Adding Symmetry Reduction Techniques to mCRL2

Shreya Adyanthaya (0754541)
s.adyanthaya@student.tue.nl

22 December, 2010

Abstract

A means to include symmetry reduction in mCRL2 has been investigated. The approach used is to detect symmetry in the specification level and then generate equivalence classes from the linear process specification, following which the symmetry reduced state space can be generated on the fly using a modified state space generation algorithm. The process has been illustrated with the Dining Philosophers example. The symmetry reduced state space is equivalent to the original state space under a new notion of bisimulation called Bisimulation under Permutations. It has also been proved that the expansion of the symmetry reduced state space generates the original state space. The largest possible reduction in a state space with S states and n processes is S/n!.

1 Introduction

State space explosion is one of the biggest problems associated with model checking. Model checking algorithms mainly require the construction of the state space with all possible states of the system. Since the number of states increases exponentially with the number of processes in the system, large number of processes result in huge state spaces that cannot be verified with the existing resources. There are several methods that try to overcome this problem such as symbolic model checking [1] and partial order reduction [2]. Another approach is Symmetry reduction which aims at grouping together states, which exhibit symmetry, into equivalence classes and replacing all the states in a class with one chosen representative state. In order to overcome the state space explosion problem to any extent, it is necessary that instead of the original large state space the symmetry reduced state space be generated on the fly.

In this paper, Section 2 contains some background theoretical concepts on symmetry. Section 3 gives an overview of the mcrl2 language and the toolset and the approach adopted toward symmetry reduction in this paper. Section 4 describes the dining philosophers problem that will be used as an example to explain the process throughout the paper. Section 5, 6 and 7 describe the three steps towards symmetry reduction in mCRL2. Section 8 gives another example for symmetry reduction on the mutual exclusion protocol. Section 9 gives a new notion of bisimulation which relates the original and the symmetry reduced state space. Section 10 gives a proof of isomorphism between the original state

space and the expansion of the symmetry reduced state space with section 11 concluding the paper.

2 Preliminaries

2.1 Symmetry

The state space corresponding to a system can be represented as a labeled transition system (LTS) given by $M = \langle S, A, T, s_0 \rangle$, where S is the set of states, A the set of actions, $T \subseteq S \times A \times S$ the set of transitions and s_0 is the initial state. The main idea behind symmetry reduction is to partition the states in the LTS into equivalence classes of symmetric states with one representative state each. Thus, the state space can be reduced to an LTS containing just the representative states and transitions.

There also exists another approach towards symmetry reduction which makes use of the concept of scalar sets [3]. A scalar set is a new data type which allows only restricted operations on its elements which do not affect the symmetry of the system. The user needs to modify the specification of the system such that symmetric processes are defined using scalar sets. But this approach has not been adopted in this paper.

2.2 Automorphism

A permutation, π is a bijection such that π: $S \rightarrow S$. It is a function that maps symmetric states onto each other.

A permutation, π, which preserves the transition relation of a system (M) such that, $s \xrightarrow{a} t \in T$ implies $\pi(s) \xrightarrow{a} \pi(t) \in T$, is called an automorphism of M, if $\pi(s_0) = s_0$. The set of all automorphisms for a model of a system form a group under functional composition denoted by $AUTM$ [4]. This means that $AUTM$ has an identity permutation and every other permutation in $AUTM$ has an inverse permutation associated with it. Any subgroup $G \in AUTM$ which is also a group under functional composition produces a symmetric structure of the system.

2.3 Equivalence Classes: Orbits

Two states s and s′ are said to be symmetric if there exists a permutation $\pi \in G$ such that $\pi(s) = s'$ for some $G \in AUTM$. States that are symmetric to each other can be put together into equivalence classes also called orbits, with one representative state being chosen for each class. The representative state is chosen such that it satisfies the transition relation of the system. Given this information of the equivalence classes and the representative states an annotated quotient structure can be constructed for a system.

2.4 Annotated Quotient structure

For a subgroup $G \in AUTM$, the annotated quotient structure (AQS) for a system M, is a labeled transition system given by $M_G = \langle \overline{S}, A, \overline{T}, s_0 \rangle$, where $\overline{S}$ is the set of representative states (one per equivalence class) and $\overline{T} = \{\bar{s} \xrightarrow{\pi(a),\pi}$

$\bar{t} \mid \pi \in G, \bar{s} \in \overline{S} \wedge \bar{t} \in \overline{S} \wedge \bar{s} \xrightarrow{a} \pi(\bar{t}) \in T\}$, which means that if there is a transition from a representative state, $\bar{s}$, to another state, t, with an action, a, then in the annotated quotient structure there will be a transition from $\bar{s}$ to the representative of t ($\bar{t}$) with the action label containing the permuted action π(a) appended with the permutation π itself [4]. This means that the annotated quotient structure only consists of the representative states and the transitions between them.

3 Overview of mCRL2

3.1 Language and Tool

mCRL2 is a formal specification language with an associated toolset [5]. The toolset can be used for modeling, validation and verification of concurrent systems and protocols. The toolset supports a collection of tools for linearization, simulation, state-space exploration and generation and tools to optimize and analyze specifications. Moreover, state spaces can be manipulated, visualized and analyzed.

