Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Exploiting Symmetry in Protocol Testing

J.M.T. Romijn, J.G. Springintveld

Software Engineering (SEN)

SEN-R9918 August 31, 1999

Report SEN-R9918
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

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 [28, 7] for test derivation. Finally, we focus on symmetries defined in terms of repeating patterns.

1991 Mathematics Subject Classification: 68M15, 68Q05, 68Q68, 94C12
1991 ACM Computing 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 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

* 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.

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 [16, 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 [22], similar techniques have been developed for a test generation methodology based on labeled transition systems, success trees and canonical testers [3, 27]. 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 [22] start from different testing methodologies, it is not easy to compare them. In [22], 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 [22].

In [21], 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 [28, 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 [26]. Finally, we discuss future work in Section 8. Code listings for the examples can be found in Appendices A, B and C.

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 *first*(σ) returns the first element of σ and *last*(σ) 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 a (possibly empty) 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*(γ) 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*(γ) $= s$, *last*(γ) $= s'$, and *trace*(γ) $= \sigma$. 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, 19], 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*.

For our basic notion of symmetry on an FSM $\mathcal{A}$, we use an equivalence relation on $(\Sigma_{\mathcal{A}})^*$, such that $\mathcal{A}$ is *closed* under the relation, i.e., if a sequence of actions is related 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 equivalence relation on the traces of the specification. In order to be able to do this, we define a symmetry to be the pair consisting of the equivalence relation and a representative-choosing function. We 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 $(\sigma_1\, \sigma_3)^r = \sigma_1\, \sigma_3$.

□

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.

Proposition 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 minimal 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 under $\simeq$, $F \subseteq E$ and if E is non-empty, F is non-empty. This justifies the function $\mathit{choose}(F)$ which nondeterministically chooses an element from F. In the algorithm, the global variable $\mathcal{K}$ is the growing state space which is returned at the end of the algorithm, and updated during the execution of the procedure Build_It. The local variable F for Build_It is not significant for the execution of the algorithm but is useful for proving correctness.

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$(s', \sigma', \mathit{Seen}')$ within Build_It$(s, \sigma, \mathit{Seen})$, $\mathit{Seen}' = \mathit{Seen} \cup \{s'\}$ with $s' \notin \mathit{Seen}$. So there can be only 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*$(s^0_{\mathcal{A}}, \epsilon, \emptyset)$*, automaton* $\mathcal{K}$ *is a subautomaton of* $\mathcal{A}$*, and* $\mathcal{K}$ *grows monotonically.*

```
 1  function Kernel(𝒜, S): FSM;
 2  var 𝒦: FSM;
 3
 4     procedure Build_It(s, σ, Seen);
 5     var a, b, s′, E, F;
 6     begin
 7        if s ∉ Seen then
 8           E := enabled(s, 𝒜);
 9           F := ∅;
10           while E ≠ ∅ do
11              a := choose(repr(σ, E));
12              s′ := eff(s, a, 𝒜);
13              S_𝒦 := S_𝒦 ∪ {s′};
14              Σ_𝒦 := Σ_𝒦 ∪ {a};
15              E_𝒦 := E_𝒦 ∪ {(s, a, s′)};
16              Build_It(s′, σ a, Seen ∪ {s});
17              F := F ∪ {a};
18              for each b ∈ E . σ a ≃ σ b do
19                 E := E \ {b};
20              od;
21           od;
22        fi;
23     end;
24
25  begin
26     s⁰_𝒦 := s⁰_𝒜;
27     S_𝒦 := {s⁰_𝒜};
28     Σ_𝒦 := ∅;
29     E_𝒦 := ∅;
30     Build_It(s⁰_𝒜, ϵ, ∅);
31     return 𝒦;
32  end.
```

Figure 1: The algorithm Kernel

Proof. Obvious from the algorithm. ⊠

The following lemma concerns the value of the variable $\mathcal{K}$ at the moment that the call to Build_It is made.

Lemma 4.3. *When Kernel($\mathcal{A}$,S) during its execution calls Build_It(s,σ,Seen), then at that moment* $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ *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* $= \emptyset$. In the initialization of $\mathcal{K}$, $s^0_{\mathcal{K}}$ has been defined equal to $s^0_{\mathcal{A}}$. As $s^0_{\mathcal{K}} \xrightarrow{\epsilon}_{\mathcal{K}} s$ and $(\epsilon)^r = \epsilon$, we are done.

- $n = m + 1$.

  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*(σ', 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)=$\mathcal{K}$ and during its execution has called Build_It(s,σ,Seen), then* $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ *and* $\sigma^r = \sigma$.

Proof. Follows immediately from Lemmas 4.2 and 4.3. ⊠

Lemma 4.5. *If Kernel($\mathcal{A}$,S)=$\mathcal{K}$ and* $s \xrightarrow{a}_{\mathcal{K}} t$ *then there is a* σ *such that* $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ *and* $(\sigma a)^r = \sigma a$.

Proof. If $s \xrightarrow{a}_{\mathcal{K}} t$, then we know by Lemma 4.2 and by the fact that execution starts with $\mathcal{K}$ empty, that the transition (s, a, t) has been added to $\mathcal{K}$ by execution of line 15 of algorithm Kernel. This happens during the execution of the call Build_It(s,σ,*Seen*) for some σ and some *Seen*, so by Lemma 4.4 we may conclude that upon completion, $s^0_{\mathcal{K}} \xrightarrow{\sigma}_{\mathcal{K}} s$ and $\sigma^r = \sigma$. By the definition of a during execution of the call Build_It(s,σ,*Seen*) at line 11, we see that $(\sigma a)^r = \sigma a$. ⊠

Lemma 4.6. *When Kernel($\mathcal{A}$,S) calls Build_It(s,σ,Seen), then during the execution of Build_It the following holds:*

1. *at termination of the while loop, the following property holds:*

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

2. *at termination of the while loop, the following property holds:*

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

Proof.

1. When the while loop is started, F is empty. The only statement that adds a to F is at line 17, which is executed after lines 12 through 16 have been executed, hence the edge (s, a, s') has been added to $E_{\mathcal{K}}$ and Build_It$(s', \sigma a, \mathit{Seen} \cup \{s\})$ has been called.

2. At the start of the while loop, $E = \mathit{enabled}(s, \mathcal{A},)$, so $b \in E$ at the start of the while loop. At termination of the while loop, E is empty. Actions are never added to E, only removed from E at line 19. So during the execution of the while loop, certainly $E \subseteq \mathit{enabled}(s, \mathcal{A},)$. We observe that during the execution of the while loop for each a, if $a \in \mathit{repr}(\sigma, E)$, then $a \in \mathit{repr}(\sigma, \mathit{enabled}(s, \mathcal{A},))$. At the moment that b is removed from E (at line 19), the condition that $\sigma\, a \simeq \sigma\, b$ holds. Since a was defined at line 11 and has not changed since, and by our observation, it holds that $a \in \mathit{repr}(\sigma, \mathit{enabled}(s, \mathcal{A},))$. So there is a $c \in \mathit{enabled}(s, \mathcal{A},)$ such that $\sigma\, a = (\sigma\, c)^r$. Since $\sigma\, a \simeq \sigma\, b$, and by definition of representative, $\sigma\, b \simeq \sigma\, c$. By uniqueness of representative, $\sigma\, a = (\sigma\, b)^r$, and we are done.

⊠

Lemma 4.7. *If Kernel($\mathcal{A}$,S) during execution 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 $0 \leq i \leq n$, $\mathit{Seen}_i = \bigcup_{j \in \{0,1,\ldots,i-1\}} \{s_j\}$*
3. *$s = s_n$ and $\mathit{Seen} = \mathit{Seen}_n$*

Proof. By induction on the length n of σ.

- $n = 0$. Then $\sigma = \epsilon$, and the result follows immediately.
- $n = m + 1$.

  Suppose $\sigma = a_1 a_2 \ldots a_{m+1}$ and Kernel($\mathcal{A}$,S) calls Build_It(s,σ,Seen) with $s_0 \xrightarrow{a_1}_{\mathcal{A}} s_1 \xrightarrow{a_2}_{\mathcal{A}} s_2 \ldots \xrightarrow{a_{m+1}}_{\mathcal{A}} s_{m+1}$ and $s_0 = s^0_{\mathcal{A}}$. Let $\sigma' = a_1 a_2 \ldots a_m$. 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 $\mathit{Seen} = \mathit{Seen}' \cup \{s'\}$. By the induction hypothesis, we know that $\mathit{Seen}' = \mathit{Seen}_m$, 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 4.8. *If Kernel($\mathcal{A}$,S) during execution 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. From Lemmas 4.2 and 4.4 it follows that $\sigma \in traces(\mathcal{A})$. The lemma then follows from Lemma 4.7. ⊠

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 4.9. *Let* $\mathcal{K} = \text{Kernel}(\mathcal{A},S)$.

1. *If $\mathcal{K}'$ is a kernel of $\mathcal{A}$ w.r.t. S, then $\mathcal{K}$ is a subautomaton of $\mathcal{K}'$.*
2. *If $\sigma \in traces(\mathcal{A})$, then $\sigma^r \in traces(\mathcal{K})$.*

Proof. First we prove Item 1. From the algorithm Kernel it is obvious that for each state s in $\mathcal{K}$, either s is the initial state, or there is a transition leading to s. Since $\mathcal{K}$ and $\mathcal{K}'$ are subautomata of $\mathcal{A}$, their initial states are equal. Since $\mathcal{K}$ and $\mathcal{K}'$ are deterministic and representative traces are unique, and since by Lemma 4.5 each transition in $\mathcal{K}$ is part of a representative trace, we see that each transition in $\mathcal{K}$ must be present in $\mathcal{K}'$. Combining all these observations, we see that each state in $\mathcal{K}$ is present in $\mathcal{K}'$. We conclude that $\mathcal{K}$ is a subautomaton of $\mathcal{K}'$.

As to Item 2. 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).

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

- $n = 0$.

  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): $0 \leq |\rho| \leq m \Rightarrow Inv(\rho^r)$

  Suppose $\sigma = \sigma' \, b$, $\tau = \tau' \, c$, and $|\sigma| = |\tau| = m+1$. Since $()^r$ is prefixed closed, $(\sigma')^r = \tau'$. Since $\tau \in traces(\mathcal{A})$, $\tau' \in traces(\mathcal{A})$. 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$(t, \tau' \, c, Seen)$. By Lemma 4.4 we then know that $\tau' \, c \in traces(\mathcal{K})$, which proves $Inv(\tau)$.

    Assume $s^0_{\mathcal{A}} \xrightarrow{\tau'}_{\mathcal{A}} t'$. Since $(\sigma')^r = \tau'$, $Inv(\tau')$ holds by IH, so $\tau' \in traces(\mathcal{K})$. There is no looping trace in τ', and by $Inv(\tau')$, for some set $Seen'$, Kernel$(\mathcal{A}, S)$ calls Build_It$(t', \tau', Seen')$. We now inspect the execution of procedure Build_It for this call. By Lemma 4.8, we know that $t' \notin Seen'$. By Lemma 4.6 we know that upon completion of the while loop Build_It$(t''$,$\tau' \, c'$,$Seen' \cup \{t'\})$ has been 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$(t$,$\tau' \, c$,$Seen' \cup \{t'\})$ has been 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$(t', \tau_1 \, \tau_2 \, a, Seen)$, and that $\tau \in traces(\mathcal{K})$. 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 $Inv(\tau_1 \, \tau_2 \, a)$ and $Inv(\tau_1 \, \tau_3 \, c)$ hold. This means that $\tau_1 \, \tau_2 \, a \in traces(\mathcal{K})$, $\tau_1 \, \tau_3 \, c \in traces(\mathcal{K})$, and since there is no looping trace in $\tau_1 \, \tau_2$, that, for some set *Seen*, Kernel$(\mathcal{A}, S)$ calls Build_It$(t', \tau_1 \, \tau_2 \, a, Seen)$. 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 $\tau_1 \, \tau_2 \, a \tau_3 \, c \in traces(\mathcal{K})$.

⊠

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 [28, 7, 5, 1]. We will show how the classical W-method [28, 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 [24] 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

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

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 $s \overset{\xi}{\Longrightarrow}_\mathcal{A} s'$ 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 [23]. 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 $\mathcal{A} \leftrightarrows \mathcal{B}$, 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 $s_1 \leftrightarrows_\mathcal{A} s_2$, if there exists a bisimulation R on $\mathcal{A}$ (and $\mathcal{A}$) such that $R(s_1, s_2)$. The relation $\leftrightarrows_\mathcal{A}$ is an equivalence relation on $S_\mathcal{A}$; a *bisimulation class* of $\mathcal{A}$ is an equivalence class of $S_\mathcal{A}$ under $\leftrightarrows_\mathcal{A}$. □

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 $s \not\simeq_\mathcal{A} s'$, 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 how long the suffix of a distinguishing sequence can be which starts with the first step 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:

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

□

In the following definition, the parameter n is the upper bound to the length of that part of the distinguishing sequence which steps outside the image of the kernel.

Definition 5.5. A subautomaton $\mathcal{B}$ of a Mealy machine $\mathcal{A}$ is n*-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, and for every pair of

states s, s' of $\mathcal{B}$ such that $s \not\approx_{\mathcal{A}} s'$, there exist input sequences ξ_1, ξ_2 of $\mathcal{A}$ of length at most m, n, 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. *The number of bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by m_1.*
2. *$\mathcal{C}$ is m_2-self-contained in $\mathcal{B}$.*
3. *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 $s \not\approx_{\mathcal{B}} s'$, $I^{m_1-n}\, I^{m_2}\, W$ 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 2, the number of bisimulation classes Q of $\mathcal{B}$ such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by 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 $k + n + 1$ 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 $s_1 \not\approx_{\mathcal{B}} s_2$ but $I^k\, I^{m_2}\, W$ does not distinguish s_1 from s_2. Since $\mathcal{C}$ is 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 $s \stackrel{(i/o)}{\longrightarrow} s'$ of $\mathcal{K}$, P contains input sequences ξ and $\xi\, i$ such that $s^0_{\mathcal{K}} \stackrel{\xi}{\Longrightarrow}_{\mathcal{K}} s$. □

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. *The number of bisimulation classes Q of Impl such that $Q \cap S_{\mathcal{C}} \neq \emptyset$ is bounded by m_1.*
3. *$\mathcal{C}$ is m_2-self-contained in Impl.*
4. *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 $s^0_{Impl} \xRightarrow{\xi}_{Impl} u'$ 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 $\tau \in traces(Impl)$. As to item 2). Consider an input sequence $\xi \in P$ such that $s^0_{\mathcal{K}} \xRightarrow{\xi}_{\mathcal{K}} s^0_{\mathcal{K}}$ and assume $s^0_{Impl} \xRightarrow{\xi}_{Impl} u'$. We have to show that $s^0_{Impl} \leftrightarrows_{\mathcal{I}} u'$.

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

- $n > 0$. Write $\sigma = \sigma'\,(i/o)$. By induction hypothesis $(\sigma')^r = \tau' \in traces(\mathcal{K}) \cap traces(Impl)$. Say that $s^0_{\mathcal{K}} \xrightarrow{\tau'}_{\mathcal{K}} r'$. 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, $r' \xrightarrow{i'/o'}_{\mathcal{K}} r$. Since $r' \in S_{\mathcal{K}}$, there exist input sequences ξ', $\xi'\,i' \in P$ such that $s^0_{\mathcal{K}} \xRightarrow{\xi'}_{\mathcal{K}} r'$.

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

  Finally, we have to prove, for all $\xi \in P$ such that $s^0_{\mathcal{K}} \xRightarrow{\xi}_{\mathcal{K}} r$: for the unique state w such that $s^0_{Impl} \xRightarrow{\xi}_{Impl} w$, 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 $\rho \in Z$, $outcome_{Impl}(\rho, v) = outcome_{Impl}(\rho, w)$. Hence, by Lemma 5.6, $v' \leftrightarrows_{\mathcal{I}} w$.

⊠

⊠

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 [21] 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 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.

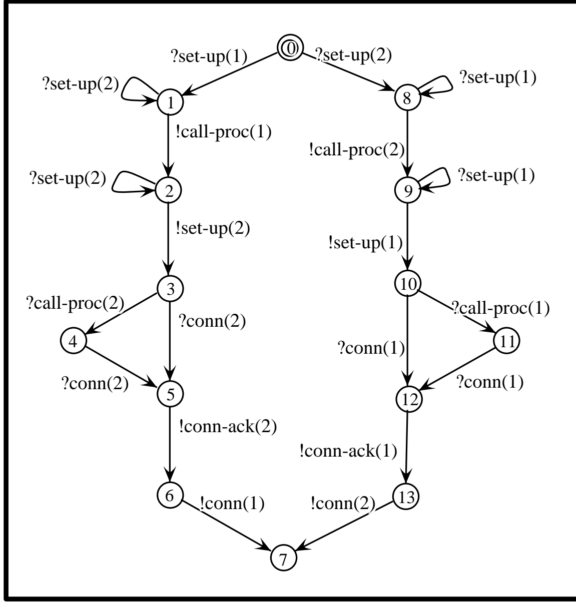

Figure 2: The ATM call setup protocol

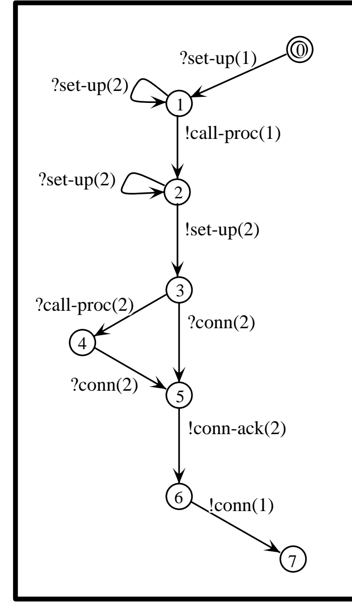

Figure 3: A template

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 $\langle f_1, \ldots, f_n \rangle$ of (partial) functions $f_1, \ldots, f_n : \Pi \to E_{\mathcal{T}}$, we denote with *exec*$(\langle f_1, \ldots, f_n \rangle, \pi)$ 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 explain 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 intuitive 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 $\mathcal{T}$, both the intuitive requirement and the formal rephrasing 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 $\pi \in \Pi$,

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

$\square$

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 $cov_1 = \langle f_1, \dots, f_n \rangle$ and $cov_2 = \langle g_1, \dots, g_n \rangle$, 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:

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

It is easy to check that $\simeq_\mathcal{P}$ is an equivalence relation. As in Section 3, we will write $\simeq$ 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 index (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 indices 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 $\langle f_1, \dots, f_n\rangle$, then $\mathcal{P}$ covers σ by $\langle f_1, \dots, f_{n-1}\rangle$.*

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 $\langle f_1, \dots, f_n\rangle$ and $\langle g_1, \dots, g_n\rangle$, respectively. By Lemma 6.5, these coverings induce the coverings $\langle f_1, \dots, f_{n-1}\rangle$ and $\langle g_1, \dots, g_{n-1}\rangle$ of σ and τ, respectively, which are clearly variants of each other. ⊠

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 $\sigma\, a \in$ *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 $\{\tau | \sigma \simeq \tau\}$ 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 $(\tau\, b)^r = \sigma\, a \in$ 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 $\simeq$, $\tau\, b \in$ *traces*$(\mathcal{A})$. By persistence of $\simeq$, $\rho \simeq \tau$ implies that there exists an action c such that $\rho\, c \simeq \tau\, b$. Since $\simeq$ is prefix closed on $\mathcal{A}$ (Corollary 6.7) and $\sigma\, a \simeq \tau\, b$, it follows that $\sigma \simeq \tau$. 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 $1 \leq i \leq n$, $[f_i(\pi) \mid \pi \in \Pi] = [g_i(\pi) \mid \pi \in \Pi]$. Now 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*

$\sigma_1\, \sigma_3 \simeq \sigma_1\, \tau$ *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 $\langle f_1, \ldots, f_n, g_1, \ldots, g_m, h_1, \ldots, h_k \rangle$ cover $\sigma_1\, \sigma_2\, \sigma_3$.
By Lemma 6.5, $\langle f_1, \ldots, f_n, g_1, \ldots, g_m \rangle$ covers $\sigma_1\, \sigma_2$ and $\langle f_1, \ldots, f_n \rangle$ covers σ_1. Since $\simeq$ is loop preserving on $\mathcal{A}$, we know that for every $\pi \in \Pi$

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

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

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

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

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

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

Now 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 "⇒", 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 $\rho \neq \rho'$. ⊠

7 Examples

In this section we report on some initial experiments in the application of symmetry to the testing of three examples. Section 7.1 presents the example of a chatbox, Section 7.2 presents the example of a cyclic train, and Section 7.3 presents the example of a ring leader election protocol. The code listings for these examples can be found in Appendices A, B and C.

