Exploiting Symmetry in Protocol Testing *

Judi Romijn[1] and Jan Springintveld[1,2]

[1] *CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
{judi,spring}@cwi.nl

[2] *Computing Science Institute*
University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

ABSTRACT

Test generation and execution are often hampered by the large state spaces of the systems involved. In automata (or transition system) based test algorithms, taking advantage of symmetry in the behavior of specification and implementation may substantially reduce the amount of tests. We present a framework for describing and exploiting symmetries in black box test derivation methods based on finite state machines (FSMs). An algorithm is presented that, for a given symmetry relation on the traces of an FSM, computes a subautomaton that characterizes the FSM up to symmetry. This machinery is applied to the classical W-method [26, 7] for test derivation. Finally, we focus on symmetries defined in terms of repeating patterns.

1991 Mathematics Subject Classification: 68M15, 68Q05, 68Q68, 94C12
1991 Computing Reviews Classification System: B.4.5, D.2.5, F.1.1
Keywords and Phrases: conformance testing, automated test generation, state space reduction, symmetry
Note: The research of the first author was carried out as part of the project "Specification, Testing and Verification of Software for Technical Applications" at the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps. The research of the second author was partially supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-33-006. His current affiliation is: Philips Research Laboratories Eindhoven, Prof. Holstlaan 4, 5656 AA, Eindhoven, The Netherlands.

1. Introduction

It has long been recognized that for the proper functioning of components in open and distributed systems, these components have to be thoroughly tested for interoperability

* A short version of this report appeared in S. Budkowski, A. Cavalli and E. Najm, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE XI/PSTV XVIII '98)*, pages 337–352. Kluwer Academic Publishers, 1998.

and conformance to internationally agreed standards. For thorough and efficient testing, a high degree of automation of the test process is crucial. Unfortunately, methods for automated test generation and execution are still seriously hampered by the often very large state spaces of the implementations under test. One of the ways to deal with this problem is to exploit structural properties of the implementation under test that can be safely assumed to hold. In this paper we focus on taking advantage of *symmetry* that is present in the structure of systems. The symmetry, as it is defined here, may be found in any type of parameterized system: such parameters may for example range over IDs of components, ports, or the contents of messages.

We will work in the setting of test theory based on finite state machines (FSMs). Thus, we assume that the specification of an implementation under test is given as an FSM and the implementation itself is given as a black box. From the explicitly given specification automaton a collection of tests is derived that can be applied to the black box. Exploiting symmetry will allow us to restrict the test process to subautomata of specification and implementation that characterize these systems up to symmetry and will often be much smaller. The symmetry is defined in terms of an equivalence relation over the trace sets of specification and implementation. Some requirements are imposed to ensure that such a symmetry indeed allows to find the desired subautomata. We instantiate this general framework by focusing on symmetries defined in terms of repeating *patterns*. Some experiments with pattern-based symmetries, supported by a prototype tool implemented using the OPEN/CÆSAR tool set [14], have shown that substantial savings may be obtained in the number of tests.

Since we assume that the black box system has some symmetrical structure (cf. the *uniformity hypothesis* in [15, 6]), it is perhaps more appropriate to speak of *gray* box testing. For the specification FSM it will generally be possible to *verify* that a particular relation is a symmetry on the system, but for the black box implementation one has to *assume* that this is the case. The reliability of this assumption is the tester's responsibility. In this respect, one may think of exploiting symmetry as a structured way of test case selection [13, 4] for systems too large to be tested exhaustively, where at least some subautomata are tested thoroughly.

This paper is not the first to deal with symmetry in protocol testing. In [20], similar techniques have been developed for a test generation methodology based on labeled transition systems, success trees and canonical testers [3, 25]. Like in our case, symmetry is an equivalence relation between traces, and representatives of the equivalence classes are used for test generation. Since our approach and the approach in [20] start from different testing methodologies, it is not easy to compare them. In [20], the symmetry relation is defined through bijective renamings of action labels; our pattern-based definition generalizes this approach. On the other hand, since in our case a symmetry relation has to result in subautomata of specification and implementation that characterize these systems up to the symmetry, we have to impose certain requirements, which are absent in [20].

In [19], symmetrical structures in the product automaton of interoperating systems

are studied. It is assumed that the systems have already been tested in isolation and attention is focused on pruning the product automaton by exploiting symmetry arising from the presence of identical peers. In the present paper, we abstract away from the internal composition of the system and focus on defining a *general* framework for describing and using symmetries on FSMs.

This paper is organized as follows. Section 2 contains some basic definitions concerning FSMs and their behavior. In Section 3, we introduce and define a general notion of trace based symmetry. We show how, given such a symmetry on the behavior of a system, a subautomaton of the system can be computed, a so-called *kernel*, that characterizes the behavior of the system up to symmetry. In Section 5 we apply the machinery to the classical W-method [26, 7] for test derivation. In Section 6 we will instantiate the general framework by focusing on symmetries defined in terms of repeating patterns. Section 7 contains an extensive example, inspired by [24]. Finally, we discuss future work in Section 8.

2. Finite state machines

In this section, we will briefly present some terminology concerning finite state machines and their behavior, that we will need in the rest of this paper.

We let N denote the set of natural numbers. (Finite) Sequences are denoted by greek letters. Concatenation of sequences is denoted by juxtaposition; ϵ denotes the empty sequence and the sequence containing a single element a is simply denoted a. If σ is non-empty then $\mathit{first}(\sigma)$ returns the first element of σ and $\mathit{last}(\sigma)$ returns the last element of σ.

If V and W are sets of sequences and σ is a sequence, then $\sigma\, W = \{\sigma\, \tau \mid \tau \in W\}$ and $V\, W = \bigcup_{\sigma \in V} \sigma\, W$. For X a set of symbols, we define $X^0 = \{\epsilon\}$ and, for $i > 0$, $X^i = X^{i-1} \cup X\, X^{i-1}$. As usual, $X^* = \bigcup_{i \in \mathsf{N}} X^i$.

Definition 2.1. A *finite state machine (FSM)* is a structure $\mathcal{A} = (S, \Sigma, E, s^0)$ where

- S is a finite set of *states*
- Σ a finite set of *actions*
- $E \subseteq S \times \Sigma \times S$ is a finite set of *edges*
- $s^0 \in S$ is the *initial state*

We require that $\mathcal{A}$ is *deterministic*, i.e., for every pair of edges (s, a, s'), (s, a, s'') in $E_{\mathcal{A}}$, $s' = s''$.

We write $S_{\mathcal{A}}$, $\Sigma_{\mathcal{A}}$, etc., for the components of an FSM $\mathcal{A}$, but often omit subscripts when they are clear from the context. We let s, s' range over states, $a, a', b, c, \ldots$ over actions, and e, e' over edges. If $e = (s, a, s')$ then $\mathsf{act}(e) = a$. We write $s \xrightarrow{a} s'$ if $(s, a, s') \in E$ and with $s \xrightarrow{a}$ we denote that $s \xrightarrow{a} s'$ for some state s'. A *subautomaton* of an FSM $\mathcal{A}$ is an FSM $\mathcal{B}$ such that $s^0_{\mathcal{B}} = s^0_{\mathcal{A}}$, $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$, $\Sigma_{\mathcal{B}} \subseteq \Sigma_{\mathcal{A}}$, and $E_{\mathcal{B}} \subseteq E_{\mathcal{A}}$.

An *execution fragment* of an FSM $\mathcal{A}$ is an alternating sequence $\gamma = s_0\, a_1\, s_1 \cdots a_n\, s_n$ of states and actions of $\mathcal{A}$, beginning and ending with a state, such that for all i, $0 \leq i < n$, we have $s_i \xrightarrow{a_{i+1}} s_{i+1}$. If $s_0 = s_n$ then γ is a *loop*, if $n \neq 0$ then γ is a *non-empty* loop. An *execution* of $\mathcal{A}$ is an execution fragment that begins with the initial state of $\mathcal{A}$.

For $\gamma = s_0\, a_1\, s_1 \cdots a_n\, s_n$ an execution fragment of $\mathcal{A}$, $trace(\gamma)$ is defined as the sequence $a_1\, a_2 \cdots a_n$. If σ is a sequence of actions, then we write $s \xrightarrow{\sigma} s'$ if $\mathcal{A}$ has an execution fragment γ with $first(\gamma) = s$, $last(\gamma) = s'$, and $trace(\gamma) = \sigma$. If γ is a loop, then σ is a *looping trace*. We write $s \xrightarrow{\sigma}$ if there exists an s' such that $s \xrightarrow{\sigma} s'$, and write $traces(s)$ for the set $\{\sigma \in (\Sigma_{\mathcal{A}})^* \mid s \xrightarrow{\sigma}\}$. We write $traces(\mathcal{A})$ for $traces(s^0_{\mathcal{A}})$. □

3. Symmetry

In this section we introduce the notion of symmetry employed in this paper.

We want to be able to restrict the test process to subautomata of specification and implementation that characterize these systems up to symmetry. In papers on exploiting symmetry in model checking [2, 8, 10, 11, 12, 18], such subautomata are constructed for explicitly given FSMs by identifying and collapsing symmetrical *states*. We are concerned with black box testing, and, by definition, it is impossible to refer directly to the states of a black box. In traditional FSM based test theory, FSMs are assumed to be deterministic and hence a state of a black box is identified as the unique state of the black box that is reached after a certain trace of the system. Thus it seems natural to define symmetry as a relation over *traces*.

Our basic notion of symmetry on an FSM $\mathcal{A}$, then, is an equivalence relation on $(\Sigma_{\mathcal{A}})^*$, such that $\mathcal{A}$ is *closed* under the symmetry, i.e., if a sequence of actions is symmetrical to a trace of $\mathcal{A}$ then the sequence is a trace of $\mathcal{A}$ too.

The idea is to construct from the specification automaton an automaton such that its trace set is included in the trace set of the specification and contains a representative trace for each equivalence class of the symmetry relation on the traces of the specification. In order to be able to do this, we need to impose some requirements on the symmetry. For the specification we demand (1) that each equivalence class of the symmetry is represented by a unique trace, (2) that prefixes of a trace are represented by prefixes of the representing trace, and (3) that representative traces respect loops. The third requirement means that if a representative trace is a looping trace, then the trace with the looping part removed is also a representative trace. This requirement introduces some state-based information in the definition of symmetry.

These requirements enable us to construct a subautomaton of the specification, a so-called *kernel*, such that every trace of the specification is represented by a trace from the kernel. Of course, it will often be the case that the symmetry itself is preserved under prefixes and respects loops, so the requirements will come almost for free.