The most fundamental concept in mCRL2 is the process. A system, in mCRL2, is specified in the form of several processes running in parallel. A process can carry several data parameters, combinations of which give the state of the process. A state change occurs upon execution of an action. Every process has a corresponding state space or Labeled Transition System (LTS) which contains all states that the process can reach, along with the possible transitions between those states.

Another fundamental concept in mCRL2 is the linear process which is a process from which all parallelism has been removed to produce a choice among several condition-action-effect rules. The mCRL2 toolset allows us to translate a large set of mCRL2 specifications to linear process specifications(LPS) consisting of linear processes, using the command *mcrl22lps*. The linear processes can then be manipulated and operated on. The state space associated with a linear process specification is generated using the *lps2lts* command, which can then be visualized and manipulated.

3.2 Specification in mCRL2

A specification in mCRL2 consists of various sections which together describe the operations of the system being modeled. In order to explain the various sections, the specification for the dining philosophers problem in section 4 has been taken as reference.

- The **sort** section consists of the data types that are to be used in the specification. It consists of the user defined data types that are defined using the **struct** construct and given some specific instances. For example, 'sort Philosopher= struct p1|p2' specifies a user defined sort called 'Philosopher' which has two instances p1 and p2. Similarly, 'Fork' is also a user defined sort. Standard data types such as natural numbers, integers, booleans are predefined.

- The **map** section contains functions that will be used in the processes in the specification. The return type and the type of the parameters

to these functions need to be defined. For instance, 'map rightFork: Philosopher→Fork' defines a function named rightFork thats maps a Philosopher to a Fork.

- The **var** sections contains variables of the different sort types that are to be used in the **eqn** section.
- The **eqn** section contains equations that define the functions in the **map** section. This is analogous to the function body where it is specified what the function returns when given specific values as arguments. For instance, 'eqn rightFork(p1) = f1' maps specific instance p1 of Philosopher sort to specific instance f1 of Fork sort.
- The **act** sections contains the actions that are to performed by the processes. The actions can have 0 or more parameters, the types of which are specified in this section. For instance, 'get : Philosopher # Fork' is an action 'get' between a Philosopher and Fork.
- The **proc** section is the most important section in an mCRL2 specification since it describes the processes that make up the system being modeled. For example, 'P_Philosopher' is a process that describes the behavior of a philosopher by means of the actions in the **act** section. There is a recursive call back to P_Philosopher at the end of the description in order to describe a real life system which does not contain deadlocks and where operations can happen in a repeated manner.
- The **init** block is the main block of the specification which initializes the various processes in the specification while also instantiating them with specific initial values when needed. For example, P_Philosopher(p1) || P_Philosopher(p2) declares two instances of the P_Philosopher process. It is also possible to describe the communications between the various processes using the **comm** subsection, where actions of different processes synchronize with each other and are renamed to single actions. These synchronizing actions if blocked in the **block** subsection will cause the processes to proceed only when those actions synchronize on the same parameter values. For instance, 'get|up→lock' defines an interaction between processes P_Philosopher and P_Fork where actions get and up synchronize into action lock in the system. The actions get and up are also blocked in the **block** section. Alternatively, the **allow** block can be used to obtain the same behaviour by including in this section all actions except the ones that need to be blocked due to communication, which also includes the renamed actions in the **comm** block. It is also possible to hide actions as internal tau actions by including them in the **hide** subsection of the **init** block.

3.3 Symmetry in mCRL2

A system in mCRL2 is made of a set of process instances. A permutation, π, in that case is a function that maps a process index onto another, such that when it is applied to the set of process indices in the system it produces a rearrangement of the indices in some manner. So given a set of process indices in a system, I, a permutation is a bijection such that π: I→I. For example, a

permutation π={2,1,4,3} works on four processes such that $\pi(1)$=2, $\pi(2)$=1, $\pi(3)$=4 and $\pi(4)$=3.

When a permutation is applied to a state it permutes all the process parameters of the state as well as the outgoing actions of the state. Permutation of an action returns the same action but with permuted process parameters and unchanged data parameters.

A permutation, π, of process indices is called an automorphism of a system, M, if it satisfies the following three properties :

1. It preserves the types of the processes it maps which means that it does not map dissimilar processes onto each other.

2. It preserves the associations of the processes it maps which means that given the process associations in a system, if a permutation is applied to a process then it must also be applied to all processes associated with it and the resulting system should be valid with respect to the given associations.

3. It preserves the transition relation of the system such that if there exists a transition between two states s and t in T, then there should be a transition between the states π(s) and π(t) in T. Also, for the start state s_0, $\pi(s_0)$=s_0.

It must also be noted that when a permutation is applied to a process with parameters then the process parameters should also be permuted and the data parameters kept unchanged.

To obtain the annotated quotient structure, all the states in the orbit are replaced by the representative states. All the outgoing transitions of the representative states are retained and the transitions from the non representative states are included by just appending the respective permutation to the action labels of the transitions from the corresponding representative state. In this manner, by applying the permutations appended to action labels to the states and actions, the original state space can be derived from the symmetry reduced state space.

