern. Then

$$
\begin{array}{ll}
& \pi(\mathsf{eval}\,\rho\,(\lambda\, pat \cdot e)) \\
= & \pi(\lambda\, v \cdot \text{if}\ \mathsf{matches}\,\rho\, pat\ v\ \text{then}\ \mathsf{eval}(\mathsf{bind}\,\rho\, pat\ v)\ e\ \text{else}\ error) \\
= & \lambda\, w \cdot \pi(\text{if}\ \mathsf{matches}\,\rho\, pat\ (\pi^{-1}(w)) \\
& \qquad\quad \text{then}\ \mathsf{eval}(\mathsf{bind}\,\rho\, pat\ (\pi^{-1}(w)))\ e \\
& \qquad\quad \text{else}\ error) \\
= & \qquad \big(\text{inductive hypothesis, parts 1, 4}\big) \\
& \lambda\, w \cdot \text{if}\ \mathsf{matches}(\pi \circ \rho)\ pat\ w \\
& \qquad \text{then}\ \mathsf{eval}(\pi \circ \mathsf{bind}\,\rho\, pat\ (\pi^{-1}(w)))\ e \\
& \qquad \text{else}\ error \\
= & \qquad \big(\text{inductive hypothesis, part 2}\big) \\
& \lambda\, w \cdot \text{if}\ \mathsf{matches}(\pi \circ \rho)\ pat\ w\ \text{then}\ \mathsf{eval}(\mathsf{bind}(\pi \circ \rho)\ pat\ w)\ e \\
& \qquad \text{else}\ error \\
= & \mathsf{eval}(\pi \circ \rho)(\lambda\, pat \cdot e).
\end{array}
$$

- Case **let** *decls* **within** e, where *decls* is a list of declarations. If any pattern matching within *decls* fails in environment ρ, the same is true in environment $\pi \circ \rho$, by part 1; in this case, both sides evaluate to *error*. If all pattern matching succeeds, then

$$
\begin{array}{ll}
& \pi(\mathsf{eval}\,\rho\,(\mathbf{let}\ decls\ \mathbf{within}\ e)) \\
= & \pi(\text{let}\ \rho' = \mathsf{bindDecls}\,\rho\, decls\ \text{in}\ \mathsf{eval}\,\rho'\ e) \\
= & \text{let}\ \rho' = \mathsf{bindDecls}\,\rho\, decls\ \text{in}\ \pi(\mathsf{eval}\,\rho'\ e) \\
= & \qquad \big(\text{inductive hypothesis}\big) \\
& \text{let}\ \rho' = \mathsf{bindDecls}\,\rho\, decls\ \text{in}\ \mathsf{eval}(\pi \circ \rho')\ e \\
= & \qquad \big(\text{letting}\ \rho'' = \pi \circ \rho'\big) \\
& \text{let}\ \rho'' = \pi \circ \mathsf{bindDecls}\,\rho\, decls\ \text{in}\ \mathsf{eval}\,\rho''\ e \\
= & \qquad \big(\text{part 3}\big) \\
& \text{let}\ \rho'' = \mathsf{bindDecls}(\pi \circ \rho)\ decls\ \text{in}\ \mathsf{eval}\,\rho''\ e \\
= & \mathsf{eval}(\pi \circ \rho)\ (\mathbf{let}\ decls\ \mathbf{within}\ e).
\end{array}
$$

- Case extensions. By the assumption of well-typing, extensions must be applied to an argument of the form $A.v$ (where v may be empty), with $\rho(A)$ of the form *dtcons* S or *channel* S. Note that if v is not a proper prefix of a member of S then extensions returns an error. We abuse notation slightly and write $\rho(A)$ for this S. Note that $\rho(A) = \rho(\pi(A))$: if $A \in \mathcal{T}$ then both equal $\{\varepsilon\}$; otherwise $A = \pi(A)$. Below, we write "$\lambda\, A.v \cdot$" for a pattern-matching

lambda-abstraction.

$$
\begin{array}{rl}
& \pi(\mathsf{eval}\,\rho\,\mathsf{extensions}) \\
= & \pi(\lambda\, A.v \cdot \text{if}\,\exists\, w \neq \varepsilon \cdot v.w \in \rho(A) \\
& \qquad\qquad \text{then } \{w | v.w \in \rho(A)\} \\
& \qquad\qquad \text{else } \mathit{error}) \\
= & \qquad \big(\text{letting } A' = \pi(A),\;\; v' = \pi(v)\big) \\
& \lambda\, A'.v' \cdot \text{if}\,\exists\, w \neq \varepsilon \cdot \pi^{-1}(v').w \in \rho(\pi^{-1}(A')) \\
& \qquad\qquad \text{then } \pi\{w | \pi^{-1}(v').w \in \rho(\pi^{-1}(A'))\} \\
& \qquad\qquad \text{else } \pi(\mathit{error}) \\
= & \qquad \left(\begin{array}{l}\pi^{-1}(v').w \in \rho(\pi^{-1}(A')) \Leftrightarrow v'.\pi(w) \in (\pi \circ \rho)(A'), \\ \text{since } \rho(\pi^{-1}(A')) = \rho(A')\end{array}\right) \\
& \lambda\, A'.v' \cdot \text{if}\,\exists\, w \neq \varepsilon \cdot v'.\pi(w) \in (\pi \circ \rho)(A') \\
& \qquad\qquad \text{then } \{\pi(w) | v'.\pi(w) \in (\pi \circ \rho)(A')\} \\
& \qquad\qquad \text{else } \mathit{error} \\
= & \qquad \big(\text{letting } w' = \pi(w)\big) \\
& \lambda\, A'.v' \cdot \text{if}\,\exists\, w' \neq \varepsilon \cdot v'.w' \in (\pi \circ \rho)(A') \\
& \qquad\qquad \text{then } \{w' | v'.w' \in (\pi \circ \rho)(A')\} \\
& \qquad\qquad \text{else } \mathit{error} \\
= & \mathsf{eval}(\pi \circ \rho)\,\mathsf{extensions}.
\end{array}
$$

- The case for productions is similar to the previous case.
- The case for Events follows immediately from the assumption that ρ respects π.
- The cases for lists $<e_1, \ldots, e_n>$, ranged integer lists $<e_1..e_2>$, infinite integer lists $<e..>$, sets $\{e_1, \ldots, e_n\}$, ranged integer sets $\{e_1..e_2\}$, infinite integer sets $\{e..\}$, enumerated sets $\{\!|e_1, \ldots, e_n|\!\}$ and maps $(\!|e_1 => e'_1, \ldots, e_n => e'_n|\!)$ are each a straightforward induction (using the result for productions in the case of enumerated sets).
- List comprehensions, $<e|\mathit{stmts}>$, where *stmts* is a sequence of statements (generators or qualifiers). Below, the list comprehensions in the first and last lines are syntax, and the comprehensions

in other lines are semantic.

$$
\begin{array}{ll}
 & \pi(\mathsf{eval}\,\rho\,{<}e|stmts{>}) \\
= & \pi(\langle \mathsf{eval}\,\rho'\;e|\rho' \leftarrow \mathsf{evalStmts}\,\rho\;stmts\rangle) \\
= & \langle \pi(\mathsf{eval}\,\rho'\;e)|\rho' \leftarrow \mathsf{evalStmts}\,\rho\;stmts\rangle \\
= & \qquad \big(\text{inductive hypothesis}\big) \\
 & \langle \mathsf{eval}(\pi\circ\rho')\;e|\rho' \leftarrow \mathsf{evalStmts}\,\rho\;stmts\rangle \\
= & \qquad \big(\text{letting } \rho'' = \pi\circ\rho'\big) \\
 & \langle \mathsf{eval}\,\rho''\;e|\rho'' \leftarrow \pi(\mathsf{evalStmts}\,\rho\;stmts)\rangle \\
= & \qquad \big(\text{part 5}\big) \\
 & \langle \mathsf{eval}\,\rho''\;e|\rho'' \leftarrow \mathsf{evalStmts}(\pi\circ\rho)\;stmts\rangle \\
= & \mathsf{eval}(\pi\circ\rho)\,{<}e|stmts{>}.
\end{array}
$$

- Other comprehensions are similar, namely finite or infinite ranged list comprehensions, set comprehensions, finite or infinite ranged set comprehensions, and enumerated set comprehensions.