For the black box implementation, we will, w.r.t. symmetry, only demand that it is closed under symmetry. So if tests have established that the implementation displays certain behavior, then by assumption it will also display the symmetrical behavior. In

Section 5, where the theory is applied to Mealy machines, we will in addition need a way to identify a subautomaton of the implementation that is being covered by the tests derived from the kernel of the specification.

Definition 3.1. A *symmetry* S on an FSM $\mathcal{A}$ is pair $\langle \simeq, ()^r \rangle$ where $\simeq$ is a binary equivalence relation on $(\Sigma_\mathcal{A})^*$, and $()^r : (\Sigma_\mathcal{A})^* \to (\Sigma_\mathcal{A})^*$ is a *representative function* for $\simeq$ such that:

1. $\mathcal{A}$ is closed under $\simeq$: If $\sigma \in \mathit{traces}(\mathcal{A})$ and $\sigma \simeq \tau$, then $\tau \in \mathit{traces}(\mathcal{A})$.
2. Only traces of the same length are related: If $\sigma \simeq \tau$, then $|\sigma| = |\tau|$.
3. $()^r$ satisfies:
   (a) $\sigma^r \simeq \sigma$
   (b) $\tau \simeq \sigma \Rightarrow \tau^r = \sigma^r$
   (c) $()^r$ is prefix closed on $\mathcal{A}$: If $\sigma\, a \in \mathit{traces}(\mathcal{A})$ and $(\sigma\, a)^r = \tau\, b$, then $\sigma^r = \tau$
   (d) $()^r$ is loop respecting on representative traces: If $(\sigma_1\, \sigma_2\, \sigma_3)^r = \sigma_1\, \sigma_2\, \sigma_3 \in \mathit{traces}(\mathcal{A})$ and σ_2 is a looping trace, then $(\sigma_1\, \sigma_3)^r = \sigma_1\, \sigma_3$.

The class of traces τ such that $\tau \simeq \sigma$ is denoted with $[\sigma]_S$, or, if S is clear from the context, $[\sigma]$. □

As mentioned above, we will demand that there exists a symmetry on the specification, while the implementation under test is required only to be closed under the symmetry.

Lemma 3.2. $(\sigma^r)^r = \sigma^r$

Definition 3.3. Let $S = \langle \simeq, ()^r \rangle$ be a symmetry on FSM $\mathcal{A}$. A *kernel* of $\mathcal{A}$ w.r.t. S is a subautomaton $\mathcal{K}$ of $\mathcal{A}$, such that for every $\sigma \in \mathit{traces}(\mathcal{A})$, $\sigma^r \in \mathit{traces}(\mathcal{K})$. □

4. Construction of a kernel

In this section, we fix an FSM $\mathcal{A}$ and a symmetry $S = \langle \simeq, ()^r \rangle$ on $\mathcal{A}$. Figure 1 presents an algorithm that constructs a kernel of $\mathcal{A}$ w.r.t. S. It basically explores the state space of $\mathcal{A}$, while keeping in mind the trace that leads to the currently visited state. As soon as such a trace contains a loop, the algorithm will not explore it any further.

In Figure 1, $\mathit{enabled}(s, \mathcal{A})$ denotes the set of actions a such that $E_\mathcal{A}$ contains an edge (s, a, s'), and for such an a, $\mathit{eff}(s, a, \mathcal{A})$ denotes s'. Furthermore, $\mathit{repr}(\sigma, E)$ denotes the set F of actions such that $a \in F$ iff there exists an action $b \in E$ such that $\sigma^r\, a = (\sigma\, b)^r$. We will only call this function for σ such that $\sigma^r = \sigma$(see Lemma 4.3). By definition of $()^r$, for some action c, $(\sigma\, b)^r = \sigma^r\, c = \sigma\, c$. So, since $\mathcal{A}$ is deterministic and closed

```
function Kernel(𝒜, S): FSM;
  var 𝒦;
    procedure Build_It(s, σ, Seen);
    var a, b, s, s′, E, F;
    begin
          if  s ∉ Seen
          then  E := enabled(s, 𝒜);
                F := ∅;
                while E ≠ ∅
                do  a := choose(repr(σ, E));
                    s′ := eff(s, a, 𝒜);
                    S_𝒦 := S_𝒦 ∪ {s′};
                    Σ_𝒦 := Σ_𝒦 ∪ {a};
                    E_𝒦 := E_𝒦 ∪ {(s, a, s′)};
                    Build_It(s′, σ a, Seen ∪ {s});
                    F := F ∪ {a};
                    E := E \ {a};
                    for each  b ∈ E . σ a ≃ σ b
                    do  E := E \ {b};
                    od;
                od;
          fi;
    end;

  begin
        s⁰_𝒦 := s⁰_𝒜;
        S_𝒦 := {s⁰_𝒜};
        Σ_𝒦 := ∅;
        E_𝒦 := ∅;
        Build_It(s⁰_𝒜, ϵ, ∅);
        return 𝒦;
  end.
```

Figure 1: The algorithm Kernel

under $\simeq$, $F \subseteq E$ and if E is non-empty, F is non-empty. This justifies the function $choose(F)$ which nondeterministically chooses an element from F.

The remainder of this section is devoted to the correctness of algorithm Kernel. In order to prove that the algorithm works properly, we first prove that it terminates, that it creates a subautomaton of $\mathcal{A}$ and that build_it uses its parameters properly.

Lemma 4.1. *The execution of the algorithm Kernel*($\mathcal{A}, S$) *terminates.*

Proof. The number of states in $\mathcal{A}$ is finite, and for each nested call of build_it(s', σ', $Seen'$) within build_it($s, \sigma, Seen$), $Seen' = Seen \cup \{s'\}$ with $s' \notin Seen$. So there can be finitely many levels of such nested calls. Furthermore, the number of enabled transitions in s is finite, so the while loop that empties E (E decreases strictly monotonically during this loop until it's empty) can make finitely many nested calls to build_it. ⊠

Lemma 4.2. *During execution of build_it*($s^0_{\mathcal{A}}$,ϵ,$\emptyset$), *automaton* $\mathcal{K}$ *is a subautomaton of* $\mathcal{A}$, *and* $\mathcal{K}$ *grows monotonically.*

Lemma 4.3. *If Kernel*($\mathcal{A}$,S) *calls build_it*(s,σ,$Seen$) *then* $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ *and* $\sigma^r = \sigma$.

Proof. By induction on the length n of σ.

- $n = 0$. Then $\sigma = \epsilon$. From observing the algorithm Kerneland procedure build_it, it is clear that the only call of build_it(s,ϵ,$Seen$) is with $s = s^0_{\mathcal{A}}$ and $Seen = \emptyset$. At this point in Kernel($\mathcal{A}$,S), $s^0_{\mathcal{A}}$ has just been added to $\mathcal{K}$ as $s^0_{\mathcal{K}}$. So $s^0_{\mathcal{K}} \xrightarrow{\epsilon}_{\mathcal{K}} s$. Certainly, $(\epsilon)^r = \epsilon$.

- $n = m + 1$.

  Suppose $\sigma = \sigma'\, a$ is a trace of length $m + 1$ and Kernel($\mathcal{A}$,S) calls build_it(s,σ, $Seen$). Since $\sigma \neq \epsilon$, the call build_it(s,σ,$Seen$) must occur within the execution of a call build_it(s',σ',$Seen'$). By the induction hypothesis, we know that $s^0_{\mathcal{K}} \xrightarrow{\sigma'}_{\mathcal{K}} s'$. When build_it($s'$,$\sigma'$,$Seen'$) calls build_it($s$,$\sigma'\, a$,$Seen$) then (s', a, s) has just been added to $E_{\mathcal{K}}$, with a from $enabled(s, \mathcal{A})$ and $s = eff(s', a, \mathcal{A})$. So $s' \xrightarrow{a}_{\mathcal{K}} s$ when the call build_it(s,σ,$Seen$) is made, and it follows that $\sigma \in traces(\mathcal{K})$.

  As to $\sigma^r = \sigma$. When build_it(s',σ',$Seen'$) calls build_it(s,$\sigma'\, a$,$Seen$) then by definition of $choose(repr(\sigma', E))$, $(\sigma')^r\, a = (\sigma'\, a)^r$. Since, by induction hypothesis, $(\sigma')^r = \sigma'$, $(\sigma'\, a)^r = \sigma'\, a$, which completes the proof.

⊠

Lemma 4.4. *If Kernel*($\mathcal{A}$,S) *calls build_it*(s,σ,$Seen$), *then* $\sigma \in traces$(*Kernel*($\mathcal{A}, S$)).

Proof. Follows immediately from Lemmas 4.2 and 4.3. ⊠

Lemma 4.5. *If Kernel($\mathcal{A}$,S) calls build_it(s,σ,$Seen$), then during the execution of build_it the following holds:*

1. *during and after the while loop, the following property holds:*

$$a \in F \Rightarrow \text{ Kernel}(\mathcal{A}, S) \text{ calls build_it}(\mathit{eff}(s, a, \mathcal{K}), \sigma\, a, \mathit{Seen} \cup \{s\})$$

2. *during and after the while loop, the following property holds:*

$$s \xrightarrow{a}_{\mathcal{A}} \Rightarrow a \in E \vee \exists b \in F.\, \sigma\, b = (\sigma\, a)^r$$

3. *after the while loop, E is empty.*

Proof.

1. When the while loop is started, F is empty. The only statement that adds b to F follows right after the statements that first add the edge (s, a, s') to $E_{\mathcal{K}}$ and then call build_it$(s', \sigma a, \mathit{Seen} \cup \{s\})$.

2. Suppose $a \notin E$. At the start of the while loop, E contains each enabled action in s, including a. So a has been removed from E by one of the last two statements of the while loop. In case a was removed from E by the first of these two statements, then it was added to F one statement earlier, and, by definition of $\mathit{choose}(\mathit{repr}(\sigma, E))$ and Lemma 4.3, $\sigma\, a = (\sigma\, a)^r$. In case a was removed from E by the last of the two statements, then, for $a' = \mathit{repr}(\sigma, E)$ and Lemma 4.3, $(\sigma\, a)^r = \sigma\, a'$.

3. Trivial, since it is the stop condition for the while loop, and the while loop is the last statement in the procedure.