Part of the test generation trajectory was implemented: we used the tool environment OPEN/CÆSAR[14] for prototyping the algorithm Kernel from Section 3. Section 7.4 relates some prototyping experiences.

We work with a pattern based symmetry (Section 6) and apply the test derivation method from Section 5.

7.1 A chatbox service

In this section we report on some experiments in the application of symmetry to the testing of a *chatbox*.

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 [26]. 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. A Lotos [20] model and a μCRL [18] model can be found in Appendix A.

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

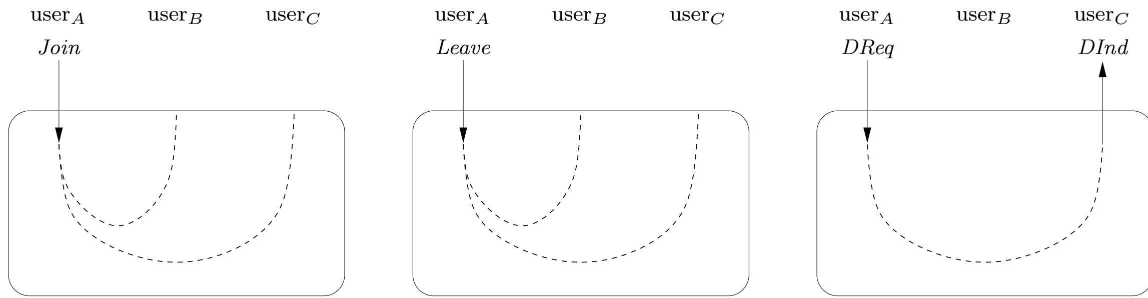

Figure 4: The chatbox protocol service

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.

We start the test generation by computing a kernel for these specifications. In Table 1, the results of applying our prototype implementation of the algorithm Kernel can be found. 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 4.9 states. This technique of dealing with traces is reminiscent of partial ordering techniques [17].

From Table 1 we see that the kernel size is relatively smaller when considering chatboxes without joining and leaving 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.

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

model				kernel	
	states	*trans*	*minimal?*	*states*	*trans*
3 users	512	12288	yes	213	3722
4 users	65536	2621440	yes	16385	263000
no joining/leaving					
3 users	64	1152	yes	10	84
4 users	4096	131072	yes	112	1296

Table 1: Kernel statistics for the chatbox

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.

7.2 A cyclic train

In this section we report on some initial experiments in the application of symmetry to the testing of a *cyclic train*. This example was inspired by the elevator specification used by Frits Vaandrager in the course 'Declarative Specifications and Systems' at the University of Nijmegen in spring 1998. Since the symmetry in an elevator obviously must be sought in the floor numbers, and at the lowest (highest) floor it is not possible to go any lower (higher), we modified the example a little to make the elevator cyclic: from the lowest floor, the elevator can reach the highest floor by moving one floor down, and vice versa. To make the example a bit more intuitive, we rename the cyclic elevator to a cyclic train, and floors to stations.

See Figure 5. The train runs on a cyclic track, from station to station. It can change direction if needed, and can be sent to a destination if a button inside the train is pressed, and called to a station if a button in the station is pressed. We consider as running example a cyclic train running between four stations. In Figure 5, the train is moving from station 3 to station 2.

The symmetry inherent in the protocol is immediate: the behavior of the train requested to go to a station or moving from one station to another is symmetric to the same behavior when other stations are involved. As an example, the trace in which the train starts at station 1, is called to station 2 and then sent to station 0 is symmetric to the trace in which the train starts at station 3, is called to station 0 and then sent to station 2. Thus the template $\mathcal{T}$ used for the symmetry basically consists of the train arriving at the current station (from left or

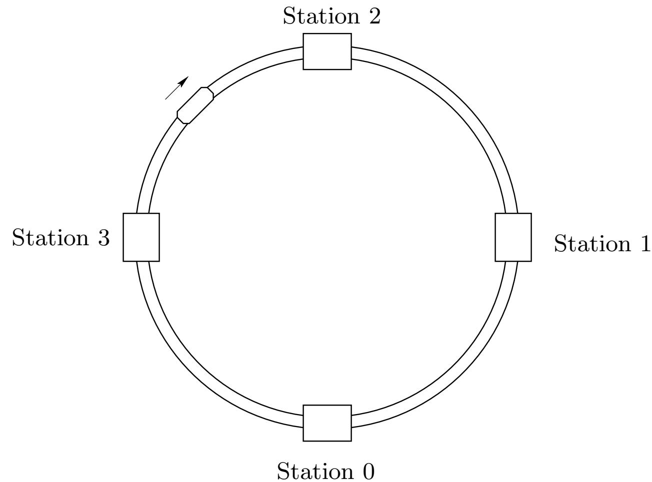

Figure 5: A cyclic train with 4 stations

right), opening its doors, closing its doors and moving away again, while the transformations π in the set Π shuffle the identity of the station.

We have applied the machinery to trains with up to 8 stations. We also considered a version of the train in the Mealy style, where each transition consists of an input and an output action. Here we have assumed that in each state, one can give an input by pressing a button, and that the output for such an input depends on the state of the train. If no input is given, the train may still want to move from station to station. This is modeled with the input action `WAIT`.

In Table 2, the results of applying our prototype implementation of the algorithm Kernel can be found. We work with state spaces generated from μCRL code (See Appendix B) which have not been minimized. The kernel is significantly smaller 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 orderings in the table refer to the ordering of symmetric request actions for the train to go to a certain station. The numbers indicate the indentity of the station to which the train should go. For the Mealy style models, it turns out that the orderings listed in the table work better than others. For the models with 4 and 5 stations these are in fact the best orderings. For the other models, only some orderings were tested. Naturally, if the state space from which the kernel is constructed and the kernel itself get larger, the process takes longer.

7.3 A ring leader election protocol

In this section we report on some initial experiments in the application of symmetry to the testing of a *ring leader election protocol*. This example was taken from [15].

The protocol is used in the setting of a number of stations, connected with a unidirectional ring on which messages can be passed, and one central resource to which no more than one station should have access at the same time. Exclusive access is guaranteed by a token which is passed around by the stations. Since the links that connect the stations are unreliable, and

model				kernel		
	states	*trans*	*minimal?*	*states*	*trans*	*representant ordering*
7 stations	286725	4300874	no	12685	79594	`6<5<4<3<2<1<0`
8 stations	1310725	22282324	no	30945	195404	`7<6<5<4<3<2<1<0`
Mealy style						
4 stations	2808	24312	no	1548		`0<2<1<3`
5 stations	13950	148650	no	5193		`0<2<4<3<1`
6 stations	72036	912276	no	16959		`0<2<5<3<4<1`
7 stations	336042	4927734	no	49941	79594	`0<3<6<4<5<2<1`
8 stations	1563696	26060592	no	146394	887208	`0<2<7<3<6<4<5<1`

Table 2: Kernel statistics for the cyclic train

model				kernel	
	states	*trans*	*minimal?*	*states*	*trans*
3 stations	37764	292254	yes	17501	100941
3 stations	224152	1710534	no	40927	281544

Table 3: Kernel statistics for the ring leader election protocol

the stations themselves may crash at any moment, the token can be lost and a new token should be generated. The leader election protocol ensures that a leader is chosen who may use the resource and generate a new token. At any moment a new election may be started.

The protocol is proved correct and explained in more detail in [15]. Our μCRL model (See Appendix C) is translated from the Lotos code at pages 23–24 in [15].

Again, the symmetry inherent in the protocol is immediate: the behavior for one station is symmetric to the same behavior for another station. The pattern for our symmetry is equal to the behavior of a station operating in isolation. Therefore we have generated the state space of the pattern from the μCRL description. The resulting state space can be found in Appendix C.

We have applied the machinery to rings with 3 stations. In Table 3, the results of applying our prototype implementation of the algorithm Kernel can be found. We work with state spaces generated from μCRL code (See Appendix C) which have not been minimized. The kernel is significantly smaller 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. This ordering depends on the parameters of the action. For the minimized state space, an ordering descending in the values of the parameters worked better, whereas for the state space which was not minimized, an ordering ascending in the values of the parameters worked slightly better. We have not tried all possible orderings.

7.4 Implementing the algorithm Kernel

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 [25], 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):178–188, 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)*, volume 697 of *Lecture Notes in Computer Science*. 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)*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34. 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 [25], pages 68–84. For more information on the tool set, see `http://www.inrialpes.fr/vasy/pub/cadp.html`.

[15] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1–2):171–197, July 1997. Full version available as INRIA Research Report RR-2986 from `http://www.inrialpes.fr/vasy/Publications/`.

[16] 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*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1995.

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

[18] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing. Springer-Verlag, 1995.

[19] V. Gyuris and A.P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In O. Grumberg, editor, *Proceedings* 9th *International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1997.

[20] ISO. Information processing systems – Open Systems Interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807, 1989.

[21] 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.

[22] 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.

[23] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[24] 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.

[25] B. Steffen, editor. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[26] 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.

[27] J. Tretmans. A theory for the derivation of tests. In *Formal Description Techniques (FORTE II '89)*. North-Holland, 1989.

[28] M.P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, 1973.

A Chatbox code listings

Lotos model for 3 users

```
1    specification chatbox[JOIN, LEAVE, DREQ] : noexit
2
3    library
4       X_BOOLEAN, X_NATURAL
5    endlib
6
7    type CHATBOX_TYPES is CHATBOX_RENAME_TYPES, BOOLEAN, NATURALNUMBER
8       sorts MES_TYPE,
9             VECTOR_TYPE (*! implementedby C_VECTOR_TYPE comparedby COMPAREV \
10                            enumeratedby ENUMV printedby PRINTV external *),
11            MATRIX_TYPE (*! implementedby C_MATRIX_TYPE comparedby COMPAREM \
12                            enumeratedby ENUMM printedby PRINTM external *),
13            OUTPUT
14      opns
15           NO_OUTPUT (*! constructor *),
16           OK (*! constructor *),
17           DIND (*! constructor *): -> OUTPUT
18           mes (*! constructor *),
19           ack (*! constructor *) : -> MES_TYPE
20           _eq_ : MES_TYPE, MES_TYPE -> BOOL
21           vector (*! implementedby c_vector constructor external *)
22             : -> VECTOR_TYPE
23           setv (*! implementedby c_setv external *)
24             : USR_TYPE,BOOL,VECTOR_TYPE -> VECTOR_TYPE
25           getv (*! implementedby c_getv external *)
26             : USR_TYPE,VECTOR_TYPE -> BOOL
27           matrix (*! implementedby c_matrix constructor external *)
28             : -> MATRIX_TYPE
29           setm (*! implementedby c_setm external *)
```

```
30             : USR_TYPE,USR_TYPE,BOOL,MATRIX_TYPE -> MATRIX_TYPE
31           getm (*! implementedby c_getm external *)
32             : USR_TYPE,USR_TYPE,MATRIX_TYPE -> BOOL
33           updatem (*! implementedby c_updatem external *)
34             : USR_TYPE,BOOL,VECTOR_TYPE,MATRIX_TYPE -> MATRIX_TYPE
35      eqns
36         ofsort BOOL
37           ack eq mes = false;
38           mes eq ack = false;
39           ack eq ack = true;
40           mes eq mes = true;
41   endtype
42
43   type CHATBOX_RENAME_TYPES is NATURALNUMBER renamedby
44      sortnames USR_TYPE for NAT
45   endtype
46
47   behaviour
48
49   CHAT[JOIN, LEAVE, DREQ]
50       (vector,matrix,matrix)
51
52   where
53
54   process CHAT
55         [JOIN, LEAVE, DREQ]
56         (PC:VECTOR_TYPE, SENT_TO,TO_ACK:MATRIX_TYPE)
57    : noexit :=
58
59     (choice u:USR_TYPE []
60      ( [getv(u,PC)] ->
61           JOIN ! u ! NO_OUTPUT ;
62              CHAT[JOIN, LEAVE, DREQ](PC,SENT_TO,TO_ACK)
63       []
64        [not(getv(u,PC))] ->
65           JOIN ! u ! OK ;
66              CHAT[JOIN, LEAVE, DREQ]
67                (setv(u,true,PC),SENT_TO,TO_ACK)
68       []
69        [not(getv(u,PC))] ->
70           LEAVE ! u ! NO_OUTPUT ;
71              CHAT[JOIN, LEAVE, DREQ](PC,SENT_TO,TO_ACK)
72       []
73        [getv(u,PC)] ->
74           LEAVE ! u ! OK ;
75              CHAT[JOIN, LEAVE, DREQ]
76                (setv(u,false,PC),SENT_TO,TO_ACK)
77       ))
78    []
79      (choice u1:USR_TYPE, u2:USR_TYPE []
80       (
```

```
81    DREQ ! u1 ! u2 ! mes ! NO_OUTPUT
82    [((u1 eq u2) or not(getv(u1,PC) and getv(u2,PC))) or getm(u1,u2,SENT_TO)];
83       CHAT[JOIN, LEAVE, DREQ](PC,SENT_TO,TO_ACK)
84       []
85    DREQ ! u1 ! u2 ! ack ! NO_OUTPUT
86    [((u1 eq u2) or not(getv(u1,PC) and getv(u2,PC))) or not(getm(u1,u2,TO_ACK))];
87       CHAT[JOIN, LEAVE, DREQ](PC,SENT_TO,TO_ACK)
88       []
89    DREQ ! u1 ! u2 ! mes ! DIND
90    [(not(u1 eq u2) and getv(u1,PC) and getv(u2,PC)) and not(getm(u1,u2,SENT_TO))];
91       CHAT[JOIN, LEAVE, DREQ]
92         (PC,setm(u1,u2,true,SENT_TO),setm(u2,u1,true,TO_ACK))
93       []
94    DREQ ! u1 ! u2 ! ack ! DIND
95    [(not(u1 eq u2) and getv(u1,PC) and getv(u2,PC)) and getm(u1,u2,TO_ACK)];
96       CHAT[JOIN, LEAVE, DREQ]
97         (PC,setm(u2,u1,false,SENT_TO),setm(u1,u2,false,TO_ACK))
98     ))
99
100   endproc
101
102   endspec (* chatbox *)
```

