A Representative Function Approach to Symmetry Exploitation for CSP Refinement Checking

Nick Moffat[1], Michael Goldsmith[2], Bill Roscoe[3]

[1]QinetiQ, Malvern Technology Centre, St Andrews Road, Malvern. UK
and Kellogg College, University of Oxford.
[2]Formal Systems (Europe) Ltd, Oxford. UK
and Worcester College, University of Oxford.
[3]Oxford University Computing Laboratory, Parks Road, Oxford. UK.
nick.moffat@kellogg.ox.ac.uk, michael@fsel.com, bill@comlab.ox.ac.uk

Abstract. Effective temporal logic model-checking algorithms exist that exploit symmetries arising from parallel composition of multiple identical components. These algorithms often employ a function *rep* from states to representative states under the symmetries exploited. We adapt this idea to the context of refinement checking for the process algebra CSP. In so doing, we must cope with refinement-style specifications. The main challenge, though, is the need for access to sufficient local information about states to enable definition of a useful *rep* function, since compilation of CSP processes to Labelled Transition Systems (LTSs) renders state information a global property instead of a local one. Using a structured form of implementation transition system, we obtain an efficient symmetry exploiting CSP refinement-checking algorithm, generalise it in two directions, and demonstrate all three variants on simple examples.

Keywords: Symmetry, CSP, Process Algebra, Refinement, Model Checking.

1. Introduction

Model checking suffers from the state explosion problem, which is the tendency for state space to grow exponentially in size (number of states) as the size of the model (system description in the modelling language) grows. A simple example is the exponential state space growth that can occur when adding parallel components.

A popular approach to combating the state explosion problem is to exploit state space symmetries. This approach has received much attention in the context of temporal logic state-based model checking ([1] contains a survey), but little has been published in the context of refinement checking ("refinement-style model checking") for process algebras.

For temporal logic model checking, effective algorithms exist that exploit symmetries arising from parallel composition of multiple identical components. The most common approach uses a function *rep* from states to representative states and requires full symmetry of the model and the property. We adapt this idea for Communicating Sequential Processes (CSP) [2,3] refinement checking. The main challenge, which may be considered a significant obstacle, is the need for local information about states to enable use of a *rep* function; compiling CSP processes to Labelled Transition Systems (LTSs) makes state information a global property, not a local one.

By exploiting a richer notation than LTSs, namely 'structured machines' (already used internally by the FDR [4] refinement checker for other reasons), we can define a suitable *rep* function. We obtain a refinement checking algorithm that explores a reduced state space efficiently for systems that have parallel components.

We then generalise this algorithm, dropping the requirement for full symmetry and then making it less restrictive, in a different sense, about the need for property symmetries. Restricting to symmetric temporal logic formulae effectively requires that the future behaviour is always symmetric, regardless of what has happened in the past; in contrast, our second generalisation only needs the specification process (corresponding to a property formula) to express symmetric behaviour starting at the initial state.

An earlier paper [5] outlined some of our work aimed at efficient identification of CSP process symmetries, including an approach to exploit symmetries when refinement checking. The exploitation approach in this paper is quite different.

For brevity, we restrict attention to refinement in CSP's traces model, which allows one to check safety properties; the algorithms extend to other semantic models.

Section 2 provides background regarding the process algebra CSP and refinement checking between CSP processes. Section 3 defines CSP process symmetry. Section 4 recaps the representative function approach to symmetry exploitation for temporal logic model checking. Section 5 describes structured machines and identification of their symmetries. Section 6 gives our basic symmetry exploiting refinement-checking algorithm and Section 7 extends it in two directions. Section 8 presents experimental results and Section 9 concludes. An appendix contains correctness proofs.

2. CSP Language, Refinement, LTSs and Refinement Checking

2.1 CSP and Refinement

Process algebras allow systems to be modelled as processes, which may be atomic (such as CSP's STOP process) or may be defined as compositions of other, child, pro-

cesses using available process operators. CSP [2,3] has a variety of process operators, including: interleaving (|||), generalised parallel ($||_X$), alphabetised parallel ($_X||_Y$), where processes must synchronise on alphabet X or alphabet X∩Y; internal choice (∏); external choice(□); hiding (\X); and renaming([[R]]), for relation R on events.

Refinement of a process Spec by a process Impl amounts to all behaviours (of some particular kind, such as the finite traces) of Impl being behaviours of Spec. In the traces semantic model, $\mathcal{T}$, a behaviour is a finite trace the process can perform.

2.2 Labelled Transition Systems

A widely used operational form for CSP processes is the Labelled Transition System (LTS). An LTS is a tuple (S,T,s_0) where S is a set of states (sometimes called nodes), $T : S \times \Sigma \times S$ (for universal event set Σ) is a *labelled* transition relation, and state s_0 is the initial state. An LTS path $\langle s_0, e_1, s_1, \ldots, e_n, s_n \rangle$ has the trace $\langle e_1, \ldots, e_n \rangle$.

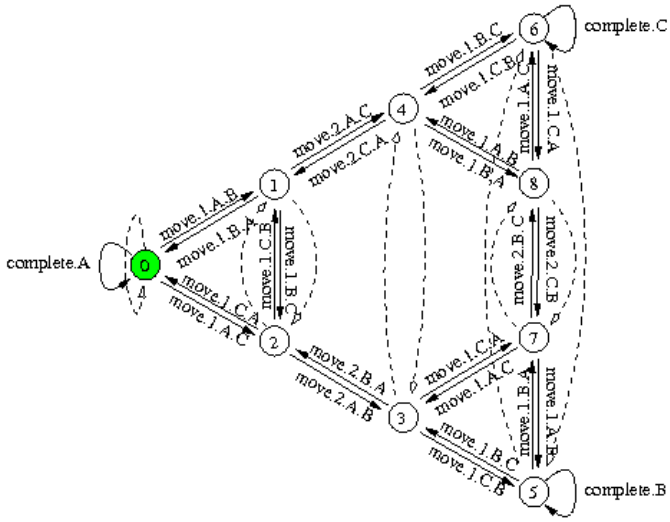

Fig. 1. An LTS for Towers of Hanoi with 3 poles and 2 discs. The initial state is shaded. Dashed arrows show a (B C)-bisimulation relation, which can be ignored until Section 3.

2.3 Refinement Checking

CSP refinement-checking algorithms operate over transition systems T_{Spec} and T_{Impl} of a specification process, Spec, and an implementation process, Impl. Each transition system is a compiled form of the process and supports calculation of the initial state

and the set of transitions. Transition system T_{Spec} is required to be an LTS in normal form [3,6], to ensure no two paths of T_{Spec} with the same trace end at different states.

The usual refinement-checking algorithm [6] explores the product space of (spec state, impl state) pairs such that a common trace can take the spec and impl to the respective states. Exploration starts at the initial state pair and continues until a counterexample has been reached or all sucessors of reached pairs have been found.

Although it is usual to refer to these as 'pairs' (and we do so throughout), refinement-checking algorithms gernerally record tuples of at least four values; they explore a product space and record extra information as they go, as explained below.