We now consider process expressions. In the following, we will sometimes apply a CSP operator to AGLTSs to represent the operation on AGLTSs corresponding to the CSP operator. We will take care to distinguish between syntactic and semantic operations.

We will assume, without loss of generality, that subprocesses are implemented as augmented GLTSs.

Recall that we simplify labels on states. We make this more precise here. We say that environments ρ and ρ' are *compatible* if they agree upon the values of all common variables; note that in this case $\rho \cup \rho'$ is an environment (i.e. a function). We define a function *simplify* that merges labels when the sub-environments are compatible. For example, for $\oplus$ one of $\Box$, $|||$, $\rhd$ or $\triangle$:

$$
\begin{array}{l}
simplify((e,\rho)\oplus(e',\rho')) = \\
\qquad \begin{array}{ll}
(e\oplus e', \rho\cup\rho'), & \text{if } \rho \text{ and } \rho' \text{ are compatible} \\
(e,\rho)\oplus(e',\rho'), & \text{otherwise.}
\end{array}
\end{array}
$$

Other clauses are similar. It is then easy to show that

$$\pi \circ simplify \;=\; simplify \circ \pi.$$

- Basic processes, STOP, SKIP, **div**: these are all trivial.
- Prefixing. Consider $e_c\, f_1 \ldots f_n \rightarrow P$, where e_c is an expression that should evaluate to a possibly incomplete event (i.e. a channel name with zero or more fields supplied), $n \geq 0$, and each f_i is a field of one of the following forms:

- $?pat$, where *pat* is a pattern; this offers an external choice between all values matching *pat* and consistent with the channel type.
- $?pat : E$, where *pat* is a pattern, and E should evaluate to a set; this is like the previous case, but restricts values to elements of E.
- $!e$, where e is an expression; this matches only the value of e.
- $\$pat$; this performs an internal choice between all values matching *pat* and consistent with the channel type.
- $\$pat : E$; this is like the previous case, but restricts values to elements of E.

The semantics of pattern matching by fields depends upon the field's location: a variable pattern (e.g. $?x$) matches a *single* complete value, *except* in the final field where it matches the whole of the rest of the event. For example, the prefix construct $c?x?y$ matches the event $c.1.2.3$, binding x to 1, and y to 2.3.

If there are one or more \$-fields, then the process has initial τ-transitions to resolve the internal choices; each such field is replaced by a field $!x$, where x is a fresh variable bound in the environment to the value chosen; also the environment is updated corresponding to any other variables bound in the field. The definition of CSP_M requires that every \$-field precedes every !- or ?-field.

Recall that we assume that distinct variables have distinct names. In particular, this means that the initial expression e_c will use no variables bound by a \$-field; this is required, since we will evaluate e_c *after* the \$-fields.

We perform a case analysis on whether or not the prefixing construct contains any \$-fields.

Case of no \$-fields. Consider, first, a prefixing construct $e_c\, fs \rightarrow P$ that contains no \$-fields.

We define the semantics of prefixing using a function

$$\begin{aligned}\mathsf{evalField} :: Env \rightarrow Bool \rightarrow Value \rightarrow Field \rightarrow \\ (\mathbf{P}(Value \times Env) \cup \{error\}).\end{aligned}$$

If $c.v$ is an incomplete event on channel c, and f is a field, then, for a correct usage, $\mathsf{evalField}\, \rho\, last\, (c.v)\, f$ gives a set of $(c.v.w, \rho')$

pairs; each $c.v.w$ is an extension of $c.v$, compatible with c, corresponding to adding this field, and ρ' is the updated environment caused by binding any variables in patterns in f. Compatibility will mean that if $\rho(c) = \mathit{channel}\, S$ then $v.w$ is a prefix of some element of S; we denote this $v.w \leq \rho(c)$. However, an error will occur if a ! field gives a value not compatible with the channel. The argument *last* of evalField indicates whether this is the final field in the prefixing construct, which affects the semantics of pattern matching, as explained above.

We prove the following

$$\pi(\mathsf{evalField}\, \rho\, \mathit{last}\, (c.v)\, f) = \mathsf{evalField}(\pi \circ \rho)\, \mathit{last}\, (c.\pi(v))\, f. \qquad (1)$$

We perform a case analysis on f.

- Case $!e$.

$$\begin{array}{cl} & \pi(\mathsf{evalField}\, \rho\, \mathit{last}\, (c.v)\, (!e)) \\ = & \pi(\text{let } w = \mathsf{eval}\, \rho\, e \text{ in} \\ & \quad \text{if } v.w \leq \rho(c) \text{ then } \{(c.v.w, \rho)\} \text{ else } \mathit{error}) \\ = & \text{let } w = \mathsf{eval}\, \rho\, e \text{ in} \\ & \text{if } \pi(v).\pi(w) \leq (\pi \circ \rho)(c) \text{ then } \{(c.\pi(v).\pi(w), \pi \circ \rho)\} \\ & \text{else } \pi(\mathit{error}) \\ = & \qquad \big(\text{letting } w' = \pi(w)\big) \\ & \text{let } w' = \pi(\mathsf{eval}\, \rho\, e) \text{ in} \\ & \text{if } \pi(v).w' \leq (\pi \circ \rho)(c) \text{ then } \{(c.\pi(v).w', \pi \circ \rho)\} \\ & \text{else } \mathit{error} \\ = & \qquad \big(\text{inductive hypothesis applied to } e\big) \\ & \text{let } w' = \mathsf{eval}(\pi \circ \rho)\, e \text{ in} \\ & \text{if } \pi(v).w' \leq (\pi \circ \rho)(c) \text{ then } \{(c.\pi(v).w', \pi \circ \rho)\} \\ & \text{else } \mathit{error} \\ = & \mathsf{evalField}(\pi \circ \rho)\, \mathit{last}\, (c.\pi(v))\, (!e). \end{array}$$

- Case $?pat$. We write $\mathit{arity}(pat)$ for the number of fields matched by *pat* in the case that this is not the final field (so each variable matches a single value). Similarly, we write $\mathit{arity}(w)$ for the number of complete values within w. Below, W is the set of candidate values that *pat* could be matched against: it contains all values w, with the appropriate arity, that can be appended to v to give a prefix of an element

of $\rho(c)$.