μCRL model for 4 users

```
1     sort    Bool
2     func    T,F : -> Bool
3     map     not: Bool -> Bool
4             and: Bool # Bool -> Bool
5     var     b : Bool
6     rew     not(T) = F
7             not(F) = T
8             and(F,b) = F
9             and(b,F) = F
10            and(T,b) = b
11            and(b,T) = b
12
13    sort    Nat
14    func    0,1,2,3: -> Nat
15
16    sort    Mes
17    func    MES, ACK: -> Mes
18
19    sort    Output
20    func    NO_OUTPUT, OK, DIND : -> Output
21
22    act     JOIN, LEAVE: Nat # Output
23            DREQ: Nat # Nat # Mes # Output
24
25    proc    Chatbox(pres0: Bool, pres1: Bool, pres2: Bool, pres3: Bool,
26                    sentto01: Bool, sentto02: Bool, sentto03: Bool,
```

```
27                    sentto10: Bool, sentto12: Bool, sentto13: Bool,
28                    sentto20: Bool, sentto21: Bool, sentto23: Bool,
29                    sentto30: Bool, sentto31: Bool, sentto32: Bool)
30            =
31              JOIN(0,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
32                                  sentto01, sentto02, sentto03,
33                                  sentto10, sentto12, sentto13,
34                                  sentto20, sentto21, sentto23,
35                                  sentto30, sentto31, sentto32)
36                 <| pres0 |>
37              JOIN(0,OK) . Chatbox(not(pres0), pres1, pres2, pres3,
38                           sentto01, sentto02, sentto03,
39                           sentto10, sentto12, sentto13,
40                           sentto20, sentto21, sentto23,
41                           sentto30, sentto31, sentto32)
42             +
43              JOIN(1,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
44                                  sentto01, sentto02, sentto03,
45                                  sentto10, sentto12, sentto13,
46                                  sentto20, sentto21, sentto23,
47                                  sentto30, sentto31, sentto32)
48                 <| pres1 |>
49              JOIN(1,OK) . Chatbox(pres0, not(pres1), pres2, pres3,
50                           sentto01, sentto02, sentto03,
51                           sentto10, sentto12, sentto13,
52                           sentto20, sentto21, sentto23,
53                           sentto30, sentto31, sentto32)
54             +
55              JOIN(2,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
56                                  sentto01, sentto02, sentto03,
57                                  sentto10, sentto12, sentto13,
58                                  sentto20, sentto21, sentto23,
59                                  sentto30, sentto31, sentto32)
60                 <| pres2 |>
61              JOIN(2,OK) . Chatbox(pres0, pres1, not(pres2), pres3,
62                           sentto01, sentto02, sentto03,
63                           sentto10, sentto12, sentto13,
64                           sentto20, sentto21, sentto23,
65                           sentto30, sentto31, sentto32)
66             +
67              JOIN(3,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
68                                  sentto01, sentto02, sentto03,
69                                  sentto10, sentto12, sentto13,
70                                  sentto20, sentto21, sentto23,
71                                  sentto30, sentto31, sentto32)
72                 <| pres3 |>
73              JOIN(3,OK) . Chatbox(pres0, pres1, pres2, not(pres3),
74                           sentto01, sentto02, sentto03,
75                           sentto10, sentto12, sentto13,
76                           sentto20, sentto21, sentto23,
77                           sentto30, sentto31, sentto32)
```

```
78            +
79              LEAVE(0,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
80                                           sentto01, sentto02, sentto03,
81                                           sentto10, sentto12, sentto13,
82                                           sentto20, sentto21, sentto23,
83                                           sentto30, sentto31, sentto32)
84                 <| not(pres0) |>
85              LEAVE(0,OK) . Chatbox(not(pres0), pres1, pres2, pres3,
86                                    sentto01, sentto02, sentto03,
87                                    sentto10, sentto12, sentto13,
88                                    sentto20, sentto21, sentto23,
89                                    sentto30, sentto31, sentto32)
90            +
91              LEAVE(1,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
92                                           sentto01, sentto02, sentto03,
93                                           sentto10, sentto12, sentto13,
94                                           sentto20, sentto21, sentto23,
95                                           sentto30, sentto31, sentto32)
96                 <| not(pres1) |>
97              LEAVE(1,OK) . Chatbox(pres0, not(pres1), pres2, pres3,
98                                    sentto01, sentto02, sentto03,
99                                    sentto10, sentto12, sentto13,
100                                   sentto20, sentto21, sentto23,
101                                   sentto30, sentto31, sentto32)
102           +
103             LEAVE(2,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
104                                          sentto01, sentto02, sentto03,
105                                          sentto10, sentto12, sentto13,
106                                          sentto20, sentto21, sentto23,
107                                          sentto30, sentto31, sentto32)
108                <| not(pres2) |>
109             LEAVE(2,OK) . Chatbox(pres0, pres1, not(pres2), pres3,
110                                   sentto01, sentto02, sentto03,
111                                   sentto10, sentto12, sentto13,
112                                   sentto20, sentto21, sentto23,
113                                   sentto30, sentto31, sentto32)
114           +
115             LEAVE(3,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
116                                          sentto01, sentto02, sentto03,
117                                          sentto10, sentto12, sentto13,
118                                          sentto20, sentto21, sentto23,
119                                          sentto30, sentto31, sentto32)
120                <| not(pres3) |>
121             LEAVE(3,OK) . Chatbox(pres0, pres1, pres2, not(pres3),
122                                   sentto01, sentto02, sentto03,
123                                   sentto10, sentto12, sentto13,
124                                   sentto20, sentto21, sentto23,
125                                   sentto30, sentto31, sentto32)
126           +
127             DREQ(0,0,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
128                                               sentto01, sentto02, sentto03,
```

```
129                                         sentto10, sentto12, sentto13,
130                                         sentto20, sentto21, sentto23,
131                                         sentto30, sentto31, sentto32)
132              <| T |>
133            delta
134          +
135            DREQ(1,1,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
136                                         sentto01, sentto02, sentto03,
137                                         sentto10, sentto12, sentto13,
138                                         sentto20, sentto21, sentto23,
139                                         sentto30, sentto31, sentto32)
140              <| T |>
141            delta
142          +
143            DREQ(2,2,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
144                                         sentto01, sentto02, sentto03,
145                                         sentto10, sentto12, sentto13,
146                                         sentto20, sentto21, sentto23,
147                                         sentto30, sentto31, sentto32)
148              <| T |>
149            delta
150          +
151            DREQ(3,3,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
152                                         sentto01, sentto02, sentto03,
153                                         sentto10, sentto12, sentto13,
154                                         sentto20, sentto21, sentto23,
155                                         sentto30, sentto31, sentto32)
156              <| T |>
157            delta
158          +
159            DREQ(0,1,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
160                                  not(sentto01), sentto02, sentto03,
161                                  sentto10, sentto12, sentto13,
162                                  sentto20, sentto21, sentto23,
163                                  sentto30, sentto31, sentto32)
164              <| and(pres0,and(pres1,not(sentto01))) |>
165            DREQ(0,1,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
166                                         sentto01, sentto02, sentto03,
167                                         sentto10, sentto12, sentto13,
168                                         sentto20, sentto21, sentto23,
169                                         sentto30, sentto31, sentto32)
170          +
171            DREQ(0,2,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
172                                  sentto01, not(sentto02),sentto03,
173                                  sentto10, sentto12, sentto13,
174                                  sentto20, sentto21, sentto23,
175                                  sentto30, sentto31, sentto32)
176              <| and(pres0,and(pres2,not(sentto02))) |>
177            DREQ(0,2,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
178                                         sentto01, sentto02, sentto03,
179                                         sentto10, sentto12, sentto13,
```

```
180                                         sentto20, sentto21, sentto23,
181                                         sentto30, sentto31, sentto32)
182          +
183            DREQ(0,3,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
184                                 sentto01, sentto02, not(sentto03),
185                                 sentto10, sentto12, sentto13,
186                                 sentto20, sentto21, sentto23,
187                                 sentto30, sentto31, sentto32)
188               <| and(pres0,and(pres3,not(sentto03))) |>
189            DREQ(0,3,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
190                                         sentto01, sentto02, sentto03,
191                                         sentto10, sentto12, sentto13,
192                                         sentto20, sentto21, sentto23,
193                                         sentto30, sentto31, sentto32)
194          +
195            DREQ(1,0,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
196                                 sentto01, sentto02, sentto03,
197                                 not(sentto10), sentto12, sentto13,
198                                 sentto20, sentto21, sentto23,
199                                 sentto30, sentto31, sentto32)
200               <| and(pres1,and(pres0,not(sentto10))) |>
201            DREQ(1,0,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
202                                         sentto01, sentto02, sentto03,
203                                         sentto10, sentto12, sentto13,
204                                         sentto20, sentto21, sentto23,
205                                         sentto30, sentto31, sentto32)
206          +
207            DREQ(1,2,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
208                                 sentto01, sentto02, sentto03,
209                                 sentto10, not(sentto12), sentto13,
210                                 sentto20, sentto21, sentto23,
211                                 sentto30, sentto31, sentto32)
212               <| and(pres1,and(pres2,not(sentto12))) |>
213            DREQ(1,2,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
214                                         sentto01, sentto02, sentto03,
215                                         sentto10, sentto12, sentto13,
216                                         sentto20, sentto21, sentto23,
217                                         sentto30, sentto31, sentto32)
218          +
219            DREQ(1,3,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
220                                 sentto01, sentto02, sentto03,
221                                 sentto10, sentto12, not(sentto13),
222                                 sentto20, sentto21, sentto23,
223                                 sentto30, sentto31, sentto32)
224               <| and(pres1,and(pres3,not(sentto13))) |>
225            DREQ(1,3,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
226                                         sentto01, sentto02, sentto03,
227                                         sentto10, sentto12, sentto13,
228                                         sentto20, sentto21, sentto23,
229                                         sentto30, sentto31, sentto32)
230          +
```

```
231             DREQ(2,0,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
232                                        sentto01, sentto02, sentto03,
233                                        sentto10, sentto12, sentto13,
234                                        not(sentto20), sentto21, sentto23,
235                                        sentto30, sentto31, sentto32)
236                <| and(pres2,and(pres0,not(sentto20))) |>
237             DREQ(2,0,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
238                                               sentto01, sentto02, sentto03,
239                                               sentto10, sentto12, sentto13,
240                                               sentto20, sentto21, sentto23,
241                                               sentto30, sentto31, sentto32)
242           +
243             DREQ(2,1,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
244                                        sentto01, sentto02, sentto03,
245                                        sentto10, sentto12, sentto13,
246                                        sentto20, not(sentto21), sentto23,
247                                        sentto30, sentto31, sentto32)
248                <| and(pres2,and(pres1,not(sentto21))) |>
249             DREQ(2,1,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
250                                               sentto01, sentto02, sentto03,
251                                               sentto10, sentto12, sentto13,
252                                               sentto20, sentto21, sentto23,
253                                               sentto30, sentto31, sentto32)
254           +
255             DREQ(2,3,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
256                                        sentto01, sentto02, sentto03,
257                                        sentto10, sentto12, sentto13,
258                                        sentto20, sentto21, not(sentto23),
259                                        sentto30, sentto31, sentto32)
260                <| and(pres2,and(pres3,not(sentto23))) |>
261             DREQ(2,3,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
262                                               sentto01, sentto02, sentto03,
263                                               sentto10, sentto12, sentto13,
264                                               sentto20, sentto21, sentto23,
265                                               sentto30, sentto31, sentto32)
266           +
267             DREQ(3,0,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
268                                        sentto01, sentto02, sentto03,
269                                        sentto10, sentto12, sentto13,
270                                        sentto20, sentto21, sentto23,
271                                        not(sentto30), sentto31, sentto32)
272                <| and(pres3,and(pres0,not(sentto30))) |>
273             DREQ(3,0,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
274                                               sentto01, sentto02, sentto03,
275                                               sentto10, sentto12, sentto13,
276                                               sentto20, sentto21, sentto23,
277                                               sentto30, sentto31, sentto32)
278           +
279             DREQ(3,1,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
280                                        sentto01, sentto02, sentto03,
281                                        sentto10, sentto12, sentto13,
```

```
282                                     sentto20, sentto21, sentto23,
283                                     sentto30, not(sentto31), sentto32)
284               <| and(pres3,and(pres1,not(sentto31))) |>
285             DREQ(3,1,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
286                                            sentto01, sentto02, sentto03,
287                                            sentto10, sentto12, sentto13,
288                                            sentto20, sentto21, sentto23,
289                                            sentto30, sentto31, sentto32)
290           +
291             DREQ(3,2,MES,OK) . Chatbox(pres0, pres1, pres2, pres3,
292                                     sentto01, sentto02, sentto03,
293                                     sentto10, sentto12, sentto13,
294                                     sentto20, sentto21, sentto23,
295                                     sentto30, sentto31, not(sentto32))
296               <| and(pres3,and(pres2,not(sentto32))) |>
297             DREQ(3,2,MES,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
298                                            sentto01, sentto02, sentto03,
299                                            sentto10, sentto12, sentto13,
300                                            sentto20, sentto21, sentto23,
301                                            sentto30, sentto31, sentto32)
302           +
303             DREQ(0,0,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
304                                            sentto01, sentto02, sentto03,
305                                            sentto10, sentto12, sentto13,
306                                            sentto20, sentto21, sentto23,
307                                            sentto30, sentto31, sentto32)
308               <| T |>
309             delta
310           +
311             DREQ(1,1,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
312                                            sentto01, sentto02, sentto03,
313                                            sentto10, sentto12, sentto13,
314                                            sentto20, sentto21, sentto23,
315                                            sentto30, sentto31, sentto32)
316               <| T |>
317             delta
318           +
319             DREQ(2,2,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
320                                            sentto01, sentto02, sentto03,
321                                            sentto10, sentto12, sentto13,
322                                            sentto20, sentto21, sentto23,
323                                            sentto30, sentto31, sentto32)
324               <| T |>
325             delta
326           +
327             DREQ(3,3,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
328                                            sentto01, sentto02, sentto03,
329                                            sentto10, sentto12, sentto13,
330                                            sentto20, sentto21, sentto23,
331                                            sentto30, sentto31, sentto32)
332               <| T |>
```

```
333             delta
334           +
335             DREQ(0,1,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
336                                        sentto01, sentto02, sentto03,
337                                        not(sentto10), sentto12, sentto13,
338                                        sentto20, sentto21, sentto23,
339                                        sentto30, sentto31, sentto32)
340                <| and(pres0,and(pres1,sentto10)) |>
341             DREQ(0,1,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
342                                               sentto01, sentto02, sentto03,
343                                               sentto10, sentto12, sentto13,
344                                               sentto20, sentto21, sentto23,
345                                               sentto30, sentto31, sentto32)
346           +
347             DREQ(0,2,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
348                                        sentto01, sentto02,sentto03,
349                                        sentto10, sentto12, sentto13,
350                                        not(sentto20), sentto21, sentto23,
351                                        sentto30, sentto31, sentto32)
352                <| and(pres0,and(pres2,sentto20)) |>
353             DREQ(0,2,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
354                                               sentto01, sentto02, sentto03,
355                                               sentto10, sentto12, sentto13,
356                                               sentto20, sentto21, sentto23,
357                                               sentto30, sentto31, sentto32)
358           +
359             DREQ(0,3,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
360                                        sentto01, sentto02, sentto03,
361                                        sentto10, sentto12, sentto13,
362                                        sentto20, sentto21, sentto23,
363                                        not(sentto30), sentto31, sentto32)
364                <| and(pres0,and(pres3,sentto30)) |>
365             DREQ(0,3,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
366                                               sentto01, sentto02, sentto03,
367                                               sentto10, sentto12, sentto13,
368                                               sentto20, sentto21, sentto23,
369                                               sentto30, sentto31, sentto32)
370           +
371             DREQ(1,0,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
372                                        not(sentto01), sentto02, sentto03,
373                                        sentto10, sentto12, sentto13,
374                                        sentto20, sentto21, sentto23,
375                                        sentto30, sentto31, sentto32)
376                <| and(pres1,and(pres0,sentto01)) |>
377             DREQ(1,0,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
378                                               sentto01, sentto02, sentto03,
379                                               sentto10, sentto12, sentto13,
380                                               sentto20, sentto21, sentto23,
381                                               sentto30, sentto31, sentto32)
382           +
383             DREQ(1,2,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
```

```
384                                 sentto01, sentto02, sentto03,
385                                 sentto10, sentto12, sentto13,
386                                 sentto20, not(sentto21), sentto23,
387                                 sentto30, sentto31, sentto32)
388                <| and(pres1,and(pres2,sentto21)) |>
389             DREQ(1,2,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
390                                          sentto01, sentto02, sentto03,
391                                          sentto10, sentto12, sentto13,
392                                          sentto20, sentto21, sentto23,
393                                          sentto30, sentto31, sentto32)
394           +
395             DREQ(1,3,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
396                                 sentto01, sentto02, sentto03,
397                                 sentto10, sentto12, sentto13,
398                                 sentto20, sentto21, sentto23,
399                                 sentto30, not(sentto31), sentto32)
400                <| and(pres1,and(pres3,sentto31)) |>
401             DREQ(1,3,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
402                                          sentto01, sentto02, sentto03,
403                                          sentto10, sentto12, sentto13,
404                                          sentto20, sentto21, sentto23,
405                                          sentto30, sentto31, sentto32)
406           +
407             DREQ(2,0,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
408                                 sentto01, not(sentto02), sentto03,
409                                 sentto10, sentto12, sentto13,
410                                 sentto20, sentto21, sentto23,
411                                 sentto30, sentto31, sentto32)
412                <| and(pres2,and(pres0,sentto02)) |>
413             DREQ(2,0,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
414                                          sentto01, sentto02, sentto03,
415                                          sentto10, sentto12, sentto13,
416                                          sentto20, sentto21, sentto23,
417                                          sentto30, sentto31, sentto32)
418           +
419             DREQ(2,1,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
420                                 sentto01, sentto02, sentto03,
421                                 sentto10, not(sentto12), sentto13,
422                                 sentto20, sentto21, sentto23,
423                                 sentto30, sentto31, sentto32)
424                <| and(pres2,and(pres1,sentto12)) |>
425             DREQ(2,1,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
426                                          sentto01, sentto02, sentto03,
427                                          sentto10, sentto12, sentto13,
428                                          sentto20, sentto21, sentto23,
429                                          sentto30, sentto31, sentto32)
430           +
431             DREQ(2,3,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
432                                 sentto01, sentto02, sentto03,
433                                 sentto10, sentto12, sentto13,
434                                 sentto20, sentto21, sentto23,
```

```
435                                sentto30, sentto31, not(sentto32))
436              <| and(pres2,and(pres3,sentto32)) |>
437            DREQ(2,3,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
438                                              sentto01, sentto02, sentto03,
439                                              sentto10, sentto12, sentto13,
440                                              sentto20, sentto21, sentto23,
441                                              sentto30, sentto31, sentto32)
442          +
443            DREQ(3,0,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
444                                sentto01, sentto02, not(sentto03),
445                                sentto10, sentto12, sentto13,
446                                sentto20, sentto21, sentto23,
447                                sentto30, sentto31, sentto32)
448              <| and(pres3,and(pres0,sentto03)) |>
449            DREQ(3,0,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
450                                              sentto01, sentto02, sentto03,
451                                              sentto10, sentto12, sentto13,
452                                              sentto20, sentto21, sentto23,
453                                              sentto30, sentto31, sentto32)
454          +
455            DREQ(3,1,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
456                                sentto01, sentto02, sentto03,
457                                sentto10, sentto12, not(sentto13),
458                                sentto20, sentto21, sentto23,
459                                sentto30, sentto31, sentto32)
460              <| and(pres3,and(pres1,sentto13)) |>
461            DREQ(3,1,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
462                                              sentto01, sentto02, sentto03,
463                                              sentto10, sentto12, sentto13,
464                                              sentto20, sentto21, sentto23,
465                                              sentto30, sentto31, sentto32)
466          +
467            DREQ(3,2,ACK,OK) . Chatbox(pres0, pres1, pres2, pres3,
468                                sentto01, sentto02, sentto03,
469                                sentto10, sentto12, sentto13,
470                                sentto20, sentto21, not(sentto23),
471                                sentto30, sentto31, sentto32)
472              <| and(pres3,and(pres2,sentto23)) |>
473            DREQ(3,2,ACK,NO_OUTPUT) . Chatbox(pres0, pres1, pres2, pres3,
474                                              sentto01, sentto02, sentto03,
475                                              sentto10, sentto12, sentto13,
476                                              sentto20, sentto21, sentto23,
477                                              sentto30, sentto31, sentto32)
478
479
480  init    Chatbox(F,F,F,F, F,F,F, F,F,F, F,F,F, F,F,F)
481
```