⊠

Lemma 4.6. *Suppose Kernel($\mathcal{A}$,S) calls build_it(s,σ,$Seen$) with $\sigma = a_1 a_2 \ldots a_n$, $s_0 \xrightarrow{a_1}_{\mathcal{A}} s_1 \xrightarrow{a_2}_{\mathcal{A}} s_2 \ldots \xrightarrow{a_n}_{\mathcal{A}} s_n$, and $s_0 = s^0_{\mathcal{A}}$. Then*

1. *Kernel($\mathcal{A}$,S) calls build_it(s_0,ϵ,$\emptyset$)*

2. *for $0 \leq i < n$, build_it(s_i,σ_i,$Seen_i$) calls build_it(s_{i+1},σ_{i+1},$Seen_{i+1}$) with $\sigma_i = a_1 a_2 \ldots a_i$ and for $0 \leq i \leq n$, $Seen_i = \bigcup_{j \in \{0,1,\ldots,i-1\}} \{s_j\}$*

3. *$s = s_n$ and $Seen = Seen_n$*

Proof. By induction on the length n of σ.

- $n = 0$. Then $\sigma = \epsilon$, and the result follows immediately.

- $n = m + 1$.

  Suppose $\sigma = a_1 a_2 \dots a_{m+1}$ and Kernel($\mathcal{A}$,S) calls build_it(s,σ,$Seen$) with $s_0 \xrightarrow{a_1}_{\mathcal{A}} s_1 \xrightarrow{a_2}_{\mathcal{A}} s_2 \dots \xrightarrow{a_{m+1}}_{\mathcal{A}} s_{m+1}$ and $s_0 = s^0_{\mathcal{A}}$. Let $\sigma' = a_1 a_2 \dots a_m$. Since $\sigma \neq \epsilon$, the call build_it(s,σ,$Seen$) must occur within the execution of a call build_it(s',σ',$Seen'$) and $Seen = Seen' \cup \{s'\}$. By the induction hypothesis, we know that $Seen' = Seen_m$, that $s' = s_m$, that Kernel($\mathcal{A}$,S) calls build_it(s_0,ϵ,$\emptyset$), that for $0 \leq i < m$, build_it(s_i,σ_i,$Seen_i$) calls build_it(s_{i+1},σ_{i+1},$Seen_{i+1}$) with $\sigma_i = a_1 a_2 \dots a_i$ and that for $0 \leq i \leq m$, $Seen_i = \bigcup_{j \in \{0,1,\dots,i-1\}} \{s_j\}$.

  So build_it(s_m,σ_m,$Seen_m$) calls build_it(s_{m+1},σ_{m+1},$Seen$), and we need to prove that $s = s_{m+1}$ and $Seen_{m+1} = Seen = \bigcup_{j \in \{0,1,\dots,m\}} \{s_j\}$. Looking at the statements in build_it(s_m,σ_m,$Seen_m$) that call build_it(s_{m+1},σ_{m+1},$Seen$), we see that $s = s_{m+1}$ and $Seen = Seen_m \cup \{s_m\}$. So $Seen = (\bigcup_{j \in \{0,1,\dots,m-1\}} \{s_j\}) \cup \{s_m\} = \bigcup_{j \in \{0,1,\dots,m\}} \{s_j\}$ and the result follows.

⊠

Lemma 4.7. *If Kernel($\mathcal{A}$,S) calls build_it(s,σ,$Seen$), then*

$$s \in Seen \Leftrightarrow \exists \sigma_1, \sigma_2.\, \sigma = \sigma_1\, \sigma_2 \wedge \sigma_2 \neq \epsilon \wedge s^0_{\mathcal{A}} \xrightarrow{\sigma_1}_{\mathcal{A}} s \xrightarrow{\sigma_2}_{\mathcal{A}} s$$

Proof. Follows immediately from Lemma 4.6 and Lemma 4.4. ⊠

The next theorem completes the proof of the fact that the algorithm Kernel($\mathcal{A}$,S) returns a kernel for $\mathcal{A}$ w.r.t. S.

Theorem 4.8. *Let $\mathcal{K} =$ Kernel($\mathcal{A}$,S). If $\sigma \in traces(\mathcal{A})$, then $\sigma^r \in traces(\mathcal{K})$.*

Proof. Let $\tau = \sigma^r$. Since $\mathcal{A}$ is closed under S, $\tau \in traces(\mathcal{A})$; say that $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} t$. We prove a stronger property $Inv(\tau)$ by induction on the length n of σ (= the length of τ).

$$\begin{array}{lll} Inv(\tau) & = & \wedge\ \ \tau \in traces(\mathcal{K}) \\ & & \wedge\ \ \exists\, Seen. \\ & & \qquad \vee\ \ \text{Kernel}(\mathcal{A}, S) \text{ calls build_it}(t, \tau, Seen) \\ & & \qquad \vee\ \ \ \wedge\ \ \tau = \tau_1\, \tau_2\, a\, \tau_3 \\ & & \qquad \qquad \wedge\ \ s^0_{\mathcal{A}} \xrightarrow{\tau_1}_{\mathcal{A}} t' \xrightarrow{\tau_2 a}_{\mathcal{A}} t' \xrightarrow{\tau_3}_{\mathcal{A}} t \\ & & \qquad \qquad \wedge\ \ \tau_1\, \tau_2 \text{ contains no non-empty looping trace in } \mathcal{A} \\ & & \qquad \qquad \wedge\ \ \text{Kernel}(\mathcal{A}, S) \text{ calls build_it}(t', \tau_1\, \tau_2\, a, Seen) \end{array}$$

- $n = 0$.

  Then $\sigma = \epsilon$, and also $\tau = \epsilon$. So $t = s^0_{\mathcal{A}}$. Since $s^0_{\mathcal{A}} \in S_{\mathcal{K}}$, $\epsilon \in traces(\mathcal{K})$. It suffices to observe that Kernel($\mathcal{A}$,S) calls build_it($s^0_{\mathcal{A}}$,ϵ,$\emptyset$).

- $n = m + 1$.

  Induction Hypothesis (IH): $0 \leq |\rho| \leq m \Rightarrow \mathit{Inv}(\rho^r)$

  Suppose $\sigma = \sigma' b$, $\tau = \tau' c$, and $|\sigma| = |\tau| = m + 1$. Since $()^r$ is prefixed closed, $(\sigma')^r = \tau'$. Since $\tau \in \mathit{traces}(\mathcal{A})$, $\tau' \in \mathit{traces}(\mathcal{A})$. We distinguish two cases.

  - τ' does not contain a non-empty looping trace.

    We show that, for some set Seen, Kernel$(\mathcal{A}, S)$ calls build_it$(t, \tau' c, \mathit{Seen})$. By Lemma 4.4 we then know that $\tau' c \in \mathit{traces}(\mathcal{K})$, which proves $\mathit{Inv}(\tau)$.

    Assume $s^0_{\mathcal{A}} \xrightarrow{\tau'}_{\mathcal{A}} t'$. Since $(\sigma')^r = \tau'$, $\mathit{Inv}(\tau')$ holds by IH. There is no looping trace in τ', so $\tau' \in \mathit{traces}(\mathcal{K})$ and, for some set Seen', Kernel$(\mathcal{A}, S)$ calls build_it$(t', \tau', \mathit{Seen}')$. We now inspect the execution of procedure build_it for this call. By Lemma 4.7, we know that $t' \notin \mathit{Seen}'$. By Lemma 4.5 we know that after the while loop build_it$(t''$,$\tau' c'$,$\mathit{Seen}' \cup \{t'\})$ is called, for some state t'' and action c' such that $t' \xrightarrow{c'}_{\mathcal{K}} t''$ and $(\tau' c)^r = \tau' c'$. By Lemma 3.2, we know that $(\tau' c)^r = \tau' c$, so $c' = c$ and hence $t'' = t$. Thus, build_it$(t$,$\tau' c$,$\mathit{Seen}' \cup \{t'\})$ is called.

  - τ' contains a non-empty looping trace.

    Then there exist τ_1, τ_2, τ_3, $\sigma_1, \sigma_2, \sigma_3$, a, and t' such that

    $$\begin{array}{ll} \wedge & \tau = \tau_1\, \tau_2\, a\, \tau_3\, c \wedge \sigma = \sigma_1\, \sigma_2\, \sigma_3 \\ \wedge & |\tau_1| = |\sigma_1| \wedge |\tau_2\, a| = |\sigma_2| \wedge |\tau_3\, c| = |\sigma_3| \\ \wedge & s_0 \xrightarrow{\tau_1}_{\mathcal{A}} t' \xrightarrow{\tau_2 a}_{\mathcal{A}} t' \xrightarrow{\tau_3 c}_{\mathcal{A}} t \\ \wedge & \tau_1 \tau_2 \text{ contains no non-empty looping trace in } \mathcal{A} \end{array}$$

    We show that, for some set Seen, Kernel$(\mathcal{A}, S)$ calls build_it$(t', \tau_1\, \tau_2\, a, \mathit{Seen})$, and that $\tau \in \mathit{traces}(\mathcal{K})$. Trivially, $|\tau_1\, \tau_2\, a| < |\tau_1\, \tau_2\, a\, \tau_3\, c|$, and $|\tau_1\, \tau_3\, c| < |\tau_1\, \tau_2\, a\, \tau_3\, c|$. Since $()^r$ is prefix closed and $\tau^r = \tau$, $\tau_1\, \tau_2\, a = (\tau_1\, \tau_2\, a)^r$. Since $()^r$ is loop respecting, $\tau_1\, \tau_3\, c = (\tau_1\, \tau_3\, c)^r$. So we may apply IH and obtain that $\mathit{Inv}(\tau_1\, \tau_2\, a)$ and $\mathit{Inv}(\tau_1\, \tau_3\, c)$ hold. This means that $\tau_1\, \tau_2\, a \in \mathit{traces}(\mathcal{K})$, $\tau_1\, \tau_3\, c \in \mathit{traces}(\mathcal{K})$, and since there is no looping trace in $\tau_1\, \tau_2$, that, for some set Seen, Kernel$(\mathcal{A}, S)$ calls build_it$(t', \tau_1\, \tau_2\, a, \mathit{Seen})$. Since $\mathcal{K}$ is a subautomaton of $\mathcal{A}$ (Lemma 4.2), we know that $s^0_{\mathcal{K}} \xrightarrow{\tau_1}_{\mathcal{K}} t' \xrightarrow{\tau_2 a}_{\mathcal{K}} t' \xrightarrow{\tau_3 c}_{\mathcal{K}} t$, and hence $\tau_1\, \tau_2\, a \tau_3\, c \in \mathit{traces}(\mathcal{K})$.

⊠

5. Test derivation from symmetric Mealy machines

In this section we will apply the machinery developed in the previous sections to Mealy machines. There exists a wealth of test generation algorithms based on the Mealy machine model [26, 7, 5, 1]. We will show how the classical W-method [26, 7] can be adapted to a setting with symmetry. The main idea is that test derivation is not based

on the entire specification automaton, but only on a kernel of it. A technical detail here is that we do not require Mealy machines to be minimal (as already observed by [22] for the setting without symmetry). We will use the notation from Chow's paper.