At each pair reached: (i) 'compatibility' of the implementation state with the specification state is checked; (ii) all successor state pairs are added to the set of pairs seen so far. The compatibility test depends on the semantic model used for the check; for the traces model it simply checks that all events labelling outgoing transitions of the impl state appear among the labels on outgoing transitions from the spec state.[1]

If an incompatible state pair is reached a counterexample trace to this pair is recovered by stepping through the implementation transition system backwards until its initial state is reached; this is possible since the identifier of a parent pair is recorded with each newly reached state pair, plus an event from the parent to this pair.

3. CSP Symmetry and Permutation Bisimulations

3.1 Algebraic and Denotational Permutation Symmetry

In a process algebraic context symmetry acts principally on events/actions; 'states' are identified with particular sets of possible future behaviours (indeed, a state represents a process). Event permutations lift naturally to state (or process) permutations.

Perhaps the simplest definition of CSP symmetry is in the algebraic semantics. Let σ be any permutation of events in some universal event set Σ, where we insist that $\tau\sigma = \tau$ (i.e., that the special CSP event τ, denoting an internal action, is unaffected by σ). Then we say that a process P is σ-*symmetric* in the traces semantic model, $\mathcal{T}$, when $P =_{\mathcal{T}} P\sigma$. Throughout the paper, $P\sigma$ denotes the functional renaming[2] of P according to σ, so $P\sigma$ is the process that can perform event $x\sigma$ whenever P can perform

1 Internal transitions, labelled by special event τ, are removed from the spec transition system by normalising it, which ensures that no two spec states are reachable by the same trace. Pair (u,v') is treated as a successor to (u,v) if v has a τ transition to v'. For details, see [3] or [6].

2 Functional renaming for injective functions is defined on page 87 of [3].

an event x (equivalently, using the relational renaming operator, we may write Pσ as P[[σ]]). Also, $=_{\mathcal{T}}$ denotes traces equivalence.

Notice that we do not restrict σ to preserve channels: we allow permutations that map, say, a.2 to b.44. However, we may anticipate that a common form of symmetry permutation will be the canonical lifting of a datatype permutation: e.g., if σ' is a permutation of a datatype D and c is a CSP channel carrying data of type D, then the canonical lifting of σ' to an event permutation σ maps events c.x to c.(xσ). When events have complex datatypes, the canonical lifting applies the datatype permutation to all fields of that type. Sometimes we will denote an event permutation σ by the datatype permutation σ' such that σ is the canonical lifting of σ'.

The equivalent denotational definition of CSP symmetry is also straightforward: process P is σ-symmetric in $\mathcal{T}$ if the mathematical object that P denotes in $\mathcal{T}$ (the set of finite traces of P) is itself symmetric according to σ, that is, if permuting this object by the lifted permutation leaves it unchanged.

3.2 Operational Permutation Symmetry

Before defining LTS symmetries we remark that, as one would expect, permutation symmetries of LTSs imply the same symmetries of the processes they represent (though structurally asymmetric LTSs can represent symmetric processes).

Our definition of LTS symmetries uses the more general notion of *permutation bisimulations*, or *pbisims* for short, which were introduced in [5]. Permutation bisimulation extends the classical notion of (strong) bisimulation [7,8]. For permutation σ, a binary relation R over the nodes S of an LTS L is a σ-*bisimulation* if R is a σ-simulation of L and R^{-1} is a σ^{-1}-simulation of L. Permutation simulation extends the classical notion of simulation: classical simulation requires that (1) if $pRp' \wedge p \text{ –(a)→ } q \in L$, then $\exists\, p' \text{ –(a)→ } q' \in L$ s.t. qRq'; and (2) $\forall p \in S, \exists\, p' \in S$ s.t. pRp'; instead, σ-simulation requires $p' \text{ –(a}\sigma\text{)→ } q'$ in the consequent of the first condition.

Here we treat τ events the same way as visible events; when a = τ we require that $p' \text{ –(}\tau\text{)→ } q'$ (recall that our event permutations do not affect τ). A possible generalisation is to consider the permutation analogue of weak bisimulation [8], but we use (strong) bisimulation here; our simpler definition admits fewer symmetries.

Two nodes are σ-*bisimilar* if there is a σ-bisimulation that relates them. Permutation bisimilarity captures the equivalence of processes represented by LTS nodes in the following sense: if state x is σ-*bisimilar* to state y, then the process represented by y equals the process represented by x (P, say) renamed by σ (i.e., Pσ).

LTS symmetry can be defined in terms of permutation bisimulation: for permutation σ, an LTS L for a process P is σ-*symmetric* iff some σ-bisimulation relates L's

initial state s_0 to itself (i.e., s_0 is σ-bisimilar to itself). The LTS of Figure 1 is (B C)-symmetric as the (B C)-bisimulation shown relates the initial node to itself.

3.3 Group Symmetry

The above definitions lift easily to group symmetry, as follows. Let G be a group of event permutations. Then a process P, or LTS L, is G-*symmetric* if it is σ-symmetric for each σ in G. (It is clearly sufficient to be σ-symmetric for each of a set of generators of G.)

4. Symmetry and Temporal Logic Model Checking

This section summarises the temporal logic model-checking problem and outlines what may be called the "representative function" approach to symmetry exploitation, broadly following the presentation in [1].

A *Kripke structure* over a set *AP* of atomic propositions is a tuple $M = (S,R,L,S_0)$ where: (1) S is a non-empty finite set of states; (2) $R \subseteq S \times S$ is a total transition relation; (3) L: $S \rightarrow 2^{AP}$ is a mapping that labels each state in S with the set of atomic propositions true in that state; and (4) $S_0 \subseteq S$ is a set of initial states. Temporal logic model checking determines whether a given Kripke structure M satisfies a given formula φ expressed in some temporal logic (often CTL* or one of its sub-logics LTL or CTL); this is denoted $M \models \varphi$ and amounts to φ holding in each initial state of M.

The representative function approach to symmetry exploitation in this context is applicable with *symmetric* formulae φ w.r.t. a group G of *automorphisms* of M (which are state permutations that preserve the transition relation R). A symmetric CTL* formula φ w.r.t. a group G of state permutations is one where, for every maximal propositional subformula f in φ, f holds in a state s iff it holds in state λ(s) for each λ in G. So, symmetric formulae are such that the validity of each maximal propositional subformula is unaffected by permutations in G.

Further, this symmetry exploitation approach requires that M represents a parallel composition of identical components and that each element of G permutes the values of state variables according to some permutation of the component indices.

The idea is to use a 'representative' function, usually called *rep*, chosen according to a symmetry group G s.t. φ is known to be symmetric w.r.t. G. This function maps each state s of the Kripke structure to a representative state *rep*(s) in the same G-orbit as s, where G-orbits are equivalence classes induced by the relation "is related to by

some permutation in G". That is, *rep* maps each state to a representative state to which it is related by some permutation in G.