State space of symmetry for user 1

```
1    des (0, 44, 6)
```

```
2    (0, "1 !JOIN !0 !OK", 1)
3    (0, "15 !LEAVE !0 !NO_OUTPUT", 0)
4    (0, "16 !DREQ !0 !0 !MES !NO_OUTPUT", 0)
5    (0, "17 !DREQ !0 !0 !ACK !NO_OUTPUT", 0)
6    (0, "18 !DREQ !0 !1 !MES !NO_OUTPUT", 0)
7    (0, "19 !DREQ !0 !1 !ACK !NO_OUTPUT", 0)
8    (1, "2 !LEAVE !0 !OK", 0)
9    (1, "3 !DREQ !0 !1 !MES !DIND", 2)
10   (1, "5 !DREQ !1 !0 !MES !DIND", 2)
11   (1, "20 !JOIN !0 !NO_OUTPUT", 1)
12   (1, "21 !DREQ !0 !0 !MES !NO_OUTPUT", 1)
13   (1, "22 !DREQ !0 !0 !ACK !NO_OUTPUT", 1)
14   (1, "23 !DREQ !0 !1 !MES !NO_OUTPUT", 1)
15   (1, "24 !DREQ !0 !1 !ACK !NO_OUTPUT", 1)
16   (2, "4 !DREQ !1 !0 !ACK !DIND", 1)
17   (2, "6 !DREQ !0 !1 !ACK !DIND", 1)
18   (2, "7 !LEAVE !0 !OK", 3)
19   (2, "9 !DREQ !1 !0 !MES !DIND", 4)
20   (2, "12 !DREQ !0 !1 !MES !DIND", 4)
21   (2, "25 !JOIN !0 !NO_OUTPUT", 2)
22   (2, "26 !DREQ !0 !0 !MES !NO_OUTPUT", 2)
23   (2, "27 !DREQ !0 !0 !ACK !NO_OUTPUT", 2)
24   (2, "28 !DREQ !0 !1 !MES !NO_OUTPUT", 2)
25   (2, "29 !DREQ !0 !1 !ACK !NO_OUTPUT", 2)
26   (3, "8 !JOIN !0 !OK", 2)
27   (3, "30 !LEAVE !0 !NO_OUTPUT", 3)
28   (3, "31 !DREQ !0 !0 !MES !NO_OUTPUT", 3)
29   (3, "32 !DREQ !0 !0 !ACK !NO_OUTPUT", 3)
30   (3, "33 !DREQ !0 !1 !MES !NO_OUTPUT", 3)
31   (3, "34 !DREQ !0 !1 !ACK !NO_OUTPUT", 3)
32   (4, "10 !DREQ !0 !1 !ACK !DIND", 2)
33   (4, "11 !DREQ !1 !0 !ACK !DIND", 2)
34   (4, "13 !LEAVE !0 !OK", 5)
35   (4, "35 !JOIN !0 !NO_OUTPUT", 4)
36   (4, "36 !DREQ !0 !0 !MES !NO_OUTPUT", 4)
37   (4, "37 !DREQ !0 !0 !ACK !NO_OUTPUT", 4)
38   (4, "38 !DREQ !0 !1 !MES !NO_OUTPUT", 4)
39   (4, "39 !DREQ !0 !1 !ACK !NO_OUTPUT", 4)
40   (5, "14 !JOIN !0 !OK", 4)
41   (5, "40 !LEAVE !0 !NO_OUTPUT", 5)
42   (5, "41 !DREQ !0 !0 !MES !NO_OUTPUT", 5)
43   (5, "42 !DREQ !0 !0 !ACK !NO_OUTPUT", 5)
44   (5, "43 !DREQ !0 !1 !MES !NO_OUTPUT", 5)
45   (5, "44 !DREQ !0 !1 !ACK !NO_OUTPUT", 5)
```

Lotos model for 3 users without joining/leaving

```
1    specification chatbox[JOIN, LEAVE, DREQ] : noexit
2
3    library
4       X_BOOLEAN, X_NATURAL
```

```
5    endlib
6
7    type CHATBOX_TYPES is CHATBOX_RENAME_TYPES, BOOLEAN, NATURALNUMBER
8       sorts MES_TYPE,
9             VECTOR_TYPE (*! implementedby C_VECTOR_TYPE comparedby COMPAREV \
10                            enumeratedby ENUMV printedby PRINTV external *),
11            MATRIX_TYPE (*! implementedby C_MATRIX_TYPE comparedby COMPAREM \
12                            enumeratedby ENUMM printedby PRINTM external *),
13            OUTPUT
14      opns
15           NO_OUTPUT (*! constructor *),
16           DIND (*! constructor *): -> OUTPUT
17           mes (*! constructor *),
18           ack (*! constructor *) : -> MES_TYPE
19           _eq_ : MES_TYPE, MES_TYPE -> BOOL
20           vector (*! implementedby c_vector constructor external *)
21             : -> VECTOR_TYPE
22           setv (*! implementedby c_setv external *)
23             : USR_TYPE,BOOL,VECTOR_TYPE -> VECTOR_TYPE
24           getv (*! implementedby c_getv external *)
25             : USR_TYPE,VECTOR_TYPE -> BOOL
26           matrix (*! implementedby c_matrix constructor external *)
27             : -> MATRIX_TYPE
28           setm (*! implementedby c_setm external *)
29             : USR_TYPE,USR_TYPE,BOOL,MATRIX_TYPE -> MATRIX_TYPE
30           getm (*! implementedby c_getm external *)
31             : USR_TYPE,USR_TYPE,MATRIX_TYPE -> BOOL
32           updatem (*! implementedby c_updatem external *)
33             : USR_TYPE,BOOL,VECTOR_TYPE,MATRIX_TYPE -> MATRIX_TYPE
34      eqns
35         ofsort BOOL
36           ack eq mes = false;
37           mes eq ack = false;
38           ack eq ack = true;
39           mes eq mes = true;
40   endtype
41
42   type CHATBOX_RENAME_TYPES is NATURALNUMBER renamedby
43      sortnames USR_TYPE for NAT
44   endtype
45
46   behaviour
47
48   CHAT[DREQ]
49       (matrix,matrix)
50
51   where
52
53   process CHAT
54         [DREQ]
55         (SENT_TO,TO_ACK:MATRIX_TYPE)
```

```
56     : noexit :=
57
58       (choice u1:USR_TYPE, u2:USR_TYPE []
59        (
60     DREQ ! u1 ! u2 ! mes ! NO_OUTPUT
61     [(u1 eq u2) or getm(u1,u2,SENT_TO)];
62        CHAT[DREQ](SENT_TO,TO_ACK)
63        []
64     DREQ ! u1 ! u2 ! ack ! NO_OUTPUT
65     [(u1 eq u2) or not(getm(u1,u2,TO_ACK))];
66        CHAT[DREQ](SENT_TO,TO_ACK)
67        []
68     DREQ ! u1 ! u2 ! mes ! DIND
69     [not(u1 eq u2) and not(getm(u1,u2,SENT_TO))];
70        CHAT[DREQ]
71          (setm(u1,u2,true,SENT_TO),setm(u2,u1,true,TO_ACK))
72        []
73     DREQ ! u1 ! u2 ! ack ! DIND
74     [not(u1 eq u2) and getm(u1,u2,TO_ACK)];
75        CHAT[DREQ]
76          (setm(u2,u1,false,SENT_TO),setm(u1,u2,false,TO_ACK))
77      ))
78
79    endproc
80
81    endspec (* chatbox *)
```

State space of symmetry for user 1 without joining/leaving

```
1     des (0, 20, 3)
2     (0, "1 !DREQ !0 !1 !MES !DIND", 1)
3     (0, "3 !DREQ !1 !0 !MES !DIND", 1)
4     (0, "9 !DREQ !0 !1 !MES !NO_OUTPUT", 0)
5     (0, "10 !DREQ !0 !1 !ACK !NO_OUTPUT", 0)
6     (0, "11 !DREQ !0 !0 !MES !NO_OUTPUT", 0)
7     (0, "12 !DREQ !0 !0 !ACK !NO_OUTPUT", 0)
8     (1, "2 !DREQ !1 !0 !ACK !DIND", 0)
9     (1, "4 !DREQ !0 !1 !ACK !DIND", 0)
10    (1, "5 !DREQ !1 !0 !MES !DIND", 2)
11    (1, "7 !DREQ !0 !1 !MES !DIND", 2)
12    (1, "13 !DREQ !0 !1 !MES !NO_OUTPUT", 1)
13    (1, "14 !DREQ !0 !1 !ACK !NO_OUTPUT", 1)
14    (1, "15 !DREQ !0 !0 !MES !NO_OUTPUT", 1)
15    (1, "16 !DREQ !0 !0 !ACK !NO_OUTPUT", 1)
16    (2, "6 !DREQ !0 !1 !ACK !DIND", 1)
17    (2, "8 !DREQ !1 !0 !ACK !DIND", 1)
18    (2, "17 !DREQ !0 !1 !MES !NO_OUTPUT", 2)
19    (2, "18 !DREQ !0 !1 !ACK !NO_OUTPUT", 2)
20    (2, "19 !DREQ !0 !0 !MES !NO_OUTPUT", 2)
21    (2, "20 !DREQ !0 !0 !ACK !NO_OUTPUT", 2)
```