Definition 5.1. A *Mealy machine* is a (deterministic) FSM $\mathcal{A}$ such that

$$\Sigma_{\mathcal{A}} = \{(i/o) \mid i \in I_{\mathcal{A}} \wedge o \in O_{\mathcal{A}}\}$$

where $I_{\mathcal{A}}$ and $O_{\mathcal{A}}$ are two finite and disjoint sets of *inputs* and *outputs*, respectively. We require that $\mathcal{A}$ is *input enabled* and *input deterministic*, i.e., for every state $s \in S_{\mathcal{A}}$ and input $i \in I_{\mathcal{A}}$, there exists precisely one output $o \in O_{\mathcal{A}}$ such that $s \xrightarrow{(i/o)}$.

Input sequences of $\mathcal{A}$ are elements of $(I_{\mathcal{A}})^*$. For ξ an input sequence of $\mathcal{A}$ and $s, s' \in S_{\mathcal{A}}$, we write $s \stackrel{\xi}{\Longrightarrow}_{\mathcal{A}} s'$ if there exists a trace σ such that $s \xrightarrow{\sigma}_{\mathcal{A}} s'$ and ξ is the result of projecting σ onto $I_{\mathcal{A}}$. In this case we write $outcome_{\mathcal{A}}(\xi, s) = \sigma$; the execution fragment γ with $first(\gamma) = s$ and $trace(\gamma) = \sigma$ is denoted by $exec_{\mathcal{A}}(s, \xi)$. A *distinguishing sequence* for two states s, s' of $\mathcal{A}$ is an input sequence ξ such that $outcome_{\mathcal{A}}(\xi, s) \neq outcome_{\mathcal{A}}(\xi, s')$. We say that ξ distinguishes s from s'. □

In Chow's paper, conformance is defined as the existence of an isomorphism between specification and implementation. Since we do not assume automata to be minimal, we will show the existence of a *bisimulation* between specification and implementation. Bisimilarity is a well-known process equivalence from concurrency theory [21]. For minimal automata, bisimilarity is equivalent to isomorphism, while for deterministic automata, bisimilarity is equivalent to equality of trace sets.

Definition 5.2. Let $\mathcal{A}$ and $\mathcal{B}$ be FSMs. A relation $R \subseteq S_{\mathcal{A}} \times S_{\mathcal{B}}$ is a *bisimulation on $\mathcal{A}$ and $\mathcal{B}$* iff

- $R(s_1, s_2)$ and $s_1 \xrightarrow{a}_{\mathcal{A}} s_1'$ implies that there is a $s_2' \in S_{\mathcal{A}}$ such that $s_2 \xrightarrow{a}_{\mathcal{B}} s_2'$ and $R(s_1', s_2')$,
- $R(s_1, s_2)$ and $s_2 \xrightarrow{a}_{\mathcal{B}} s_2'$ implies that there is a $s_1' \in S_{\mathcal{A}}$ such that $s_1 \xrightarrow{a}_{\mathcal{A}} s_1'$ and $R(s_1', s_2')$.

$\mathcal{A}$ and $\mathcal{B}$ are *bisimilar*, notation $\mathcal{A} \leftrightarroweq \mathcal{B}$, if there exists a bisimulation R on $\mathcal{A}$ and $\mathcal{B}$ such that $R(s^0_{\mathcal{A}}, s^0_{\mathcal{B}})$. We call two states $s_1, s_2 \in S_{\mathcal{A}}$ bisimilar, notation $s_1 \leftrightarroweq_{\mathcal{A}} s_2$, if there exists a bisimulation R on $\mathcal{A}$ (and $\mathcal{A}$) such that $R(s_1, s_2)$. The relation $\leftrightarroweq_{\mathcal{A}}$ is an equivalence relation on $S_{\mathcal{A}}$; a *bisimulation class* of $\mathcal{A}$ is an equivalence class of $S_{\mathcal{A}}$ under $\leftrightarroweq_{\mathcal{A}}$. □

The main ingredient of Chow's test suite is a *characterizing set* for the specification, i.e., a set of input sequences that distinguish inequivalent states by inducing different output behavior from them. In our case, two states are inequivalent if they are non-bisimilar, i.e. have different trace sets. In the presence of symmetry we will need a

characterizing set not for the entire specification automaton but only for a kernel of it. However, a kernel need not be input enabled, so two inequivalent states need not have a common input sequence that distinguishes between them. Instead we will use a characterizing set that contains for every two states of the kernel that are inequivalent in the original specification automaton, an input sequence that these states have in common in the specification and distinguishes between them.

Constructing distinguishing sequences in the specification automaton rather than in the smaller kernel is of course potentially as expensive as in the setting without symmetry, and may lead to large sequences. However, if the number of states of the kernel is small we will not need many of them, so test *execution* itself may still benefit considerably from the restriction to the kernel. Moreover, we expect that in most cases distinguishing sequences can be found in a well marked out subautomaton of the specification that envelopes the kernel.

Definition 5.3. A *test pair* for a Mealy machine $\mathcal{A}$ is a pair $\langle \mathcal{K}, W \rangle$ where $\mathcal{K}$ is a kernel of $\mathcal{A}$ and W is a set of input sequences of $\mathcal{A}$ such that the following holds. For every pair of states $s, s' \in S_{\mathcal{K}}$ such that $s \not\approx_{\mathcal{A}} s'$, W contains an input sequence ξ such that $outcome_{\mathcal{A}}(\xi, s) \neq outcome_{\mathcal{A}}(\xi, s')$. □

The proof that Chow's test suite has complete fault coverage crucially relies on the assumption that (an upper bound to) the number of states of the black box implementation is correctly estimated. Since specification and implementation are also assumed to have the same input sets and to be input enabled, this is equivalent to a correct estimate of the number of states of the implementation that can be reached from the start state by an input sequence from the specification. Similarly, we will assume that we can give an upper bound to the number of states of the black box that are reachable from the start state by an input sequence from the kernel of the specification. We call the subautomaton of the implementation generated by these states the *image* of the kernel.

Technically, the assumption on the state space of the black box is used in [7] to bound the maximum length of distinguishing sequences needed for a characterizing set for the implementation. Since, like the kernel, the image of the kernel need not be input enabled, it may be that distinguishing sequences for states of the image cannot be constructed in the image itself. Thus, it is not sufficient to estimate the number of states of the image, but we must in addition estimate the number of steps distinguishing sequences may have to take outside the image of the kernel.

Definition 5.4. Let $\mathcal{A}$ and $\mathcal{B}$ be Mealy machines with the same input set and let $\mathcal{K}$ be a kernel of $\mathcal{A}$. A $\mathcal{K}$*-sequence* is an input sequence ξ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}}$. A state s of $\mathcal{B}$ is called $\mathcal{K}$*-related* if there exists a $\mathcal{K}$-sequence ξ such that $s^0_{\mathcal{B}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{B}} s$.
We define $im_{\mathcal{K}}(\mathcal{B})$ as the subautomaton (S, Σ, E, s^0) of $\mathcal{B}$ defined by:

- $S = \{s \in S_{\mathcal{B}} \mid s \text{ is } \mathcal{K}\text{-related}\}$

- $E = \{(s, a, s') \in E_{\mathcal{B}} \mid s, s' \in S\}$
- $\Sigma = \{a \in \Sigma_{\mathcal{B}} \mid \exists\, s, s'.\, (s, a, s') \in E\}$
- $s^0 = s^0_{\mathcal{B}}$

□

Definition 5.5. A subautomaton $\mathcal{B}$ of a Mealy machine $\mathcal{A}$ is *(m_1, m_2)-self-contained in $\mathcal{A}$* when the number of bisimulation classes Q of $\mathcal{A}$ such that $Q \cap S_{\mathcal{B}} \neq \emptyset$ is m_1, and for every pair of states s, s' of $\mathcal{B}$ such that $s \not\simeq_{\mathcal{A}} s'$, there exist input sequences ξ_1, ξ_2 of $\mathcal{A}$ of length at most m_1, m_2, respectively, such that $s \stackrel{\xi_1}{\Longrightarrow}_{\mathcal{B}}$, $s' \stackrel{\xi_1}{\Longrightarrow}_{\mathcal{B}}$, and $outcome_{\mathcal{A}}(\xi_1\xi_2, s) \neq outcome_{\mathcal{A}}(\xi_1\xi_2, s')$. □

The next lemma is a generalization of [7]'s Lemma 0.

Lemma 5.6. *Let $\mathcal{A}$ and $\mathcal{B}$ be Mealy machines with the same input set I and let $\langle \mathcal{K}, W \rangle$ be a test pair for $\mathcal{A}$. Let $\mathcal{C} = im_{\mathcal{K}}(\mathcal{B})$. Suppose that:*

1. *$\mathcal{C}$ is (m_1, m_2)-self-contained in $\mathcal{B}$.*
2. *W distinguishes between n bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$.*

Then for every two states s and s' of $\mathcal{C}$ such that $s \not\simeq_{\mathcal{B}} s'$, $I^{m_1-n}\, I^{m_2}\, W$ distinguishes s from s'.

Proof. By induction on $j \in \{0, \ldots, m_1 - n\}$ we prove that there exist $j + n$ bisimulation classes Q of $\mathcal{B}$ with $Q \cap S_{\mathcal{C}} \neq \emptyset$ such that $I^j\, I^{m_2}\, W$ distinguishes between them. This proves the result, since, by assumption 1, the number of bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is m_1.

- $j = 0$. By assumption 3, W already distinguishes between n bisimulation classes of $\mathcal{B}$ with $Q \cap S_{\mathcal{C}} \neq \emptyset$, so surely $I^{m_2}\, W$ distinguishes at least these n classes.
- $j = k + 1$. If $I^k\, I^{m_2}\, W$ already distinguishes between $k + 1$ bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$, we are done. So suppose not. Then there exist two distinct bisimulation classes Q_1 and Q_2 of $\mathcal{B}$ whose intersection with $S_{\mathcal{C}}$ is non-empty, such that $I^k\, I^{m_2}\, W$ does not distinguish Q_1 from Q_2. So there exist states $s_1 \in Q_1 \cap S_{\mathcal{C}}$ and $s_2 \in Q_2 \cap S_{\mathcal{C}}$ of $\mathcal{C}$ such that $s_1 \not\simeq_{\mathcal{B}} s_2$ but $I^k\, I^{m_2}\, W$ does not distinguish s_1 from s_2. Since $\mathcal{C}$ is (m_1, m_2)-self-contained in $\mathcal{B}$, we can define the smallest number $l \leq m_1$ such that $I^l\, I^{m_2}\, W$ contains an input sequence ξ such that $outcome_{\mathcal{B}}(\xi, s_1) \neq outcome_{\mathcal{B}}(\xi, s_2)$. So there exist states t_1 and t_2 of $\mathcal{C}$ (among the $(l - (k + 1))^{th}$ successors of s_1 and s_2, respectively) such that $I^k\, I^{m_2}\, W$ does not distinguish t_1 from t_2 whereas $I^{k+1}\, I^{m_2}\, W$ does distinguish t_1 from t_2. Hence $I^{k+1}\, I^{m_2}\, W$ distinguishes the bisimulation classes of $\mathcal{B}$ to which t_1 and t_2 belong.