3.4 Linear Process Equations

A linear process is a process of a restricted form in mCRL2. It consists of a single process name on the left hand side with precisely one action in front of the recursive invocation of the process variable at the right hand side.

A linear process equation has the form shown in Equation 1.

$$X(d:D) = \sum_{i:I} \sum_{e_i:D_i} c_i(d,e_i) \rightarrow a_i(f_i(d,e_i)).X(g_i(d,e_i)) \tag{1}$$

In the equation, d is the state parameter and can be a list of parameters in the domain D of states. I is a finite index set and for $i \in I$, e_i is the local sum variable per summand i in the domain D_i, c_i is the condition for summand i, $a_i \in Act$ is the action label for summand i, f_i is the parameter for action a_i and g_i is the next state.

[X(e)] defines the Labeled transition system given by [X(e)]=$\langle$S, A, $\rightarrow$, $s_0\rangle$, where:

- $S = D$ is the state space
- s_0 = e is the initial state
- A = $\{a_i(d)|\ i \in I \wedge d \in D_{a_i}\}$. This means that the set of actions labels in the linear process equations, A, gives the set of actions in the LTS.
- $d \xrightarrow{a} d'$ iff for some $i \in I$, $\exists e_i{:}D_i.\ c_i(d, e_i) \wedge d' = g_i(d, e_i) \wedge a = a_i(f_i(d,e_i))$. This means that if condition c(d,e) holds, then there exists, in the LTS, a transition from state given by 'd' to the state given by 'g(d,e)' which is labeled by the action 'a(f(d,e))'.

3.5 Approach adopted towards Symmetry Reduction in mCRL2

In order to perform symmetry reduction in mCRL2, symmetry needs to first be detected in the system at the specification level. The approach used here is to extract the permutations from the initial mCRL2 specification and based on this information, generate the equivalence classes from the linear process specification. The above two steps together constitute **Symmetry Detection**. Once symmetry detection has been performed, **Symmetry Reduction** is performed to generate the symmetry reduced state space on the fly using a modified state space generation algorithm. It must be noted that the symmetry reduced state space is generated on the fly in this approach as opposed to symmetry detection and reduction being performed after the original state space has been generated. This is essential because the basic requirement is to avoid generating large state spaces.

4 Dining Philosophers Problem

In order to explain how symmetry reduction can be performed in mCRL2, the Dining Philosophers example has been considered. It consists of several philosophers who are seated around a table and several forks placed in a manner that there is one fork between every pair of philosophers. A philosopher needs both the left as well as the right fork to begin eating.

In order to model this problem in mCRL2, the philosophers and forks have been modeled as sorts (*Phil*,*Fork*) and have been mapped onto each other by means of equations which give the leftfork (*leftfork()*) and rightfork (*rightfork()*) for any philosopher. Actions have been used to describe the philosopher getting a fork (*get*), putting it down (*put*) or eating (*eat*) as well as the fork being up (*up*) or down (*down*). The entire set of operations performed by the philosophers and the forks have been encapsulated into two processes. Communications have been established between the philosophers and the forks, in the init block, by communicating the actions *get* and *up* to the action *lock*, saying the fork has been locked from use. Similarly, *put* and *down* synchronize to form the action *free*, saying the fork is free for use. The processes have also been instantiated in parallel in the init block which is the main block in the specification.

In order to show the process of symmetry reduction, a specification for a simple case of dining philosophers with just two philosophers and forks, which contains deadlock, has been considered. Equations have been used to specify

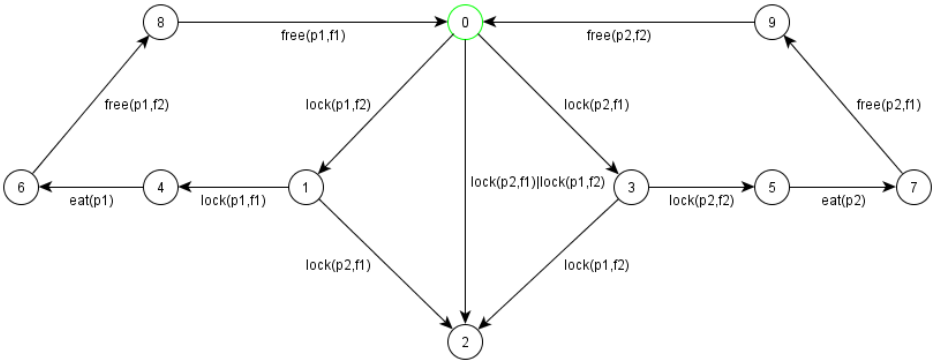

Figure 1: Original State Space: Dining Philosophers Problem with 2 philosophers and 2 forks

the setting where philosopher 1 has fork 1 on the right and fork 2 on the left. Similarly, philosopher 2 has fork 2 on the right and fork 1 on the left. This specification, when run in the mCRL2 tool, generates the state space shown in Figure 1. It can be observed that there exists symmetry in the state space owing to the symmetric behavior of the philosophers and the forks.