Algorithm Kernel for 4 users

```
1  /***************************************************************************
2   *                           O P E N / C A E S A R
3   *--------------------------------------------------------------------------
4   *   INRIA - Unite de Recherche Rhone-Alpes
5   *   655, avenue de l'Europe
6   *   38330 Montbonnot Saint Martin
7   *   FRANCE
8   *--------------------------------------------------------------------------
9   *   Module          :       kernel_4_users.c (adapted from reductor.c)
10  *   Auteur          :       Judi Romijn (e-mail: judi@cwi.nl)
11  *   Date            :       98/03/20 (97/09/23 18:08:37)
12  **************************************************************************/
13
14
15 /* This program can be used by the following command:
16       bcg_open <file1>.bcg kernel_4_users.c <file2>.bcg <file3>.bcg
17    where
18         <file1>.bcg = LTS Spec in bcg format (input)
19         <file2>.bcg = LTS Symm in bcg format (input)
20         <file3>.bcg = LTS Result in bcg format (output)
21
22    This program computes an LTS Result which is a sub-LTS of Spec, more
23    precisely, it computes a Kernel for Spec, based on the symmetry given
24    in LTS Symm.
25    More information on the theory of kernels and symmetry is found
26    in the paper:
27         Exploiting Symmetry in Protocol Testing
28         Judi Romijn and Jan Springintveld.
29    An extended abstract and the full version of this paper are available
30    at
31         http://www.cwi.nl/~judi/
32 */
33
34 /* ASSUMPTIONS:
35    -> the pair <Symm, Permutations> with the representation function r()
36       is a symmetry on Spec
37       (r(sigma) is defined by taking the smallest trace tau such that
38          tau is the smallest trace that is symmetric to sigma,
39          `smallest' = minimal element wrt lexigraphic ordering
40                       induced by the ranking of action labels as
41                       given in MY_RANK_LABEL)
42    -> action labels in Spec are of form <label>
43       with
44           <label> = <input string> "!" { <offer> "!" }+ <output string>
45    -> action labels in Symm are of form
46           <edgenr> "!" <label>
47       with
48           <edgenr> = unique edgenumber of the transition in Symm
49    -> the <label> part in Symm must also occur in Spec (subset)
```

```
50        -> function MY_PERMUTE_LABEL assumes that MY_PERMUTATION_SIZE
51           is 4, so 2 parameters are permuted (nr 0 by nr 1 and nr 2 by nr 3).
52        -> typedefs MY_TYPE_PERMUTATION and function MY_PERMUTE_LABEL
53           assume that the parameter to be permuted is a single character
54           (see also the static constant Permutations)
55        -> typedef MY_TYPE_COLOUR and function MY_ADD_ITEM_COLOUR_LIST
56           assume the edge nr from LTS Symm is stored as int
57
58        -> Lemma 6.15 from the full paper (\ref{La:efficiente kernel stap 1})
59           implies that for each two visits of one state with the same states
60           for each permutation of LTS Symm, we can skip one of the
61           two visits. This makes the implementation far more efficient.
62
63        -> The search tables caesar_t1 and caesar_t2 keep track of all the
64           states visited until now. The table caesar_t1 stores each state
65           that needs to be visited and explored, instead of storing a
66           state and on visit decide whether it should be explored.
67           So a state needs to be stored only if
68             - the transition leading to it is the representative one
69               (all other transitions symmetric to this one are skipped)
70             - the path leading to it does not contain the state (a loop)
71               (this is stored with pointers throughout the table, of
72                course a NULL pointer for the initial state with empty trace)
73             - there is not an occurrence in the table of the same state
74               (unique state id!) where also all the permutations of LTS
75               Symm are in the same state (Lemma 6.15 from the full paper)
76           Still, multiple occurrences of one state can occur in the table
77           caesar_t1. So the requirements from the OPEN/CAESAR manual
78           (Hubert Garavel, October 18, 1997) are not obeyed here.
79           During execution this gives no errors.
80
81        -> The constant MY_BRANCHING_SIZE is used to denote the branching
82           factor of LTS Spec, this number is needed for finite administration
83           of whether transitions from Spec have been written to Result or not
84           (to avoid writing more than once, which is very inefficient!).
85     */
86
87     /* ASSUMPTIONS for this example CHATBOX:
88        -> In the action labels in Spec en Symm:
89               <input string> = "JOIN" | "LEAVE" | "DREQ"
90               <output string> = "OK" | "DIND" | "NO_OUTPUT"
91               <offer> = [0..9] | "MES" | "ACK"
92
93           "JOIN", "LEAVE" have 1 offer: [0..9]
94           "JOIN", "LEAVE" have output:  "OK" | "NO_OUTPUT"
95           "DREQ" has 3 offers:          [0..9] "!" [0..9] "!" ("MES" | "ACK")
96           "DREQ" has output:            "DIND" | "NO_OUTPUT"
97
98        -> constant MY_VECTOR_SIZE is 12: there are 12 permutations of the
99           symmetry template LTS Symm, because there are 4 users of the chatbox.
100
```

```
101  */
102
103
104  /*----PRELIMINARY DEFS, INCLUDES, GLOBAL VARS--------------------------------*/
105
106  static char caesar_sccs_what_kernel_algo[] =
107  "@(#)open/caesar -- 2.15 -- 98/03/07 18:08:37 -- kernel_4_users.c";
108
109  static char *LDFLAGS = "@(#)LDFLAGS = \
110      -L${BCG:-$CADP}/bin.‘$CADP/com/arch‘ -lBCG_IO -lBCG -lm";
111
112  #include <string.h>
113  #include "caesar_graph.h"
114  #include "caesar_edge.h"
115  #include "caesar_table_1.h"
116
117  #include <signal.h>
118  #include "bcg_user.h"
119  /* bcg_user.h includes bcg_io_write_bcg.h */
120
121    /* Global variable for exploring LTS Symm, since it is used in some (one)
122       of the following functions */
123  BCG_TYPE_OBJECT_TRANSITION bcg_object_transition;
124
125  /*----POP-UP WINDOW WITH STATISTICS------------------------------------------*/
126
127  static void caesar_abort (bcg_signal)
128  int  bcg_signal;
129
130  {
131       signal (SIGHUP, SIG_IGN);
132       signal (SIGINT, SIG_IGN);
133       signal (SIGQUIT, SIG_IGN);
134       signal (SIGILL, SIG_IGN);
135       signal (SIGABRT, SIG_IGN);
136       signal (SIGFPE, SIG_IGN);
137       signal (SIGBUS, SIG_IGN);
138       signal (SIGSEGV, SIG_IGN);
139  #ifdef SIGSYS
140       signal (SIGSYS, SIG_IGN);
141  #endif
142       signal (SIGTERM, SIG_IGN);
143       signal (SIGPIPE, SIG_IGN);
144       BCG_IO_WRITE_BCG_END ();
145       CAESAR_ERROR ("caught interrupt: closing BCG file");
146  }
147
148  /*----TYPEDEFS FOR PERMUTATION-----------------------------------------------*/
149
150  #define MY_PERMUTATION_SIZE 4  /* 2 action parameters to be permuted */
151  #define MY_VECTOR_SIZE 12  /* 12 permutations of LTS Symm (incl identity pm) */
```

```
152
153   /* parameters are to be permuted in a string label, so type char
154      in our case, we'll just deal with parameters 0 .. n with n<10,
155      so 1 char suffices */
156 typedef char MY_TYPE_PERMUTATION [MY_PERMUTATION_SIZE];
157 typedef MY_TYPE_PERMUTATION MY_TYPE_PERMUTATION_VECTOR[MY_VECTOR_SIZE];
158
159   /* the permutations are stored such that
160      Permutations[i] gives the i-th permutation, and
161      Permutations[i][j] with j even, gives value to be permuted, and
162      Permutations[i][j] with j odd, gives new value for Permutations[i][j-1]
163      Here you have to know what the default values are!!! */
164 static MY_TYPE_PERMUTATION_VECTOR Permutations = {
165         {'0','0','1','1'},    /* (0,1) (identity perm) */
166         {'0','0','1','2'},    /* (0,2) */
167         {'0','0','1','3'},    /* (0,3) */
168         {'0','1','1','0'},    /* (1,0) */
169         {'0','1','1','2'},    /* (1,2) */
170         {'0','1','1','3'},    /* (1,3) */
171         {'0','2','1','0'},    /* (2,0) */
172         {'0','2','1','1'},    /* (2,1) */
173         {'0','2','1','3'},    /* (2,3) */
174         {'0','3','1','0'},    /* (3,0) */
175         {'0','3','1','1'},    /* (3,1) */
176         {'0','3','1','2'}     /* (3,2) */
177 };
178
179 /*----TYPEDEFS FOR COLOURING-----------------------------------------------*/
180
181 #define MY_LABEL_SIZE   100     /* reserve space big enough for action labels */
182 typedef char MY_TYPE_LABEL [MY_LABEL_SIZE];
183
184   /* state vector of BCG states for exploring LTS Symm
185      there is a state for each permutation of LTS Symm  */
186 typedef BCG_TYPE_NATURAL MY_TYPE_STATE_VECTOR[MY_VECTOR_SIZE];
187
188 typedef int MY_TYPE_COLOUR; /* the edge nr from LTS Symm is stored as int */
189
190 typedef struct tag_colour *MY_TYPE_COLOUR_LIST;
191
192 typedef struct tag_colour {
193    MY_TYPE_COLOUR colour;
194    MY_TYPE_COLOUR_LIST next;
195 } MY_TYPE_COLOUR_BODY;
196
197 /*----FUNCTIONS FOR COLOURING-----------------------------------------------*/
198
199 char *MY_GET_EDGENR(MY_TYPE_LABEL label)
200 /* Note: the contents of label are changed! */
201 { /* the edges of the bcg of the symmetry format have labels with first
202      the edgenr, then a "!", then the action label so this function must
```

```
203       get the characters of label before the first "!" */
204
205    char *tmp;
206
207    tmp = strchr ((char *) label, '!');
208    (* tmp)= '\0';
209
210    return (char *) label;
211  }
212
213  char *MY_GET_ACTION(MY_TYPE_LABEL label)
214  /* Note: the contents of label may be changed! */
215  { /* the edges of the bcg of the symmetry format have labels with first
216       the edgenr, then a "!", then the action label so this function must
217       get the characters of label after the first "!" */
218
219    char *tmp;
220
221    tmp = strchr ((char *) label, '!');
222
223    return ++tmp;
224  }
225
226  CAESAR_TYPE_BOOLEAN MY_NO_OUTPUT(CAESAR_TYPE_LABEL caesar_l)
227  { /* determine whether the last offer in caesar_l
228       is equal to "NO_OUTPUT" */
229
230    char *tmp,separator[]="!";
231    MY_TYPE_LABEL label;
232
233    CAESAR_DUMP_LABEL (label, caesar_l);
234
235    tmp = strtok((char *) label, separator);
236    switch(*tmp){
237     case 'J' :  /* "JOIN" or "JOIN " or "LEAVE" or "LEAVE " */
238     case 'L' : tmp = strtok(NULL, separator);  /* 1 parameter/offer */
239                 tmp = strtok(NULL, separator);
240                break;
241     case 'D' : tmp = strtok(NULL, separator);  /* "DREQ" or "DREQ " */
242                tmp = strtok(NULL, separator);  /* 3 parameters/offers */
243                tmp = strtok(NULL, separator);
244                tmp = strtok(NULL, separator);
245                break;
246     default: tmp = strtok(NULL, separator); /* edge number taken off */
247              switch(*tmp){
248              case 'J' :  /* "JOIN" or "JOIN " or "LEAVE" or "LEAVE " */
249              case 'L' : tmp = strtok(NULL, separator);
250                         tmp = strtok(NULL, separator);
251                         break;
252              case 'D' : tmp = strtok(NULL, separator);  /* "DREQ" or "DREQ " */
253                         tmp = strtok(NULL, separator);
```

```
254                         tmp = strtok(NULL, separator);
255                         tmp = strtok(NULL, separator);
256                         break;
257              }
258   }
259   return (CAESAR_TYPE_BOOLEAN) ((strcmp(tmp,"NO_OUTPUT")==0)? 1 : 0);
260 }
261
262 int MY_RANK_LABEL(CAESAR_TYPE_LABEL caesar_l)
263 { /* determine ranking of label conform following:
264      JOIN!i!o = 100 + i
265      LEAVE!i!o = 210 - i
266      DREQ!i!j!m!o =
267            if (m=="MES")
268            then for (ij): 01<10<02<20<03<30<12<21<13<31<23<32
269            else for (ij): 32<23<31<13<21<12<30<03<20<02<10<01
270      (in this case such an ordering is more easily implemented
271       with a case statement than one arithmetic expression)
272
273   Note: these labels get same ranking for o="OK" or "DIND" or "NO_OUTPUT";
274         since labels with different outputs will never be symmetric (they
275         are covered by different edges), this is safe
276   */
277
278
279     char *tmp,separator[]="!";
280     int rnk,rnk1,rnk2;
281     MY_TYPE_LABEL label;
282
283     CAESAR_DUMP_LABEL (label, caesar_l);
284     tmp = strtok((char *) label, separator);
285     switch(*tmp){
286      /* "JOIN" or "JOIN " */
287      case 'J' : tmp = strtok(NULL, separator);
288                 rnk = 100 + atoi(tmp);
289                 break;
290
291      /* "LEAVE" or "LEAVE " */
292      case 'L' : tmp = strtok(NULL, separator);
293                 rnk = 210 - atoi(tmp);
294                 break;
295
296      /* "DREQ" or "DREQ " */
297      case 'D' : tmp = strtok(NULL, separator);
298                 rnk1 = 10*atoi(tmp);
299                 tmp = strtok(NULL, separator);
300                 rnk1 += atoi(tmp);
301                 tmp = strtok(NULL, separator);
302                 if ((*tmp)=='M')  /* this offer is MES or ACK */
303                   switch(rnk1){
304                     case 01: rnk = 301; break;
```

```
305                         case 10: rnk = 302; break;
306                         case 02: rnk = 303; break;
307                         case 20: rnk = 304; break;
308                         case 03: rnk = 305; break;
309                         case 30: rnk = 306; break;
310                         case 12: rnk = 307; break;
311                         case 21: rnk = 308; break;
312                         case 13: rnk = 309; break;
313                         case 31: rnk = 310; break;
314                         case 23: rnk = 311; break;
315                         case 32: rnk = 312; break;
316                       }
317                   else
318                     switch(rnk1){
319                       case 01: rnk = 350; break;
320                       case 10: rnk = 349; break;
321                       case 02: rnk = 348; break;
322                       case 20: rnk = 347; break;
323                       case 03: rnk = 346; break;
324                       case 30: rnk = 345; break;
325                       case 12: rnk = 344; break;
326                       case 21: rnk = 343; break;
327                       case 13: rnk = 342; break;
328                       case 31: rnk = 341; break;
329                       case 23: rnk = 340; break;
330                       case 32: rnk = 339; break;
331                     }
332                   break;
333     }
334     return rnk;
335 }
336
337 char *MY_PERMUTE_LABEL(char *str,MY_TYPE_PERMUTATION perm)
338 /* Note: the contents of str are changed */
339 {
340   /* Each occurrence in string str of perm[0] must be replaced by perm[1]
341      and each occ of perm[2] must be replaced by perm[3], simultaneously
342      Assumptions: replace only single characters,
343                   only 2 replacements to be made (nr 1 for nr 0, nr 3 for nr 2)
344   */
345
346   char *tmp;
347
348   tmp = str;
349   while ((*tmp) != '\0'){
350     if ((*tmp) == perm[0])
351        (*tmp) = perm[1];
352     else
353       if ((*tmp) == perm[2])
354          (*tmp) = perm[3];
355     tmp++;
```

```
356     }
357     return str;
358   }
359
360   void MY_CREATE_COLOUR_LIST(MY_TYPE_COLOUR_LIST *cl)
361   { /* create empty list at which pointer cl points */
362
363     (* cl) = (MY_TYPE_COLOUR_LIST) malloc(sizeof(MY_TYPE_COLOUR_BODY));
364   }
365
366   void MY_DELETE_COLOUR_LIST(MY_TYPE_COLOUR_LIST *cl)
367   { /* free memory space corresponding to list pointed to by cl. */
368
369     MY_TYPE_COLOUR_LIST tmp1,tmp2;
370
371     tmp1 = tmp2 = (*cl);
372     if (tmp1!=NULL) {
373       while (tmp1!=NULL) {
374         tmp1 = tmp1->next;
375         free(tmp2);
376         tmp2 = tmp1;
377       }
378     }
379     (*cl) = NULL;
380   }
381
382   int MY_LENGTH_COLOUR_LIST(MY_TYPE_COLOUR_LIST cl)
383   { /* return the number of elements of the list cl */
384
385     int n;
386
387     for (n=0; cl!=NULL;n++)
388       cl = cl->next;
389     return n;
390   }
391
392   MY_TYPE_COLOUR MY_HEAD_COLOUR_LIST(MY_TYPE_COLOUR_LIST cl)
393   { /* return the colour elt in the first elt of the list cl */
394
395     return cl->colour;
396   }
397
398   void MY_TAIL_COLOUR_LIST(MY_TYPE_COLOUR_LIST *cl)
399   { /* delete the first elt from the list cl and store a pointer
400     to the tail of the list in cl */
401
402     MY_TYPE_COLOUR_LIST tmp;
403
404     tmp = *cl;
405     (*cl) = tmp->next;
406     free(tmp);
```

```
407  }
408
409  void MY_ADD_ITEM_COLOUR_LIST(char *str,MY_TYPE_COLOUR_LIST *cl)
410  { /* add item str at the end of list pointed at by cl. */
411
412    MY_TYPE_COLOUR_LIST tmp;
413
414    tmp = *cl;
415    if (tmp != NULL){
416      MY_TYPE_COLOUR_LIST tmp2;
417
418      while (tmp->next!=NULL){
419        tmp = tmp->next;
420      }
421
422      MY_CREATE_COLOUR_LIST(&tmp2);
423      tmp2->colour = atoi(str); /* blanks at the end of str are deleted? */
424      tmp2->next = NULL;
425      tmp->next = tmp2;
426    } else {
427      MY_CREATE_COLOUR_LIST(&tmp);
428      tmp->colour = atoi(str);
429      tmp->next = NULL;
430      (*cl) = tmp;
431    }
432  }
433
434  void MY_MAKE_COLOUR_LIST(cursymm,caesar_l,cl,newsymm)
435  /* Note: this function uses global variable bcg_object_transition, which
436           must therefore be defined earlier! */
437  /* Note: no problems are caused by using the string-changing functions
438           MY_GET_ACTION and MY_GET_EDGENR on the same string ‘label2’,
439           because the order in which these functions are used ensures that
440           they don’t mess with each other’s information */
441   MY_TYPE_STATE_VECTOR cursymm;
442   CAESAR_TYPE_LABEL caesar_l;
443   MY_TYPE_COLOUR_LIST *cl;
444   MY_TYPE_STATE_VECTOR *newsymm;
445  {
446  /* The label caesar_l may be an enabled transition in several of
447     the states in cursymm. For each of these states, add the edge nr
448     of the enabled transition as colour to the colourlist cl. */
449
450    int j,step = 0;
451    MY_TYPE_LABEL label1,label2;
452    BCG_TYPE_STATE_NUMBER bcg_symm_state0, bcg_symm_state1;
453    BCG_TYPE_LABEL_NUMBER bcg_label_number;
454
455    CAESAR_DUMP_LABEL (label1, caesar_l);
456
457    for(j=0;j<MY_VECTOR_SIZE;j++){ /* compute all succ for each permtion */
```

```
458     bcg_symm_state0 = (BCG_TYPE_STATE_NUMBER) cursymm[j];
459     BCG_OT_ITERATE_P_LN(
460          bcg_object_transition,
461          bcg_symm_state0,
462          bcg_label_number,
463          bcg_symm_state1){ /* body of the state iterator */
464       char *tmp;
465
466         /* get the label string from LTS Symm belonging to this label number */
467       strcpy((char *)label2,
468              (char *)BCG_OT_LABEL_STRING(bcg_object_transition,
469                                           bcg_label_number));
470       tmp = MY_PERMUTE_LABEL(MY_GET_ACTION(label2),Permutations[j]);
471
472       if (strcmp(tmp,(char *)label1) == 0){
473
474         (* newsymm)[j] = (BCG_TYPE_NATURAL) bcg_symm_state1;
475         MY_ADD_ITEM_COLOUR_LIST(MY_GET_EDGENR(label2),cl);
476         step = 1;
477       }   /* end if */
478     } BCG_OT_END_ITERATE;  /* ?This END should be used with new CADP? */
479     if (!step)
480       (* newsymm)[j] = cursymm[j];
481     step = 0;
482   }  /* end for */
483 }
484
485 CAESAR_TYPE_BOOLEAN MY_FIND_DELETE_COLOUR_LIST(col,cl)
486  MY_TYPE_COLOUR col;
487  MY_TYPE_COLOUR_LIST *cl;
488 { /* find entry col in list cl, delete entry from cl, return true
489      and store changed list in pointer cl.
490      if entry not found, return false (cl not changed) */
491   CAESAR_TYPE_BOOLEAN found=(CAESAR_TYPE_BOOLEAN) 0;
492   MY_TYPE_COLOUR_LIST tmp;
493
494   tmp = (*cl);
495   if (tmp!=NULL){
496     if (tmp->colour == col){
497       found = (CAESAR_TYPE_BOOLEAN) 1;
498       (*cl) = tmp->next;
499       free(tmp);
500     } else {
501       while ((tmp->next!=NULL) && (tmp->next->colour != col))
502         tmp = tmp->next;
503       if (tmp->next!=NULL){
504         MY_TYPE_COLOUR_LIST tmp2;
505         found = (CAESAR_TYPE_BOOLEAN) 1;
506         tmp2 = tmp->next;
507         tmp->next = tmp2->next;
508         free(tmp2);
```

```
509         }
510       }
511     }
512     return found;
513   }
514
515   CAESAR_TYPE_BOOLEAN MY_SYMMETRIC_COLOUR_LIST(cl1,cl2)
516   MY_TYPE_COLOUR_LIST cl1;
517   MY_TYPE_COLOUR_LIST *cl2;
518   { /* compare the lists of colours cl1 and cl2. These lists should induce
519        equal multisets of colours, we check this by taking the first elt
520        of cl1, see if it's in cl2 and if so, delete it. Then look the
521        first elt of the tail of cl1 and so on.