⊠

This result allows us to construct a characterizing set $Z = I^{m_1-n}\, I^{m_2}\, W$ for the image of the kernel in the implementation. The test suite resulting from the W-method consists of all concatenations of sequences from a *transition cover* P for the specification with sequences from Z.

Definition 5.7. A *transition cover* for the kernel of a Mealy machine $\mathcal{A}$ is a finite collection P of input sequences of $\mathcal{A}$, such that $\epsilon \in P$ and, for all transitions $s \xrightarrow{(i/o)} s'$ of $\mathcal{K}$, P contains input sequences ξ and $\xi\, i$ such that $s^0_{\mathcal{K}} \overset{\xi}{\Longrightarrow}_{\mathcal{K}} s$. □

Now follows the main theorem.

Theorem 5.8. *Let Spec and Impl be Mealy machines with the same input set I, and assume $\langle \simeq, ()^r \rangle$ is a symmetry on Spec such that Impl is closed under $\simeq$. Let $\langle \mathcal{K}, W \rangle$ be a test pair for Spec. Write $\mathcal{C} = im_{\mathcal{K}}(Impl)$. Suppose*

1. *The number of bisimulation classes Q of Spec such that $Q \cap S_{\mathcal{K}} \neq \emptyset$ is n.*
2. *$\mathcal{C}$ is (m_1,m_2)-self-contained in Impl.*
3. *For all $\sigma \in P$ and $\tau \in I^{m_1-n}\, I^{m_2}\, W$*

$$outcome_{Spec}(\sigma\,\tau, s^0_{Spec}) = outcome_{Impl}(\sigma\,\tau, s^0_{Impl}) \qquad (\star)$$

Then Spec $\leftrightarrows$ Impl.

Proof. *Spec* and *Impl* are deterministic, so it suffices to prove $traces(Spec) = traces(Impl)$. Since *Spec* is input enabled and *Impl* is input deterministic, it then suffices to prove that $traces(Spec) \subseteq traces(Impl)$. Using that *Impl* is closed under S, this follows immediately from the first item of the following claim.

Claim. *For every $\sigma \in traces(Spec)$, with $\sigma^r = \tau$ and $s^0_{\mathcal{K}} \xrightarrow{\tau}_{\mathcal{K}} r$ we have:*

1. *$\tau \in traces(Impl)$*
2. *For every $\xi \in P$ such that $s^0_{\mathcal{K}} \xrightarrow{\xi}_{\mathcal{K}} r$: if $s^0_{Impl} \xrightarrow{\tau}_{Impl} u$ and $s^0_{Impl} \overset{\xi}{\Longrightarrow}_{Impl} u'$ then $u \leftrightarrows_{\mathcal{I}} u'$.*

where $\mathcal{I}$ abbreviates Impl.

Proof of claim. Write $Z = I^{m_1-n}\, I^{m_2}\, W$. Note that, by construction of W, W distinguishes between n bisimulation classes of *Spec* whose intersection with $S_{\mathcal{K}}$ is non-empty. So, since ($\star$) holds, W distinguishes between at least n bisimulation classes of *Impl* whose intersection with $S_{\mathcal{C}}$ is non-empty. Thus we can use Lemma 5.6.

The proof of the claim proceeds by induction on the length n of σ.

- $n = 0$. So $\sigma = \epsilon = \tau$. Then certainly $\tau \in \mathit{traces}(\mathit{Impl})$. As to item 2). Consider an input sequence $\xi \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}} s^0_{\mathcal{K}}$ and assume $s^0_{\mathit{Impl}} \stackrel{\xi}{\Longrightarrow}_{\mathit{Impl}} u'$. We have to show that $s^0_{\mathit{Impl}} \leftrightarrows_{\mathcal{I}} u'$.

  Since ξ and ϵ are elements of P and lead in *Spec* to the same state, it follows from $(\star)$ that for all $\rho \in Z$, $\mathit{outcome}_{\mathit{Impl}}(\rho, s^0_{\mathit{Impl}}) = \mathit{outcome}_{\mathit{Impl}}(\rho, u')$. Hence, by Lemma 5.6, $s^0_{\mathit{Impl}} \leftrightarrows_{\mathcal{I}} u'$.

- $n > 0$. Write $\sigma = \sigma'\,(i/o)$. By induction hypothesis $(\sigma')^r = \tau' \in \mathit{traces}(\mathcal{K}) \cap \mathit{traces}(\mathit{Impl})$. Say that $s^0_{\mathcal{K}} \stackrel{\tau'}{\longrightarrow}_{\mathcal{K}} r'$. Since $\mathcal{K}$ is a kernel of *Spec*, there exists an action (i'/o') such that $(\sigma'\,(i/o))^r = \tau'\,(i'/o')$ and, for some state r, $r' \stackrel{i'/o'}{\longrightarrow}_{\mathcal{K}} r$. Since $r' \in S_{\mathcal{K}}$, there exist input sequences ξ', $\xi'\,i' \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi'}{\Longrightarrow}_{\mathcal{K}} r'$.

  Let $s^0_{\mathit{Impl}} \stackrel{\tau'}{\longrightarrow}_{\mathit{Impl}} u$ and $s^0_{\mathit{Impl}} \stackrel{\xi'}{\Longrightarrow}_{\mathit{Impl}} u'$. By induction hypothesis, item 3), $u \leftrightarrows_{\mathcal{I}} u'$. Since $\mathit{outcome}_{\mathit{Spec}}(\xi'\ i', s^0_{\mathit{Spec}}) = \mathit{outcome}_{\mathit{Impl}}(\xi'\ i', s^0_{\mathit{Impl}})$, there exists a (unique) state v' such that $u' \stackrel{i'/o'}{\longrightarrow}_{\mathcal{I}} v'$. Since $u \leftrightarrows_{\mathcal{I}} u'$, there exists a (unique) state v such that $u \stackrel{i'/o'}{\longrightarrow}_{\mathcal{I}} v$. So $\tau'\,(i'/o') \in \mathit{traces}(\mathit{Impl})$. Because *Impl* is input deterministic, $v \leftrightarrows_{\mathcal{I}} v'$.

  Finally, we have to prove, for all $\xi \in P$ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}} r$: for the unique state w such that $s^0_{\mathit{Impl}} \stackrel{\xi}{\Longrightarrow}_{\mathit{Impl}} w$, we have $w \leftrightarrows_{\mathcal{I}} v$. Consider such a ξ. Since $v' \leftrightarrows_{\mathcal{I}} v$ it suffices to prove that $w \leftrightarrows_{\mathcal{I}} v'$. Since $\xi'\,i'$ and ξ are elements of P and lead to the same state in *Spec*, it follows from $(\star)$ that, for all $\rho \in Z$, $\mathit{outcome}_{\mathit{Impl}}(\rho, v) = \mathit{outcome}_{\mathit{Impl}}(\rho, w)$. Hence, by Lemma 5.6, $v' \leftrightarrows_{\mathcal{I}} w$.

$\boxtimes$

$\boxtimes$

6. Patterns

In this section we describe symmetries based on *patterns*. A pattern is an FSM, together with a set of permutations of its set of actions, so-called *transformations*. The FSM is a *template* for the behavior of a system, while the transformations indicate how this template may be filled out to obtain symmetric variants that cover the full behavior of the system.

In [19] an interesting example automaton is given for a symmetric protocol, representing the behavior of two peer hosts that may engage in the ATM call setup procedure. This behavior is completely symmetric in the identity of the peers. An FSM representation is given in Figure 2. Here, !<action>(i) means output of the ATM service to caller i, and ?<action>(i) means input from caller i to the ATM service. So, action ?set_up(1) denotes the request from caller 1 to the ATM service, to set up a call to caller 2. A set_up request is followed by an acknowledgement in the form of call_proc if the service can be performed. Then, action conn indicates that the called side is ready

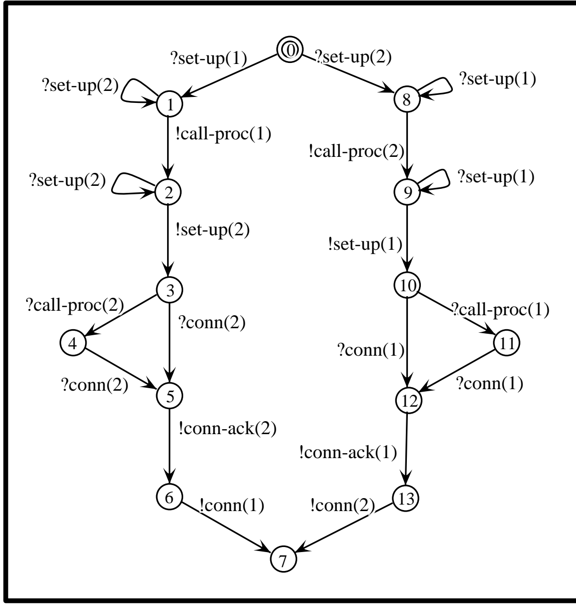

Figure 2: The ATM call setup protocol

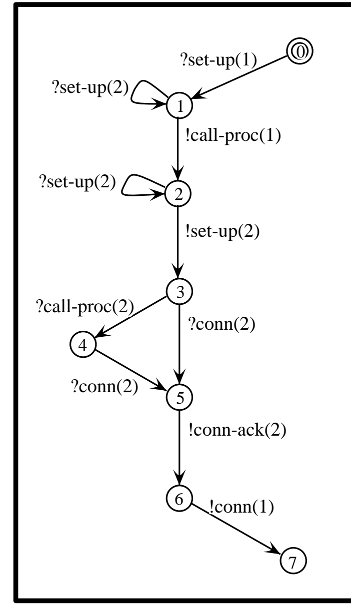

Figure 3: A template

for the connection, which is acknowledged by conn_ack. A caller may skip sending call_proc, if it can already send conn instead (transition from state 3 to 5 and from 10 to 12 in Figure 2).

Here, a typical template is the subautomaton representing the call set up as initiated by a single initiator (e.g. caller 1), and the transformation will be the permutation of actions generated by swapping the roles of initiator and responder. Such a template is displayed in Figure 3.