$$
\begin{aligned}
& \pi(\mathsf{evalField}\,\rho\,\mathit{last}\,(c.v)\,(?\mathit{pat})) \\
= \; & \pi(\text{let } W = \text{if } \mathit{last} \text{ then } \{w | v.w \in \rho(c)\} \\
& \qquad\qquad\quad \text{else } \{w | v.w.u \in \rho(c), \mathit{arity}(\mathit{pat}) = \mathit{arity}(w)\} \\
& \quad \text{in } \{(c.v.w, \mathsf{bind}\,\rho\,\mathit{pat}\,w) | w \in W, \mathsf{matches}\,\rho\,\mathit{pat}\,w\}) \\
= \; & \text{let } W = \text{if } \mathit{last} \text{ then } \{w | v.w \in \rho(c)\} \\
& \qquad\qquad \text{else } \{w | v.w.u \in \rho(c), \mathit{arity}(\mathit{pat}) = \mathit{arity}(w)\} \text{ in} \\
& \{(c.\pi(v).\pi(w), \pi \circ \mathsf{bind}\,\rho\,\mathit{pat}\,w) | w \in W, \mathsf{matches}\,\rho\,\mathit{pat}\,w\} \\
= \; & \qquad \big(\text{parts 1 and 2}\big) \\
& \text{let } W = \text{if } \mathit{last} \text{ then } \{w | v.w \in \rho(c)\} \\
& \qquad\qquad \text{else } \{w | v.w.u \in \rho(c), \mathit{arity}(\mathit{pat}) = \mathit{arity}(w)\} \text{ in} \\
& \{(c.\pi(v).\pi(w), \mathsf{bind}(\pi \circ \rho)\,\mathit{pat}\,(\pi(w))) | \\
& \qquad w \in W, \mathsf{matches}(\pi \circ \rho)\,\mathit{pat}\,(\pi(w))\} \\
= \; & \qquad \big(\text{letting } w' = \pi(w), \;\; W' = \pi(W)\big) \\
& \text{let } W' = \text{if } \mathit{last} \text{ then } \{\pi(w) | \pi(v).\pi(w) \in (\pi \circ \rho)(c)\} \\
& \qquad\qquad \text{else } \{\pi(w) | \pi(v).\pi(w).\pi(u) \in (\pi \circ \rho)(c), \\
& \qquad\qquad\qquad\qquad \mathit{arity}(\mathit{pat}) = \mathit{arity}(w)\} \text{ in} \\
& \{(c.\pi(v).w', \mathsf{bind}(\pi \circ \rho)\,\mathit{pat}\,w') | \\
& \qquad w' \in W', \mathsf{matches}(\pi \circ \rho)\,\mathit{pat}\,w'\} \\
= \; & \qquad \begin{pmatrix} \text{letting } w' = \pi(w), \;\; u' = \pi(u); \\ \mathit{arity}(w) = \mathit{arity}(\pi(w)) \text{ by Lemma 68} \end{pmatrix} \\
& \text{let } W' = \text{if } \mathit{last} \text{ then } \{w' | \pi(v).w' \in (\pi \circ \rho)(c)\} \\
& \qquad\qquad \text{else } \{w' | \pi(v).w'.u' \in (\pi \circ \rho)(c), \\
& \qquad\qquad\qquad\quad \mathit{arity}(\mathit{pat}) = \mathit{arity}(w')\} \text{ in} \\
& \{(c.\pi(v).w', \mathsf{bind}(\pi \circ \rho)\,\mathit{pat}\,w') | \\
& \qquad w' \in W', \mathsf{matches}(\pi \circ \rho)\,\mathit{pat}\,w'\} \\
= \; & \mathsf{evalField}(\pi \circ \rho)\,\mathit{last}\,(c.\pi(v))\,(?\mathit{pat}).
\end{aligned}
$$

– Case $?\mathit{pat} : E$. This is very similar to the previous case, except in the main set comprehensions, w ranges over $W \cap \mathsf{eval}\,\rho\,E$. Then w' ranges over $W' \cap \pi(\mathsf{eval}\,\rho\,E) = W' \cap \mathsf{eval}(\pi \circ \rho)\,E$ (by the inductive hypothesis).

We now define a function that accumulate over the fields to get the resulting events and environments (but handling errors appro-

priately).

$$
\begin{aligned}
&\mathsf{evalFields} :: \mathit{Env} \to \mathit{Value} \to \mathit{Field}^* \to \\
&\qquad\qquad (\mathbf{P}(\mathit{Value} \times \mathit{Env}) \cup \{\mathit{error}\}) \\
&\mathsf{evalFields}\,\rho\,(c.v)\,\langle\rangle = \{(c.v, \rho)\} \\
&\mathsf{evalFields}\,\rho\,(c.v)\,(\langle f\rangle^\frown \mathit{fs}) = \\
&\quad \text{let } S = \mathsf{evalField}\,\rho\,(\mathit{fs} = \langle\rangle)\,(c.v)\,f \text{ in} \\
&\quad \text{if } S = \mathit{error} \text{ then } \mathit{error} \\
&\quad \text{else } \bigcup\{\mathsf{evalFields}\,\rho'\,(c.w)\,\mathit{fs} | (c.w, \rho') \in S\}.
\end{aligned}
$$

(The term $\mathit{fs} = \langle\rangle$ tests whether f is the last field.) We show

$$\pi(\mathsf{evalFields}\,\rho\,(c.v)\,\mathit{fs}) \quad = \quad \mathsf{evalFields}(\pi \circ \rho)\,(c.\pi(v))\,\mathit{fs}, \qquad (2)$$

by an induction on fs.

- Case $\langle\rangle$. Straightforward: both sides give $\{(c.\pi(v), \pi \circ \rho)\}$.
- Case $\langle f\rangle^\frown \mathit{fs}$.

$$
\begin{aligned}
& \pi(\mathsf{evalFields}\,\rho\,(c.v)\,(\langle f\rangle^\frown \mathit{fs})) \\
= \;& \text{let } S = \mathsf{evalField}\,\rho\,(\mathit{fs} = \langle\rangle)\,(c.v)\,f \text{ in} \\
& \text{if } S = \mathit{error} \text{ then } \pi(\mathit{error}) \\
& \text{else } \bigcup\{\pi(\mathsf{evalFields}\,\rho'\,(c.w)\,\mathit{fs}) | (c.w, \rho') \in S\} \\
= \;& \qquad \big(\text{inductive hypothesis}\big) \\
& \text{let } S = \mathsf{evalField}\,\rho\,(\mathit{fs} = \langle\rangle)\,(c.v)\,f \text{ in} \\
& \text{if } S = \mathit{error} \text{ then } \mathit{error} \\
& \text{else } \bigcup\{\mathsf{evalFields}(\pi \circ \rho')\,(c.\pi(w))\,\mathit{fs} | \\
& \qquad\qquad \pi(c.w, \rho') \in \pi(S)\} \\
= \;& \qquad \big(\text{letting } w' = \pi(w),\ \rho'' = \pi \circ \rho',\ S' = \pi(S)\big) \\
& \text{let } S' = \pi(\mathsf{evalField}\,\rho\,(\mathit{fs} = \langle\rangle)\,(c.v)\,f) \text{ in} \\
& \text{if } S' = \mathit{error} \text{ then } \mathit{error} \\
& \text{else } \bigcup\{\mathsf{evalFields}\,\rho''\,(c.w')\,\mathit{fs} | (c.w', \rho'') \in S'\} \\
= \;& \qquad \big(\text{equation (1)}\big) \\
& \text{let } S' = \mathsf{evalField}(\pi \circ \rho)\,(\mathit{fs} = \langle\rangle)\,(c.\pi(v))\,f \text{ in} \\
& \text{if } S' = \mathit{error} \text{ then } \mathit{error} \\
& \text{else } \bigcup\{\mathsf{evalFields}\,\rho''\,(c.w')\,\mathit{fs} | (c.w', \rho'') \in S'\} \\
= \;& \mathsf{evalFields}(\pi \circ \rho)\,(c.\pi(v))\,(\langle f\rangle^\frown \mathit{fs}).
\end{aligned}
$$

