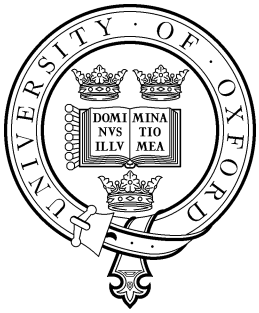
Verifying methods of bell ringing with FDR

Endre Moen

St. Cross College

Supervisor: Dr. Jeff Sanders



Thesis submitted for the degree of MSc in Computer Science
at the University of Oxford

24. August, 2004

Abstract

This project has investigated approaches to verifying methods for bell ringing. In doing so we have formally defined some of the language used by bell-ringers in describing methods, with the use of Z. A program has been developed in CSP where the verification was expressed as assertions of trace refinement. The project has also produced two Java programs. One illustrates what a method looks like, and how it sounds. Another program has been developed to search for how many methods there are on 4 bells. This program has given us a feel for how this NP complete problem of finding all methods grows with the number of bells.

Acknowledgments

I would like to thank my supervisor, Dr Jeff Sanders, for challenging me, providing me with good feedback, for being approachable, and for committing a lot of energy to make me produce a rigorous product.

I would like to thank my girlfriend, Kristin, for taking time to read through my report, and pointing out errors in spelling, grammar and style.

I would also like to thank Terry Brown and Ian Collier at the ComLab for helping me recover valuable code from my laptop. I would also like to thank Ian Collier for supplying me with excellent information on the TEX and LATEX system. He also made the front page possible by making TEX code from his thesis available to me. He is truly a wizard with computers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Outlining the Contents of the Project

What is a method? A method is a sequence of all the permutations on n bells. There are $n!$ of them. The project will produce a formal definition of important glossary used by the bell ringing community, a CSP program to represent methods on n bells, a program to exhaustively search for all methods on four bells and an illustrative program sounding the changes on a method.

A method is defined by constraints such as restricting permutations so that only consecutive bells can swap places. This is a physical constraint because the time it takes to ring a bell does not allow for bigger swaps, as it takes ca 2 seconds for a bell to swing back to its normal position. There are a total of six constraints expressed informally in English. These will make the goal to formalize. On the way we will define glossary and explain group theoretic ideas used on the way to producing specifications. The formal definitions will then be used to document and produce algorithms for different methods. Formal definitions will be written in Z. CSP and Java will be used to illustrate different approaches to algorithms.

An important process when developing these algorithms is to consider different ways of representing methods. We will consider two ways; one encouraged by the way a bell-ringer thinks about a method, which is implemented in CSP. The other is motivated by group theory, and supports a global view of a method. This is implemented in Java.

The purpose of this project is to investigate the dichotomy between global and distributed algorithms, and the two representations make the distinction of the algorithmic understanding. Typically a global algorithm is what we implement in a 3. generation programming language, and the distributed algorithm is more easily understood in terms of CSP processes.

The project will also make a search for all methods on 4 bells as this is a small enough search space that it can be attempted by a computer. It is worth noticing that on 3 bells there are 6 permutations but potentially $6! = 720$ ways

of order them so potentially 720 methods exists on 3 bells.

1.2 An illustrative example

So what does a method look like? The smallest method defined is on 3 bells, and there exists only two methods. They are called quick - and slow six:

quick	slow
123	123
132	213
312	231
321	321
231	312
213	132
123	123

Table 1.1: Methods on three bells; quick - and slow six

We see that one is the reverse of the other. As stated in the section above, there is a potential 720 methods on 3 bells, but you have now been told there exists only 2 methods on 3 bells. Hence the constraints on methods reduces the number of potential methods considerably.

Despite the fact that the constraints reduce the number of valid methods considerably, the search space for finding them all is still vast. To illustrate this, look at table 1.2. There you find a list of methods on a number of bells and their associated names, and the number of permutations in the method. To illustrate that there are many methods, and the search space for finding them is big, you will see that the number of potential methods grows $n!!$ of the number of bells. In other words $p!$ of the number in the left most column, where p is the number of permutations.

1.3 Ringing a method

Bell ringers use a variety of words to describe a method. To illustrate we will look at plain bob minimus depicted in table 1.3.