A *quotient* Kripke structure $M_G = (S_G, R_G, L_G, S^0_G)$ is generated where: $S_G = \{rep(s) \mid s \in S\}$, $R_G = \{(rep(s), rep(s')) \mid (s,s') \in R\}$, $L_G(rep(s)) = L(rep(s))$, $S^0_G = \{rep(s) \mid s \in S_0\}$. The quotient structure is then checked against the original formula φ. It has been proved that $M \models \varphi$ iff $M_G \models \varphi$ [9,10]. The quotient check is up to n! times faster than the original, for n identical components, and can consume significantly less memory.

5. Structured Machines and their Symmetries

5.1 Structured Machines

A *structured machine* represents an LTS as an operator tree with a CSP process operator at each non-leaf node and an LTS at each leaf. Alphabets are associated with child nodes as appropriate for the parent node's CSP operator (i.e., according to the number of operand alphabets). Structured machines reflect an upper part of a process expression's algebraic structure. They are called *configurations* in [3,6]. They can be much smaller than equivalent LTSs, being linear in the number of component processes of indexed parallel composition; they can often be operated on very efficiently.

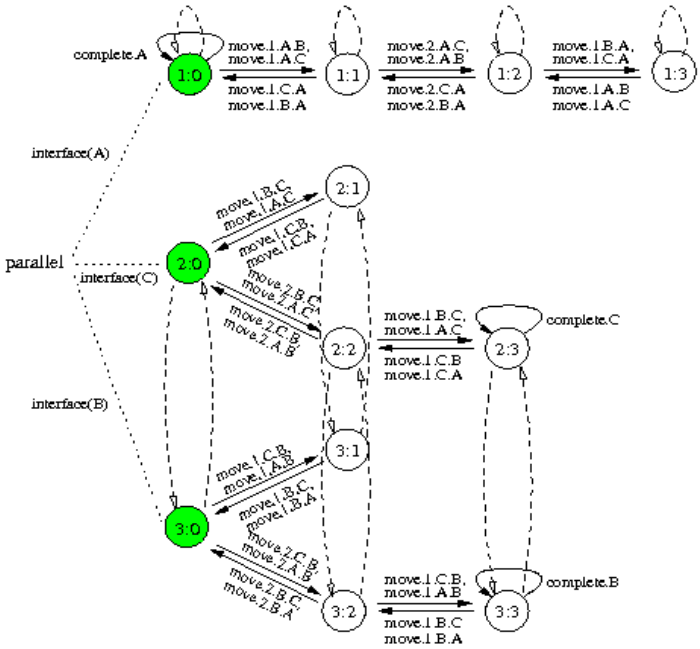

Fig. 2. A structured machine for Towers of Hanoi with 3 poles and 2 discs, with alphabetised parallel at the root. A (B C)-bisimulation relation on LTS nodes is shown using dashed arrows.

The example in Figure 2 represents a process P = || p : PEGS @ [interface(p)] POLE(p) for a datatype PEGS = {A,B,C} and alphabet- and process-valued functions interface/1 and POLE/1. The definition of interface/1 is not shown, but that of POLE/1 is implied by the three leaf LTSs in the right-hand portion of Figure 2, one for each of the child processes POLE(p). The initial node of each leaf LTS is shaded. The same process P is represented explicitly by the LTS of Figure 1.

For simplicity, we consider only single-configuration processes, which has the effect of allowing only a subset of CSP process operators outside recursive definitions: parallel operators, hiding and renaming. In practice many processes have this form. A structured machine with a top level parallel operator has states in tuple form – each component denotes the local state of a particular leaf LTS. See [3,6] for more details.

5.2 Structured Machine Symmetries

Symmetries of a structured machine can be represented conveniently using permutation bisimulations between the nodes of its leaf LTSs, as demonstrated by Figure 2. A single permutation bisimulation may relates nodes of a single leaf LTS, or nodes of

different leaf LTSs. Permutation bisimulations can often be found by exploiting the structure of CSP process expressions, as explained below. Operational and algebraic approaches to identifying (finding/checking) symmetries and permutation bisimulations were discussed briefly in [5]. The algebraic approach is well suited to efficient identification of structured machine symmetries, so it is described here.

Table 1 expands the table in [5]. It gives a selection of rules that relate trace symmetries of processes to those of sub-processes and alphabets. Due to space limitations, Table 1 is incomplete and we omit our proofs of these results. Throughout, σ is taken to be an event permutation. For an alphabet X (A, H or A(i) in the table), $X\sigma$ denotes the set $\{x\sigma \mid x \in X\}$. In rules 9-13, σ permutes indices in the set I and permutes events according to the corresponding canonical lifting.

Rules 4 and 5 are alternative instances of rule 10 for two sub-processes: rule 4 is obtained when σ maps P to P and Q to Q, and rule 5 is obtained when σ swaps P and Q; rechristening P as P(1) and Q as P(2), the distinction is how σ acts on the indices 1 and 2 in rule 10, i.e. on whether σ maps 1 to 1 and 2 to 2, or swaps 1 and 2. Rule 3 is an instance of rule 12. In this way, specialised rules can be derived easily from rules 9-13. Rule 8 uses the 'exact alphabet' function α.

Rules 9-13 allow one to infer symmetries that are (liftings of) index permutations. These rules can be generalised, replacing $\forall i \in I,\ P(i\sigma) =_{\mathcal{T}} P(i)\sigma$ by $\exists$ an index permutation $\rho \bullet \forall i \in I,\ P(i\rho) =_{\mathcal{T}} P(i)\sigma$, where ρ permutes indices and σ permutes events.

Most of the rules are deliberately approximate. Informally, they only allow 'easy' symmetries to be identified – symmetries one would expect to hold 'at first glance'. This helps to make them simple and easy to implement. Reasoning with such rules will generally miss some symmetries, but we expect they would find most that arise in practice. Some approximation is necessary, as finding all symmetries would in general be too computationally demanding.

One approach to cope with recursive definitions would be to calculate conditions iteratively and terminate on reaching a fixed point. This would require some supporting theory to argue termination and perhaps uniqueness of the fixed point. We take the simpler approach of identifying symmetry of recursive processes operationally [5], by examining transition systems (LTSs, in fact) that represent them.

We have developed a prototype tool which implements extended versions of these rules, for deciding whether any given processes $Proc_1$ and $Proc_2$ are mutually permutation symmetric by a given event permutation σ (i.e., whether $Proc_1\sigma =_{\mathcal{T}} Proc_2$). By choosing $Proc_1 = P(x)$ and $Proc_2 = P(x\sigma)$, the extended rules can also be used for checking *permutation transparency* conditions $P(x)\sigma =_{\mathcal{T}} P(x\sigma)$. Such conditions occur at lines 2 and 9-13 of Table 1.