522     NOTE: cl2 is changed with this function, might be totally emptied */
523
524     int j,n,m;
525     MY_TYPE_COLOUR col1,col2;
526   MY_TYPE_COLOUR_LIST tmp;
527
528     n = MY_LENGTH_COLOUR_LIST(cl1);
529     m = MY_LENGTH_COLOUR_LIST(*cl2);
530
531     if (n==m)
532     { for(j=0;j<n;j++)
533       { col1 = MY_HEAD_COLOUR_LIST(cl1);
534
535         if (!(MY_FIND_DELETE_COLOUR_LIST(col1,cl2)))
536           return (CAESAR_TYPE_BOOLEAN) 0;
537         cl1 = cl1->next;
538       }
539       return (CAESAR_TYPE_BOOLEAN) 1;
540     }
541     return (CAESAR_TYPE_BOOLEAN) 0;
542   }
543
544   /*----VARS/TYPEDEFS FOR SEARCH TABLES----------------------------------------*/
545
546   CAESAR_TYPE_TABLE_1 caesar_t1; /* base = CAESAR_TYPE_STATE,
547                                     mark = MY_TYPE_MARK1_FIELD */
548   CAESAR_TYPE_TABLE_1 caesar_t2; /* base field = CAESAR_TYPE_STATE,
549                                     mark = MY_TYPE_MARK2_FIELD */
550
551     /* caesar_t1 keeps track of states that need to be explored by
552        the algorithm. Per state we also need information of
553        - the path that led to this state
554        - the current state of each permutation of LTS Symm
555
556        caesar_t2 will be used to store each state from caesar_t1
557        only at the first visit to this state. The index of the
558        state in caesar_t2 can then be used as a unique id for the
559        state. It is also used as id when states are written to the
```

```
560       output LTS Result.
561       Furthermore, it is useless and inefficient to write
562       transitions to output LTS Result more than once. So in
563       caesar_t2, we keep track of which transitions have
564       already been written. The branching factor of LTS Spec is
565       an upperbound for the number of transitions that will
566       maximally be written to LTS Result, so we can use a finite
567       structure (vector) for this.
568
569    */
570
571    /* the mark field of table caesar_t1 must store
572       prev:       an int flag indicating whether there is (1) a prev
573                   state or not (0)
574       prev_index: index in t1 of state from which current state was reached
575       symmstates: a vector for current state of each permutation of LTS Symm */
576  typedef struct tag_mark1 *MY_TYPE_MARK1_FIELD;
577  typedef struct tag_mark1 {
578     int prev;
579     CAESAR_TYPE_INDEX_TABLE_1 prev_index;
580     MY_TYPE_STATE_VECTOR symmstates;
581  } MY_TYPE_MARK1_BODY;
582
583  #define MY_BRANCHING_SIZE 80  /* LTS Spec: max 80 outgoing trans per state */
584  typedef struct {
585     MY_TYPE_LABEL label;
586     CAESAR_TYPE_INDEX_TABLE_1 succ;
587  } MY_TYPE_TRANS;
588  typedef MY_TYPE_TRANS MY_TYPE_TRANS_VECTOR[MY_BRANCHING_SIZE];
589
590    /* the mark field of table caesar_t2 must store
591       howmany: an integer indicating how many transitions departing
592                from the state in corr. base field have already been
593                written to the output BCG file
594       trans:   a vector with fixed size of which the first #howmany fields
595                are filled with transitions already written to BCG output */
596  typedef struct tag_mark2 *MY_TYPE_MARK2_FIELD;
597  typedef struct tag_mark2 {
598     int howmany;
599     MY_TYPE_TRANS_VECTOR trans;
600  } MY_TYPE_MARK2_BODY;
601
602    /* when iterating successor states, the edges use the mark field
603       for indicating:
604       '0': this transition has not been explored or found symmetric
605       '1': transition is symmetric to another but not repr, do not explore
606       '2': transition is representant for all symmetric trans, explore!   */
607  typedef char MY_TYPE_EDGE_MARK_FIELD;
608
609  /*----MAIN-------------------------------------------------------------------*/
610
```

```
611 main (argc, argv)
612 int  argc;
613 char *argv[];
614
615 { /*----variable decls-----------------------------------------------------*/
616   int j;
617
618   CAESAR_TYPE_BOOLEAN caesar_monitor;
619   CAESAR_TYPE_STATE caesar_s0, caesar_s1, caesar_s2;
620   CAESAR_TYPE_STATE prev_state, tmp_state;
621   int prev, found, tmp_found;
622   CAESAR_TYPE_EDGE caesar_e1_en, caesar_e1, caesar_e2;
623   CAESAR_TYPE_EDGE *caesar_e_ptr_next;
624   CAESAR_TYPE_LABEL caesar_l1, caesar_l2, caesar_l_r;
625   CAESAR_TYPE_STRING caesar_label;
626   CAESAR_TYPE_POINTER caesar_m1, caesar_m2, pointer_dummy;
627   MY_TYPE_MARK1_BODY *mark1, *mark1_1, *mark1_2, *mark1_p, *prev_mark1, *tmp_mark1;
628   MY_TYPE_MARK1_BODY mark1_dummy;
629   MY_TYPE_MARK1_BODY mark1_body, mark1_body_prev;
630   MY_TYPE_MARK2_BODY *mark2, *mark2_1, *mark2_2, *mark2_p, *prev_mark2;
631   MY_TYPE_MARK2_BODY mark2_dummy;
632   MY_TYPE_MARK2_BODY mark2_body, mark2_body_prev;
633   MY_TYPE_EDGE_MARK_FIELD edge_mark1,edge_mark2;
634   MY_TYPE_EDGE_MARK_FIELD *edge_mark_ptr1,*edge_mark_ptr2, *edge_mark_ptr_r;
635
636   CAESAR_TYPE_INDEX_TABLE_1 bcg_spec_initial_state, bcg_spec_state0, bcg_spec_state1;
637   CAESAR_TYPE_INDEX_TABLE_1 prev_index, caesar_dummy, run;
638   CAESAR_TYPE_INDEX_TABLE_1 get_index_table1, put_index_table1;
639
640   char bcg_filename_output[4096];
641   char bcg_filename_symm[4096];
642   char tmp_label[40];
643   BCG_TYPE_C_STRING bcg_label_string;
644   BCG_TYPE_BOOLEAN bcg_visible;
645
646   MY_TYPE_COLOUR_LIST cl1, cl2;
647   MY_TYPE_STATE_VECTOR cursymmstates,newsymmstates1,newsymmstates2;
648
649 setbuf (stdout, NULL);
650
651   /*----opening LTS Spec----------------------------------------------------*/
652   CAESAR_TOOL = argv[0];
653   --argc;
654   ++argv;
655
656   if (argc != 2)
657           CAESAR_ERROR ("two BCG filenames expected as argument,
658      first LTS Symmetry input file, then LTS Result output file");
659
660   if ((strlen (argv[0]) > 4) &&
661          (strcmp (argv[0] + strlen (argv[0]) - 4, ".bcg") == 0))
```

```
662             sprintf (bcg_filename_symm, "%s", argv[0]);
663     else
664             sprintf (bcg_filename_symm, "%s.bcg", argv[0]);
665
666     --argc;++argv;
667     if ((strlen (argv[0]) > 4) &&
668            (strcmp (argv[0] + strlen (argv[0]) - 4, ".bcg") == 0))
669             sprintf (bcg_filename_output, "%s", argv[0]);
670     else
671             sprintf (bcg_filename_output, "%s.bcg", argv[0]);
672
673     /*----opening LTS Symm-----------------------------------------------*/
674     BCG_INIT (); /* one or more occurrences of BCG_INIT doesn't matter */
675
676     BCG_OT_READ_BCG_BEGIN (bcg_filename_symm, &bcg_object_transition, 1);
677
678     CAESAR_INIT_GRAPH ();
679
680     /* ?? I'm taking a guess at the alignment mark value here */
681     CAESAR_INIT_EDGE (CAESAR_FALSE, CAESAR_TRUE, CAESAR_TRUE, sizeof(char), 16);
682
683     /*----building exploration tables-----------------------------------------*/
684
685     CAESAR_CREATE_TABLE_1 (&caesar_t1,
686                            0, /* use standard base field with CAESAR_TYPE_STATE */
687                            sizeof(MY_TYPE_MARK1_BODY),
688                            0, 0, CAESAR_TRUE,
689                            NULL, NULL,
690                            CAESAR_PRINT_STATE, NULL);
691     if (caesar_t1 == NULL)
692       CAESAR_ERROR ("not enough memory for table 1");
693
694     CAESAR_CREATE_TABLE_1 (&caesar_t2,
695                            0, /* use standard base field with CAESAR_TYPE_STATE */
696                            sizeof(MY_TYPE_MARK2_BODY),
697                            0, 0, CAESAR_TRUE,
698                            NULL, NULL, NULL, NULL);
699     if (caesar_t2 == NULL)
700       CAESAR_ERROR ("not enough memory for table 2");
701
702     /*----storing initial state-----------------------------------------------*/
703     CAESAR_START_STATE ((CAESAR_TYPE_STATE) CAESAR_PUT_BASE_TABLE_1 (caesar_t1));
704
705     CAESAR_START_STATE ((CAESAR_TYPE_STATE) CAESAR_PUT_BASE_TABLE_1 (caesar_t2));
706
707     mark2 = (MY_TYPE_MARK2_BODY*) CAESAR_PUT_MARK_TABLE_1 (caesar_t2);
708     mark2_dummy.howmany = 0;
709     (*mark2) = (mark2_dummy);
710     CAESAR_PUT_TABLE_1 (caesar_t2);
711
712     bcg_spec_initial_state = (BCG_TYPE_STATE_NUMBER)
```

```
713                                  CAESAR_GET_INDEX_TABLE_1(caesar_t2);
714
715     mark1 = (MY_TYPE_MARK1_BODY*) CAESAR_PUT_MARK_TABLE_1 (caesar_t1);
716
717       /* initialise symmstates on initial states of LTS Symm */
718     mark1_dummy.prev = 0;
719     mark1_dummy.prev_index = 0;
720     for (j=0; j<MY_VECTOR_SIZE; j++)
721       mark1_dummy.symmstates[j] = BCG_OT_INITIAL_STATE(bcg_object_transition);
722
723     (*mark1) = (mark1_dummy);
724     CAESAR_PUT_TABLE_1 (caesar_t1);
725
726
727     /*----opening LTS Kernel--------------------------------------------------*/
728     BCG_INIT ();  /* one or more occurrences of BCG_INIT doesn't matter */
729
730     BCG_IO_WRITE_BCG_BEGIN (bcg_filename_output, bcg_spec_initial_state, 1,
731                             "created by kernel_algo", caesar_monitor);
732
733     /*----setting interrupts--------------------------------------------------*/
734     signal (SIGHUP, caesar_abort);
735     signal (SIGINT, caesar_abort);
736     signal (SIGQUIT, caesar_abort);
737     signal (SIGILL, caesar_abort);
738     signal (SIGABRT, caesar_abort);
739     signal (SIGFPE, caesar_abort);
740     signal (SIGBUS, caesar_abort);
741     signal (SIGSEGV, caesar_abort);
742   #ifdef SIGSYS
743     signal (SIGSYS, caesar_abort);
744   #endif
745     signal (SIGTERM, caesar_abort);
746     signal (SIGPIPE, caesar_abort);
747
748     CAESAR_CREATE_LABEL(&caesar_l_r);
749
750     /*----main loop for exploring table---------------------------------------*/
751     while ((!CAESAR_EXPLORED_TABLE_1 (caesar_t1)) ) {
752
753       /*----current size of table---------------------------------------------*/
754       get_index_table1 = CAESAR_GET_INDEX_TABLE_1 (caesar_t1);
755       put_index_table1 = CAESAR_PUT_INDEX_TABLE_1 (caesar_t1);
756
757       /*----get current state from table--------------------------------------*/
758       CAESAR_COPY_STATE(caesar_s0,
759                         (CAESAR_TYPE_STATE) CAESAR_GET_BASE_TABLE_1 (caesar_t1));
760
761       mark1 = (MY_TYPE_MARK1_BODY*) CAESAR_GET_MARK_TABLE_1 (caesar_t1);
762         /* we need the state id of the current state for BCG_IO_WRITE later */
763       found = CAESAR_SEARCH_TABLE_1(caesar_t2,
```

```
764                              (CAESAR_TYPE_POINTER) caesar_s0,
765                              &bcg_spec_state0, &pointer_dummy);
766
767       if (!found)
768         CAESAR_ERROR("At get index %d in table 1, state not found in table 2",
769                      get_index_table1);
770
771       mark1_body = (*mark1);
772       for(j=0;j<MY_VECTOR_SIZE;j++){
773         cursymmstates[j] = mark1_body.symmstates[j];
774       }
775
776       CAESAR_GET_TABLE_1 (caesar_t1);
777
778       /*----create successor list for current state---------------------------*/
779       CAESAR_CREATE_EDGE_LIST (caesar_s0, &caesar_e1_en, 1);
780
781       if (CAESAR_TRUNCATION_EDGE_LIST () != 0)
782         CAESAR_ERROR ("not enough memory for edge lists");
783
784         /* put '0' in the mark field of every edge in the list */
785       CAESAR_ITERATE_LNM_EDGE_LIST(caesar_e1_en,caesar_e1,caesar_l1,
786                                    caesar_s1,caesar_m1) {
787         edge_mark_ptr1 = (MY_TYPE_EDGE_MARK_FIELD*) caesar_m1;
788         (*edge_mark_ptr1) = '0';
789       }
790
791       /*----iterate loop for each successor-----------------------------------*/
792       CAESAR_ITERATE_LNM_EDGE_LIST(caesar_e1_en,caesar_e1,caesar_l1,
793                                      caesar_s1,caesar_m1) {
794         edge_mark_ptr1 = (MY_TYPE_EDGE_MARK_FIELD*) caesar_m1;
795
796         CAESAR_DUMP_LABEL((CAESAR_TYPE_STRING) tmp_label,caesar_l1);
797
798         /*----skip if not representative--------------------------------------*/
799         if ((*edge_mark_ptr1) !='1') {
800
801           MY_MAKE_COLOUR_LIST(cursymmstates,caesar_l1,&cl1,&newsymmstates1);
802           if (!MY_LENGTH_COLOUR_LIST(cl1))
803               CAESAR_ERROR ("Main: colour list for label %s in state %d with
804       symm states [%d,%d,%d,%d,%d,%d] is empty!\n",
805                             tmp_label, bcg_spec_state0,
806                             cursymmstates[0], cursymmstates[1], cursymmstates[2],
807                             cursymmstates[3], cursymmstates[4], cursymmstates[5]);
808
809           /*----if representative not yet known------------------------------*/
810           if ((*edge_mark_ptr1) == '0'){ /* then we need to find repr */
811
812             /* Now walk through the rest of the edge list and change the
813                explore flag for each successor state for which the transition
814                is symmetric to the transition from caesar_s0 to caesar_s1.
```

```
815                 If there is one with lower ranking, then current one will
816                 *not* be explored!
817                 This ensures that symmetric successors are not explored.
818                 Starting point in the edge list is the successor of edge
819                 caesar_e1, the current elt */
820
821               CAESAR_COPY_LABEL(caesar_l_r,caesar_l1);
822               edge_mark_ptr_r = edge_mark_ptr1;
823               caesar_e_ptr_next =  CAESAR_SUCCESSOR_EDGE(caesar_e1);
824               if (caesar_e_ptr_next != NULL){
825
826                 CAESAR_ITERATE_LM_EDGE_LIST((*caesar_e_ptr_next),
827                                             caesar_e2,caesar_l2,caesar_m2){
828                   edge_mark_ptr2 = (MY_TYPE_EDGE_MARK_FIELD*) caesar_m2;
829                   if ((*edge_mark_ptr2) == '0'){
830
831                     MY_MAKE_COLOUR_LIST(cursymmstates,caesar_l2,&cl2,&newsymmstates2);
832
833                     if (MY_SYMMETRIC_COLOUR_LIST(cl1,&cl2)){
834
835                       if (MY_RANK_LABEL(caesar_l2)<MY_RANK_LABEL(caesar_l_r)){
836                         (*edge_mark_ptr_r) = '1';
837                         CAESAR_COPY_LABEL(caesar_l_r,caesar_l2);
838                         edge_mark_ptr_r = edge_mark_ptr2;
839                       } else
840                         (*edge_mark_ptr2) = '1'; /* this edge should not be explored later! */
841                     }   /* end if */
842                     MY_DELETE_COLOUR_LIST(&cl2);
843                   }   /* end if */
844                 }   /* end CAESAR_ITERATE */
845               } /* end if */
846               (*edge_mark_ptr_r) = '2';
847
848             } /* end if */
849
850               /* Even if it's empty, deleting this list is safe (i hope) */
851             MY_DELETE_COLOUR_LIST(&cl1);
852
853             /*----explore representative----------------------------------------*/
854             if((*edge_mark_ptr1) =='2'){  /* only explore repr */
855
856               /* we need the state id of the succ state: search caesar_t2
857                  if this is first visit, initialise mark field */
858             found = CAESAR_SEARCH_TABLE_1 (caesar_t2,
859                                           (CAESAR_TYPE_POINTER) caesar_s1,
860                                              &bcg_spec_state1,
861                                            &caesar_dummy);
862
863             if(!found){
864               CAESAR_COPY_STATE ((CAESAR_TYPE_STATE)
865                                  CAESAR_PUT_BASE_TABLE_1(caesar_t2),
```

```
866                              caesar_s1);
867             mark2 = (MY_TYPE_MARK2_BODY *) CAESAR_PUT_MARK_TABLE_1(caesar_t2);
868             mark2->howmany = 0;
869             bcg_spec_state1 = CAESAR_PUT_INDEX_TABLE_1(caesar_t2);
870             CAESAR_PUT_TABLE_1 (caesar_t2);
871           }
872
873           /*----decide if current successor was Seen--------------------------*/
874             /* only store succ state in table 1 if this is first visit
875                on the current path from initial state */
876             /* walk back via the path that led to caesar_s1 to the
877                initial state or until caesar_s0 is encountered on the way */
878           found = 0;
879           prev = 1;  /* the prev is the state we just came from */
880           prev_index = get_index_table1; /* and it's index is get index from t1 */
881
882           while ((prev!=0) && (found == 0)){
883             CAESAR_RETRIEVE_I_B_TABLE_1(caesar_t1,
884                                         prev_index,
885                                         (CAESAR_TYPE_POINTER *) &prev_state);
886             if(CAESAR_COMPARE_STATE(caesar_s1,prev_state))
887                found = 1;
888             else {
889               CAESAR_RETRIEVE_I_M_TABLE_1(caesar_t1,
890                                           prev_index,
891                                           (CAESAR_TYPE_POINTER *) &prev_mark1);
892               mark1_body_prev = (MY_TYPE_MARK1_BODY) (*prev_mark1);
893               prev = mark1_body_prev.prev;
894               prev_index = mark1_body_prev.prev_index;
895             }
896           }
897
898           /*----skip if Seen or if exact state already in caesar_t1-----------*/
899           if (!found){ /* first time caesar_s1 is visited on this path */
900             /* we are going to try to find the same state with the same
901                symmstates in table 1, because such a combination just
902                needs to be explored once, not more often! */
903             put_index_table1 = CAESAR_PUT_INDEX_TABLE_1(caesar_t1);
904             for (run=0; ((run<put_index_table1) && (!found)); run++){
905               CAESAR_RETRIEVE_I_BM_TABLE_1(caesar_t1,
906                                           run,
907                                           (CAESAR_TYPE_POINTER *) &tmp_state,
908                                           (CAESAR_TYPE_POINTER *) &tmp_mark1);
909               if(CAESAR_COMPARE_STATE(caesar_s1,tmp_state)){
910                 tmp_found = 1;
911                 for (j=0; ((j<MY_VECTOR_SIZE) && tmp_found); j++)
912                   tmp_found = (tmp_mark1->symmstates[j] == newsymmstates1[j]);
913                 found = tmp_found;
914               }
915             }
916           }
```

```
917
918
919         caesar_label = CAESAR_STRING_LABEL (caesar_l1);
920
921         /*----store in caesar_t1 to be explored later-----------------------*/
922         if (!found){ /* first time caesar_s1 is visited on this path */
923
924             /* prepare base field values for storing the new state in table 1 */
925           put_index_table1 = CAESAR_PUT_INDEX_TABLE_1(caesar_t1);
926           CAESAR_COPY_STATE((CAESAR_TYPE_STATE)
927                                CAESAR_PUT_BASE_TABLE_1(caesar_t1),
928                              caesar_s1);
929             /* prepare mark field to be put with new values for symmstates
930                from colouring */
931           mark1_p = (MY_TYPE_MARK1_BODY*) CAESAR_PUT_MARK_TABLE_1(caesar_t1);
932           mark1_dummy.prev = 1;
933           mark1_dummy.prev_index = get_index_table1;
934           for (j=0; j<MY_VECTOR_SIZE; j++)
935             mark1_dummy.symmstates[j] = newsymmstates1[j];
936           (*mark1_p) = mark1_dummy;
937             /* finally, store new elt in table 1 */
938           CAESAR_PUT_TABLE_1 (caesar_t1);
939         } else {
940         }   /* end if */
941
942         /*----store also in caesar_t2---------------------------------------*/
943           /* explored edges must be added to table 2 and LTS Kernel */
944           /* if they're not already in table 2 */
945         found = 0;
946         CAESAR_RETRIEVE_I_M_TABLE_1(caesar_t2,
947                                     (CAESAR_TYPE_INDEX_TABLE_1) bcg_spec_state0,
948                                     (CAESAR_TYPE_POINTER *) &mark2_1);
949
950         for(run=0; ((run<mark2_1->howmany) && (!found)); run++){
951
952           found = ((mark2_1->trans[run].succ == bcg_spec_state1)
953                    && (strcmp(mark2_1->trans[run].label,caesar_label) == 0));
954         }
955
956         /*----write to LTS Result-------------------------------------------*/
957         if (!found){
958           run = mark2_1->howmany;
959           mark2_1->trans[run].succ =bcg_spec_state1;
960           strcpy(mark2_1->trans[run].label,caesar_label);
961           mark2_1->howmany++;
962           BCG_IO_WRITE_BCG_EDGE(bcg_spec_state0, caesar_label, bcg_spec_state1);
963         }
964
965         } /* end if */
966       } else {
967       }   /* end if */
```

```
968      }   /* end CAESAR_ITERATE */
969
970      /*----finished iterating successors------------------------------------*/
971      CAESAR_DELETE_EDGE_LIST (&caesar_e1_en);
972    }   /* end while */
973
974    /*----finished exploring table--------------------------------------------*/
975
976  printf("Done!!\n");
977    BCG_IO_WRITE_BCG_END ();
978    exit (0);
979  }   /* end main */
```