The second and third column shows permutations in the cycle notation. We will delve into this in more detail in the next chapter. This method is given by the row column. The top of one line to the bottom of the next is referred to as a hunt. The first and last permutation is called a round, and from one row to the next the new permutation is referred to as a change. Throughout when defining the glossary, we will illustrate the use mostly on methods of 4 bells for convenience of length. When ringing any method in the bell tower, the smallest bell is referred to as the treble and the largest, tenor. There are still many terms used by the bell ringing community to describe methods like bob and single, which are called at the end of a hunt, and has the effect of introducing a new permutation to the pattern of changes in the current hunt. They have the effect

bells	stage name	permutation
3 bells:	Singles	6
4 bells:	Minimus	24
5 bells:	Doubles	120
6 bells:	Minor	720
7 bells:	Triples	5040
8 bells:	Major	40320
9 bells:	Caters	362880
10 bells:	Royal	3628800
11 bells:	Cinques	39916800
12 bells:	Maximus	479001600
13 bells:	Sextuples	13!
14 bells:	14	14!
15 bells:	Septuples	15!
16 bells:	16	16!

Table 1.2: Method names and number of changes

of breaking the symmetry between hunts. As these definitions are not directly related to the work presented here we will not describe them in any more detail.

Referring to table1.2 again, ringing all the permutations on 7 bells is called a peal and takes about 3 hours. It would take years to ring all the permutations on say 12 bells. For this reason, there are many methods, even on 4 bells, which do not include all the permutations. In this project we will only consider those methods which do include all possible permutations.

row	change	permutation
1234	a	I
2143	b	(12)(34)
2413	a	(1243)
4231	b	(14)
4321	a	(14)(23)
3412	b	(13)(24)
3142	a	(1342)
1324	c	(23)
1342	a	(234)
3124	b	(132)
3214	a	(13)
2341	b	(1234)
2431	a	(124)
4213	b	(143)
4123	a	(1432)
1432	c	(24)
1423	a	(243)
4132	b	(142)
4312	a	(1423)
3421	b	(1324)
3241	a	(134)
2314	b	(123)
2134	a	(12)
1243	c	(34)
1234		

Table 1.3: plain bob minimus

Chapter 2

Bell ringing and permutations

This chapter describes theory applied to bell ringing. The purpose of this chapter is to introduce the theory applied in chapter 3 and to give examples of how it is used. This is the theory of permutations which is described in group theory. The chapter first introduces concepts and definitions in group theory. Examples making use of these definitions will be given, and finally the theory is applied as it will be used to describe methods and algorithms. The theory described here is also helpful to understand chapter 3, where we give a formal definition of methods with the use of Z. For example the definition of the dihedral group helps explain some of the Z schemas defined.

The theory outlined here is standard theory and is collected from [4] and [5]. The theory on dihedral groups and application to bell ringing is collected from [3].

2.1 Groups, Subgroups and Cosets

A group is defined to consist of a set G and a binary operation $\circ$ on G satisfying the following four properties.

1. associative: if $f, g, h \in G$ then $(f \circ g) \circ h = f \circ (g \circ h)$
2. identity: $\exists\, I \in G \bullet \forall\, a \in G \bullet a \circ I = I \circ a = a$
3. inverse: $\forall f \in G \bullet \exists f^{-1} \in G \bullet f \circ f^{-1} = I$
4. closed: if $f, g \in G$ then $f \circ g \in G$

Then the pair $G, \circ)$ is said to form a group. In this thesis we consider only examples in which the set G is finite.

We now move towards defining a coset. To define a coset we need to know what a subgroup is. Let G be a group and $S \subset G$. Then S is a subgroup if it preserves:

1. identity
2. inverse
3. closure

The subgroup 'inherits' associativity from the fact that it is a subset of a group.

If H is a subgroup then an equivalence relation is defined: $x \sim y$ iff $x \circ y^{-1} \in H$. The equivalence classes of H are called the right cosets of H and the coset containing x is written Hx. If the group is not abelian then that equivalence differs from the one defined by $x^{-1} \circ y \in H$, whose equivalence classes are called left cosets and written xH.