---

sort Philosopher = struct p1 | p2;
Fork = struct f1 | f2;
map leftFork, rightFork: Phil − > Fork;
eqn rightFork(p1) = f1;
leftFork(p1) = f2;
rightFork(p2) = f2;
leftFork(p2) = f1;
act get, put, up, down, lock, free: Phil # Fork;
eat: Phil;
proc P_Philosopher(p: Phil) = get(p,leftFork(p)).get(p,rightFork(p)).
eat(p).put(p,leftFork(p)).put(p,rightFork(p)).P_Phil(p);
P_Fork(f: Fork) = sum p:Phil. up(p,f) . down(p,f) . P_Fork(f);
init block({ get, put, up, down },
comm({ get|up→lock, put|down→free },
P_Fork(f1) || P_Fork(f2) || P_Philosopher(p1) || P_Philosopher(p2)));

---

5 Step 1: Symmetry Detection in the specification

In order to detect symmetry in the specification, it is required to extract all valid permutations (Automorphisms) from the specification and form the group $AUTM$. Algorithm 1 shows how this can be performed on a given specification. Any specification has a set of process instances which are assigned indices and the set of all possible permutations of the indices, P, is considered as a starting

point and given as input to this algorithm. For each permutation in this set P, the algorithm checks if it is a valid permutation which means that it is an automorphism of the model. Only such permutations are added to the set $AUTM$.

Algorithm 1 Symmetry Detection in the specification

```
1: Check_Permutations(P)
2: for each π in P do
3:   for each π(p_i)==p_j where p_i, p_j∈P & i, j∈I do
4:     if Type(p_i)==Type(p_j); then
5:       if Check_associations(p_i, p_j) then
6:         if Check_parameters(p_i, p_j) then
7:           π ∈ AUTM
8:         end if
9:       end if
10:    end if
11:  end for
12: end for
13: return AUTM
```

In order to check if a permutation is valid the algorithm checks for three properties. First property says that the permutation preserves the types of the processes being mapped onto each other. This means that if a process P_1 is mapped onto a process P_2 then they must be of the same sort in the mCRL2 specification. In the dining philosophers problem considered, there are four processes, namely the two philosophers (p_1, p_2) and the two forks (f_1, f_2). Assuming p_1, p_2 are assigned indices of 1, 2 and f_1, f_2 are assigned indices 3, 4 respectively, all possible permutations of the set of indices {1, 2, 3, 4} are obtained. This results in 24 permutations which need to be checked for validity. After checking for the first property and eliminating all permutations that map a philosopher to a fork and vice versa, we are left with 4 permutations namely, {1,2,3,4}, {2,1,3,4}, {1,2,4,3}, {2,1,4,3}.

Second property to be tested checks if associations between the process instances which are specified as equations in the specification are not violated in the system obtained after applying the permutations to the processes. This happens in the function Check_associations(p_i, p_j). It says that if a permutation, π, maps p_i to p_j then the permutations of all processes associated with p_i are the processes associated with p_j in the new system obtained by replacing p_i with p_j. These associations must be consistent with the equations in the specification otherwise π is not an automorphism. In the example, property 2 is violated by {2,1,3,4} and {1,2,4,3}. In {2,1,3,4}, the philosophers get interchanged but the forks do not. So after applying the permutations, philosopher 2 will have fork 2 on the right and fork 1 on the left which violates the system setting given in the equations in the specification.

Third property to be tested checks if the processes being mapped onto each other have symmetry in their parameters as well. This is checked in the function Check_parameters(p_i, p_j). Parameters can be of process type or of data type. If p_1 is a process parameter of p_i then the corresponding parameter of p_j should be

Type(p) returns the sort that p belongs to.

Check_associations(p_i, p_j)
if processes associated with p_i = {p_1, p_2,...} **then**
  processes associated with p_j = {$\pi(p_1)$, $\pi(p_2)$,...};
  if {$\pi(p_1)$, $\pi(p_2)$,...} consistent with equations of p_j **then**
    return true;
  endif
endif

Check_parameters(p_i, p_j)
for all process parameters p_{ik} of p_i and corresponding p_{jk} of p_j
  if $pr_{ik} \neq \pi(pr_{jk})$ **then**
    return false;
  endif
endfor
for all non-process parameters e_{ik} for p_i and corresponding e_{jk} for p_j
  if $np_{ik} \neq np_{ik}$ **then**
    return false;
  endif
endfor
return true;

$\pi(p_1)$. On the other hand, d_1 is a data parameter of p_i then the corresponding parameter of p_j should be a data parameter with the same value as d_1. If a permutation satisfies the above three properties then it is an automorphism of the structure and is added to the set $AUTM$ of automorphisms. Since in this particular example the processes do not have parameters, the remaining 2 permutations satisfy property 3. Hence, $AUTM$ consists of two permutations, namely π_1 = {1,2,3,4} and π_2 = {2,1,4,3}.