Now, let $c.v = \mathsf{eval}\,\rho\,e_c$. Then $\mathsf{eval}(\pi \circ \rho)\,e_c = c.\pi(v)$, by the inductive hypothesis. Let

$$
\begin{aligned}
L &= \mathsf{eval}\,\rho\,(e_c\,\mathit{fs} \to P), \\
S &= \mathsf{evalFields}\,\rho\,(c.v)\,\mathit{fs}, \\
L' &= \mathsf{eval}(\pi \circ \rho)\,(e_c\,\mathit{fs} \to P), \\
S' &= \mathsf{evalFields}(\pi \circ \rho)\,(c.\pi(v))\,\mathit{fs}.
\end{aligned}
$$

The above result shows $\pi(S) = S'$. We need to show $\pi(L) = L'$.

If $S = error$ then $L = error$ and $\pi(L) = error$. But then $S' = error$, so $L' = error$, as required.

Otherwise, L is an AGLTS as follows.

- The initial state is $(e_c\, fs \rightarrow P, \rho)$.
- For each $(a, \rho') \in S$, there is a transition labelled with a from the initial state.
- For each $(a, \rho') \in S$, the sub-GLTS following the transition of the previous item equals $\mathsf{eval}\, \rho'\, P$.

Then $\pi(L)$ is an augmented LTS as follows.

- The initial state is $(e_c\, fs \rightarrow P, \pi \circ \rho)$. This is the same as the initial state of L'.
- For each $(a, \rho') \in S$, there is a transition labelled with $\pi(a)$ from the initial state. Letting $a' = \pi(a)$ and $\rho'' = \pi \circ \rho'$, this is equivalent to saying that for every $(a', \rho'') \in \pi(S) = S'$, there is a transition labelled with a' from the initial state. This is the same as the initial transitions of L'.
- For each $(a, \rho') \in S$, the sub-LTS after the transition of the previous item equals $\pi(\mathsf{eval}\, \rho'\, P)$. But this equals $\mathsf{eval}(\pi \circ \rho')\, P$, by the inductive hypothesis. Letting $a' = \pi(a)$ and $\rho'' = \pi \circ \rho'$, this is equivalent to saying that for every $(a', \rho'') \in \pi(S) = S'$, the sub-LTS after this transition equals $\mathsf{eval}\, \rho''\, P$. This is the same as for L'.

Hence $\pi(L) = L'$, as required.

Case of at least one \$-field. Now suppose that the prefixing construct contains at least one \$-field. We define the semantics using a function

$$\mathsf{eval\$Field} :: Env \rightarrow Bool \rightarrow Value \rightarrow Field \rightarrow \mathbf{P}(Value \times Env).$$

If $c.v$ is an incomplete event on channel c, and f is a \$-field, then, for a correct usage, $\mathsf{eval\$Field}\, \rho\, last\, (c.v)\, f$ gives a set of (w, ρ') pairs; each w is a value that could be communicated in the place of f, compatible with $c.v$, and ρ' is the updated environment caused by binding any variables in patterns in f. Compatibility again means that if $\rho(c) = channel\, S$ then $v.w$ is a prefix of some element of S, denoted $v.w \leq \rho(c)$. The argument $last$

of $\mathsf{evalField}$ again indicates whether this is the final field in the prefixing construct.

We can prove the following, for f a \$-field:

$$\pi(\mathsf{eval\$Field}\,\rho\, last\,(c.v)\,f) = \mathsf{eval\$Field}(\pi \circ \rho)\, last\,(c.\pi(v))\,f. \quad (3)$$

The proof is almost identical to the cases for ?-fields in the proof of equation (1), and so is omitted.

We now define a function that, given an environment ρ, an incomplete event $c.v$, and a list of fields fs, gives a set of pairs (fs', ρ') such that fs' is the result of substituting each \$-field of fs with a field $!x$ where x is a fresh variable, and ρ' is the environment resulting from binding x to a value w that could instantiate this field, and also binding any variables in the \$-fields with the corresponding values. The definition makes use of the fact that every \$-field must precede every !- or ?-field.

$$\begin{array}{l}
\mathsf{eval\$Fields} :: Env \to Value \to Field^* \to \mathbf{P}(Field^* \times Env),\\
\mathsf{eval\$Fields}\,\rho\,(c.v)\,\langle\rangle = \{(\langle\rangle, \rho)\},\\
\mathsf{eval\$Fields}\,\rho\,(c.v)\,(\langle f\rangle^\frown fs) =\\
\quad \text{if}\, f \text{ is a \$-field}\\
\quad \text{then } \{(\langle !x\rangle^\frown fs', \rho') | (w, \rho_f) \in \mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f,\\
\qquad\qquad\quad (fs', \rho') \in \mathsf{eval\$Fields}(\rho_f \cup \{x \mapsto w\})\,(c.v.w)\,fs\}\\
\qquad\quad \text{where } x \text{ is fresh}\\
\quad \text{else } \{(\langle f\rangle^\frown fs, \rho)\}.
\end{array}$$

We now show

$$\pi(\mathsf{eval\$Fields}\,\rho\,(c.v)\,fs) \;=\; \mathsf{eval\$Fields}(\pi \circ \rho)\,(c.\pi(v))\,fs, \quad (4)$$

by an induction on fs.

– Case $\langle\rangle$. Straightforward: both sides equal $\{(\langle\rangle, \pi \circ \rho)\}$.
– Case $\langle f\rangle^\frown fs$ where f is a \$-field. We require that the same

fresh variable x is used in both cases below.