Finally we will state a theorem that will later help us explain properties on methods.

Theorem 1 (Lagrange) *Let G be a finite group of order n. If H is a subgroup of order m, then $m \mid n$.*

Where the order of a group is the number of elements it contain. We shall see that this explains how and why a method can be decomposed.

2.2 Application of group theory - Permutations

A permutation is a rearrangement of a sequence. E.g. $\langle 1,2,3\rangle, \rightarrow \langle 2,1,3\rangle$ is a permutation. Since we will only consider permutations on finite sequences, we can define a permutation as a bijection ($S \rightarrowtail\!\!\!\!\!\rightarrow S$) on a finite set S. It is an injection on itself ($S \rightarrowtail S$) for any set S, and on finite sets also a surjection ($S \twoheadrightarrow S$), hence a bijection.

Here is a conventional way of representing a permutation. Let $f : S \rightarrow S$, then

$$f = \begin{pmatrix} 1 & 2 & ... & n \\ f_1 & f_2 & ... & f_n \end{pmatrix}$$

where $f_1, f_2, ..., f_n$ is a rearrangement of the integers $1, 2, ..., n$. This means, $f_1 = f(1), f_2 = f(2), ..., f_n = f(n)$, and $f_i \neq f_j$ unless $i = j$. For example

$$f = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

and $f(1) = 3, f(2) = 1$, and $f(3) = 2$. So f sends 1 to $f(1)$, 2 to $f(2)$, and 3 to $f(3)$. The identity permutation then becomes:

$$I = \begin{pmatrix} 1 & 2 & ... & n \\ 1 & 2 & ... & n \end{pmatrix}$$

We now introduce the cycle notation as it is more compact than the array notation. Let $S = 1..n$ and if $a_1, a_2, ..., a_r \in S, (r \leq n)$ then the symbol

$$(a_1, a_2, ..., a_r)$$

denotes the permutation f on S which sends a_1 to a_2, sends a_2 to $a_3, ...$, and sends a_r to a_1.

The cycle notation gives us a succinct way of representing permutations on rows in methods, and explains what column - 'changes' and 'permutation' in table 1.3 represents. So why have we included group theory in the previous section? Because permutations form a group under the follow by operation, $\fatsemi$, and all the permutation on a sequence forms a group. This group has to be finite as we can only produce permutations on finite sequences, and it is called a symmetric group, S_n. Let X be any finite sequence and let S_n denote the set of all permutations of X, $(n = \#X)$, then:

$$S_n = X \rightarrowtail\!\!\!\!\!\twoheadrightarrow X$$

The symmetric group is the set with all the elements of the group, and it is finite. As stated in the first chapter, we will only consider complete methods and if we make each row of a method an element in a set, then that set is the symmetric group, S_n where n is the number of bells in the method.

A transposition is a cycle (i, j) of length 2, a 2-cycle, which interchanges i and j. The following theorem explains why methods may be produced from sequential composition of transpositions, as illustrated in table 1.3.

Theorem 2 (Steinhaus) *Any permutation can be written as a sequence of transpositions.*

So every method may be expressed as a sequence of transpositions.

When expressing methods as products of transpositions, we abbreviate the composition operator, $\fatsemi$. So to express plain bob minimus we write $I = ((ab)^3 ac)^3$ where $a = (12)(34), b = (2, 3)$ and $c = (3, 4)$. It is implicit that both $a = (12)(34)$ means $a = (12) \fatsemi (34)$, and that when a and b are transpositions, (ab) reads $(a \fatsemi b)$. To explain the notation, lets consider the first part of the expression, $(ab)^3$, which reads; apply transposition a to the permutation then apply b and repeat three times. This non-trivial expression of the identity permutation, I, defines plain bob minimus.

Because there also exists a non-trivial expression for the identity permutation, which only produces the first hunt of plain bob minimus, $I = (ab)^4$. We need another group theoretic definition applicable to describing methods. As we will understand from chapter 3, this holds for any method preserving constraint 4. This set of elements is called the dihedral group, D_n. A dihedral group, D_n is a subgroup of S_n, and as we have learned from Lagrange's theorem all subgroups of the same order partition the group.