The final set $AUTM$ thus obtained forms a group under functional composition. This means that, $AUTM$ had an identity element π_1 such that $\pi_1(p)$=p. Also, every other permutation π_i in $AUTM$ has an inverse element π_i^{-1} such that $\pi_i^{-1}\pi_i$ gives the identity element or in other words, $\pi_i^{-1}\pi_i(p)$=p=$\pi_1(p)$. This is because the permutations are generated based on symmetric properties of the processes and hence if there is a permutation which maps a process p to another process q, then there must be a permutation which maps q back to p, since p and q are symmetric. In the dining philosophers example, $AUTM$ consists of the identity permutation, π_1 = {1,2,3,4} as well as the permutation, π_2 = {2,1,4,3}, which in this case is the inverse of itself. In other words, for some process p, $\pi_2(\pi_2(p))$=p=$\pi_1(p)$.

6 Step 2: Generation of equivalence classes

Two states s and s' are said to be symmetric iff for each action a from s leading to a state t, there is a corresponding action $\pi(a)$ from s' leading to a state $\pi(t)$ for some $\pi \in AUTM$ and vice versa.

Algorithm 2 Generation of equivalence classes from LPS

```
1: Equivalence_Classes()
2: for each state s do
3:    EClass(s):={s};
4:    for all π∈AUTM do
5:       π(s) := s;
6:    end for
7: end for
8: repeat
9:    for a pair of states s_i, s_j & π∈AUTM, s.t. EClass(s_i)≠EClass(s_j) do
10:      if c_i & c_j are symmetric under π and syntactically equivalent then
11:         if a_i & a_j are symmetric under π then
12:            if EClass(s'_i)==EClass(s'_j) then
13:               π(s_i):=s_j;
14:               π^{-1}(s_j):=s_i;
15:               Merge EClass(s_i) and EClass(s_j);
16:            end if
17:         end if
18:      end if
19:   end for
20: until (no change)
```

The procedure of generating equivalence classes, given the LPS, is given in Algorithm 2. Initially every state has a separate equivalence class containing itself as the only member and application of any permutation on a state returns itself. The algorithm then repeatedly combines the equivalence classes of states which perform symmetric transitions to symmetric states until all symmetric states have been grouped together into separate equivalence classes. The algorithm also sets the permutations that map states to each other. The permutation (if any) that maps the outgoing actions and next states of a particular state, s_1, to those of another state, s_2, is the permutation that maps s_1 to s_2.

A simplified version of the LPS generated for the dining philosophers specification is shown in Table 1. It can be observed that state 8 and 9 perform symmetric actions, free(p1,f1) and free(p2,f2) (π_2(free(p1,f1)) = free(π_2(p1), π_2(f1)) = (free(p2,f2))) to state 0 which is trivially symmetric to itself. Hence, their equivalence classes are combined into one. Similarly, state 6 and 7 perform symmetric actions free(p1,f2) and free(p2,f1) leading to symmetric states 8 and 9 which have already been put into one equivalence class. Hence, their equivalence classes are combined together too. This process takes place repeatedly and the system is reduced to 6 equivalence classes namely, {0}, {1, 3}, {2}, {4, 5}, {6, 7}, {8, 9}.

7 Step 3: Generation of Symmetry Reduced state space

Using the information of the permutations and the equivalence classes, the symmetry reduced state space can be directly generated on the fly by using a mod-

X(n) = (n≈0)→ lock(p1,f2).X(1)
+(n≈0)→ lock(p2,f1).X(3)
+(n≈0)→ (lock(p2,f1)|lock(p1,f2)).X(2)
+(n≈1)→ lock(p1,f1).X(4)
+(n≈1)→ lock(p2,f1).X(2)
+(n≈3)→ lock(p2,f2).X(5)
+(n≈3)→ lock(p1,f2).X(2)
+(n≈4)→ eat(p1).X(6)
+(n≈5)→ eat(p2).X(7)
+(n≈6)→ free(p1,f2).X(8)
+(n≈7)→ free(p2,f1).X(9)
+(n≈8)→ free(p1,f1).X(0)
+(n≈9)→ free(p2,f2).X(0)

Table 1: Simplified LPS for Dining Philosophers Specification

ified state space generation algorithm. The resulting state space has permutations appended to the action labels. The original state space generation algorithm is given in Algorithm 3. There are two sets maintained namely, *toVisit* which includes all states that need to be visited and *visited* which includes all states that have already been visited. Initially, *visited* is set to *null* and *toVisit* contains the start state of the system, s_0. For each state in *toVisit*, the corresponding transitions and resulting states are drawn from the LPS, along with it being removed from *toVisit* and added to *visited*. The resultant states are added to *toVisit* to be visited next. This process repeats until *toVisit* is empty.

Algorithm 3 Original State Space Generation

```
1:  state_space_generation()
2:  toVisit:={s_o}
3:  visited:=∅
4:  repeat
5:    for each s∈toVisit do
6:      toVisit:=toVisit\{s};
7:      if s∉visited then
8:        for each transition s -a-> s' do
9:          transitions:=transitions ∪ {(s, a, s')};
10:         toVisit:=toVisit ∪ {s'};
11:       end for
12:       visited:=visited ∪ {s};
13:     end if
14:   end for
15: until (toVisit=∅)
```