	Proc	op	(Proc)σ $=_{\mathcal{T}}$ Proc	Explanation of Proc
1	STOP	⇔	True	STOP has only the empty trace
2	?x : A → P(x)	⇔	Aσ = A ∧ ∀x∈A, P(xσ)$=_{\mathcal{T}}$ P(x)σ	Accept x in A, then act as P(x)
3	P □ Q	⇐	Pσ $=_{\mathcal{T}}$ P ∧ Qσ $=_{\mathcal{T}}$ Q	External choice between P, Q
4	P [| A |] Q	⇐	Pσ $=_{\mathcal{T}}$ P ∧ Qσ $=_{\mathcal{T}}$ Q ∧ Aσ = A	P and Q synchronised on A
5	P [| A |] Q	⇐	Pσ $=_{\mathcal{T}}$ Q ∧ Qσ $=_{\mathcal{T}}$ P ∧ Aσ = A	P and Q synchronised on A
6	P ; Q	⇐	Pσ $=_{\mathcal{T}}$ P ∧ Qσ $=_{\mathcal{T}}$ Q	P then (on termination) Q
7	P \ H	⇐	Pσ $=_{\mathcal{T}}$ P ∧ Hσ $=_{\mathcal{T}}$ H	P with events in H hidden
8	P [[R]]	⇐	∃ρ • Pρ $=_{\mathcal{T}}$ P ∧ ∀a∈αP, aRb ⇒ (aρ)R(bσ^{-1})	P renamed by event relation R
9	||| i ∈ I • P(i)	⇐	σ:I→ I ∧ ∀ i ∈ I, P(iσ) $=_{\mathcal{T}}$ P(i)σ	Interleaving of all 'P(i)'s
10	[| A |] i ∈ I • P(i)	⇐	σ:I→ I ∧ ∀ i ∈ I, P(iσ) $=_{\mathcal{T}}$ P(i)σ ∧ Aσ = A	Generalised parallel of 'P(i)'s
11	|| i∈I • [|A(i)|] P(i)	⇐	σ:I→ I ∧ ∀i ∈ I, P(iσ) $=_{\mathcal{T}}$ P(i)σ ∧ A(iσ) $=_{\mathcal{T}}$ A(i)σ	'P(i)'s synchronised on 'A(i)'s
12	□ i ∈ I • P(i)	⇐	σ:I→ I ∧ ∀ i ∈ I, P(iσ) $=_{\mathcal{T}}$ P(i)σ	External choice of all 'P(i)'s
13	|~| i ∈ I • P(i)	⇐	σ:I→ I ∧ ∀ i ∈ I, P(iσ) $=_{\mathcal{T}}$ P(i)σ	Non-det choice of all 'P(i)'s

Table 1. Some exact (1 and 2) and sufficient (3-13) conditions for CSP process symmetry.

The most significant rules for this paper are those for the replicated parallel operators: rows 9-11 in the table. This is because structured machines with these operators have effective state spaces with states being tuples of local states, one per child machine. Sections 6 and 7 will define *rep* functions on such tuple states.

An alternative, promising approach to finding permutation transparencies (and so symmetries) is to look for data independence (d.i.) [11] of a parametrized process expression P(x) in the type X, say, of its parameter. This is because d.i. – a simple syntactic property – implies transparency with respect to all permutations of the type.

It appears possible to liberalise the notion of data independence to yield a syntactic characterisation of a large class of transparent processes: one would remove conditions (notably banning of parallel composition indexed by the d.i. type) designed to prevent d.i. processes 'counting' the datatype. Having identified transparency syntactically – using standard d.i. or a liberalised version – one could deduce symmetries using the rules above. This is motivated further in [5], in particular for d.i. index sets. However, this symmetry identification short cut is outside the scope of the paper.

The algebraic rules in Table 1 can be extended to yield a compiled representation of the process as a structured machine, plus permutation bisimulation relations on the nodes of its LTSs (not just knowledge of whether the process is symmetric). Such permutation bisimulations will justify the *rep* functions defined in the next sections.

6. Basic Symmetry Exploiting Algorithm

Recall that symmetry of a CTL* formula *f* w.r.t. group G means *f* never discriminates between mutually symmetric behaviours, regardless of the number of steps already taken. The corresponding condition on a specification process is that it is G-symmetric *in each state* (each process to which it can evolve is G-symmetric); if this holds we say the specification process is *universally G-symmetric*. Similarly, a specification transition system (LTS or structured machine) is universally G-symmetric if each of its states is G-symmetric, implying universal G-symmetry of the process it represents.

The *product space* for a specification Spec with states S_{Spec} and implementation Impl with states S_{Impl} is the subspace of $S_{Spec} \times S_{Impl}$ reachable under lock-step synchronisation on all visible events. This state space is explored during a standard refinement check; each 'state' of the product space is really a state pair (u,v), say, where u is a specification state and v is an implementation state. A *path* through a transition system is an alternating sequence $\langle s_0,e_1,s_1,...,e_n,s_n\rangle$ of states and events, starting and ending with states s.t. for each $0 \le i < n$, there is a transition from s_i to s_{i+1} labelled e_{i+1}.

A *twisted path* through a Spec-Impl product space is a sequence $\langle s_0,e_1,\sigma_1,s_1,...,e_n,\sigma_n,s_n\rangle$ of (product) states, events and permutations, starting and ending with states, with the following well formedness condition between successive states: $\forall\ 0 \le i < n$, there is a product space transition labelled e_{i+1} from $s_i = (u_i,v_i)$ to $\text{pre-}s_{i+1} = s_{i+1}\sigma_{i+1}^{-1} = (u_{i+1}\sigma_{i+1}^{-1},v_{i+1}\sigma_{i+1}^{-1})$. Intuitively, non-trivial permutations σ 'twist' the search away from paths that the usual refinement checking algorithm would follow.

Given a function *repPair* from state pairs to state pairs, a *repPair-twisted path* is a twisted path $\langle s_0,e_1,\sigma_1,s_1,...,e_n,\sigma_n,s_n\rangle$ such that $\forall\ 0 < i \le n$, $s_i = repPair(\text{pre-}s_i)$, where $\text{pre-}s_i = s_i\sigma_i^{-1}$. (We let *repPair* return a permutation too, which this definition ignores.)

6.1 TwistedCheck

The symmetry exploiting algorithms will be defined in terms of a curried function TwistedCheck (see Figure 3) parametrized by a function *repPair*. Ignoring counterexample recovery for now, TwistedCheck(*repPair*) is obtained by changing the usual refinement checking algorithm (Section 2.3) as follows: during exploration, instead of recording a reached state pair (u,v), record (rep_u, rep_v) where $(rep_u, rep_v, \sigma) = repPair(u,v)$. Note that TwistedCheck(*repPair*) does not need Spec or Impl G-symmetry.