There is a pictorial way to describe a dihedral group, let $n \geq 2$ and let R be a regular n-gon centered at the origin of a co-ordinate system. For example an equilateral triangle is a 3-gon and a square is a 4-gon. Let G denote the set of all binary transformations of the plane that preserve the figure R. This may be flipping a square over the x-axis. An example is given in figure 2.1 which produces D_4 on the symmetry lines represented as dashed lines.

A hunt has $2 * n$ elements. The dihedral group, D_n has $2 * n$ elements and if we consider the first hunt of plain bob minimus it is given by

$$D_4 = \{I, a, ab, aba, (ab)^2, a(ab)^2, (ab)^3, (ab)^3 a\}$$

Figure 2.1 produces this set by compositions of rotations on the symmetry lines. To show that it is a subset of S_4, consider the Cayley table below. Cayley tables are like the multiplication table. To find the product $a \circ b$, find a in the row and find b in the column. The cell intersecting these two lines are the product. This way a Cayley table can be used to easily verify if the set preserve the **closure** property. If there is an element in any of the cells which is not in the set, then the set is not a group on the chosen, $\circ$ operator. The Cayley table for S_4 contains all the elements from I to $((ab)^3 ac)^3$, and has $64 * 64$ entries in its Cayley table, and will therefore not be included here. However D_4 has only $8 * 8$ entries.

	I	**a**	**ab**	**aba**	$\mathbf{(ab)^2}$	$\mathbf{(ab)^2a}$	$\mathbf{(ab)^3}$	$\mathbf{(ab)^3a}$
I	I	a	ab	aba	$(ab)^2$	$(ab)^2 a$	$(ab)^3$	$(ab)^3 a$
a	a	I	b	ba	bab	$(ba)^2$	$(ba)^2 b$	$(ba)^3$
ab	ab	aba	$(ab)^2$	$(ab)^2 a$	$(ab)^3$	$(ab)^3 a$	I	a
aba	aba	ab	a	I	b	ba	bab	$(ba)^2$
$\mathbf{(ab)^2}$	$(ab)^2$	$(ab)^2 a$	$(ab)^3$	$(ab)^3 a$	I	a	ab	aba
$\mathbf{(ab)^2a}$	$(ab)^2 a$	$(ab)^2$	aba	ab	a	I	b	ba
$\mathbf{(ab)^3}$	$(ab)^3$	$(ab)^3 a$	I	a	ab	aba	$(ab)^2$	$(ab)^2 a$
$\mathbf{(ab)^3a}$	$(ab)^3 a$	$(ab)^3$	$(ab)^2 a$	$(ab)^2$	aba	ab	a	I

Note that $a = a^{-1}, b = b^{-1}$ and $c = c^{-1}$, hence $aba \circ ababa = ba$, and $b = (ab)^3 a, ba = (ab)^3, bab = (ab)^2 a, (ba)^2 = (ab)^2$ and $(ba)^a = aba$, and since I is also in the set. This set has the **identity**, and with the equalities shown also **inverse** and **closure**. Hence this is a subgroup of S_4.

Are all the hunts of plain bob minimus dihedral groups? Yes, the next transposition of plain bob minimus is $(ab)^3 ac = w$. We see that $w(D_4)$ is the second hunt, and $w^2(D_4)$ is the third, which both produce different D_4. Moreover these sets have been produced as left cosets of the first dihedral group.

We now give examples of what we have stated. You may have to refer back to table 1.3. If we index each row in the three hunts from 1 to 8 ($n = 4$ and $2 * 4 = 8$ which is the size of a hunt), and combine any permutation in the first hunt with $w = (234)$ we get the permutation to the corresponding index in the second hunt. For example if we choose index 1:

$$(234);\ (12)(34) = (132)$$

If we do the same for $w^2 = (243)$ we get

$$(243);\ (12)(34) = (1423)$$

We see from table 2.1 and figure 2.1 the correspondence between a 4-gon and the first hunt of plain bob minimus. The table is extracted from table 1.3.