$$
\begin{array}{ll}
 & \pi(\mathsf{eval\$Fields}\,\rho\,(c.v)\,(\langle f\rangle ^\frown fs) \\
= & \pi\{(\langle !x\rangle ^\frown fs', \rho') | \\
 & \quad (w, \rho_f) \in \mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f, \\
 & \quad (fs', \rho') \in \mathsf{eval\$Fields}(\rho_f \cup \{x \mapsto w\})\,(c.v.w)\,fs\} \\
= & \{(\langle !x\rangle ^\frown fs', \pi\circ\rho') | \\
 & \quad (w, \rho_f) \in \mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f, \\
 & \quad \pi(fs', \rho') \in \pi(\mathsf{eval\$Fields}(\rho_f \cup \{x \mapsto w\})\,(c.v.w)\,fs)\} \\
= & \quad \big(\text{letting } \rho'' = \pi\circ\rho'\big) \\
 & \{(\langle !x\rangle ^\frown fs', \rho'') | \\
 & \quad (w, \rho_f) \in \mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f, \\
 & \quad (fs', \rho'') \in \pi(\mathsf{eval\$Fields}(\rho_f \cup \{x \mapsto w\})\,(c.v.w)\,fs)\} \\
= & \quad \big(\text{inductive hypothesis}\big) \\
 & \{(\langle !x\rangle ^\frown fs', \rho'') | \\
 & \quad \pi(w, \rho_f) \in \pi(\mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f), \\
 & \quad (fs', \rho'') \in \mathsf{eval\$Fields}\,((\pi\circ\rho_f) \cup \{x \mapsto \pi(w)\}) \\
 & \qquad\qquad (c.\pi(v).\pi(w))\,fs\} \\
= & \quad \big(\text{letting } w' = \pi(w),\ \rho'_f = \pi\circ\rho_f\big) \\
 & \{(\langle !x\rangle ^\frown fs', \rho'') | \\
 & \quad (w', \rho'_f) \in \pi(\mathsf{eval\$Field}\,\rho\,(fs = \langle\rangle)\,(c.v)\,f), \\
 & \quad (fs', \rho'') \in \mathsf{eval\$Fields}(\rho'_f \cup \{x \mapsto w'\})\,(c.\pi(v).w')\,fs\} \\
= & \quad \big(\text{equation (3)}\big) \\
 & \{(\langle !x\rangle ^\frown fs', \rho'') | \\
 & \quad (w', \rho'_f) \in \mathsf{eval\$Field}(\pi\circ\rho)\,(fs = \langle\rangle)\,(c.\pi(v))\,f, \\
 & \quad (fs', \rho'') \in \mathsf{eval\$Fields}(\rho'_f \cup \{x \mapsto w'\})\,(c.\pi(v).w')\,fs\} \\
= & \mathsf{evalFields}(\pi\circ\rho)\,(c.\pi(v))\,(\langle f\rangle ^\frown fs).
\end{array}
$$

– Case $\langle f\rangle ^\frown fs$ where f is not a \$-field. Straightforward: both sides equal $\{(\langle f\rangle ^\frown fs, \pi\circ\rho)\}$.

Now, let $c.v = \mathsf{eval}\,\rho\,e_c$. So $\mathsf{eval}(\pi\circ\rho)\,e_c = \pi(\mathsf{eval}\,\rho\,e_c) = c.\pi(v)$, using the inductive hypothesis. So let

$$
\begin{array}{rcl}
L & = & \mathsf{eval}\,\rho\,(e_c\,fs \to P), \\
S & = & \mathsf{eval\$Fields}\,\rho\,(c.v)\,fs, \\
L' & = & \mathsf{eval}(\pi\circ\rho)\,(e_c\,fs \to P), \\
S' & = & \mathsf{eval\$Fields}(\pi\circ\rho)\,(c.\pi(v))\,fs.
\end{array}
$$

The above result shows $\pi(S) = S'$. We need to show $\pi(L) = L'$. If $S = \{\}$ then $L = \mathit{error}$ and $\pi(L) = \mathit{error}$. But then $S' = \{\}$, so $L' = \mathit{error}$, as required.

Otherwise, L is an augmented LTS as follows.

- The initial state is $(e_c\,fs \to P, \rho)$.
- From the initial state, for each $(fs', \rho') \in S$, there is a τ-transition to $(e_c\,fs' \to P, \rho')$.
- For each $(fs', \rho') \in S$, the sub-LTS rooted at $(e_c\,fs' \to P, \rho')$ equals $\mathsf{eval}\ \rho'\,(e_c\,fs' \to P)$.

Then $\pi(L)$ is an augmented LTS as follows.

- The initial state is $(e_c\,fs \to P, \pi \circ \rho)$. This is the same as the initial state of L'.
- From this initial state, for each $(fs', \rho') \in S$, there is a τ-transition to $(e_c\,fs' \to P, \pi \circ \rho')$. Letting $\rho'' = \pi \circ \rho'$, this is equivalent to saying that for each $(fs', \rho'') \in \pi(S) = S'$, there is a τ-transition to $(e_c\,fs' \to P, \rho'')$. This is the same as the initial transitions of L'.
- For each $(fs', \rho') \in S$, the sub-LTS rooted at $(e_c\,fs' \to P, \pi \circ \rho')$ is equal to $\pi(\mathsf{eval}\ \rho'\,(e_c\,fs' \to P))$. But this equals $\mathsf{eval}\ (\pi \circ \rho')\,(e_c\,fs' \to P)$ using the result for the case that there are no \$-fields. Letting $\rho'' = \pi \circ \rho'$, this is equivalent to saying that for each $(fs', \rho'') \in \pi(S) = S'$, the sub-LTS rooted at $(e_c\,fs' \to P, \rho'')$ is equal to $\mathsf{eval}\ \rho''\,(e_c\,fs' \to P)$. This is the same as for L'.

Hence $\pi(L) = L'$, as required.

- Case of parallel composition.

  We define an operation on AGLTSs corresponding to parallel composition. $L_P\,[A \parallel B]_\rho\,L_Q$ runs AGLTSs L_P and L_Q in parallel, with alphabets $X = \mathsf{eval}\,\rho\,A$ and $Y = \mathsf{eval}\,\rho\,B$. If s_P and s_Q are states of L_P and L_Q, then the corresponding state of $L_P\,[A \parallel B]_\rho\,L_Q$ is labelled with $\mathit{simplify}(s_P\,[A \parallel B]_\rho\,s_Q)$.

  Then

$$
\begin{array}{ll}
 & \pi(\mathsf{eval}\,\rho\,(P[A \parallel B]Q)) \\
= & \pi(\mathsf{eval}\,\rho\,P\,[A \parallel B]_\rho\,\mathsf{eval}\,\rho\,Q) \\
= & \qquad \big(\pi \text{ distributes over parallel composition; see below}\big) \\
 & \pi(\mathsf{eval}\,\rho\,P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(\mathsf{eval}\,\rho\,Q) \\
= & \qquad \big(\text{inductive hypothesis}\big) \\
 & \mathsf{eval}(\pi \circ \rho)\,P\,[A \parallel B]_{\pi\circ\rho}\,\mathsf{eval}(\pi \circ \rho)\,Q \\
= & \mathsf{eval}(\pi \circ \rho)\,(P[A \parallel B]Q).
\end{array}
$$

Let

$$\begin{array}{rcl} X & = & \mathsf{eval}\,\rho\,A, \\ Y & = & \mathsf{eval}\,\rho\,B, \\ L & = & \mathsf{eval}\,\rho\,P\,[A \parallel B]_\rho\,\mathsf{eval}\,\rho\,Q, \\ L' & = & \pi(\mathsf{eval}\,\rho\,P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(\mathsf{eval}\,\rho\,Q). \end{array}$$

We show $\pi(L) = L'$. In L, the two components have alphabets X and Y. In L', the two components have alphabets $\mathsf{eval}(\pi \circ \rho)A = \pi(X)$ and $\mathsf{eval}(\pi \circ \rho)B = \pi(Y)$, using the inductive hypothesis.