TwistedCheck(*repPair*) explores the Spec-Impl product space by following *repPair*-twisted paths – each non-trivial permutation σ returned by *repPair* re-directs the search to continue from $(\texttt{repnext}_u, \texttt{repnext}_v)$ instead of from $(\texttt{next}_u, \texttt{next}_v)$. Each such σ is recorded for counterexample recovery.

```
TwistedCheck(repPair)(T_Spec, T_Impl)
 1 Input:  Normal Spec transition system T_Spec with states S_Spec
 2         Impl structured machine T_Impl with states S_Impl
 3         repPair: S_Spec×S_Impl -> S_Spec×S_Impl×G
 4 Output: A repPair-twisted counterexample or 'REFINES'
 5
 6 function recover2(state,vparent,e,π)
 7   if defined(vparent) then
 8     (u,v,vparent2,e2,σ) := Seen[vparent];
 9     return recover2(v,vparent2,e2,σπ)^<eπ>;
10   else
11     return <>;
12   endif
13 end
14
15 Seen := {(init(T_Spec),init(T_Impl),undef,undef,1)}; Done := {};
16
17 while Seen - Done is not empty do
18   Choose some (u,v,vparent,event,π) from Seen-Done;
19   if v is compatible with u then
20     foreach transition (v,e_v,next_v) in T_Impl do
21       e_u := e_v;
22       Let next_u be unique such that (u,e_u,next_u) ∈ T_Spec;
23       (repnext_u,repnext_v,σ) := repPair(next_u,next_v);
24       Put (repnext_u,repnext_v,v,e_v,σ) in Seen if
25         no tuple in Seen has same first two values;
26     endfor
27   else
28     bad := an event possible for v but not for u;
29     print recover2(v,vparent,event,π)^bad; abort;
30   endif
31   Done := union(Done,{(u,v,vparent,event,π)});
32 endwhile
33
34 print 'REFINES';
```

Fig. 3. Twisted refinement checking algorithm for traces refinement. Underlining shows the differences compared with the usual refinement checking algorithm.

A *bad state pair,* and a *bad event* from that pair, are a pair (u,v) and event e where v has an outward transition labelled e but u does not. We generalise the notion of bad event: a *bad trace* from a bad state pair (u,v) is a trace t such that Impl state v can perform t but Spec state u cannot.

A *counterexample trace* is a trace to a bad state pair, extended by a bad trace from that pair. It is easy to see that the counterexample traces are exactly the Impl traces that are not Spec traces.

Define *recover*(<path>) and *recover2*(<*repPair*-twisted path>) as follows:

$$recover(<s_0,e_1,s_1,...,e_n,s_n>) = <e_1, ..., e_{n-1}, e_n>$$

$$recover2(<s_0,e_1,\sigma_1,s_1,...,e_n,\sigma_n,s_n>) = <e_1\sigma_1\sigma_2...\sigma_n, ..., e_{n-1}\sigma_{n-1}\sigma_n, e_n\sigma_n>$$

So *recover*(p) is the trace of events along path p, and *recover2* also yields a trace. Let a *repPair trace to state pair s* be the result of applying *recover2* to a *repPair*-twisted path r to s.

A *repPair counterexample trace* is then a *repPair* trace to a bad state pair, extended by a bad trace from that pair. Examination of Figure 3 shows that on reaching a bad pair (u,v) the condition at line 19 fails and TwistedCheck(*repPair*) effectively applies *recover2* to a *repPair*-twisted path to (u,v), extends the result by a bad event, and so obtains a *repPair* counterexample trace.

6.2 SymCheck1

Suppose a function *rep* maps each implementation state v to a representative in the G-equivalence class of v, for some event permutation group G. Define SymCheck1 to be TwistedCheck(*repPair1*) where *repPair1* is defined in terms of a function *sortRep*:

$$repPair1(u,v) = (u,rep(v),\sigma), \text{ some } \sigma \text{ in G s.t. } v\sigma=rep(v)$$

$$rep = sortRep$$

SymCheck1 explores the Spec-Impl product space by following *repPair1*-twisted paths. Recall that universal G-symmetry of a specification transition system T_{Spec} means that each state u of T_{Spec} is G-symmetric, i.e. each u is such that $u\sigma = u$ for all σ in G. So, for state u of T_{Spec} and state v of T_{Impl}, *repPair1* maps state pair (u,v) to $(u,rep(v),\sigma)$ [for some σ in G s.t. $v\sigma=rep(v)$] = $(u\sigma,v\sigma,\sigma)$ [using universal G-symmetry of T_{spec} and that $v\sigma=rep(v)$] = $(u\sigma_{u,v},v\sigma_{u,v},\sigma_{u,v})$ for $\sigma_{u,v} = \sigma$. The significance of this is that Theorem 2, proved in the appendix, applies.

Theorem 2: *Let G be a group of event permutations and suppose Spec and Impl have G-symmetric transition systems T_{Spec} and T_{Impl} respectively. Suppose function repPair maps each state pair (u,v) to $(u\sigma_{u,v},v\sigma_{u,v},\sigma_{u,v})$ for some $\sigma_{u,v}$ in G. Then Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a counterexample trace t iff Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a repPair counterexample trace t.*

So, SymCheck1 eventually finds a *repPair* counterexample trace exactly when the refinement does not hold, and this will be a counterexample trace. If the exploration order is breadth-first, the counterexample found will clearly have minimal length.

6.3 Method *sortRep*

It remains to define a suitable function *rep* that maps each implementation state v to some G-equivalent representative. Given a group G, an implementation structured

machine T_{Impl} with n leaves and a state $v=(v_1,...,v_n)$ of T_{Impl}, we describe a method of calculating a representative *rep*(v) and a permutation σ in G such that $v\sigma$ = *rep*(v). This method and an alternative defined later both rely on knowledge of the permutation bisimulations between the nodes of T_{Impl} and in particular the σ-bisimulations for σ in G. The method *sortRep* is fast but, as discussed below, it needs all pbisims to have a simple form. Furthermore, G must be a full symmetry group.

Suppose process P is a parallel composition, by some parallel operator *op*, of n>1 processes $P(id_1)$, ..., $P(id_n)$ represented by LTSs L(1), ..., L(n). Then P can be represented by a structured machine S having a simple form if *op* is interleaving, shared parallel on some G-symmetric alphabet A, or alphabetised parallel on G-transparent alphabets (so each process P(id) is synchronised with the others on an alphabet A(id), where $A(id\sigma) = A(id)\sigma$, each σ in G). In such cases, P is representable by structured machine S having top level operator *op* and children L(1), ..., L(n). As previously stated, Figure 2 gives an example for Towers of Hanoi with 3 poles and 2 discs.

A pbisim p_σ is a *simple swap pbisim for leaf indices* i *and* j if p_σ relates each i:m to j:m, each j:m to i:m and, for $k \notin \{i,j\}$, each k:m to k:m. Consider arbitrary state $v = (v_1,...,v_n)$ of such a structured machine S having simple swap σ-bisimulation $p_{i,j}$ for i and j. Then applying $p_{i,j}$ to v has the effect of swapping the values at indices i and j of v and not changing other values, i.e., it yields the state $v' = (v_1,....,v_{i-1},v_j,v_{i+1},...,v_{j-1},v_i,v_{j+1},...,v_n)$. State v is σ-bisimilar to v' since each component of v' is σ-bisimilar to a component of v.

Suppose PBISIMS is a set of simple swap pbisims and S is a subset of {1,...,n}. Then PBISIMS is a *full set of simple swap pbisims for* S if for each i, $j \in S$ there is a simple swap pbisim $p_\sigma \in$ PBISIMS for i and j, with p_σ a σ-bisimulation relation. Let G be the group generated by such permutations σ. In this case any permutation of components v_i of $v = (v_1,...,v_n)$ with indices in S yields a G-equivalent state. The method *sortRep* sorts the components of a state $v = (v_1,...,v_n)$ that have indices in S and leaves the others unchanged; the resulting state is G-related to v by the above reasoning.