If we number clockwise the square from 1 in the upper left corner we get the sequence $\langle 1, 2, 4, 3\rangle$. It is important to note that this is only one ordering of

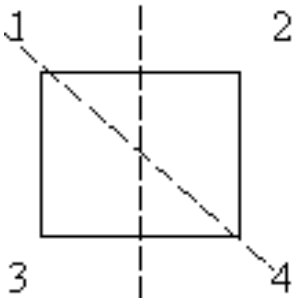


Figure 2.1: 4-gon and an example of the figure R. All combinations of rotations over the two symmetry lines produces D_4

row	change	permutation
1234	a	I
2143	b	(12)(34)
2413	a	(1243)
4231	b	(14)
4321	a	(14)(23)
3412	b	(13)(24)
3142	a	(1342)
1324	c	(23)
1342	a	(234)

Table 2.1: The row column make up the set D_4. This is an extraction from table 1.3

the square. The set of all dihedral groups is produced by unique numberings of the square but not all numberings produce a new dihedral group. For example $\langle 1,2,4,3\rangle = \langle 2,1,3,4\rangle$ as *lhs* (Left Hand Side) $\circ$ *rotate_on_vertical_line* = *rhs* (Right Hand Side), but the set of all unique dihedral groups, D_n, are the subsets of the symmetric group, S_n. This is what we saw when the first hunt of plain bob minimus was represented as D_4, and where the left cosets where produced by w and w^2 as $w(D_4), w^2(D_4)$.

In general, the decomposition of a symmetric group into dihedral groups follows by:

Theorem 3 (Campanology) *For any group G and any subgroup H, the cosets of H in G partition G.*

This theorem is a special case of the usual proof of Lagrange's theorem. We will later see that constraint 4 only consider those methods where the first hunt is a subgroup and together with a coset generator, w, produces the other hunts making up the method.

These theorems are due to [5]. The illustration is due to [2]

2.3 An illustrative example - Generating Methods

Now that we have a non-trivial expression for I on plain bob minimus, it is easy to implement the Steinhaus algorithm for $I = ((ab)^3ac)^3$ producing the method:

PlainBobMinimus

```
1  m = 4
3  bell4 = ⟨1, 2, 3, 4⟩
5  for i = 0 to m! do
6      if i + 1 ∈ odd(ℕ)
7          then transposition(bell4, a);
```

```
 8        else  if i + 1 mod 8 ≠ 0
 9                then transposition(bell4, b);
10                else  transposition(bell4, c);
11 return
```

This illustrates the importance of finding a short expression for I (except the trivial $I = (1)(2)..(n)$) when implementing methods.

2.4 Generating Sound - The Karplus Strong Algorithm

As we now have the sequence of bells to be rung, it only remains to sound them, and we will have a program to illustrate bell ringing.

$$b(t) = e^{-\alpha t}\sin(2\pi f_c t + be{-}\beta t sin(2\pi f_m t)) \tag{2.1}$$

where $f_c = 80\ Hz$, $f_m = 112\ Hz$, $\alpha = 0.06$, $\beta = 0.09$ and $b = 4$ will give the sound of a bell. This function was discovered by Alex Strong and Kevin Karplus at Stanford in 1979. See [6] for more information on this matter and how scales are built up.

The program uses this function to generate an array of $8000 * 10$ elements. The numerical array is then converted to a byte array twice the length (160.000 elements). The array is sampled between $220Hz$ and $440Hz$, which is the beginning and end of an A-major scale. This is implemented and requires between 22050 and 44100 samples per second (a Hz is sounded with 100 samples), and each sample is represented by 16 bits, so 2 elements of our byte array. This means that the tenor rings for $8000 * 10 * 2/(2 * 22050) \approx 3,63$ seconds and the treble rings for $8000 * 10 * 2/(2 * 44100) \approx 1.81$ seconds. A difference of 1.82 seconds.