In the example of Section 7, featuring a *chatbox* that supports multiple conversations between callers, the template will be the chatting between two callers, while the transformations will shuffle the identity of the callers.

The template FSM may be arbitrarily complex; intuitively, increasing complexity indicates a stronger symmetry assumption on the black box implementation.

To define pattern based symmetries, we need some terminology for partial functions and multisets. If $f : A \to B$ is a partial function and $a \in A$, then $f(a) \downarrow$ means that $f(a)$ is defined, while $f(a) \uparrow$ means that $f(a)$ is not defined. A *multiset* over A is a set of the form $\{(a_1, n_1), \ldots, (a_k, n_k)\}$ where, for $1 \leq i \leq k$, a_i is an element of A and $n_i \in \mathsf{N}$ denotes its *multiplicity*. We use $[f(x)|\ \mathrm{cond}(x)]$ as a shorthand for the multiset over A that is created by adding, for every single $x \in A$, a copy of $f(x)$ if the condition $\mathrm{cond}(x)$ holds.

Definition 6.1 *(Patterns).* A *pattern* $\mathcal{P}$ is a pair $\langle \mathcal{T}, \Pi \rangle$ where $\mathcal{T}$ is an FSM, called the *template* of $\mathcal{P}$, and Π is a finite set of permutations of $\Sigma_{\mathcal{T}}$, which we call *transformations*.

Given a sequence $\langle f_1, \dots, f_n \rangle$ of (partial) functions $f_1, \dots, f_n : \Pi \to E_{\mathcal{T}}$, we denote with $exec(\langle f_1, \dots, f_n \rangle, \pi)$ the sequence of edges obtained by taking for each function f_i, $0 \leq i \leq n$, the edge e (if any) such that $f_i(\pi) = e$.

In the remainder of this section, we fix an FSM $\mathcal{A}$ and a pattern $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$.

Below we will define how $\mathcal{P}$ defines a symmetry of the behavior of an FSM $\mathcal{A}$. Each transformation $\pi \in \Pi$ gives rise to a copy $\pi(\mathcal{T})$ of $\mathcal{T}$ obtained by renaming the actions according to π. Each such copy is a particular instantiation of the template. Intuitively, the trace set of $\mathcal{A}$ is included in the trace set of the parallel composition of the copies $\pi(\mathcal{T})$, indexed by elements of Π, with enforced synchronization over all actions of $\mathcal{A}$. Using that traces of $\mathcal{A}$ are traces of the parallel composition, we will define the symmetry relation on traces in terms of the behavior of the copies and permutations of the index set Π.

The following definition rephrases the inclusion requirement above in such a way that the relation $\simeq$ and a representative function for it can be formulated succinctly. In particular, if $\mathcal{A}$ *is* the parallel composition of the copies of $\mathcal{T}$, the requirement in this definition apply.

Definition 6.2. Let $\sigma = a_1 \cdots a_n$ be an element of $(\Sigma_{\mathcal{A}})^*$. A *covering* of σ by $\mathcal{P}$ is a sequence $\langle f_1, \dots, f_n \rangle$ of partial functions $f_i : \Pi \to E_{\mathcal{T}}$ with *non-empty* domain such that for every $\pi \in \Pi$ and $1 \leq i \leq n$:

1. If $f_i(\pi) = e$ then $a_i = \pi(\mathsf{act}(e))$.
2. The sequence $exec(\langle f_1, \dots, f_i \rangle, \pi)$ induces an execution γ_i of $\mathcal{T}$.
3. If the sequence $trace(\gamma_{i-1})\, a_i$ is a trace of $\pi(\mathcal{T})$ then $f_i(\pi) \downarrow$.

We say that $\mathcal{P}$ covers σ if there exists a covering of σ by $\mathcal{P}$.

We call $\mathcal{P}$ *loop preserving* when the following holds. Suppose $\sigma_1\, \sigma_2 \in traces(\mathcal{A})$ is covered by $\langle f_1, \dots, f_n, g_1, \dots, g_m \rangle$ and σ_2 is a looping trace. Then for all $\pi \in \Pi$,

$$last(exec(\langle f_1, \dots, f_n \rangle, \pi)) = last(exec(\langle f_1, \dots, f_n, g_1, \dots, g_m \rangle, \pi))$$

□

Intuitively, these requirements mean the following. The 'non-empty domain' requirement for the partial functions f_i ensures the inclusion of the trace set of $\mathcal{A}$ in the trace set of the parallel composition of copies of $\mathcal{T}$. Requirements 1 and 2 express that a covering should not contain 'junk'. Requirement 3 corresponds to the enforced synchronization of actions of the parallel composition.

Lemma 6.3. *For every trace σ, there exists at most one covering of σ by $\mathcal{P}$.*

Proof. Since $\mathcal{T}$ is deterministic, coverings of σ are uniquely determined by $\mathcal{T}$. $\boxtimes$

Two traces σ and τ of the same length n that are covered by $\mathcal{P}$, are *variants* of each other if at each position i, $1 \leq i \leq n$, of σ and τ the following holds. The listings for σ and τ, respectively, of the copies $\pi(\mathcal{T})$ that participate in the action at position i, the states these copies are in before participating, and the edge they follow by participating, are equal up to a permutation of Π. Then, two traces of the same length are *symmetric* iff they are either both not covered by $\mathcal{P}$ or are covered by coverings that are variants of each other.

Definition 6.4. Let σ and τ be elements of $(\Sigma_{\mathcal{A}})^n$, which $\mathcal{P}$ covers by $cov_1 = \langle f_1, \ldots, f_n \rangle$ and $cov_2 = \langle g_1, \ldots, g_n \rangle$, respectively. Then cov_1 and cov_2 are said to be *variants* of each other if for every $1 \leq i \leq n$, $[f_i(\pi) \mid \pi \in \Pi] = [g_i(\pi) \mid \pi \in \Pi]$.
We define the binary relation $\simeq_{\mathcal{P}}$ on $(\Sigma_{\mathcal{A}})^*$ by:

$$\begin{array}{lll} \sigma \simeq_{\mathcal{P}} \tau \;\Leftrightarrow & \wedge & |\sigma| = |\tau| \\ & \wedge & \vee \text{ both } \sigma \text{ and } \tau \text{ are not covered by } \mathcal{P} \\ & & \vee\ \mathcal{P} \text{ covers } \sigma \text{ and } \tau \text{ by variant coverings} \end{array}$$

It is easy to check that $\simeq_{\mathcal{P}}$ is an equivalence relation. As in Section 3, we will write $[\sigma]$ for the equivalence class of σ and $\simeq$ instead of $\simeq_{\mathcal{P}}$. $\square$

An important special case is the following. Suppose $\mathcal{A}$ consists of the parallel composition of components C_i, indexed by elements of a set I, that are identical up to their ID (which occur as parameters in the actions). Let σ and τ be traces of $\mathcal{A}$. If there exists a permutation ρ of the index set I such that for all indices $i \in I$, σ induces (up to renaming of IDs in actions) the same execution of C_i as τ induces in $C_{\rho(i)}$, then σ and τ are symmetric.

Lemma 6.5. *If $\mathcal{P}$ covers σa by $\langle f_1, \ldots, f_n \rangle$, then $\mathcal{P}$ covers σ by $\langle f_1, \ldots, f_{n-1} \rangle$.*

Lemma 6.6. *If $\mathcal{P}$ covers $\sigma\, a$ and $\tau\, b$ and $\sigma\, a \simeq \tau\, b$, then $\sigma \simeq \tau$.*

Proof. Let $\sigma\, a$ and $\tau\, b$ be covered by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_n \rangle$, respectively. By Lemma 6.5, these coverings induce the coverings $\langle f_1, \ldots, f_{n-1} \rangle$ and $\langle g_1, \ldots, g_{n-1} \rangle$ of σ and τ, respectively, which are clearly variants of each other. $\boxtimes$

The previous two lemmas together imply the following result.

Corollary 6.7. The relation $\simeq$ is *prefix closed* on $\mathcal{A}$, i.e., for every two traces $\sigma\, a$, $\tau\, b \in$ *traces*$(\mathcal{A})$, if $\sigma\, a \simeq \tau\, b$ then $\sigma \simeq \tau$.

Given the definition of $\simeq$, it is reasonable to demand that every trace of $\mathcal{A}$ is covered by $\mathcal{P}$. We will also need the following closure property. We call a binary relation R on $(\Sigma_{\mathcal{A}})^*$ *persistent on* $\mathcal{A}$ when $R(\sigma, \tau)$ and $\sigma\, a \in \mathit{traces}(\mathcal{A})$ implies that there exists an action b such that $R(\sigma\, a, \tau\, b)$.

Now we define a representative function for $\simeq$. We assume given a total, irreflexive ordering $<$ on $\Sigma_{\mathcal{A}}$. Such an ordering of course always exists, but the choice for $<$ may greatly influence the size of the kernel constructed for a symmetry based on $\mathcal{P}$.

Definition 6.8. Let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. This ordering induces a reflexive, transitive ordering $\leq$ on traces of the same length in the following way:

$$a\,\sigma \leq b\,\tau \Leftrightarrow a < b \vee (a = b \wedge \sigma \leq \tau)$$

We define σ^r as the least element of $[\sigma]$ under $\leq$. □

We will show that $()^r$ is a representative function for $\simeq$. First we prove that $()^r$ is prefix closed.

Lemma 6.9. *Suppose $\simeq$ is persistent on $\mathcal{A}$ and $\mathcal{A}$ is closed under $\simeq$. If $([\tau\, b])^r = \sigma\, a \in \mathit{traces}(\mathcal{A})$, then $([\tau])^r = \sigma$.*

Proof. By contradiction. Suppose that there exists a trace ρ such that $\rho = ([\tau])^r$ and $\rho \neq \sigma$. Note that, since $\mathcal{A}$ is closed under $\simeq$, $\tau\, b \in \mathit{traces}(\mathcal{A})$. By persistence of $\simeq$, $\rho \in [\tau]$ implies that there exists an action c such that $\rho\, c \in [\tau\, b]$. Since $\simeq$ is prefix closed on $\mathcal{A}$ (Corollary 6.7) and $\sigma\, a \in [\tau\, b]$, $\sigma \in [\tau]$. By definition of $()^r$, $\rho \leq \sigma$. But also, $\sigma a \leq \rho c$, and, by definition of $\leq$, $\sigma \leq \rho$. So $\rho = \sigma$ and we have a contradiction. ⊠

To show that $()^r$ is loop respecting, we first prove two auxiliary results.