A structured machine can be determined to have a full set of simple swap pbisims by finding a set of suitably intersecting 'cycle' pbisims for permutations $\{(e_{1,1} ... e_{1,n1}), ..., (e_{k,1} ... e_{k,nk})\}$ covering all values permuted in G.

There is scope for defining variants of this method that are more widely applicable. In particular, it would be straightforward to cope with multiple simultaneous swaps of indices – such as (1 2)(5 6) – and still use a fast sort-based method: sort a subset of the local state values (say, v_1 and v_2) and apply a corresponding permutation to the other values (v_5 and v_6 in this example).

7. Extensions

Two extended algorithms are described. SymCheck2 uses a more general *rep* function that applies to a larger class of implementation processes than does *sortRep*. SymCheck3 is even more general, requiring only G-symmetry of the Spec transition system instead of universal G-symmetry.

7.1 SymCheck2

Define SymCheck2 to be TwistedCheck(*repPair2*), where *repPair2* uses a more general *rep* function:

$$repPair2(u,v) = (u,rep(v),\sigma), \text{ some } \sigma \text{ in } G \text{ s.t. } v\sigma = rep(v)$$

$$rep = genRep$$

SymCheck2 explores the Spec-Impl product space by following *repPair2*-twisted paths. Compared to SymCheck1, SymCheck2 uses *genRep* (defined below) in place of *sortRep*. Theorem 2 also justifies use of SymCheck2 to find counterexamples when the Spec transition system is universally G-symmetric and the Impl transition system is G-symmetric; the practical difference is that SymCheck2 is less restrictive about the form of the Impl transition system and its known permutation bisimulations.

Method genRep

As already mentioned, this method is more general than *sortRep*. It works with any set of Impl permutation bisimulations such that, for each leaf index i, each pbisim p relates all nodes of LTS(i) to nodes of a distinct LTS(j), and each such LTS(j) node is the image of some LTS(i) node by p, where j depends on the pbisim (and could be the same as i). That is, we require each pbisim p to be the union of bijections $\{p_1,...,p_n\}$ with each p_i having domain the nodes of LTS(i) and range the nodes of some distinct LTS(j). We call such pbisims *uniform*. (Uniformity is a natural condition, indeed all pbisims calculated using our extended Table 1 rules are uniform, and composition of pbisims preserves uniformity.)

The method *genRep* calculates each state $(v'_1,...,v'_n)$ related to $v = (v_1,...,v_n)$ by some pbisim, using pre-calculated pbisims between nodes of the LTSs, and chooses the lexicographically smallest.

We explain how to calculate the node $v' = (v'_1,...,v'_n)$ to which v is related, as determined by a particular permutation bisimulation p. The value v'_j at position j of tuple v' is determined as follows: find the leaf number, i, of the Impl leaf LTS such

that p relates LTS(i) nodes to LTS(j) nodes, and set v'_j to the node of LTS(j) to which node v_i of LTS(i) is related. Now, v represents $Pv_1 \| ... \| Pv_n$ where each Pv_i is the process represented by node v_i of LTS(i), and by construction each Pv_i is such that $Pv_i = Pv'_j\sigma$, for some distinct (by uniformity of p) index j. So, $Pv_1 \| ... \| Pv_n = Pv'_1\sigma \| ... \| Pv'_n\sigma = (Pv'_1 \| ... \| Pv'_n)\sigma$ and hence $Pv = Pv'\sigma$.

For improved efficiency, our implementation pre-calculates, for each pbisim, the appropriate ordering of indices i to calculate the components of v' in left-to-right order. It abandons calculation of v' when a component v'_j is calculated that makes the partial v' larger than the lex-least thus far.

When using *genRep*, before exploration we transitively close the calculated pbisims in the obvious sense – this makes it sufficient to find just a *generating set of pbisims* (i.e., pbisims for a generating set of permutations of G) using the extended Table 1 rules. Transitive closure is not used for *sortRep*, since *sortRep* does not generate all related nodes – even partially – and can be determined applicable given a small number of suitable generating pbisims.

7.2 SymCheck3

Define SymCheck3 to be TwistedCheck(*repPair3*) where:

$$repPair3(u,v) = (u\sigma, rep(v), \sigma), \text{ some } \sigma \text{ in G s.t. } v\sigma = rep(v)$$

$$rep = genRep$$

So SymCheck3 explores the Spec-Impl product space by following *repPair3*-twisted paths. Theorem 2 applies directly to SymCheck3 when the Spec and Impl transition systems are each G-symmetric. We drop the condition (needed for SymCheck1 and SymCheck2) that the Spec transition system T_{Spec} is *universally* G-symmetric – this condition is not needed here because *repPair3* is defined to yield $u\sigma$ in the first part of its result, exactly as needed for Theorem 1 to apply. Hence this algorithm is more general than SymCheck2; the price paid for this extra generality is the need to calculate $u\sigma$, but this is straightforward given pbisims for T_{Spec}. Note that it would not be appropriate to use *rep*(u) instead of $u\sigma$ here, as these will be different in general.

8. Experimental Results

We present results obtained using a prototype tool written in Perl. The tool compiles given Spec and Impl processes, checks particular symmetries of them claimed by the user and in so doing finds corresponding pbisims, and checks applicability of, and

runs, refinement checking algorithms as requested by the user. The results presented are for the usual refinement checking algorithm (which we call Check) and for symmetry exploiting algorithms SymCheck1, SymCheck2 and SymCheck3.

Specification processes were chosen that are refined by the implementations, to show the full size of the (product) state space explored in each case. Three classes of refinement check are reported, distinguished by the choice of specification and implementation:

- refinement of RUN(Events) by Towers of Hanoi models with 4 discs and 4-7 poles, where RUN(Events) can always perform any event;
- refinement of RUN({| try,enter,leave |}), which can perform all events on channels try, enter and leave, by Dijkstra mutual exclusion algorithm models with 2-4 participants; and
- refinement of SpecME by these Dijkstra models, where SpecME can perform exactly the desired patterns of try.i, enter.i and leave.i events and is not universally symmetric for any non-trivial permutation.

Table 2 shows the results obtained for the most efficient of the applicable symmetry exploiting algorithms. For each check, G is the full symmetry group on pole indices (except pole A, where all discs start) or participant identifiers.

In each case the applicable SymCheck algorithms can be determined automatically based on whether there is found to be a full set of simple swap pbisims (in which case *sortRep* can be used) and whether the specification process LTS is found to be universally G-symmetric (in which case SymCheck2 applies, and so does SymCheck1 if *sortRep* can be used).

One column gives total time for compilation of the implementation process to a structured machine plus checking of the claimed implementation symmetries. Others give supercompilation[3] [6] time, and time for transitive closure of implementation transition system pbisims (i.e. for determining an implementation transition system pbisim for each permutation in G, which is needed for *genRep* and so for SymCheck2 and SymCheck3). Corresponding timings are omitted for the specification as they are much smaller. In addition, exploration times are of course reported.

Although the table does not show it, SymCheck3 has a larger overhead per state explored than does SymCheck2. The table does include evidence that SymCheck2 has a larger overhead than SymCheck1.