Algorithm 4 gives the modified state space generation algorithm that maintains another set *visitedEC* which contains the equivalence classes that have been visited and is initially set to *null*. For every state s in *toVisit*, it is checked if its equivalence class (EClass(s)) has been visited earlier or not. If not then s is chosen as the representative state for the equivalence class. Choosing

representatives in this manner ensures that they are the first encountered states in the equivalence classes that satisfy the transition relation of the system. All states in EClass(s) get their representative state set to s. Next, for each transition from s leading to another state s′, if EClass(s′) has also not been visited earlier then a transition between s and s′ with the action label (a) appended with the identity transition (π_1) is added to the state space. If EClass(s′) has been visited then s′ has a representative state in which case a transition from s to rep(s′) with the permuted action label appended with the permutation that maps s′ to rep(s′) is added to the state space. On the other hand, if EClass(s) has been visited earlier then it has a representative state and in this case, instead of adding the transitions from s, the permutation which maps s to rep(s) is appended to the action label of all the transitions from rep(s). For the dining philosophers example the symmetry reduced state space thus generated is given in Figure 2.

Algorithm 4 Symmetry Reduced State Space Generation

```
1:  state_space_generation()
2:  toVisit:={s_o}
3:  visited:=ø
4:  visitedEC:=ø
5:  repeat
6:    for each s∈toVisit do
7:      toVisit:=(toVisit\{s});
8:      if s∉visited then
9:        if EClass(s)∉ visitedEC then
10:         ∀s_i∈EClass(s), rep(s_i)=s;
11:         for each transition s -a-> s′ do
12:           if EClass(s′)∉visitedEC then
13:             transitions:=transitions ∪ {(s, {a,π_1}, s′};
14:           else if EClass(s′)∈visitedEC && rep(s′)==π_i(s′) then
15:             transitions:=transitions ∪ {(s, {π_i(a),π_i}, rep(s′)};
16:           end if
17:           toVisit:=toVisit ∪ {s′};
18:         end for
19:         visitedEC:=visitedEC ∪ {EClass(s)};
20:       else if EClass(s)∈ visitedEC && rep(s)==π_i(s) for π_i∈AUTM then
21:         for each transition s -a-> s′ do
22:           Append π_i to corresponding action label of rep(s);
23:           toVisit:=toVisit ∪ {s′};
24:         end for
25:       end if
26:       visited:=visited ∪ {s};
27:     end if
28:   end for
29: until (toVisit=ø)
```

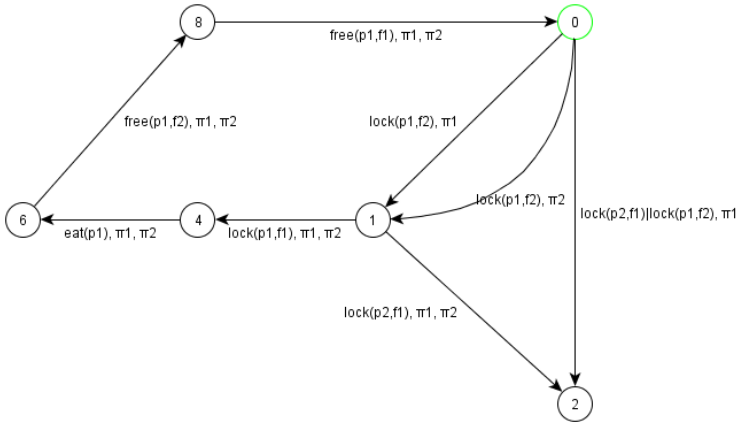

Figure 2: Symmetry Reduced State Space: Dining Philosophers Problem

8 Symmetry Reduction in Mutual Exclusion Protocol

Consider a system consisting of n processes executing in parallel [6]. Each process can either be in the Critical section or in the Non-critical section. Initially all processes are in the Non-critical section. A process enters the critical section, performs some operations and leaves the critical section. The mutual exclusion property states that no two processes can be in the critical section at the same time. The specification in mCRL2 for an instance of the system with 3 processes is given below and it consists of the three process instances as well as a monitor process that does not allow two processes to enter the critical section at the same time. The original state space generated from the specification is given in Figure 3.

---

sort process= struct p1|p2|p3;
act critical, noncritical, request_critical, allow_critical:process;
proc P(p:process) = request_critical(p).P(p);
Monitor = sum p:process.allow_critical(p).noncritical(p).Monitor;
init block({request_critical, allow_critical},
comm({request_critical|allow_critical → critical},
P(p1)||P(p2)||P(p3)||Monitor));

It can be observed that there is symmetry in this system with respect to the three process instances which exhibit the same behavior. Hence, symmetry reduction is performed on the specification and on the subsequent LPS and the reduced state space given in Figure 4 is generated.

It can also be noted that the reduction in the state space depends on the number of permutations that can be extracted from the system and is given by S/ $|AUTM|$. For a system with n processes, the maximum possible number of permutations is n!(in case of fully symmetric systems) and hence the maximum possible reduction is in the order of S/n! [7].