Lemma 6.10. *If $\mathcal{P}$ covers σ and τ by $\langle f_1, \ldots, f_n\rangle$ and $\langle g_1, \ldots, g_n\rangle$, respectively, and $\sigma \simeq \tau$, then for every $1 \leq i \leq n$:*

$$\begin{aligned}&[\mathit{last}(\mathit{exec}(\langle f_1, \ldots, f_i\rangle, \pi)) \mid \pi \in \Pi \wedge f_i(\pi)\downarrow] \\ = \quad &[\mathit{last}(\mathit{exec}(\langle g_1, \ldots, g_i\rangle, \pi)) \mid \pi \in \Pi \wedge g_i(\pi)\downarrow]\end{aligned}$$

Proof. Since $\sigma \simeq \tau$ we know that for every $1 \leq i \leq n$, $[f_i(\pi) \mid \pi \in \Pi] = [g_i(\pi) \mid \pi \in \Pi]$. From this the result follows immediately. ⊠

Lemma 6.11. *Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 6.8. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. If $\sigma_1\, \sigma_2\, \sigma_3 \in \mathit{traces}(\mathcal{A})$ and σ_2 is a looping trace, then*

$$\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau \text{ iff } \sigma_1\, \sigma_2\, \sigma_3 \simeq \sigma_1\, \sigma_2\, \tau.$$

Proof. Write $|\sigma_1| = n$, $|\sigma_2| = m$, and $|\sigma_3| = |\tau| = k$.
Let $\langle f_1, \dots, f_n, g_1, \dots, g_m, h_1, \dots, h_k \rangle$ cover $\sigma_1\,\sigma_2\,\sigma_3$.
By Lemma 6.5, $\langle f_1, \dots, f_n, g_1, \dots, g_m \rangle$ covers $\sigma_1\,\sigma_2$ and $\langle f_1, \dots, f_n \rangle$ covers σ_1. Since $\simeq$ is loop preserving on $\mathcal{A}$, we know that for every $\pi \in \Pi$

$$last(exec(\langle f_1, \dots, f_n \rangle, \pi)) = last(exec(\langle f_1, \dots, f_n, g_1, \dots, g_m \rangle, \pi)) \tag{6.1}$$

So $\langle f_1, \dots, f_n, h_1, \dots, h_k \rangle$ covers $\sigma_1\,\sigma_3$.

"$\Rightarrow$" Since $\sigma_1\,\sigma_3 \simeq \sigma_1\,\tau$ and $\sigma_1\,\sigma_3 \in traces(\mathcal{A})$, $\sigma_1\,\tau \in traces(\mathcal{A})$.
Let $\langle f_1, \dots, f_n, h'_1, \dots, h'_k \rangle$ cover $\sigma_1\,\tau$.
From Equation 6.1 and the fact that $\langle f_1, \dots, f_n, g_1, \dots, g_m \rangle$ covers $\sigma_1\,\sigma_2$, it follows that $\langle f_1, \dots, f_n, g_1, \dots, g_m, h'_1, \dots, h'_k \rangle$ covers $\sigma_1\,\sigma_2\,\tau$. Since $\sigma_1\,\sigma_3 \simeq \sigma_1\,\tau$, we obtain, for every $0 \leq i \leq k$:

$$[h_i(\pi) \mid \pi \in \Pi] = [h'_i(\pi) \mid \pi \in \Pi] \tag{6.2}$$

From this fact, it follows that $\sigma_1\,\sigma_2\,\sigma_3 \simeq \sigma_1\,\sigma_2\,\tau$.

"$\Leftarrow$" Since $\sigma_1\,\sigma_2\,\sigma_3 \simeq \sigma_1\,\sigma_2\,\tau$ and $\sigma_1\,\sigma_2\,\sigma_3 \in traces(\mathcal{A})$, $\sigma_1\,\sigma_2\,\tau \in traces(\mathcal{A})$.
Let $\langle f_1, \dots, f_n, g_1, \dots, g_m, h'_1, \dots, h'_k \rangle$ cover $\sigma_1\,\sigma_2\,\tau$. From Equation 6.1, it follows that $\langle f_1, \dots, f_n, h'_1, \dots, h'_k \rangle$ covers $\sigma_1\,\tau$. Since $\sigma_1\,\sigma_2\,\sigma_3 \simeq \sigma_1\,\sigma_2\,\tau$, we obtain, for every $0 \leq i \leq k$:

$$[h_i(\pi) \mid \pi \in \Pi] = [h'_i(\pi) \mid \pi \in \Pi] \tag{6.3}$$

From this, it follows that $\sigma_1\,\sigma_3 \simeq \sigma_1\,\tau$.

⊠

Finally, we prove that $()^r$ is loop respecting.

Lemma 6.12. *Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 6.8. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. If $([\sigma_1\,\sigma_2\,\sigma_3])^r = \sigma_1\,\sigma_2\,\sigma_3 \in traces(\mathcal{A})$ and σ_2 is a looping trace, then $([\sigma_1\,\sigma_3])^r = \sigma_1\,\sigma_3$.*

Proof. By contradiction. Suppose that $([\sigma_1\,\sigma_3])^r = \tau_1\,\tau_3$ and $\tau_1\,\tau_3 \neq \sigma_1\,\sigma_3$. By Lemma 6.9, $([\sigma_1])^r = \sigma_1$, and $\tau_1 = ([\sigma_1])^r$, so $\tau_1 = \sigma_1$. By definition of $()^r$, $\sigma_1\,\tau_3 \leq \sigma_1\,\sigma_3$ and $\sigma_1\,\tau_3 \simeq \sigma_1\,\sigma_3$. By Lemma 6.11, $\sigma_1\,\sigma_2\,\sigma_3 \simeq \sigma_1\,\sigma_2\,\tau_3$. Since $\sigma_1\,\sigma_2\,\sigma_3 = ([\sigma_1\,\sigma_2\,\sigma_3])^r$, $\sigma_1\,\sigma_2\,\sigma_3 \leq \sigma_1\,\sigma_2\,\tau_3$, and by definition of $\leq$, $\sigma_1\,\sigma_3 \leq \sigma_1\,\tau_3$. Since also $\sigma_1\,\tau_3 \leq \sigma_1\,\sigma_3$, $\sigma_1\tau_3 = \sigma_1\sigma_3$, and we have a contradiction. So $\sigma_1\,\sigma_3 = ([\sigma_1\,\sigma_3])^r$. ⊠

The next result allows us to use the pattern-approach for computing a kernel. In our example of the ATM switch, we have computed the kernel from the FSM in Figure 2, using the symmetry induced by the template in Figure 3 and an ordering $<$ that obeys the relation ?set_up(1) $<$?set_up(2). Not surprisingly, the resulting kernel is identical to the template.

Theorem 6.13. *Suppose $\mathcal{P}$ is a loop preserving pattern on $\mathcal{A}$ and let $<$ be a total, irreflexive ordering on $\Sigma_{\mathcal{A}}$. Let $()^r$ be as in Definition 6.8. Suppose every trace of $\mathcal{A}$ is covered by $\mathcal{P}$, $\mathcal{A}$ is closed under $\simeq$, and $\simeq$ is persistent on $\mathcal{A}$. Then $\langle \simeq, ()^r \rangle$ is a symmetry on $\mathcal{A}$.*

Proof. We have to show that $()^r$ is a representative function for $\simeq$. It is immediate that $\sigma^r \simeq \sigma$ and for all τ such that $\sigma \simeq \tau$, $\tau^r = \sigma^r$. The requirement that $()^r$ is prefix closed follows from Lemma 6.9. That $()^r$ is loop respecting follows from Lemma 6.12. ⊠

The following lemma is an extra ingredient for making the implementation of the algorithm Kernel from Section 4 more efficient. The implementation itself is described in Section 7.

Lemma 6.14. *Suppose $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$ is a pattern on $\mathcal{A}$, that covers σ and τ by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_m \rangle$, respectively.*
If $s^0_{\mathcal{A}} \xrightarrow{\sigma}_{\mathcal{A}} s$, $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} s$ and for each π in Π: $last(exec(\langle f_1, \ldots, f_n \rangle, \pi)) = last(exec(\langle g_1, \ldots, g_m \rangle, \pi))$, then for each ρ such that $s \xrightarrow{\rho}_{\mathcal{A}}$:

$$\langle f_1, \ldots, f_n, h_1, \ldots, h_k \rangle \text{ covers } \sigma\rho \Leftrightarrow \langle g_1, \ldots, g_m, h_1, \ldots, h_k \rangle \text{ covers } \tau\rho$$

Lemma 6.15. *Suppose $\langle \mathcal{P}, ()^r \rangle$ is a symmetry on $\mathcal{A}$, $()^r$ is as in Definition 6.8, and $\mathcal{P} = \langle \mathcal{T}, \Pi \rangle$ covers σ and τ by $\langle f_1, \ldots, f_n \rangle$ and $\langle g_1, \ldots, g_m \rangle$, respectively.*
If $s^0_{\mathcal{A}} \xrightarrow{\sigma}_{\mathcal{A}} s$, $s^0_{\mathcal{A}} \xrightarrow{\tau}_{\mathcal{A}} s$, for each π in Π: $last(exec(\langle f_1, \ldots, f_n \rangle, \pi)) = last(exec(\langle g_1, \ldots, g_m \rangle, \pi))$, and $\sigma = \sigma^r$ and $\tau = \tau^r$, then for each ρ such that $s \xrightarrow{\rho}_{\mathcal{A}}$:

$$\sigma\rho = (\sigma\rho)^r \Leftrightarrow \tau\rho = (\tau\rho)^r$$

Proof. We only prove "$\Rightarrow$", the other direction then follows immediately.
By contradiction. Suppose $s \xrightarrow{\rho}_{\mathcal{A}}$, $\sigma\rho = (\sigma\rho)^r$ and $(\tau\rho)^r = \tau\rho'$ with $\rho \neq \rho'$. By definition of $()^r$, we know that $\tau\rho \simeq \tau\rho'$. By Lemma 6.14, we know that the covering of the ρ-part in $\tau\rho$ must be equal to the covering of the ρ-part in $\sigma\rho$, and likewise for the ρ'-part in $\tau\rho'$ and $\sigma\rho'$. Then certainly $\sigma\rho \simeq \sigma\rho'$ must hold. By unicity of representatives, $\sigma\rho = (\sigma\rho')^r$. From Definition 6.8 we then obtain that $\sigma\rho \leq \sigma\rho'$ and $\tau\rho' \leq \tau\rho$, so $\rho \leq \rho'$ and $\rho' \leq \rho$. This yields a contradiction with the assumption that $\rho \neq \rho'$. ⊠

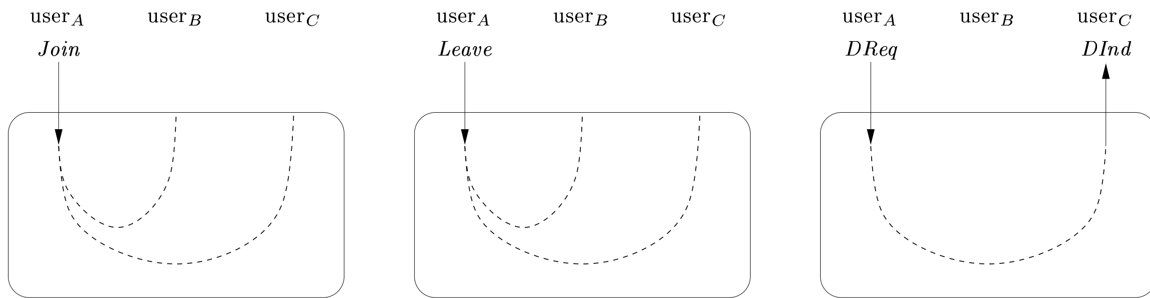

Figure 4: The chatbox protocol service

7. Example: Testing a chatbox

In this section we report on some initial experiments in the application of symmetry to the testing of a *chatbox*. Part of the test generation trajectory was implemented: we used the tool environment Open/Cæsar[14] for prototyping the algorithm Kernel from Section 3. We work with a pattern based symmetry (Section 6) and apply the test derivation method from Section 5.