The Towers of Hanoi models are very simple. Each has a structured machine with a full set of simple swap pbisims for G. Also, the specification RUN(Events) is found to be universally G-symmetric. These properties are determined quickly by the tool and hence SymCheck1 is found to apply. Compared with Check, there is a substantial reduction in the number of state pairs explored by SymCheck1 and in exploration

3 Supercompilation can reduce exploration times greatly; it is outside the scope of this paper.

time. Although the compilation effort is larger, the extra costs are small compared to the benefits of exploring fewer state pairs.

The Dijkstra mutual exclusion models were chosen partly because their structured machines do not have simple swap pbisims for the permutations in the corresponding group G. Accordingly, SymCheck1 does not apply. SymCheck2 applies when checking refinement of RUN({| try,enter,leave |}), as this specification is univerally G-symmetric. However, SymCheck2 does not apply with the merely G-symmetric specification SpecME; only SymCheck3 applies in the case of this refinement property.

Algorithm	MODEL	States		Time (secs)					Number of perm syms found	Overhead per state explored
				Impl compilation + sym checking	Impl supercompilation	Impl pbisim transitive closure	Exploration			
Check	hanoi4p4d	256	100%	3.34	0.20	-	3.58	100%	-	0%
SymCheck1	hanoi4p4d	51	19.92%	3.34	0.20	-	0.73	20%	6	2%
SymCheck2	hanoi4p4d	51	19.92%	3.34	0.20	0.01	0.73	20%	6	2%
Check	hanoi5p4d	625	100%	5.13	0.37	-	19.14	100%	-	0%
SymCheck1	hanoi5p4d	52	8.32%	5.13	0.37	-	1.53	8%	24	-4%
SymCheck2	hanoi5p4d	52	8.32%	5.13	0.37	0.05	1.72	9%	24	8%
Check	hanoi6p4d	1296	100%	7.38	0.78	-	82.29	100%	-	0%
SymCheck1	hanoi6p4d	52	4.01%	7.38	0.78	-	3.11	4%	120	-6%
SymCheck2	hanoi6p4d	52	4.01%	7.38	0.78	0.56	4.20	5%	120	27%
Check	hanoi7p4d	2401	100%	9.88	3.78	-	246.83	100%	-	0%
SymCheck1	hanoi7p4d	52	2.17%	9.88	3.78	-	5.09	2%	720	-5%
SymCheck2	hanoi7p4d	52	2.17%	9.88	3.78	5.69	15.38	6%	720	188%
Check	DijkstraME_2	445	100%	4.18	0.53	-	0.16	100%	-	0%
SymCheck2	DijkstraME_2	224	50.34%	4.18	0.53	0.00	0.11	69%	2	37%
Check	DijkstraME_3	19161	100%	22.66	2.65	-	11.77	100%	-	0%
SymCheck2	DijkstraME_3	3269	17.06%	22.66	2.65	0.01	2.94	25%	6	46%
Check	DijkstraME_4	1189379	100%	63.89	10.02	-	1103.00	100%	-	0%
SymCheck2	DijkstraME_4	51571	4.34%	63.89	10.02	0.07	118.95	11%	24	149%
Check	DijkstraME_2	445	100%	4.08	0.52	-	0.13	100%	-	0%
SymCheck3	DijkstraME_2	224	50.34%	4.08	0.52	0.00	0.12	92%	2	83%
Check	DijkstraME_3	19161	100%	22.77	2.66	-	8.82	100%	-	0%
SymCheck3	DijkstraME_3	3269	17.06%	22.77	2.66	0.01	2.96	34%	6	97%
Check	DijkstraME_4	1189379	100%	63.57	10.00	-	861.12	100%	-	0%
SymCheck3	DijkstraME_4	51571	4.34%	63.57	10.00	0.07	133.88	16%	24	259%

Table 2. Experimental results for the usual refinement checking algorithm Check and the three SymCheck algorithms.

For the larger symmetry groups, algorithms SymCheck2 and SymCheck3 suffer from the rapid increases in the size of G that result from increasing the number of poles or participants; this is because both algorithms use *genRep*, which needs a pbisim for each element of G. SymCheck1 is much less sensitive to this because it uses *sortRep*, which only requires a linear number of (simple swap) pbisims. Further,

these pbisims can be calculated efficiently from just two pbisims corresponding to any transposition (x y) and any cycle on all elements of G except for x. This was done for the SymCheck1 checks reported in the table.

9. Conclusions

We have successfully adapted the representative function approach to symmetry exploitation from the temporal logic model checking context to CSP refinement checking. The major obstacle was the need for access to sufficient local information about state during refinement checking, which is provided by representing the implementation process as a structured machine. We have also presented two generalisations of the basic algorithm. All three algorithms have been presented in a common style, in terms of a curried function TwistedCheck.

An option for future work is to characterise more precisely, in terms of processes, when alternative SymCheck variants apply and even develop methods for transforming CSP models, or their transition systems, to make the more efficient algorithms more widely applicable.

There are many other possible extensions, including: use of (a perhaps liberalized notion of) data independence to increase the efficiency of symmetry identification; development of variants of the *sortRep* function to cope efficiently with wider classes of structured machines and permutation bisimulations over them (and hence more implementation processes); extension to multiple representatives; extension to virtual symmetries [12]; and use of computational group theory to improve efficiency.

It would also be interesting to investigate the temporal logic analogue of (non-universal) G-symmetry and perhaps generalise the representative function approach to symmetry exploitation for temporal logic model checking, effectively removing the requirement that the specification is *always* symmetric.

Our experimental results illustrate that the refinement checking algorithms we have presented can give significant savings in the number of state pairs explored and in verification time. The former can be expected to lead to corresponding reductions in memory usage, which is often the dominant factor determining the sizes of problems that can be checked.

Acknowledgements

We are grateful to the reviewers for suggesting several helpful clarifications.

References

[1] A. Miller, A. F. Donaldson and M. Calder. *Symmetry in Temporal Logic Model Checking*, ACM Comput. Surv., 38(3), Article 8, 2006.

[2] C. A. R. Hoare. *Communicating Sequential Processes*, Communications of the ACM, 21(8): 666-677, 1978.

[3] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[4] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 User Manual*, 1992-2008.

[5] N. Moffat, M. Goldsmith and A. W. Roscoe. *Towards Symmetry Aware Refinement Checking (Extended Abstract)*, Proceedings of International Symmetry Conference (ISC 2007), Edinburgh, UK. January 2007.

[6] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe and A. W. Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2001.

[7] D. M. Park. *Concurrency on automata and infinite sequences*. In P. Deussen, editor, Proceedings of the Conference on Theoretical Computer Science, LNCS vol. 104. Springer Verlag, 1981.

[8] R. Milner. *Communication and concurrency*. Prentice Hall, 1989.

[9] E. Clarke, R. Enders, T. Filkhorn and S. Jha. *Exploiting Symmetry in Temporal Logic Model Checking*. Formal Methods in System Design, 9(1/2):77-104, August 1996.

[10] E. A. Emerson and A. Prasad Sistla. *Symmetry and model checking*. Formal Methods in System Design, 9(1/2):105-131, August 1996.