B Cyclic train code listings

μCRL model for 8 stations

```
1    sort    Bool
2    func    T,F : -> Bool
3    map     not : Bool -> Bool
4            and : Bool # Bool -> Bool
5            or : Bool # Bool -> Bool
6            if : Bool # Bool # Bool -> Bool
7    var     b, b1, b2: Bool
8    rew     not(T) = F
9            not(F) = T
10           and(F,b) = F
11           and(b,F) = F
12           and(T,b) = b
13           and(b,T) = b
14           or(T,b) = T
15           or(b,T) = T
16           or(F,b) = b
17           or(b,F) = b
18           if(T,b1,b2) = b1
19           if(F,b1,b2) = b2
20
21   sort    Station
22   func    0,1,2,3,4,5,6,7 : -> Station
23   map     eq : Station # Station -> Bool
24           left : Station -> Station
25           right : Station -> Station
26           left : Station # Station -> Bool
27           right : Station # Station -> Bool
28   var     s1, s2 : Station
29   rew     left(0) = 7
30           left(1) = 0
31           left(2) = 1
32           left(3) = 2
33           left(4) = 3
```

```
34          left(5) = 4
35          left(6) = 5
36          left(7) = 6
37          right(0) = 1
38          right(1) = 2
39          right(2) = 3
40          right(3) = 4
41          right(4) = 5
42          right(5) = 6
43          right(6) = 7
44          right(7) = 0
45          eq(0,0) = T
46          eq(0,1) = F
47          eq(0,2) = F
48          eq(0,3) = F
49          eq(0,4) = F
50          eq(0,5) = F
51          eq(0,6) = F
52          eq(0,7) = F
53          eq(1,0) = F
54          eq(1,1) = T
55          eq(1,2) = F
56          eq(1,3) = F
57          eq(1,4) = F
58          eq(1,5) = F
59          eq(1,6) = F
60          eq(1,7) = F
61          eq(2,0) = F
62          eq(2,1) = F
63          eq(2,2) = T
64          eq(2,3) = F
65          eq(2,4) = F
66          eq(2,5) = F
67          eq(2,6) = F
68          eq(2,7) = F
69          eq(3,0) = F
70          eq(3,1) = F
71          eq(3,2) = F
72          eq(3,3) = T
73          eq(3,4) = F
74          eq(3,5) = F
75          eq(3,6) = F
76          eq(3,7) = F
77          eq(4,0) = F
78          eq(4,1) = F
79          eq(4,2) = F
80          eq(4,3) = F
81          eq(4,4) = T
82          eq(4,5) = F
83          eq(4,6) = F
84          eq(4,7) = F
```

```
85          eq(5,0) = F
86          eq(5,1) = F
87          eq(5,2) = F
88          eq(5,3) = F
89          eq(5,4) = F
90          eq(5,5) = T
91          eq(5,6) = F
92          eq(5,7) = F
93          eq(6,0) = F
94          eq(6,1) = F
95          eq(6,2) = F
96          eq(6,3) = F
97          eq(6,4) = F
98          eq(6,5) = F
99          eq(6,6) = T
100         eq(6,7) = F
101         eq(7,0) = F
102         eq(7,1) = F
103         eq(7,2) = F
104         eq(7,3) = F
105         eq(7,4) = F
106         eq(7,5) = F
107         eq(7,6) = F
108         eq(7,7) = T
109         left(0,0) = F
110         left(0,1) = T
111         left(0,2) = T
112         left(0,3) = T
113         left(0,4) = T
114         left(0,5) = F
115         left(0,6) = F
116         left(0,7) = F
117         left(1,0) = F
118         left(1,1) = F
119         left(1,2) = T
120         left(1,3) = T
121         left(1,4) = T
122         left(1,5) = T
123         left(1,6) = F
124         left(1,7) = F
125         left(2,0) = F
126         left(2,1) = F
127         left(2,2) = F
128         left(2,3) = T
129         left(2,4) = T
130         left(2,5) = T
131         left(2,6) = T
132         left(2,7) = F
133         left(3,0) = F
134         left(3,1) = F
135         left(3,2) = F
```

```
136          left(3,3) = F
137          left(3,4) = T
138          left(3,5) = T
139          left(3,6) = T
140          left(3,7) = T
141          left(4,0) = T
142          left(4,1) = F
143          left(4,2) = F
144          left(4,3) = F
145          left(4,4) = F
146          left(4,5) = T
147          left(4,6) = T
148          left(4,7) = T
149          left(5,0) = T
150          left(5,1) = T
151          left(5,2) = F
152          left(5,3) = F
153          left(5,4) = F
154          left(5,5) = F
155          left(5,6) = T
156          left(5,7) = T
157          left(6,0) = T
158          left(6,1) = T
159          left(6,2) = T
160          left(6,3) = F
161          left(6,4) = F
162          left(6,5) = F
163          left(6,6) = F
164          left(6,7) = T
165          left(7,0) = T
166          left(7,1) = T
167          left(7,2) = T
168          left(7,3) = T
169          left(7,4) = F
170          left(7,5) = F
171          left(7,6) = F
172          left(7,7) = F
173          right(s1,s2) = left(s2,s1)
174
175  sort    ReqType
176  func    CALL, SEND : -> ReqType
177
178  sort    ReqList
179  func    er : -> ReqList
180          rl : Station # Bool # Bool # ReqList -> ReqList
181  map     if : Bool # ReqList # ReqList -> ReqList
182          isreq : Station # ReqList -> Bool
183          isleftreq : Station # ReqList -> Bool
184          isrightreq : Station # ReqList -> Bool
185          change : ReqType # Station # Bool # ReqList -> ReqList
186          changeF : Station # ReqList -> ReqList
```

```
187 var     l,l1,l2 : ReqList
188         s,s1,s2 : Station
189         b,b1,b2 : Bool
190         r : ReqType
191 rew     if(T,l1,l2) = l1
192         if(F,l1,l2) = l2
193         isreq(s,er) = F
194         isreq(s1,rl(s2,b1,b2,l)) = if(eq(s1,s2),or(b1,b2),isreq(s1,l))
195         isleftreq(s,er) = F
196         isleftreq(s1,rl(s2,b1,b2,l))
197           = if(left(s2,s1),or(b1,b2),isleftreq(s1,l))
198         isrightreq(s,er) = F
199         isrightreq(s1,rl(s2,b1,b2,l))
200           = if(right(s2,s1),or(b1,b2),isrightreq(s1,l))
201         change(r,s,b,er) = er
202         change(CALL,s1,b,rl(s2,b1,b2,l))
203           = if(eq(s1,s2),rl(s2,b,b2,l),rl(s2,b1,b2,change(CALL,s1,b,l)))
204         change(SEND,s1,b,rl(s2,b1,b2,l))
205           = if(eq(s1,s2),rl(s2,b1,b,l),rl(s2,b1,b2,change(SEND,s1,b,l)))
206         changeF(s,er) = er
207         changeF(s1,rl(s2,b1,b2,l))
208           = if(eq(s1,s2),rl(s2,F,F,l),rl(s2,b1,b2,changeF(s1,l)))
209
210 sort    Direction
211 func    right, left, none : -> Direction
212 map     eq : Direction # Direction -> Bool
213         if : Bool # Direction # Direction -> Direction
214 var     d, d1, d2 : Direction
215 rew     eq(right,right) = T
216         eq(right,left) = F
217         eq(right,none) = F
218         eq(left,right) = F
219         eq(left,left) = T
220         eq(left,none) = F
221         eq(none,right) = F
222         eq(none,left) = F
223         eq(none,none) = T
224         if(T,d1,d2) = d1
225         if(F,d1,d2) = d2
226
227 act     REQUEST : ReqType # Station
228         OPENDOOR, CLOSEDOOR, MOVERIGHT, MOVELEFT: Station
229
230 proc    CyclicTrain(stat:Station, dooropen:Bool, reqs:ReqList, dir:Direction)
231         =
232            sum(r:ReqType,
233                sum(s:Station,
234                    REQUEST(r,s)
235                    . CyclicTrain(stat,
236                                  dooropen,
237                                  if(not(and(eq(s,stat),dooropen)),
```

```
238                                 change(r,s,T,reqs),
239                                 reqs),
240                              if(eq(dir,none),
241                                 if(left(s,stat),
242                                    left,
243                                    if(right(s,stat),right,dir)),
244                                 dir))))
245          +
246            OPENDOOR(stat) . CyclicTrain(stat,T,changeF(stat,reqs),dir)
247               <| and(not(dooropen),isreq(stat,reqs)) |>
248            delta
249          +
250            CLOSEDOOR(stat) . CyclicTrain(stat,F,reqs,dir)
251               <| dooropen |>
252            delta
253          +
254            MOVERIGHT(right(stat))
255            . CyclicTrain(right(stat),dooropen, reqs,
256                          if(isrightreq(right(stat),reqs),
257                             dir,
258                             if(isleftreq(right(stat),reqs),left,dir)))
259               <| and(not(dooropen),and(eq(dir,right),not(isreq(stat,reqs)))) |>
260            delta
261          +
262            MOVELEFT(left(stat))
263            . CyclicTrain(left(stat),dooropen, reqs,
264                          if(isleftreq(left(stat),reqs),
265                             dir,
266                             if(isrightreq(left(stat),reqs),right,dir)))
267               <| and(not(dooropen),and(eq(dir,left),not(isreq(stat,reqs)))) |>
268            delta
269
270
271 init    CyclicTrain(0,
272                     F,
273                     rl(0,F,F,rl(1,F,F,rl(2,F,F,rl(3,F,F,rl(4,F,F,rl
274                       (5,F,F,rl(6,F,F,rl(7,F,F,er)))))))),
275                     none)
276
```

State space of symmetry for station 1

```
1     des (0, 20, 5)
2     (0, "1!REQUEST(CALL,1)", 1)
3     (0, "2!REQUEST(SEND,1)", 1)
4     (0, "3!MOVELEFT(1)", 2)
5     (0, "4!MOVERIGHT(1)", 2)
6     (1, "5!REQUEST(CALL,1)", 1)
7     (1, "6!REQUEST(SEND,1)", 1)
8     (1, "7!MOVELEFT(1)", 3)
9     (1, "8!MOVERIGHT(1)", 3)
```

```
10   (2, "9!REQUEST(CALL,1)", 3)
11   (2, "10!REQUEST(SEND,1)", 3)
12   (2, "11!MOVELEFT(0)", 0)
13   (2, "12!MOVERIGHT(2)", 0)
14   (3, "13!REQUEST(CALL,1)", 3)
15   (3, "14!REQUEST(SEND,1)", 3)
16   (3, "15!MOVELEFT(0)", 1)
17   (3, "16!MOVERIGHT(2)", 1)
18   (3, "17!OPENDOOR(1)", 4)
19   (4, "18!REQUEST(CALL,1)", 4)
20   (4, "19!REQUEST(SEND,1)", 4)
21   (4, "20!CLOSEDOOR(1)", 2)
```

Mealy-style μCRL model for 8 stations

```
1    sort   Bool
2    func   T,F : -> Bool
3    map    not : Bool -> Bool
4           and : Bool # Bool -> Bool
5           or : Bool # Bool -> Bool
6           if : Bool # Bool # Bool -> Bool
7    var    b, b1, b2: Bool
8    rew    not(T) = F
9           not(F) = T
10          and(F,b) = F
11          and(b,F) = F
12          and(T,b) = b
13          and(b,T) = b
14          or(T,b) = T
15          or(b,T) = T
16          or(F,b) = b
17          or(b,F) = b
18          if(T,b1,b2) = b1
19          if(F,b1,b2) = b2
20
21   sort   Station
22   func   0,1,2,3,4,5,6,7 : -> Station
23   map    eq : Station # Station -> Bool
24          left : Station -> Station
25          right : Station -> Station
26          left : Station # Station -> Bool
27          right : Station # Station -> Bool
28   var    s1, s2 : Station
29   rew    left(0) = 7
30          left(1) = 0
31          left(2) = 1
32          left(3) = 2
33          left(4) = 3
34          left(5) = 4
35          left(6) = 5
36          left(7) = 6
```

```
37          right(0) = 1
38          right(1) = 2
39          right(2) = 3
40          right(3) = 4
41          right(4) = 5
42          right(5) = 6
43          right(6) = 7
44          right(7) = 0
45          eq(0,0) = T
46          eq(0,1) = F
47          eq(0,2) = F
48          eq(0,3) = F
49          eq(0,4) = F
50          eq(0,5) = F
51          eq(0,6) = F
52          eq(0,7) = F
53          eq(1,0) = F
54          eq(1,1) = T
55          eq(1,2) = F
56          eq(1,3) = F
57          eq(1,4) = F
58          eq(1,5) = F
59          eq(1,6) = F
60          eq(1,7) = F
61          eq(2,0) = F
62          eq(2,1) = F
63          eq(2,2) = T
64          eq(2,3) = F
65          eq(2,4) = F
66          eq(2,5) = F
67          eq(2,6) = F
68          eq(2,7) = F
69          eq(3,0) = F
70          eq(3,1) = F
71          eq(3,2) = F
72          eq(3,3) = T
73          eq(3,4) = F
74          eq(3,5) = F
75          eq(3,6) = F
76          eq(3,7) = F
77          eq(4,0) = F
78          eq(4,1) = F
79          eq(4,2) = F
80          eq(4,3) = F
81          eq(4,4) = T
82          eq(4,5) = F
83          eq(4,6) = F
84          eq(4,7) = F
85          eq(5,0) = F
86          eq(5,1) = F
87          eq(5,2) = F
```

```
88              eq(5,3) = F
89              eq(5,4) = F
90              eq(5,5) = T
91              eq(5,6) = F
92              eq(5,7) = F
93              eq(6,0) = F
94              eq(6,1) = F
95              eq(6,2) = F
96              eq(6,3) = F
97              eq(6,4) = F
98              eq(6,5) = F
99              eq(6,6) = T
100             eq(6,7) = F
101             eq(7,0) = F
102             eq(7,1) = F
103             eq(7,2) = F
104             eq(7,3) = F
105             eq(7,4) = F
106             eq(7,5) = F
107             eq(7,6) = F
108             eq(7,7) = T
109             left(0,0) = F
110             left(0,1) = T
111             left(0,2) = T
112             left(0,3) = T
113             left(0,4) = T
114             left(0,5) = F
115             left(0,6) = F
116             left(0,7) = F
117             left(1,0) = F
118             left(1,1) = F
119             left(1,2) = T
120             left(1,3) = T
121             left(1,4) = T
122             left(1,5) = T
123             left(1,6) = F
124             left(1,7) = F
125             left(2,0) = F
126             left(2,1) = F
127             left(2,2) = F
128             left(2,3) = T
129             left(2,4) = T
130             left(2,5) = T
131             left(2,6) = T
132             left(2,7) = F
133             left(3,0) = F
134             left(3,1) = F
135             left(3,2) = F
136             left(3,3) = F
137             left(3,4) = T
138             left(3,5) = T
```

```
139         left(3,6) = T
140         left(3,7) = T
141         left(4,0) = T
142         left(4,1) = F
143         left(4,2) = F
144         left(4,3) = F
145         left(4,4) = F
146         left(4,5) = T
147         left(4,6) = T
148         left(4,7) = T
149         left(5,0) = T
150         left(5,1) = T
151         left(5,2) = F
152         left(5,3) = F
153         left(5,4) = F
154         left(5,5) = F
155         left(5,6) = T
156         left(5,7) = T
157         left(6,0) = T
158         left(6,1) = T
159         left(6,2) = T
160         left(6,3) = F
161         left(6,4) = F
162         left(6,5) = F
163         left(6,6) = F
164         left(6,7) = T
165         left(7,0) = T
166         left(7,1) = T
167         left(7,2) = T
168         left(7,3) = T
169         left(7,4) = F
170         left(7,5) = F
171         left(7,6) = F
172         left(7,7) = F
173         right(s1,s2) = left(s2,s1)
174
175   sort  ReqType
176   func  CALL, SEND : -> ReqType
177
178   sort  ReqList
179   func  er : -> ReqList
180         rl : Station # Bool # Bool # ReqList -> ReqList
181   map   if : Bool # ReqList # ReqList -> ReqList
182         isreq : Station # ReqList -> Bool
183         isleftreq : Station # ReqList -> Bool
184         isrightreq : Station # ReqList -> Bool
185         change : ReqType # Station # Bool # ReqList -> ReqList
186         changeF : Station # ReqList -> ReqList
187   var   l,l1,l2 : ReqList
188         s,s1,s2 : Station
189         b,b1,b2 : Bool
```

```
190          r : ReqType
191 rew      if(T,l1,l2) = l1
192          if(F,l1,l2) = l2
193          isreq(s,er) = F
194          isreq(s1,rl(s2,b1,b2,l)) = if(eq(s1,s2),or(b1,b2),isreq(s1,l))
195          isleftreq(s,er) = F
196          isleftreq(s1,rl(s2,b1,b2,l))
197            = or(and(left(s2,s1),or(b1,b2)),isleftreq(s1,l))
198          isrightreq(s,er) = F
199          isrightreq(s1,rl(s2,b1,b2,l))
200            = or(and(right(s2,s1),or(b1,b2)),isrightreq(s1,l))
201          change(r,s,b,er) = er
202          change(CALL,s1,b,rl(s2,b1,b2,l))
203            = if(eq(s1,s2),rl(s2,b,b2,l),rl(s2,b1,b2,change(CALL,s1,b,l)))
204          change(SEND,s1,b,rl(s2,b1,b2,l))
205            = if(eq(s1,s2),rl(s2,b1,b,l),rl(s2,b1,b2,change(SEND,s1,b,l)))
206          changeF(s,er) = er
207          changeF(s1,rl(s2,b1,b2,l))
208            = if(eq(s1,s2),rl(s2,F,F,l),rl(s2,b1,b2,changeF(s1,l)))
209
210 sort     Direction
211 func     right, left, none : -> Direction
212 map      eq : Direction # Direction -> Bool
213          if : Bool # Direction # Direction -> Direction
214 var      d, d1, d2 : Direction
215 rew      eq(right,right) = T
216          eq(right,left) = F
217          eq(right,none) = F
218          eq(left,right) = F
219          eq(left,left) = T
220          eq(left,none) = F
221          eq(none,right) = F
222          eq(none,left) = F
223          eq(none,none) = T
224          if(T,d1,d2) = d1
225          if(F,d1,d2) = d2
226
227 sort     Output
228 func     OPENDOOR, CLOSEDOOR, MOVERIGHT, MOVELEFT, DOORCLOSED : Station -> Output
229          NO_OUTPUT: -> Output
230 act      REQUEST : ReqType # Station # Output
231          WAIT : Output
232
233 proc     CyclicTrain(stat:Station, dooropen:Bool, closing: Bool,
234                      reqs:ReqList, dir:Direction)
235          =
236             sum(r:ReqType,
237                 sum(s:Station,
238                       REQUEST(r,s,NO_OUTPUT)
239                       . CyclicTrain(stat, dooropen, closing,
240                                     if(not(and(eq(s,stat),dooropen)),
```

```
241                                         change(r,s,T,reqs),
242                                         reqs),
243                                      if(eq(dir,none),
244                                         if(left(s,stat),
245                                            left,
246                                            if(right(s,stat),right,dir)),
247                                         dir))
248                        <| or(dooropen,
249                              and(or(not(closing),not(eq(s,stat))),
250                                  or(closing,
251                                     or(not(eq(dir,none)),isreq(stat,reqs))))) |>
252                         delta
253                       +
254                         REQUEST(r,s,OPENDOOR(stat))
255                         . CyclicTrain(stat, T, F, reqs, dir)
256                        <| and(not(dooropen),and(eq(dir,none),
257                           and(eq(s,stat),not(isreq(stat,reqs))))) |>
258                         delta
259                       +
260                         REQUEST(r,s,MOVELEFT(left(stat)))
261                         . CyclicTrain(left(stat), dooropen, F, change(r,s,T,reqs),
262                                       if(eq(s,left(stat)),none,left))
263                         <| and(not(dooropen),and(not(closing),and(eq(dir,none),
264                            and(left(s,stat),not(isreq(stat,reqs)))))) |>
265                         delta
266                       +
267                         REQUEST(r,s,MOVERIGHT(right(stat)))
268                         . CyclicTrain(right(stat), dooropen, F, change(r,s,T,reqs),
269                                       if(eq(s,right(stat)),none,right))
270                         <| and(not(dooropen),and(not(closing),and(eq(dir,none),
271                            and(right(s,stat),and(not(left(s,stat)),
272                                not(isreq(stat,reqs))))))) |>
273                         delta))
274              +
275                WAIT(NO_OUTPUT) . CyclicTrain(stat,dooropen,closing,reqs,dir)
276                   <| and(not(dooropen),and(not(closing),
277                      and(eq(dir,none),not(isreq(stat,reqs))))) |>
278                delta
279              +
280                WAIT(DOORCLOSED(stat)) . CyclicTrain(stat,dooropen,F,reqs,dir)
281                   <| and(closing,and(eq(dir,none),not(isreq(stat,reqs)))) |>
282                delta
283              +
284                WAIT(OPENDOOR(stat)) . CyclicTrain(stat,T,F,changeF(stat,reqs),dir)
285                   <| and(not(dooropen),isreq(stat,reqs)) |>
286                delta
287              +
288                WAIT(CLOSEDOOR(stat)) . CyclicTrain(stat,F,T,reqs,dir)
289                   <| dooropen |>
290                delta
291              +
```

```
292            WAIT(MOVERIGHT(right(stat)))
293            . CyclicTrain(right(stat), dooropen, F, reqs,
294                          if(isrightreq(right(stat),reqs),
295                             dir,
296                             if(isleftreq(right(stat),reqs),left,none)))
297               <| and(not(dooropen),and(eq(dir,right),not(isreq(stat,reqs)))) |>
298            delta
299          +
300            WAIT(MOVELEFT(left(stat)))
301            . CyclicTrain(left(stat), dooropen, F, reqs,
302                          if(isleftreq(left(stat),reqs),
303                             dir,
304                             if(isrightreq(left(stat),reqs),right,none)))
305               <| and(not(dooropen),and(eq(dir,left),not(isreq(stat,reqs)))) |>
306            delta
307
308
309  init    CyclicTrain(0,
310                     F,
311                     F,
312                     rl(0,F,F,rl(1,F,F,rl(2,F,F,rl(3,F,F,rl(4,F,F,rl
313                       (5,F,F,rl(6,F,F,rl(7,F,F,er)))))))),
314                     none)
315
```