A chatbox offers the possibility to talk with users connected to the chatbox. After one joins (connects to) the chatbox, one can talk with all other connected users, until one leaves (disconnects). One can only join if not already present, and one can leave at any time. For simplicity, we assume that every user can at each instance talk with at most one user. Moreover, we demand that a user waits for a reply before talking again (unless one of the partners leaves). Finally, we abstract from the contents of the messages, and consider only one message. The service primitives provided by the chatbox are thus the following; Join, Leave, DReq, and DInd, with the obvious meaning (see Figure 4). For lack of space, we do not give the full formal specification of the chatbox or its template.

What we test for is the service of the chatbox as a whole, such as it may be offered by a vendor, rather than components of its implementation, which we (the "customers") are not allowed to, or have no desire to, inspect.

This example was inspired by the conference protocol presented in [24]. Some changes were made, all stemming from the need to keep the protocol manageable for experiments without losing the symmetry pursued. We mention the absence of queues and multicasts and the restriction to the number of outstanding messages. Also, we ignore the issues of test contexts, test architectures, and points of control and observation.

The symmetry inherent in the protocol is immediate: pairs of talking users can be replaced by other pairs of talking users, as long as this is done systematically according to Definitions 6.2 and 6.4. As an example, the trace in which user 1 joins, leaves and joins again, is symmetric to the trace in which user 1 joins and leaves, after which user 2 joins. The essence is that after user 1 has joined and left, this user is at the same point as all the other users not present, so all new join actions are symmetric. Note

that this symmetry is more general than a symmetry induced solely by a permutation of actions or IDs of users. Thus the template $\mathcal{T}$ used for the symmetry basically consists of the conversation between two users, including joining and leaving, while the transformations π in the set Π shuffle the identity of users. We feel that it is a reasonable assumption that the black implementation offering the service indeed is symmetric in this sense.

We have applied the machinery to chatboxes with up to 4 users. We also considered a (much simpler) version of the protocol without joining and leaving. Still, a chatbox with only 4 users and no joining or leaving already has 4096 reachable states.

We start the test generation by computing a kernel for these specifications. Our prototype is able to find a significantly smaller Mealy machine as a kernel for each of the models, provided that it is given a suitable ordering $<$ (see Definition 6.8) on the actions symbols for its representative function. The kernels constructed consist of interleavings of transformations of the pattern, constrained by the symmetry and the ordering $<$.

For instance, in a chatbox with 3 users and no joining and leaving, we take the ordering $<$ defined as follows. "Sending a message from i_1 to j_1" $<$ "sending a message from i_2 to j_2" if $(i_1 < i_2)$ or if $(i_1 = i_2$ and $j_1 < j_2)$, and "sending a reply from i_1 to j_1" $<$ "sending a reply from i_2 to j_2" if $(i_1 > i_2)$ or if $(i_1 = i_2$ and $j_1 > j_2)$.

Using this ordering, the kernel only contains those traces in which first messages from user 1 are sent, then messages from user 2 and finally messages from user 3, while the sending of replies is handled in the reverse order. Each trace with different order of sending messages can then be computed from a trace of this kernel, which is exactly what Theorem 4.8 states. This technique of dealing with traces is reminiscent of partial order techniques [16].

Our experiments so far have revealed that for chatboxes with joining and leaving, the kernel is approximately half the size. When considering chatboxes without joining and leaving, the size of the kernel is reduced much more. This difference is due to the fact that, since one cannot send a message to a user that has left, joining and leaving obstructs the symmetry in messages being sent. Of course, the algorithm should be run on more and larger examples to get definite answers about possible size reduction.

Given the computed kernels, we can construct test pairs by determining for each kernel a set of input sequences W that constitutes a *characterizing set* for the kernel (as defined in Definition 5.3). Although this part has not yet been automated, it is easily seen by a generic argument that for every pair of inequivalent (non-bisimilar) states very short distinguishing sequences exist. It is easy to devise a transition cover for a kernel, the size of which is proportional to the size of the kernel.

As shown in Theorem 5.8, the size of the test suite to be generated will depend on the magnitude of two numbers m_1 and m_2, indicating the search space for distinguishing sequences for the image of the kernel in the implementation. This boils down to the following questions: (1) What is the size of the image part of the implementation for this kernel? (2) What is the size of a minimal distinguishing experience for each two

inequivalent (non-bisimilar) states in the image part of the implementation? (3) How many steps does a distinguishing sequence perform outside the image of the kernel? These questions are variations of the classical state space questions for black box testing. For practical reasons, these numbers are usually taken to be not much larger than the corresponding numbers for the specification.

The algorithm Kernel (see Figure 1) was implemented using the Open/Cæsar [14] tool set. An interesting detail here is that the algorithm uses two finite state machines: one for the specification that is reduced to a kernel, and one for the template of the symmetry, which is used to determine (as an oracle) whether two traces are symmetric. To enable this, Open/Cæsar interface had to be generalized somewhat so that it is now able to explore several labeled transition systems at the same time. We have the experience that Open/Cæsar is suitable for prototyping exploration algorithms such as Kernel.

8. Future work

We have introduced a general, FSM based, framework for exploiting symmetry in specifications and implementations in order to reduce the amount of tests needed to establish correctness. The feasibility of this approach has been shown in a few experiments.

However, a number of open issues remain. We see the following steps as possible, necessary and feasible. On the theoretical side we would like to (1) construct algorithms for computing and checking symmetries, and (2) determine conditions that are on the one hand sufficient to guarantee symmetry, and on the other hand enable significant optimizations of the algorithms. On the practical side we would like to (1) generate and execute tests for real-life implementations, and (2) continue prototyping for the whole test generation trajectory.

Acknowledgments

We thank Frits Vaandrager for suggesting the transfer of model checking techniques to test theory, and Radu Mateescu and Hubert Garavel for their invaluable assistance (including adding functionality!) with the Open/Cæsar tool set. We also thank Jan Tretmans and the anonymous referees for their comments on this paper.

References

1. A.V. Aho, A.T. Dahbura, D. Lee, and M.Ü. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.
2. K. Ajami, S. Haddad and J-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In Steffen [23], pages 52–67.
3. E. Brinksma. A theory for the derivation of tests. In S. Aggrawal and K. Sabani, editors, *Protocol Specification Testing and Verification*, Volume VIII, pages 63–74. North-Holland, 1988.

4. E. Brinksma, J. Tretmans and L. Verhaard. A framework for test selection. In B. Jonsson, J. Parrow and B. Pehrson, editors, *Protocol Specification Testing and Verification*, Volume XI, pages 233-248. North-Holland, 1991.

5. W.Y.L. Chan, S.T. Vuong, and M.R. Ito. An improved protocol test generation procedure based on UIOs. In *Proceedings of the ACM Symposium on Communication Architectures and Protocols*, pages 283–294, 1989.

6. O. Charles, and R. Groz. Basing test coverage on a formalization of test hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, Volume 10, pages 109–124. Chapman & Hall, 1997.

7. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

8. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [9], pages 450–462.

9. C. Courcoubetis, editor. *Proceedings* 5th *International Conference on Computer Aided Verification (CAV '93)*. Lecture Notes in Computer Science 697. Springer-Verlag, 1993.

10. E.A. Emerson, S. Jha and D. Peled. Combining partial order and symmetry reductions. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pages 19–34. Lecture Notes in Computer Science 1217. Springer-Verlag, 1997.

11. E.A. Emerson and A.P. Sistla. Symmetry and model checking. In Courcoubetis [9], pages 463–478.

12. E.A. Emerson and A.P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

13. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 16(6):591–603, 1991.

14. H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In Steffen [23], pages 68–84. For more information on the tool set, see `http://www.inrialpes.fr/vasy/pub/cadp.html`.

15. M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96. Lecture Notes in Computer Science 915. Springer-Verlag, 1995.

16. P. Godefroid. *Partial-order methods for the verification of concurrent systems – An approach to the state-explosion problem.* Lecture Notes in Computer Science 1032. Springer-Verlag, 1996.

17. O. Grumberg, editor. *Proceedings* 9th *International Conference on Computer Aided Verification (CAV '97)*. Lecture Notes in Computer Science 1254. Springer-Verlag, 1997.
18. V. Gyuris and A.P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In Grumberg [17], pages 232–243.
19. S. Kang and M. Kim. Interoperability test suite derivation for symmetric communication protocols. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/ PSTV XVII '97)*, pages 57–72. Chapman & Hall, 1997.
20. F. Michel, P. Azema, and K. Drira. Selective generation of symmetrical test cases. In B. Baumgarten, H.-J. Burkhardt and A. Giessler, editors, *Testing of Communicating Systems*, Volume 9, pages 191–206. Chapman & Hall, 1996.
21. R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.
22. A. Petrenko, T. Higashino, and T. Kaji. Handling redundant and additional states in protocol testing. In A. Cavalli and S. Budkowski, editors, *Protocol Test Systems*, Volume VIII, pages 307–322. Chapman & Hall, 1995.
23. B. Steffen, editor. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*. Lecture Notes in Computer Science 1384. Springer-Verlag, 1998.
24. R. Terpstra, L. Fereira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*, 1996. Also published as Technical Report CTIT 96-21, University of Twente, The Netherlands, 1996.
25. J. Tretmans. A theory for the derivation of tests. In *Formal Description Techniques (FORTE II '89)*. North-Holland, 1989.
26. M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.