If we split the tones between 220 Hz and 440 Hz (which both sound an A) in twelve we get:

$$\begin{gathered}\mathbf{220}, 233.08, \mathbf{246{,}94}, 261.63, \mathbf{277.18}, \mathbf{293.67}, \\ 311.13, \mathbf{329.63}, 249.23, \mathbf{369.99}, 391.00, \mathbf{415.31}, \mathbf{440}\end{gathered} \tag{2.2}$$

Each number represents a tone in hertz. These tones are given by splitting the line between $220Hz$ and $440Hz$ in twelve points with equal distance. This is given by the equations:

$$\begin{gathered}f_1 = cf_0, \\ f_2 = cf_1 = c^2 f_0, \\ . \\ . \\ . \\ f_{12} = cf_{11} = c^{12} f_0\end{gathered}$$

and $f_{12} = 2f_0$ and $c = 2^{1/12} \approx 1.0594$. The A-major scale is then given by:

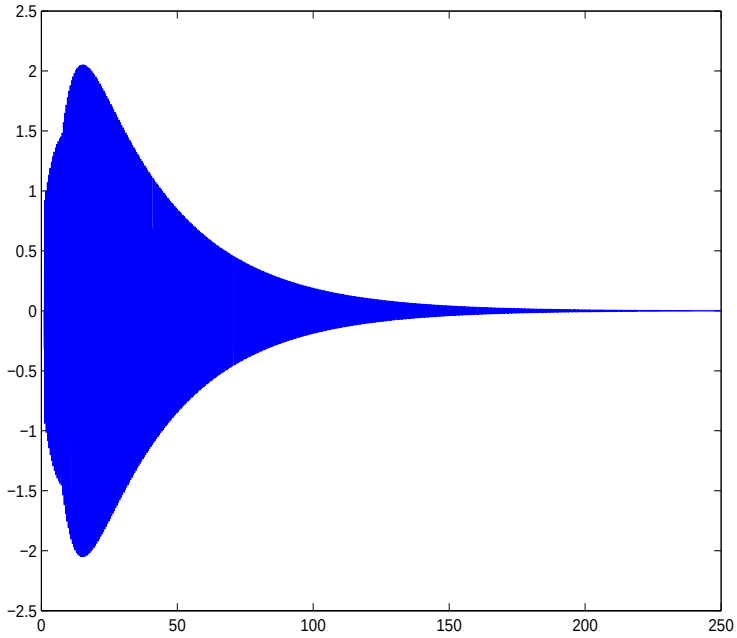


Figure 2.2: The graph of a bell, note that this graph does not range over the same length as that in the program

A	B	$C^{\#}$	D	E	$F^{\#}$	$G^{\#}$	A
f_0	f_2	f_4	f_5	f_7	f_9	f_{11}	f_{12}

Table 2.2: table of the A major, which is what is implemented in the bell ringing program.

2.5 The bell ringing program

The implementation lets there be a 0.5 second pause from the first bell is rung to the next. As we have seen above, each bell can ring between 1.8 and 3.6 seconds. There are methods from 3 to 16 bells, so a change may not take more than 1.5 seconds. The implementation therefore lets bells ring for about 1.5 seconds, and a little longer for larger bells. This is achieved by letting each bell be controlled by different threads. As the array storing the sound graph feeds the sound card with data, it stops if more than 1.5 seconds has elapsed since the start. This is an expensive operation. Therefore it is only checked after every 2000 sample has been written to the sound card, and as the sampling happens at different frequencies, a difference in the duration a bell rings is expected. The lower bells ring for a longer time. This is also how it is in the bell tower.

The Graphical User Interface (GUI) is also run as a thread to allow the user to interact with buttons and menu options as a method sounds. This gives the following picture of how the application runs when a method has been started:

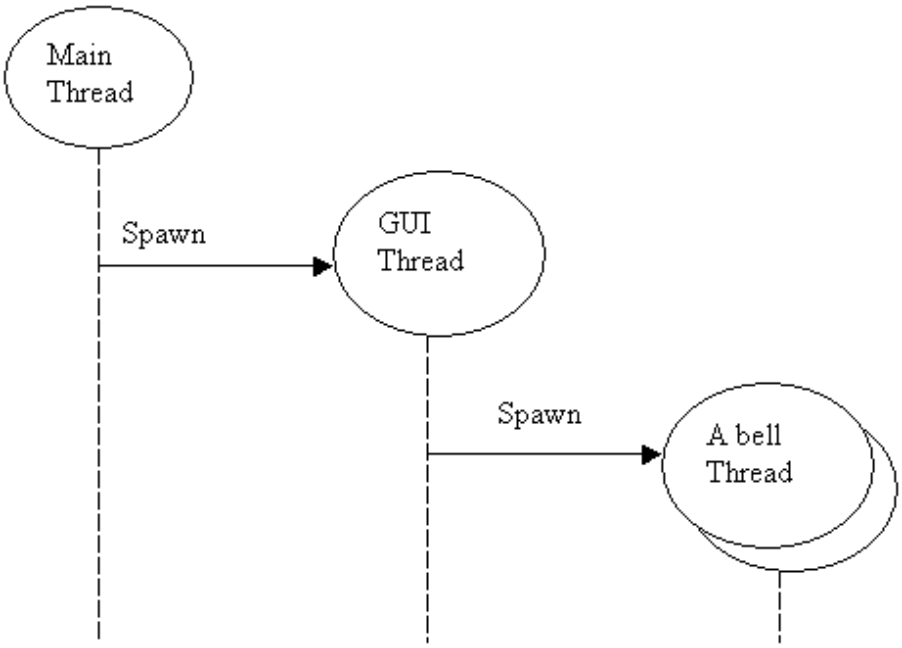


Figure 2.3: Threads started when the application is running

By double clicking on a method in the listbox with names of methods, a method will be sounded. The bell being rung will be printed as the method sounds, and a visual display is made of the key to which the method is tuned. On the "Keys" menu the user may choose to set another key. The choice is G, A, D, E and the change causes f_c to be set to 80, 85, 90 or 95. A is the default as A major has been implemented. A picture of the application can be found in figure 2.4.

A single bell can be rung by choosing one of the eight bells on the "bells" menu. The bell will be rung in either key chosen at the "Keys" menu. At any time a method may be stopped by pressing the stop button.

2.6 A program to generate all the methods on 4 bells

As part of the search for methods, a program in java was developed to search through all the possible sequences of permutations on 4 bells to find all the methods satisfying constraints 1-3. We came up with the result, 58948 methods on 4 bells.

The result of the program is presented in table 2.3 There are no methods on 22 or 23 rows but this table says something about when the constraints take effect. We see that the number of possible methods continues to rise until the 20th row, then the constraints start to reduce the numbers of possible methods.