State space of mealy-style symmetry for station 1

```
1    des (0, 62, 6)
2    (0, "1!WAIT(NO_OUTPUT)", 0)
3    (0, "2!WAIT(MOVELEFT(1))", 5)
4    (0, "3!WAIT(MOVERIGHT(1))", 5)
5    (0, "4!REQUEST(CALL,1,NO_OUTPUT)", 1)
6    (0, "5!REQUEST(SEND,1,NO_OUTPUT)", 1)
7    (0, "6!REQUEST(CALL,1,MOVELEFT(1))", 2)
8    (0, "7!REQUEST(SEND,1,MOVELEFT(1))", 2)
9    (0, "8!REQUEST(CALL,1,MOVERIGHT(1))", 2)
10   (0, "9!REQUEST(SEND,1,MOVERIGHT(1))", 2)
11   (0, "10!REQUEST(CALL,1,MOVELEFT(2))", 1)
12   (0, "11!REQUEST(SEND,1,MOVELEFT(2))", 1)
13   (0, "12!REQUEST(CALL,1,MOVELEFT(3))", 1)
14   (0, "13!REQUEST(SEND,1,MOVELEFT(3))", 1)
15   (0, "14!REQUEST(CALL,1,MOVELEFT(4))", 1)
16   (0, "15!REQUEST(SEND,1,MOVELEFT(4))", 1)
17   (0, "16!REQUEST(CALL,1,MOVERIGHT(0))", 1)
18   (0, "17!REQUEST(SEND,1,MOVERIGHT(0))", 1)
19   (0, "18!REQUEST(CALL,1,MOVERIGHT(7))", 1)
20   (0, "19!REQUEST(SEND,1,MOVERIGHT(7))", 1)
21   (0, "20!REQUEST(CALL,2,MOVERIGHT(1))", 5)
22   (0, "21!REQUEST(SEND,2,MOVERIGHT(1))", 5)
23   (0, "22!REQUEST(CALL,3,MOVERIGHT(1))", 5)
24   (0, "23!REQUEST(SEND,3,MOVERIGHT(1))", 5)
```

```
25   (0, "24!REQUEST(CALL,0,MOVELEFT(1))", 5)
26   (0, "25!REQUEST(SEND,0,MOVELEFT(1))", 5)
27   (0, "26!REQUEST(CALL,6,MOVELEFT(1))", 5)
28   (0, "27!REQUEST(SEND,6,MOVELEFT(1))", 5)
29   (0, "28!REQUEST(CALL,7,MOVELEFT(1))", 5)
30   (0, "29!REQUEST(SEND,7,MOVELEFT(1))", 5)
31   (1, "30!WAIT(MOVELEFT(1))", 2)
32   (1, "31!WAIT(MOVERIGHT(1))", 2)
33   (1, "32!REQUEST(CALL,1,NO_OUTPUT)", 1)
34   (1, "33!REQUEST(SEND,1,NO_OUTPUT)", 1)
35   (2, "34!WAIT(OPENDOOR(1))", 3)
36   (2, "35!REQUEST(CALL,1,NO_OUTPUT)", 2)
37   (2, "36!REQUEST(SEND,1,NO_OUTPUT)", 2)
38   (3, "37!WAIT(CLOSEDOOR(1))", 4)
39   (3, "38!REQUEST(CALL,1,NO_OUTPUT)", 3)
40   (3, "39!REQUEST(SEND,1,NO_OUTPUT)", 3)
41   (4, "40!WAIT(DOORCLOSED(1))", 4)
42   (4, "41!WAIT(MOVELEFT(0))", 0)
43   (4, "42!WAIT(MOVERIGHT(2))", 0)
44   (4, "43!REQUEST(CALL,1,OPENDOOR(1))", 3)
45   (4, "44!REQUEST(SEND,1,OPENDOOR(1))", 3)
46   (4, "45!REQUEST(CALL,2,MOVERIGHT(2))", 0)
47   (4, "46!REQUEST(SEND,2,MOVERIGHT(2))", 0)
48   (4, "47!REQUEST(CALL,3,MOVERIGHT(2))", 0)
49   (4, "48!REQUEST(SEND,3,MOVERIGHT(2))", 0)
50   (4, "49!REQUEST(CALL,4,MOVERIGHT(2))", 0)
51   (4, "50!REQUEST(SEND,4,MOVERIGHT(2))", 0)
52   (4, "51!REQUEST(CALL,0,MOVELEFT(0))", 0)
53   (4, "52!REQUEST(SEND,0,MOVELEFT(0))", 0)
54   (4, "53!REQUEST(CALL,5,MOVELEFT(0))", 0)
55   (4, "54!REQUEST(SEND,5,MOVELEFT(0))", 0)
56   (4, "55!REQUEST(CALL,6,MOVELEFT(0))", 0)
57   (4, "56!REQUEST(SEND,6,MOVELEFT(0))", 0)
58   (4, "57!REQUEST(CALL,7,MOVELEFT(0))", 0)
59   (4, "58!REQUEST(SEND,7,MOVELEFT(0))", 0)
60   (5, "59!WAIT(MOVELEFT(0))", 0)
61   (5, "60!WAIT(MOVERIGHT(2))", 0)
62   (5, "61!REQUEST(CALL,1,NO_OUTPUT)", 2)
63   (5, "62!REQUEST(SEND,1,NO_OUTPUT)", 2)
```

C Ring leader election code listings

μCRL model for 3 stations

```
1    sort    Bool
2    func    T,F : -> Bool
3    map     not : Bool -> Bool
4            and : Bool # Bool -> Bool
5            or : Bool # Bool -> Bool
6            eq : Bool # Bool -> Bool
```

```
7           neq : Bool # Bool -> Bool
8           if : Bool # Bool # Bool -> Bool
9    var    b, b1, b2: Bool
10   rew    not(T) = F
11          not(F) = T
12          and(F,b) = F
13          and(b,F) = F
14          and(T,b) = b
15          and(b,T) = b
16          or(T,b) = T
17          or(b,T) = T
18          or(F,b) = b
19          or(b,F) = b
20          eq(F,F) = T
21          eq(F,T) = F
22          eq(T,F) = F
23          eq(T,T) = T
24          neq(F,F) = F
25          neq(F,T) = T
26          neq(T,F) = T
27          neq(T,T) = F
28          if(T,b1,b2) = b1
29          if(F,b1,b2) = b2
30
31   sort   Address
32   func   0,1,2 : -> Address
33   map    eq : Address # Address -> Bool
34          pred : Address -> Address
35          succ : Address -> Address
36          less : Address # Address -> Bool
37          greater : Address # Address -> Bool
38   var    a1, a2 : Address
39   rew    pred(0) = 2
40          pred(1) = 0
41          pred(2) = 1
42          succ(0) = 1
43          succ(1) = 2
44          succ(2) = 0
45          eq(0,0) = T
46          eq(0,1) = F
47          eq(0,2) = F
48          eq(1,0) = F
49          eq(1,1) = T
50          eq(1,2) = F
51          eq(2,0) = F
52          eq(2,1) = F
53          eq(2,2) = T
54          less(0,0) = F
55          less(0,1) = T
56          less(0,2) = T
57          less(1,0) = F
```

```
58          less(1,1) = F
59          less(1,2) = T
60          less(2,0) = F
61          less(2,1) = F
62          less(2,2) = F
63          greater(a1,a2) = less(a2,a1)
64
65    act   REC_CLAIM, STAT_REC_CLAIM, LINK_REC_CLAIM : Address # Address # Bool
66          SND_CLAIM, STAT_SND_CLAIM, LINK_SND_CLAIM : Address # Address # Bool
67          REC_TOKEN, STAT_REC_TOKEN, LINK_REC_TOKEN : Address
68          SND_TOKEN, STAT_SND_TOKEN, LINK_SND_TOKEN : Address
69          CRASH, OPEN, CLOSE : Address
70
71    comm  LINK_REC_CLAIM | STAT_REC_CLAIM = REC_CLAIM
72          LINK_SND_CLAIM | STAT_SND_CLAIM = SND_CLAIM
73          LINK_REC_TOKEN | STAT_REC_TOKEN = REC_TOKEN
74          LINK_SND_TOKEN | STAT_SND_TOKEN = SND_TOKEN
75
76
77    proc  Station(my_add:Address)
78          = Election(my_add, T)
79
80          Election(my_add:Address, my_round:Bool)
81          =
82             CRASH(my_add)
83             . Fail(my_add)
84           +
85             STAT_SND_CLAIM(succ(my_add),my_add,my_round)
86             . Election(my_add,my_round)
87           +
88             STAT_REC_TOKEN(my_add)
89             . Privilege(my_add,my_round)
90           +
91             sum(cand:Address,
92                 sum(round:Bool,
93                     STAT_REC_CLAIM(my_add,cand,round)
94                     . (  ( CRASH(my_add)
95                              . Fail(my_add)
96                             +
97                              STAT_SND_CLAIM(succ(my_add),cand,round)
98                              . Election(my_add,my_round)
99                           )
100                         <| or(less(cand,my_add),less(my_add,cand)) |> delta
101                        +
102                         Election(my_add,my_round)
103                         <| and(eq(my_add,cand),neq(round,my_round)) |> delta
104                        +
105                         Privilege(my_add,my_round)
106                         <| and(eq(my_add,cand),eq(round,my_round)) |> delta
107                       )
108                    )
```

```
109                 )
110
111
112           Fail(my_add:Address)
113           =
114              STAT_REC_TOKEN(my_add)
115              . STAT_SND_TOKEN(succ(my_add))
116                . Fail(my_add)
117            +
118              sum(cand:Address,
119                  sum(round:Bool,
120                      STAT_REC_CLAIM(my_add,cand,round)
121                      . ( STAT_SND_CLAIM(succ(my_add),cand,round)
122                          . Fail(my_add)
123                          <| not(eq(cand,my_add)) |> delta
124                         +
125                          Fail(my_add)
126                          <| eq(my_add,cand) |> delta
127                        )
128                     )
129                 )
130
131
132           Privilege(my_add:Address, my_round:Bool)
133           =
134              CRASH(my_add)
135              . Fail(my_add)
136            +
137              OPEN(my_add)
138              . ( CRASH(my_add)
139                  . Fail(my_add)
140                 +
141                  CLOSE(my_add)
142                  . ( CRASH(my_add)
143                      . Fail(my_add)
144                     +
145                      STAT_SND_TOKEN(succ(my_add))
146                      . Election(my_add,not(my_round))
147                    )
148                )
149            +
150              STAT_SND_TOKEN(succ(my_add))
151              . Election(my_add,not(my_round))
152
153
154           Link(my_add:Address)
155           =
156              LINK_REC_TOKEN(my_add)
157              . LINK_SND_TOKEN(my_add)
158                . Link(my_add)
159            +
```

```
160            sum(cand:Address,
161                sum(round:Bool,
162                    LINK_REC_CLAIM(my_add,cand,round)
163                    . LINK_SND_CLAIM(succ(my_add),cand,round)
164                      . Link(my_add)
165                   )
166               )
167          +
168            LINK_REC_TOKEN(my_add)
169            . Link(my_add)
170          +
171            sum(cand:Address,
172                sum(round:Bool,
173                    LINK_REC_CLAIM(my_add,cand,round)
174                    . Link(my_add)
175                   )
176               )
177
178
179   init    encap( { LINK_REC_CLAIM, STAT_REC_CLAIM,
180                     LINK_SND_CLAIM, STAT_SND_CLAIM,
181                     LINK_REC_TOKEN, STAT_REC_TOKEN,
182                     LINK_SND_TOKEN, STAT_SND_TOKEN },
183                   (  Link(0)
184                    ||
185                      Station(0)
186                    ||
187                      Link(1)
188                    ||
189                      Station(1)
190                    ||
191                      Link(2)
192                    ||
193                      Station(2)
194                   )
195                 )
```

μCRL model of symmetry for station 0

```
1     sort    Bool
2     func    T,F : -> Bool
3     map     not : Bool -> Bool
4             and : Bool # Bool -> Bool
5             or : Bool # Bool -> Bool
6             eq : Bool # Bool -> Bool
7             neq : Bool # Bool -> Bool
8             if : Bool # Bool # Bool -> Bool
9     var     b, b1, b2: Bool
10    rew     not(T) = F
11            not(F) = T
12            and(F,b) = F
```

```
13          and(b,F) = F
14          and(T,b) = b
15          and(b,T) = b
16          or(T,b) = T
17          or(b,T) = T
18          or(F,b) = b
19          or(b,F) = b
20          eq(F,F) = T
21          eq(F,T) = F
22          eq(T,F) = F
23          eq(T,T) = T
24          neq(F,F) = F
25          neq(F,T) = T
26          neq(T,F) = T
27          neq(T,T) = F
28          if(T,b1,b2) = b1
29          if(F,b1,b2) = b2
30
31    sort  Address
32    func  0,1,2 : -> Address
33    map   eq : Address # Address -> Bool
34          pred : Address -> Address
35          succ : Address -> Address
36          less : Address # Address -> Bool
37          greater : Address # Address -> Bool
38    var   a1, a2 : Address
39    rew   pred(0) = 2
40          pred(1) = 0
41          pred(2) = 1
42          succ(0) = 1
43          succ(1) = 2
44          succ(2) = 0
45          eq(0,0) = T
46          eq(0,1) = F
47          eq(0,2) = F
48          eq(1,0) = F
49          eq(1,1) = T
50          eq(1,2) = F
51          eq(2,0) = F
52          eq(2,1) = F
53          eq(2,2) = T
54          less(0,0) = F
55          less(0,1) = T
56          less(0,2) = T
57          less(1,0) = F
58          less(1,1) = F
59          less(1,2) = T
60          less(2,0) = F
61          less(2,1) = F
62          less(2,2) = F
63          greater(a1,a2) = less(a2,a1)
```

```
64
65   act     REC_CLAIM : Address # Address # Bool
66           SND_CLAIM : Address # Address # Bool
67           REC_TOKEN : Address
68           SND_TOKEN : Address
69           CRASH, OPEN, CLOSE : Address
70
71   proc    Station(my_add:Address)
72           = Election(my_add, T)
73
74           Election(my_add:Address, my_round:Bool)
75           =
76              CRASH(my_add)
77              . Fail(my_add)
78            +
79              SND_CLAIM(succ(my_add),my_add,my_round)
80              . Election(my_add,my_round)
81            +
82              REC_TOKEN(my_add)
83              . Privilege(my_add,my_round)
84            +
85              sum(cand:Address,
86                  sum(round:Bool,
87                      REC_CLAIM(my_add,cand,round)
88                      . (  ( CRASH(my_add)
89                             . Fail(my_add)
90                            +
91                             SND_CLAIM(succ(my_add),cand,round)
92                             . Election(my_add,my_round)
93                           )
94                          <| or(less(cand,my_add),less(my_add,cand)) |> delta
95                         +
96                          Election(my_add,my_round)
97                          <| and(eq(my_add,cand),neq(round,my_round)) |> delta
98                         +
99                          Privilege(my_add,my_round)
100                         <| and(eq(my_add,cand),eq(round,my_round)) |> delta
101                       )
102                    )
103                )
104
105
106          Fail(my_add:Address)
107          =
108             REC_TOKEN(my_add)
109             . SND_TOKEN(succ(my_add))
110               . Fail(my_add)
111           +
112             sum(cand:Address,
113                 sum(round:Bool,
114                     REC_CLAIM(my_add,cand,round)
```

```
115                     . ( SND_CLAIM(succ(my_add),cand,round)
116                         . Fail(my_add)
117                         <| not(eq(cand,my_add)) |> delta
118                        +
119                         Fail(my_add)
120                         <| eq(my_add,cand) |> delta
121                       )
122                    )
123                )
124
125
126         Privilege(my_add:Address, my_round:Bool)
127         =
128            CRASH(my_add)
129            . Fail(my_add)
130          +
131            OPEN(my_add)
132            . ( CRASH(my_add)
133                . Fail(my_add)
134               +
135                CLOSE(my_add)
136                . ( CRASH(my_add)
137                    . Fail(my_add)
138                   +
139                    SND_TOKEN(succ(my_add))
140                    . Election(my_add,not(my_round))
141                  )
142              )
143          +
144            SND_TOKEN(succ(my_add))
145            . Election(my_add,not(my_round))
146
147
148
149 init    Station(0)
```

State space of symmetry for station 0

```
1    des (0, 54, 13)
2    (0,"1!REC_CLAIM(0,0,F)",0)
3    (0,"2!REC_CLAIM(0,1,F)",1)
4    (0,"3!REC_CLAIM(0,2,F)",1)
5    (0,"4!REC_CLAIM(0,0,T)",2)
6    (0,"5!REC_CLAIM(0,1,T)",1)
7    (0,"6!REC_CLAIM(0,2,T)",1)
8    (0,"7!SND_CLAIM(1,0,T)",0)
9    (0,"8!REC_TOKEN(0)",2)
10   (0,"9!CRASH(0)",3)
11   (1,"10!SND_CLAIM(1,1,F)",0)
12   (1,"11!SND_CLAIM(1,2,F)",0)
13   (1,"12!SND_CLAIM(1,1,T)",0)
```

```
14  (1,"13!SND_CLAIM(1,2,T)",0)
15  (1,"14!CRASH(0)",3)
16  (2,"15!SND_TOKEN(1)",4)
17  (2,"16!CRASH(0)",3)
18  (2,"17!OPEN(0)",5)
19  (3,"18!REC_CLAIM(0,0,F)",3)
20  (3,"19!REC_CLAIM(0,1,F)",6)
21  (3,"20!REC_CLAIM(0,2,F)",6)
22  (3,"21!REC_CLAIM(0,0,T)",3)
23  (3,"22!REC_CLAIM(0,1,T)",6)
24  (3,"23!REC_CLAIM(0,2,T)",6)
25  (3,"24!REC_TOKEN(0)",7)
26  (4,"25!REC_CLAIM(0,0,F)",8)
27  (4,"26!REC_CLAIM(0,1,F)",9)
28  (4,"27!REC_CLAIM(0,2,F)",9)
29  (4,"28!REC_CLAIM(0,0,T)",4)
30  (4,"29!REC_CLAIM(0,1,T)",9)
31  (4,"30!REC_CLAIM(0,2,T)",9)
32  (4,"31!SND_CLAIM(1,0,F)",4)
33  (4,"32!REC_TOKEN(0)",8)
34  (4,"33!CRASH(0)",3)
35  (5,"34!CLOSE(0)",10)
36  (5,"35!CRASH(0)",3)
37  (6,"36!SND_CLAIM(1,1,F)",3)
38  (6,"37!SND_CLAIM(1,2,F)",3)
39  (6,"38!SND_CLAIM(1,1,T)",3)
40  (6,"39!SND_CLAIM(1,2,T)",3)
41  (7,"40!SND_TOKEN(1)",3)
42  (8,"41!SND_TOKEN(1)",0)
43  (8,"42!CRASH(0)",3)
44  (8,"43!OPEN(0)",11)
45  (9,"44!SND_CLAIM(1,1,F)",4)
46  (9,"45!SND_CLAIM(1,2,F)",4)
47  (9,"46!SND_CLAIM(1,1,T)",4)
48  (9,"47!SND_CLAIM(1,2,T)",4)
49  (9,"48!CRASH(0)",3)
50  (10,"49!SND_TOKEN(1)",4)
51  (10,"50!CRASH(0)",3)
52  (11,"51!CLOSE(0)",12)
53  (11,"52!CRASH(0)",3)
54  (12,"53!SND_TOKEN(1)",0)
55  (12,"54!CRASH(0)",3)
```