Let $init_P$ and $init_Q$ be the initial states of $\mathsf{eval}\,\rho\,P$ and $\mathsf{eval}\,\rho\,Q$, respectively. Then the initial state of L is $simplify(init_P\,[A \parallel B]_\rho\, init_Q)$, and the initial state of $\pi(L)$ is

$$\begin{array}{rl} & \pi(simplify(init_P\,[A \parallel B]_\rho\, init_Q)) \\ = & simplify(\pi(init_P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(init_Q)). \end{array}$$

But the initial states of $\pi(\mathsf{eval}\,\rho\,P)$ and $\pi(\mathsf{eval}\,\rho\,Q)$ are $\pi(init_P)$ and $\pi(init_Q)$, so the initial state of L' is

$$simplify(\pi(init_P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(init_Q)),$$

which is the same as the initial state of $\pi(L)$.

Now suppose L has a transition

$$(simplify(s_P\,[A \parallel B]_\rho\, s_Q),\ a,\ simplify(s'_P\,[A \parallel B]_\rho\, s'_Q)).$$

Then $\pi(L)$ has a transition

$$\begin{array}{l} (\pi(simplify(s_P\,[A \parallel B]_\rho\, s_Q)),\ \pi(a),\ \pi(simplify(s'_P\,[A \parallel B]_\rho\, s'_Q))) \\ = (simplify(\pi(s_P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(s_Q)),\ \pi(a), \\ \qquad simplify(\pi(s'_P)\,[A \parallel B]_{\pi\circ\rho}\,\pi(s'_Q))). \end{array}$$

Further, we can show that L' has the same transition, by a case analysis on a. For example, if $a \in X - Y$, then $\mathsf{eval}\,\rho\,P$ has a transition (s_P, a, s'_P), and $s_Q = s'_Q$. Then $\pi(\mathsf{eval}\,\rho\,P)$ has a transition $(\pi(s_P), \pi(a), \pi(s'_P))$, and $\pi(s_Q) = \pi(s'_Q)$. But $\pi(a) \in \pi(X) - \pi(Y)$, so L' has the above transition. Other cases are similar. The converse holds similarly. Hence $\pi(L)$ and L' have the same transitions.

Hence $\pi(L) = L'$, as required.

- Other binary operators and hiding are similar, essentially since π distributes over each such operator.
- Timed sections are again similar, again since π distributes over the corresponding semantic operation (using the fact that $\pi(tock) = tock$).
- Case renaming. Let $\mathcal{R} = [[e_1 \leftarrow e_2 | stmts]]$. Without loss of generality, we assume that e_1 and e_2 are complete events (rather than channels), since any renaming can be easily rewritten into this form.

  We define the semantics of $\mathcal{R}$ using function evalRenaming, which produces a renaming relation, i.e. a set of pairs of events. Define *close* to extend a partial relation to a total relation over Σ as follows:

  $$close(R) \quad = \quad R \cup \{(a,a) | a \in \Sigma - \mathrm{dom}\, R\}.$$

  It is then easy to show $\pi \circ close = close \circ \pi$. We calculate as follows.

  $$\begin{array}{ll} & \pi(\mathsf{evalRenaming}\,\rho\,\mathcal{R}) \\ = & \pi(close(\{(\mathsf{eval}\,\rho'\ e_1, \mathsf{eval}\,\rho'\ e_2) | \rho' \in \mathsf{evalStmts}\,\rho\ stmts\})) \\ = & close(\{(\pi(\mathsf{eval}\,\rho'\ e_1), \pi(\mathsf{eval}\,\rho'\ e_2)) | \rho' \in \mathsf{evalStmts}\,\rho\ stmts\}) \\ = & \qquad \langle \text{inductive hypothesis} \rangle \\ & close(\{(\mathsf{eval}(\pi \circ \rho')\ e_1, \mathsf{eval}(\pi \circ \rho')\ e_2) | \rho' \in \mathsf{evalStmts}\,\rho\ stmts\}) \\ = & \qquad \langle \text{letting } \rho'' = \pi \circ \rho' \rangle \\ & close(\{(\mathsf{eval}\,\rho''\ e_1, \mathsf{eval}\,\rho''\ e_2) | \rho'' \in \pi(\mathsf{evalStmts}\,\rho\ stmts)\}) \\ = & \qquad \langle \text{part 5} \rangle \\ & close(\{(\mathsf{eval}\,\rho''\ e_1, \mathsf{eval}\,\rho''\ e_2) | \rho'' \in \mathsf{evalStmts}(\pi \circ \rho)\ stmts\}) \\ = & \mathsf{evalRenaming}(\pi \circ \rho)\,\mathcal{R}. \end{array}$$

  Note, in particular, that $\mathsf{eval}\,\rho'\ e_1$ and $\mathsf{eval}\,\rho'\ e_2$ are well defined (i.e. complete events, consistent with the channel declarations) iff $\pi(\mathsf{eval}\,\rho'\ e_1)$ and $\pi(\mathsf{eval}\,\rho'\ e_2)$ are well defined, by part 3.

  Now, let

  $$\begin{array}{rcl} L_0 & = & \mathsf{eval}\,\rho\,P, \\ L & = & \mathsf{eval}\,\rho\,(P\,\mathcal{R}), \\ R & = & \mathsf{evalRenaming}\,\rho\,\mathcal{R}, \\ L_0' & = & \mathsf{eval}(\pi \circ \rho)P, \\ L' & = & \mathsf{eval}(\pi \circ \rho)(P\,\mathcal{R}), \\ R' & = & \mathsf{evalRenaming}(\pi \circ \rho)\,\mathcal{R}. \end{array}$$

So $\pi(R) = R'$, and $\pi(L_0) = L'_0$ by the inductive hypothesis. We need to show $\pi(L) = L'$.

Each state of L is of the form $\mathit{simplify}(s\,\mathcal{R}_\rho)$ where s is the corresponding state of L_0. Then the corresponding state of $\pi(L)$ is $\pi(\mathit{simplify}(s\,\mathcal{R}_\rho)) = \mathit{simplify}(\pi(s)\,\mathcal{R}_{\pi\circ\rho})$. The corresponding state of L'_0 is $\pi(s)$, so the corresponding state of L' is also $\mathit{simplify}(\pi(s)\,\mathcal{R}_{\pi\circ\rho})$. Hence the states of $\pi(L)$ and L' have the same form. In particular, their initial states are equal.

For every transition (s, a, s') of L_0, and $(a, b) \in R$:

- L has a transition

$$(\mathit{simplify}(s\,\mathcal{R}_\rho), b, \mathit{simplify}(s'\,\mathcal{R}_\rho)).$$

- Hence $\pi(L)$ has a transition

$$\begin{array}{rl} & (\pi(\mathit{simplify}(s\,\mathcal{R}_\rho)), \pi(b), \pi(\mathit{simplify}(s'\,\mathcal{R}_\rho))) \\ = & (\mathit{simplify}(\pi(s)\,\mathcal{R}_{\pi\circ\rho}), \pi(b), \mathit{simplify}(\pi(s')\,\mathcal{R}_{\pi\circ\rho})). \end{array}$$

- L'_0 has a transition:

$$(\pi(s), \pi(a), \pi(s')).$$

- Since $(\pi(a), \pi(b)) \in R'$, L' has a transition

$$(\mathit{simplify}(\pi(s)\,\mathcal{R}_{\pi\circ\rho}), \pi(b), \mathit{simplify}(\pi(s')\,\mathcal{R}_{\pi\circ\rho})).$$

And conversely, all transitions of $\pi(L)$ and L' are as above. Hence $\pi(L)$ and L' have the same transitions.