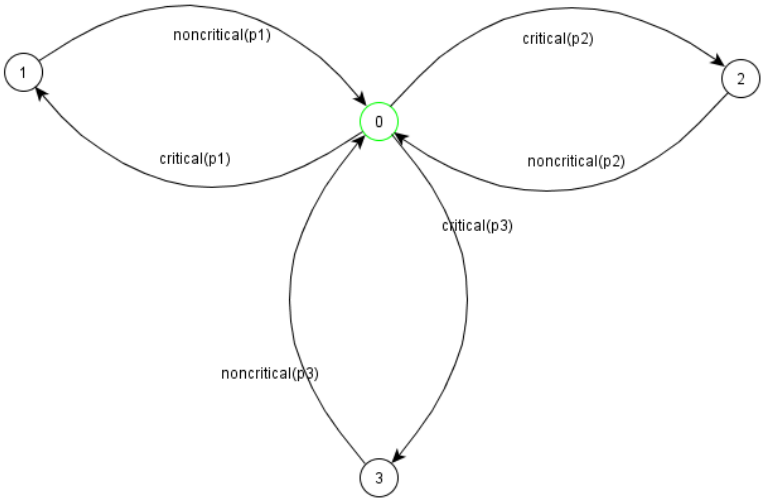

Figure 3: Original State Space: Mutual Exclusion Protocol with 3 processes

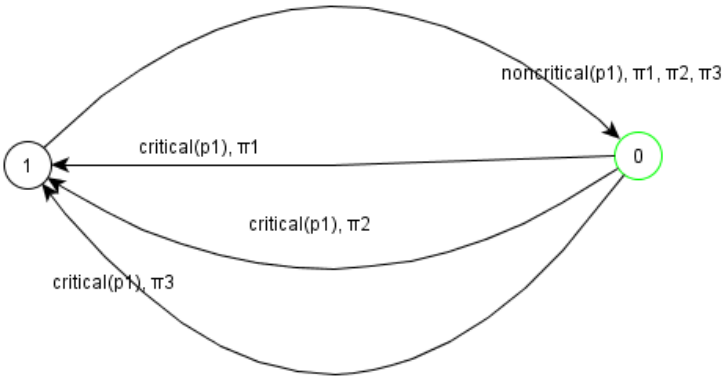

Figure 4: Symmetry Reduced State Space: Mutual Exclusion Protocol

9 Bisimulation under Permutations

The symmetry reduced state space generated will prove useful only if it has an equivalence relation with the original state space, which means that properties which hold in the original state space must hold in the reduced state space too. This relation between the two state spaces can be specified by means of a new notion of bisimulation named *Bisimulation under permutations* given in Definition 1.

Definition. Let $L_1 = \langle S_1, A_1, \rightarrow_1, s_1, T_1 \rangle$ and $L_2 = \langle S_2, A_2, \rightarrow_2, s_2, T_2 \rangle$ be labeled transition systems. A binary relation, $R_\pi \subseteq S_1 \times S_2$ is called a *bisimulation relation under Permutations* iff for all $s \in S_1$ and $t \in S_2$ such that $sR_\pi t$ holds, it also holds for all actions $a \in A_1$ that:

1. if $s \xrightarrow{a}_1 s'$, then there exists a $t' \in S_2$ such that $t \xrightarrow{\pi(a),P}_2 t'$, where P is a

set of permutations and $\pi \in P$, with $s' R_\pi t'$.

2. if $t \xrightarrow{a,P}_2 t'$, where P is a set of permutations, then there exists an $s' \in S_1$ such that $s \xrightarrow{\pi^{-1}(a)}_1 s'$, for some $\pi \in P$, with $s' R_\pi t'$.

3. $s \in T_1$ iff $t \in T_2$

Two states s and t are said to be *bisimilar under permutations*, denoted by $s \leftrightarrow_\pi t$, if there is a bisimulation relation under permutations R_π such that $sR_\pi t$ and $\pi \in AUTM$. The labeled transition systems L_1 and L_2 are *bisimilar under permutations* iff their initial states are bisimilar, i.e. $s_1 R_\pi s_2$.

10 Proving Equivalence

Definition. (Expansion) Expansion is the process of applying the permutations on the action labels to the states and the actions in the symmetry reduced state space, given by $L_r = \langle S_r, A_r, \rightarrow_r, s_r, T_r \rangle$, and generating an Expanded state space, given by $L_e = \langle S_e, A_e, \rightarrow_e, s_e, T_e \rangle$. It begins by including the start state s_0 belonging to L_r as it is into L_e. Thereafter, a transition of the form $s \xrightarrow{a,P} s'$ in L_r, for each $\pi_i \in P$, is added to L_e in the form $\pi_i^{-1}(s) \xrightarrow{a} \pi_i^{-1}(s')$.

Definition. (Isomorphism of Labeled Transition Systems) Two Labeled Transition Systems, $L_1 = \langle S_1, A_1, \rightarrow_1, s_1, T_1 \rangle$ and $L_2 = \langle S_2, A_2, \rightarrow_2, s_2, T_2 \rangle$, are isomorphic if they are structurally identical, which means that each of S_1, A_1, $\rightarrow_1$ are set-equivalent to S_2, A_2, $\rightarrow_2$ and $s_1 = s_2$ up to renaming of states and actions.