[11] R. S. Lazic. *A semantic study of data-independence with applications to the mechanical verification of concurrent systems*. Ph.D. thesis, Oxford University Computing Laboratory, 1999.

[12] E. A. Emerson, J. Havlicek, R. Trefler. *Virtual Symmetry Reduction.* In Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, 2000.

Appendix: Theory

Lemma 1: *Let G be a group of event permutations. Consider the product space for particular Spec and Impl transition systems, with initial state pair $s_0=(u_0,v_0)$. Suppose repPair is a function that maps each state pair (u,v) to $(u\sigma_{u,v},v\sigma_{u,v},\sigma_{u,v})$, some $\sigma_{u,v}$ in G. Then, for all traces t, there is a path p from s_0 to state pair s=(u,v), with recover(p) = t iff there is a repPair-twisted path r from s_0 to $s\sigma = (u\sigma,v\sigma)$, with recover2(r) = $t\sigma$, some σ in G.*

Proof: Induction on length k of t.

Base case: k = 0, so t = <>. There is exactly one path, p=<s_0>, starting at s_0 and such that *recover*(p) = <>. This path ends at (u,v)=(u_0,v_0)=s_0. Also, there is exactly one *rep-*

Pair-twisted path, r=<s_0>, starting at s_0 and such that *recover2*(r) = <>. This *repPair*-twisted path ends at $s_0 = s_0 1$ so we may choose σ=1. Then *recover2*(r) = <> = <>σ.

Inductive step: Suppose the lemma holds for all traces t of length k. We show it also holds for all t of length k+1.

(⇒) Suppose p=p' ^ <e> is a length-k+1 path from s_0 to (u,v), with *recover*(p) = t. Then p' is a length-k path to some state pair (u',v') and there is a transition (u',v') -(e)-> (u,v). Clearly, *recover*(p) = *recover*(p' ^ <e>) = *recover*(p') ^ <e>. So, defining t'=*recover*(p'), we have t = t' ^ <e>. By the induction hypothesis applied to p' and t', there is a *repPair*-twisted path r' from s_0 to (u'σ',v'σ') with *recover2*(r') = t'σ', some σ' in G. Recall there is a transition (u',v') -(e)-> (u,v), so there is a transition (u'σ',v'σ') -(eσ')-> (uσ',vσ'). Let ρ in G be such that *repPair*(uσ',vσ')=(uσ'ρ,vσ'ρ, ρ).[4] Then r = r' ^ <eσ',ρ,(uσ'ρ,vσ'ρ)> is a *repPair*-twisted path to (uσ'ρ,vσ'ρ) since r' is *repPair*-twisted and ends at (u'σ',v'σ') and there is a transition (u'σ',v'σ') -(eσ')-> (uσ',vσ') and *repPair*(uσ',vσ') = (uσ'ρ,vσ'ρ,ρ). Putting σ = σ'ρ, we obtain that r is a *repPair*-twisted path from s_0 to sσ=(uσ,vσ), where σ = σ'ρ is in G since both σ' and ρ are. It remains to show that *recover2*(r) = tσ. We have *recover2*(r) = *recover2*(r' ^ <eσ',ρ,(uσ'ρ,vσ'ρ)>) = (*recover2*(r') ^ <eσ'>)ρ = (t'σ' ^ <eσ'>)ρ = (t' ^ <e>)σ'ρ = tσ.

(⇐) Similar.

Theorem 1: *Let G be a group of event permutations and suppose Spec and Impl have transition systems T_{Spec} and T_{Impl} respectively. Suppose function repPair maps each state pair (u,v) to ($u\sigma_{u,v}$,$v\sigma_{u,v}$,$\sigma_{u,v}$) for some $\sigma_{u,v}$ in G. Then Spec $\sqsubseteq_T$ Impl has a counterexample trace t iff ∃ σ ∈ G s.t. Spec $\sqsubseteq_T$ Impl has a repPair counterexample trace tσ.*

Proof: (⇒) Let t be a counterexample trace of Spec $\sqsubseteq_T$ Impl. Then t is a trace of Impl. Let t_1 be the longest prefix of t that is a trace of Spec and t_2 be such that $t = t_1 \hat{} t_2$. Then there is a path p from initial state pair s_0 to s=(u,v), say, with *recover*(p) = t_1 and t_2 a bad trace from (u,v). By Lemma 1, there is a *repPair*-twisted path r from s_0 to sσ, some σ in G, with *recover2*(r) = t_1σ. But sσ = (uσ,vσ) is a bad state pair, and t_2σ must be a bad trace from sσ (since Impl state vσ is able to perform trace t_2σ but Spec state uσ is not). So *recover2*(r) ^ t_2σ = t_1σ ^ t_2σ = $(t_1 \hat{} t_2)$σ = tσ is a *repPair* counterexample trace, for this σ in G.

(⇐) Let t be a *repPair* counterexample trace of Spec $\sqsubseteq_T$ Impl. Then t = *recover2*(r) ^ t_2 for some *repPair*-twisted path r from initial state pair s_0 to a bad state pair s=(u,v), such that t_2 is a bad trace from (u,v). So Impl state v can perform trace t_2 but Spec state u cannot . Putting t_1 = *recover2*(r), we have $t = t_1 \hat{} t_2$. By Lemma 1, there is a path p from s_0 to $s\sigma^{-1}$, some σ^{-1} in G, with *recover*(p) = $t_1\sigma^{-1}$. Now $s\sigma^{-1}$ = ($u\sigma^{-1}$,$v\sigma^{-1}$) is

4 Such a ρ is denoted $\sigma_{u\sigma',v\sigma'}$ in the statement of the lemma.

must be a bad state pair with $t_2\sigma^{-1}$ a bad trace from $s\sigma^{-1}$, since Impl state $v\sigma^{-1}$ can perform trace $t_2\sigma^{-1}$ but Spec state $u\sigma^{-1}$ cannot. So p is a path from s_0 to bad state pair $s\sigma^{-1}$ and *recover*(p) ^ $t_2\sigma^{-1}$ = $t_1\sigma^{-1}$ ^ $t_2\sigma^{-1}$ = $(t_1 \hat{} t_2)\sigma^{-1}$ = $t\sigma^{-1}$ is a counterexample trace, for this σ^{-1} in G.

Theorem 2: *Let G be a group of event permutations and suppose Spec and Impl have G-symmetric transition systems T_{Spec} and T_{Impl} respectively. Suppose function repPair maps each state pair (u,v) to $(u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$ for some $\sigma_{u,v}$ in G. Then Spec $\sqsubseteq_T$ Impl has a counterexample trace t iff Spec $\sqsubseteq_T$ Impl has a repPair counterexample trace t.*

Proof: By Theorem 1, Spec $\sqsubseteq_T$ Impl has a counterexample trace t iff $\exists\ \sigma \in G$ s.t. Spec $\sqsubseteq_T$ Impl has a *repPair* counterexample trace $t\sigma$. Then use that, $\forall\ \sigma$ in G, Spec $\sqsubseteq_T$ Impl has a counterexample trace t iff it has a counterexample trace $t\sigma$ (which follows from G-symmetry of the Spec and Impl transition systems).