Hence $\pi(L) = L'$.

- Case of indexed parallel composition. Below we use a semantic operator $\big\|[A]$, which takes a multiset of (LTS, environment) pairs $\{(L_1, \rho_1), \ldots, (L_n, \rho_n)\}$, and combines them in parallel, with LTS L_i given alphabet $\mathsf{eval}\,\rho_i\,A$. Given states $s_1, \ldots, s_n$ of the components, the corresponding state of the parallel composition is labelled $\mathit{simplify}(\big\|[A]\{(s_1, \rho_1), \ldots, (s_n, l_n)\})$.

We have

$$
\begin{array}{rl}
& \pi(\mathsf{eval}\,\rho\,(\|\ \mathit{stmts}\cdot[A]P)) \\
= & \pi(\big\|[A]\{(\mathsf{eval}\,\rho'\,P,\rho')|\rho'\in\mathsf{evalStmts}\,\rho\,\mathit{stmts}\}) \\
= & \qquad \big(\pi \text{ distributes over parallel composition; see below}\big) \\
& \big\|[A]\{(\pi(\mathsf{eval}\,\rho'\,P),\pi\circ\rho')|\rho'\in\mathsf{evalStmts}\,\rho\,\mathit{stmts}\} \\
= & \qquad \big(\text{inductive hypothesis}\big) \\
& \big\|[A]\{(\mathsf{eval}(\pi\circ\rho')\,P),\pi\circ\rho')|\pi\circ\rho'\in\pi(\mathsf{evalStmts}\,\rho\,\mathit{stmts})\} \\
= & \qquad \big(\text{letting } \rho''=\pi\circ\rho'\big) \\
& \big\|[A]\{(\mathsf{eval}\,\rho''\,P,\rho'')|\rho''\in\pi(\mathsf{evalStmts}\,\rho\,\mathit{stmts})\} \\
= & \qquad \big(\text{part 5}\big) \\
& \big\|[A]\{(\mathsf{eval}\,\rho''\,P,\rho'')|\rho''\in\mathsf{evalStmts}(\pi\circ\rho)\,\mathit{stmts})\} \\
= & \mathsf{eval}(\pi\circ\rho)\,(\|\ \mathit{stmts}\cdot[A]P).
\end{array}
$$

We show that π distributes over parallel composition, i.e. that if R is a finite multiset of environments, then

$$
\begin{array}{l}
\pi(\big\|[A]\{(\mathsf{eval}\,\rho'\,P,\rho')|\rho'\in R\}) = \\
\qquad \big\|[A]\{(\pi(\mathsf{eval}\,\rho'\,P),\pi\circ\rho')|\rho'\in R\}.
\end{array}
$$

We prove the labels correspond. Let $R=\{\rho'_1,\ldots,\rho'_n\}$, and let $s_1,\ldots,s_n$ be states of $\mathsf{eval}\,\rho'_1\,P,\ldots,\mathsf{eval}\,\rho'_n\,P$. Then the corresponding state of the left-hand side, above, is

$$
\begin{array}{rl}
& \pi(\mathit{simplify}(\big\|[A]\{(s_1,\rho_1),\ldots,(s_n,\rho_n)\})) \\
= & \mathit{simplify}(\big\|[A]\{(\pi(s_1),\pi\circ\rho_1),\ldots,(\pi(s_n),\pi\circ\rho_n)\}).
\end{array}
$$

Then $\pi(s_1),\ldots,\pi(s_n)$ are corresponding states of $\pi(\mathsf{eval}\,\rho'_1\,P)$, $\ldots,\pi(\mathsf{eval}\,\rho'_n\,P)$. Then the corresponding state of the right-hand side, above, is

$$
\mathit{simplify}(\big\|[A]\{(\pi(s_1),\pi\circ\rho_1),\ldots,(\pi(s_n),\pi\circ\rho_n)\})
$$

which is the same as for the left-hand side.

The fact that the transitions correspond can be proven in a similar way as for binary parallel composition, noting that in the left-hand side, the alphabet for $\mathsf{eval}\,\rho'\,P$ is $\mathsf{eval}\,\rho'\,A$, and in the right-hand side, the alphabet for $\pi(\mathsf{eval}\,\rho'\,P)$ is $\mathsf{eval}(\pi\circ\rho')\,A=\pi(\mathsf{eval}\,\rho'\,A)$, using the inductive hypothesis.

- Other indexed operators are similar, essentially because π distributed over each corresponding semantic operator.

We now consider compression functions. Each such compression function corresponds to a function *compress* on LTSs. By the remarks at the start of this item, it is enough to prove that $\pi(compress(L)) = compress(\pi(L))$ for each LTS L. Most of the results hold because the compression functions are mathematical functions (i.e. deterministic) that act in a data-independent way upon labels of transitions; i.e. the only operations performed on such labels are equality tests with other labels and with τ (and with *tock*, in the case of timed_priority).

- Case sbisim. Let *sbisim* be the corresponding function on AGLTSs. This function calculates the maximal strong bisimulation over the AGLTS, and reduces the AGLTS according to this bisimulation. Clearly, if two states are strongly bisimilar in L, the corresponding states are strongly bisimilar in $\pi(L)$. Hence if E is a bisimulation-equivalence class of states in L, then $\pi(E)$ is a bisimulation-equivalence class of states in $\pi(L)$.

  Each state s of $sbisim(L)$ corresponds to a bisimulation-equivalence class E of states in L; s is labelled with $simplify(sbsim\, E)$. By the above, there is a state $\pi(s)$ of $sbisim(\pi(L))$ corresponding to the equivalence class $\pi(E)$; then $\pi(s)$ is labelled with $simplify(sbsim\,(\pi(E))) = \pi(simplify(sbsim\, E))$, as required.

  It is then easy to show that the transitions of $sbisim(L)$ and $sbisim(\pi(L))$ correspond appropriately.

- The other bisimulation-based compressions (dbisim and wbisim) are similar.

- Case normal. This case is similar to previous cases. It is straightforward to see that the prenormal forms of L and $\pi(L)$ are themselves π-bisimilar.

- Case tau_loop_factor. This case is again similar: corresponding states are within τ-loops in L and $\pi(L)$, and so are identified.

- Case diamond. This compression starts by applying tau_loop_factor. Letting *tau_loop_factor* be the corresponding semantic operation, the previous case tells us that $\pi \circ tau_loop_factor = tau_loop_factor \circ \pi$.

  The compression then applies a function *pre_diamond*. Given a GLTS $L = (S, \Delta, init, \mathsf{minaccs}, \mathsf{div})$, *pre_diamond* calculates a GLTS $L' = (S', \Delta', init, \mathsf{minaccs}', \mathsf{div}')$ where

$$S' = \{s_2 | \exists s_1 \in S, a \in \Sigma \cdot s_2 \in min\{s | s_1 \xrightarrow{\tau^* a}_L s\}\}.$$

Here $min(X)$ represents the τ-minimal states of X:

$$min(X) \quad = \quad \{s | s \in X \wedge \nexists\, s' \in X \cdot s' \overset{\tau^+}{\longmapsto} s\},$$

Further, for $a \in \Sigma$,

$$s_1 \xrightarrow{a}_{L'} s_2 \quad \textit{iff} \quad s_2 \in min\{s | s_1 \overset{\tau^* a}{\longmapsto}_L s\}.$$

Finally, minaccs$'$ and div$'$ are the restrictions of minaccs and div to S'.