Theorem 1. *Expansion of the symmetry reduced state space generates the original state space*

Proof: Let the LTS, $L = \langle S, A, \rightarrow, s_0, T \rangle$ represent the original state space and $L_r = \langle S_r, A_r, \rightarrow_r, s_r, T_r \rangle$ represent the symmetry reduced state space. Expansion is performed on the symmetry reduced state space to generate an expanded state space given by, $L_e = \langle S_e, A_e, \rightarrow_e, s_e, T_e \rangle$. Thereafter we need to show that the expanded state space, L_e, is isomorphic to the original state space which means that S_r, A_r, $\rightarrow_r$ are set-equivalent to S_e, A_e, $\rightarrow_e$ and $s_r = s_e$ up to renaming of states and actions. To prove this we need to first show that $L \subseteq L_e$ and $L_e \subseteq L$. $L \subseteq L_e$ implies that $S_r \subseteq S_e$, $A_r \subseteq A_e$, $\rightarrow_r \subseteq \rightarrow_e$ and $s_r = s_e$.

- First, we show that $L \subseteq L_e$. We know that in order to generate the reduced state space every transition in the original state space L of the form $s \xrightarrow{a} s'$, is transformed into a transition of the form $\pi_i(s) \xrightarrow{a,P} \pi_i(s')$ in L_r for some $\pi_i \in P \in AUTM$. Let state $\pi_i(s)$ be denoted by a state p and $\pi_i(s')$ be denoted by a state p'. A transition of the form $p \xrightarrow{a,\pi_i} p'$ in L_r is transformed into a transition of the form $\pi_i^{-1}(p) \xrightarrow{a} \pi_i^{-1}(p')$ in the expanded state space L_e. Now, $\pi_i^{-1}(p)$ is equal to $\pi_i^{-1}(\pi_i(s))$ which is equivalent to s itself from the property of inverse element in Group theory. Similarly, $\pi_i^{-1}(p')$ is equivalent to $\pi_i^{-1}(\pi_i(s'))$, which is s'. Hence, we observe that every transition of the form $s \xrightarrow{a} s'$ in L is present in L_e as $s \xrightarrow{a} s'$ itself. Hence, $S \subseteq S_e$, $A \subseteq A_e$, $\rightarrow \subseteq \rightarrow_e$ and we also have that $s = s_e$ since the start state is unique in the system. This implies that $L \subseteq L_e$.

- Now, we need to show $L_e \subseteq L$. In order to prove this we need to show that L_e does not have any extra states and transitions than the ones that match L. This can be proved by contradiction. Assume that there exists a transition in L_e given by $s \xrightarrow{a} s'$, such that there is no corresponding transition in the original state space. This transition should have been derived from some corresponding transition in L_r given by $\pi_i(s) \xrightarrow{a,\pi_i} \pi_i(s')$ for some $\pi_i \in AUTM$. Let $\pi_i(s)$ be u and $\pi_i(s')$ be u′. Since every transition in L_r is derived from some transition in L, there has to be a transition in L corresponding to the transition $u \xrightarrow{a,\pi_i} u'$ in L_r given by $\pi_i^{-1}(u) \xrightarrow{a} \pi_i^{-1}(u')$. Now, $\pi_i^{-1}(u)$ is the same as $\pi_i^{-1}(\pi_i(s))$, which is s itself. Similarly, $\pi_i^{-1}(u')$ is the same as $\pi_i^{-1}(\pi_i(s'))$, which is s′. Hence, the transition in L is $s \xrightarrow{a} s'$. But this contradicts our assumption that there was no such transition in L. Hence our assumption is wrong which means that $L_e \subseteq L$.

From the above deductions it is clear that $L = L_e$, or in other words they are isomorphic.

11 Conclusion

In this paper, a means of performing symmetry reduction in mCRL2 has been demonstrated. The advantage of the approach chosen here is that the user need not perform any modifications to the specification in order to detect symmetry in the system. It is also possible to use an alternative approach where the user can specify the existence of symmetry in the system by means of scalar sets, which is a new data type with some restricted operations allowed that preserve symmetry in the system. However, this approach has not been used in this work. The state space generated after symmetry reduction is reduced by the order of the number of permutations that are extracted from the specification. Hence, in highly symmetric systems this approach can be very beneficial in reducing the size of the state space.

References

[1] K. L. McMillan, *Symbolic Model Checking*. Norwell, MA, USA: Kluwer Academic Publishers, 1993.

[2] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.

[3] C.-W. N. Ip, D. L. Dill, and J. C. Mitchell, "State reduction methods for automatic formal verification," 1996.

[4] M. M. Jaghoori, M. Sirjani, M. Mousavi, E. Khamespanah, and A. Movaghar, "Symmetry and partial order reduction techniques in model checking rebeca," *Acta Informatica*, vol. 47, pp. 33–67, 2010.

[5] J. F. Groote, A. Mathijssen, M. Reniers, Y. S. Usenko, and M. van Weerdenburg, "The formal specification language mcrl2." in

MMOSS, ser. Dagstuhl Seminar Proceedings, E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, Eds., vol. 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. [Online]. Available: http://dblp.uni-trier.de/db/conf/dagstuhl/P6351.htmlGrooteMRUW06

[6] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Formal Methods in System Design*, vol. 9, pp. 105–131, August 1996.

[7] A. Miller, A. Donaldson, and M. Calder, "Symmetry in temporal logic model checking," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, 2006.