If is then clear that $\pi \circ pre_diamond = pre_diamond \circ \pi$: if $pre_diamond$ retains a state s of L, then it also retains the state $\pi(s)$ of $\pi(L)$; and if it includes a transition $s_1 \xrightarrow{a} s_2$ when applied to L, it also includes a transition $\pi(s_1) \xrightarrow{\pi(a)} \pi(s_2)$ when applied to $\pi(L)$.

Finally, diamond restricts to reachable states. It is clear that π also commutes with this operation.

- Case explicate and lazyenumerate. These functions simply change the representation of a process, from a supercombinator to a low-level machine. As such, the result holds trivially.
- Case prioritise. Let *prioritise* be the corresponding semantic function on LTSs. It is enough to show that

  $$\pi(prioritise(L, \langle S_1, \ldots, S_N \rangle)) = \\ prioritise(\pi(L), \langle \pi(S_1), \ldots, \pi(S_N) \rangle),$$

  for each LTS L and sets of events $S_1, \ldots, S_N$.

  Suppose in L there is a transition from state s labelled with event $a \in S_i$. In $\pi(L)$, the corresponding state $\pi(s)$ has a transition labelled with $\pi(a) \in \pi(S_i)$. The transition in L is retained in the prioritised LTS iff $i = 1$ or s has no transition in L labelled with an event in $\bigcup_{j=1}^{i-1} S_j \cup \{\tau, \surd\}$; which holds iff $i = 1$ or $\pi(s)$ has no transition in $\pi(L)$ labelled with an event in $\bigcup_{j=1}^{i-1} \pi(S_j) \cup \{\tau, \surd\}$; which holds iff the transition from $\pi(s)$ in $\pi(L)$ is retained.

  A similar argument applies for transitions labelled with τ or $\surd$, or with events not in $\bigcup_{j=1}^{N} S_j$.

  Hence $prioritise(L, \langle S_1, \ldots, S_N \rangle)$ and $prioritise(\pi(L), \langle \pi(S_1), \ldots, \pi(S_N) \rangle)$ are π-bisimilar, as required.
- prioritise_nocache, prioritisepo and timed_priority are similar to the previous case (using the fact that $\pi(tock) = tock$ for timed_priority).

- Case trace_watchdog and failure_watchdog. These cases hold since the transformations described in [GMR+03] commute with π.

5. We prove

$$\pi(\mathsf{evalStmts}\,\rho\,\mathit{stmts}) \;=\; \mathsf{evalStmts}(\pi\circ\rho)\mathit{stmts}.$$

in the case of evalStmts producing a *sequence* of environments; the case of a multiset of environments is similar. We prove this result by induction over *stmts*.

- Case $\langle\rangle$.

$$\pi(\mathsf{evalStmts}\,\rho\,\langle\rangle) = \langle\pi\circ\rho\rangle = \mathsf{evalStmts}(\pi\circ\rho)\langle\rangle.$$

- Case $\langle \mathit{pat} \leftarrow e\rangle^\frown \mathit{stmts}$.

$$\begin{array}{rl}
& \pi(\mathsf{evalStmts}\,\rho\,(\langle \mathit{pat} \leftarrow e\rangle^\frown \mathit{stmts})) \\
= & \pi(\mathit{concat}\langle \mathsf{evalStmts}(\mathsf{bind}\,\rho\,\mathit{pat}\,v)\;\mathit{stmts} \mid \\
& \qquad\qquad v \leftarrow \mathsf{eval}\,\rho\,e, \mathsf{matches}\,\rho\,\mathit{pat}\,v\rangle) \\
= & \mathit{concat}\langle \pi(\mathsf{evalStmts}(\mathsf{bind}\,\rho\,\mathit{pat}\,v)\;\mathit{stmts}) \mid \\
& \qquad\qquad v \leftarrow \mathsf{eval}\,\rho\,e, \mathsf{matches}\,\rho\,\mathit{pat}\,v\rangle \\
= & \qquad \big(\text{inductive hypothesis}\big) \\
& \mathit{concat}\langle \mathsf{evalStmts}(\pi\circ\mathsf{bind}\,\rho\,\mathit{pat}\,v)\;\mathit{stmts} \mid \\
& \qquad\qquad v \leftarrow \mathsf{eval}\,\rho\,e, \mathsf{matches}\,\rho\,\mathit{pat}\,v\rangle \\
= & \qquad \big(\text{parts 1 and 2}\big) \\
& \mathit{concat}\langle \mathsf{evalStmts}(\mathsf{bind}(\pi\circ\rho)\;\mathit{pat}\;(\pi(v)))\;\mathit{stmts} \mid \\
& \qquad\qquad v \leftarrow \mathsf{eval}\,\rho\,e, \mathsf{matches}(\pi\circ\rho)\;\mathit{pat}\;(\pi(v))\rangle \\
= & \qquad \big(\text{putting } w = \pi(v)\big) \\
& \mathit{concat}\langle \mathsf{evalStmts}(\mathsf{bind}(\pi\circ\rho)\;\mathit{pat}\;w)\;\mathit{stmts} \mid \\
& \qquad\qquad w \leftarrow \pi(\mathsf{eval}\,\rho\,e), \mathsf{matches}(\pi\circ\rho)\;\mathit{pat}\;w\rangle \\
= & \qquad \big(\text{part 4}\big) \\
& \mathit{concat}\langle \mathsf{evalStmts}(\mathsf{bind}(\pi\circ\rho)\;\mathit{pat}\;w)\;\mathit{stmts} \mid \\
& \qquad\qquad w \leftarrow \mathsf{eval}(\pi\circ\rho)\;e, \mathsf{matches}(\pi\circ\rho)\;\mathit{pat}\;w\rangle \\
= & \mathsf{evalStmts}(\pi\circ\rho)\;(\langle \mathit{pat} \leftarrow e\rangle^\frown \mathit{stmts}).
\end{array}$$

- Case $\langle \mathit{pred}\rangle^\frown \mathit{stmts}$.

$$\begin{array}{rl}
& \pi(\mathsf{evalStmts}\,\rho\,(\langle \mathit{pred}\rangle^\frown \mathit{stmts})) \\
= & \pi(\text{if}\,\mathsf{eval}\,\rho\,\mathit{pred}\text{ then }\mathsf{evalStmts}\,\rho\,\mathit{stmts}\text{ else }\langle\rangle) \\
= & \qquad \big(\mathsf{eval}\,\rho\,\mathit{pred}\text{ is invariant under }\pi\big) \\
& \text{if}\,\pi(\mathsf{eval}\,\rho\,\mathit{pred})\text{ then }\pi(\mathsf{evalStmts}\,\rho\,\mathit{stmts})\text{ else }\pi\langle\rangle \\
= & \qquad \big(\text{item 4; inductive hypothesis}\big) \\
& \text{if}\,\mathsf{eval}(\pi\circ\rho)\;\mathit{pred}\text{ then }\mathsf{evalStmts}(\pi\circ\rho)\;\mathit{stmts}\text{ else }\langle\rangle \\
= & \mathsf{evalStmts}(\pi\circ\rho)\;(\langle \mathit{pred}\rangle^\frown \mathit{stmts}).
\end{array}$$

□