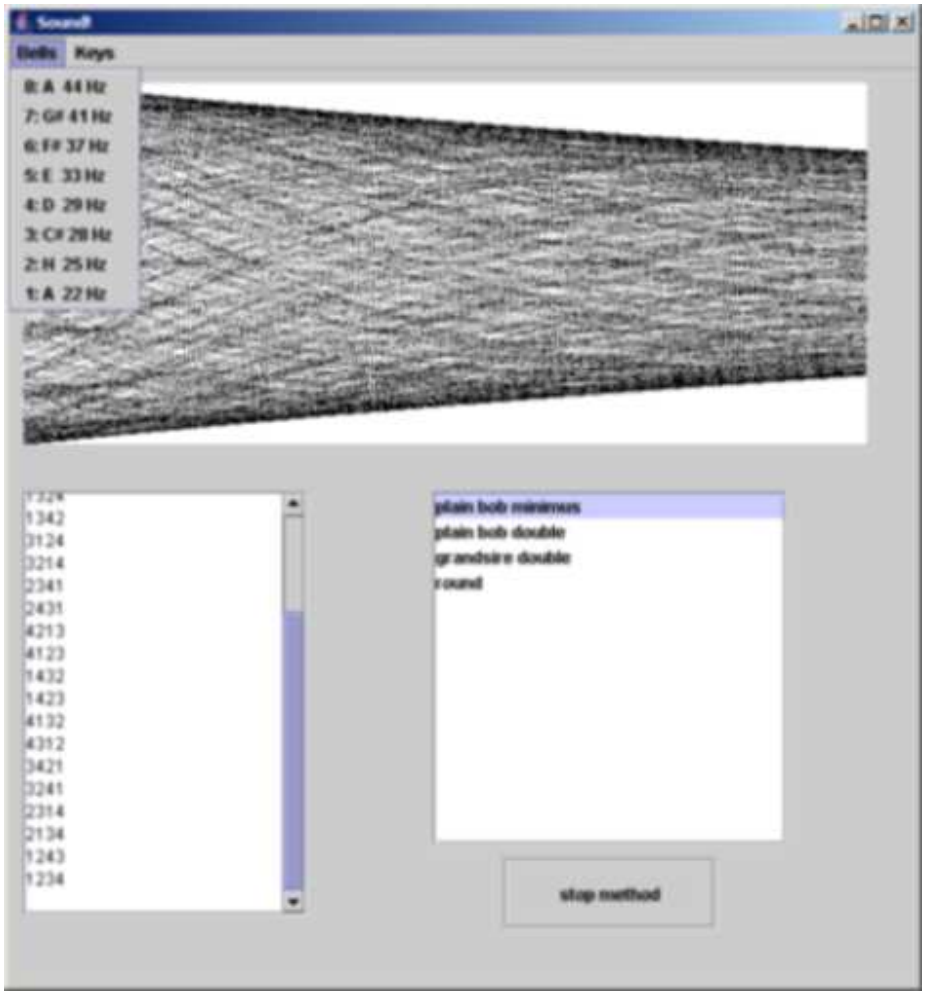


Figure 2.4: The java program - running

This program will not be discussed in any detail. It was developed out of curiosity and to illustrate that this data is very sparse. It is worth nothing that despite the exponential nature of the program, it only took 10-20 seconds to calculate these numbers. The complete program can be found in appendix C.

This program was developed early on in the project. It could have been extended further to explore the number of methods on 5 and 6 bells. However I decided to develop the project in another direction, and this is what we will discuss in the next chapter.

Number of rows	Number of methods
25	0
24	58,948
23	256,688
22	585,740
21	930,728
20	1,086,948
19	1,014,616
18	800,636

Table 2.3: Calculation from program `TransMinimusOld.java`

Chapter 3

Formalizing methods for bell ringing

3.1 Looking at constrained methods

The aim of this chapter is to formally define any method. The bell ringing community has defined methods on three to sixteen bells. We will settle on three to twelve bells, even though in the bibliography researched for this project, no conceptual difference has been found between methods on three to twelve and three to sixteen bells. We find the background for these definitions in [2]. The constraints are given by:

1. The first and last change are both rounds.
2. No other change is repeated.
3. From one change to the next, no bell changes its position more than by one.
4. The working bells each do the same work.
5. No bell occupies the same position for more than two successive changes.
6. Each hunt of the method, and perhaps the whole method, is palindromic in its sequence of transpositions. If the whole method is palindromic, this applies only to the $m!-1$ transpositions on a total of $m!$ changes.

Remember from chapter 1, that we have made the assumption that all permutations on a particular number of bells must appear in the method.

Constraints 1-3 are mandatory in any method, constraints 4-6 can sometimes be broken even though no rule has been found to when an exception is applied.

3.2 Z specification

Here we define global variables that are used in most schemas to follow.

$n : \mathbb{N}$
$m : 3..12$
$m > n$
$size\_hunt = 2 * m$
$round = \langle 1..m \rangle$

n is the beginning of an index, and m is the end of the index. This index is normally a sequence where $n = 1$, and the sequence is given by $\langle 1..m \rangle$ which represent a row in a method. The index for the beginning has been added to define the cyclic permutation $\langle 2..m \rangle$ which is used to define the working bells below, see schema $working\_cycle[m]$. $Size\_hunt$ has been included to document why $2 * m$ is used within schemas. $round$ is the first and last row of any method.

$data[n, m]$

$d : \text{iseq}(n..m)$

This schema is the foundation for all other schema defined later, and it is used to define all rows of methods. For example row $\langle 1, 2, 3, 4 \rangle$ of plain bob minimus.

$S_n[n', m]$

$ds : \text{iseq } data[n', m]$

$\#ds = (m - (n' - 1))!$

S_n is the schema that holds all the row entries in a method. It can be justified to be a symmetric group, hence the name, by letting the set be given by the imaginary function $Set(ds)$ (defined in CSP), which converts a sequence to a set and by letting the operation be $\fatsemi$. n in S_n is given by $n = m - (n' - 1)$. It will later be used in other schemas to enforce constraints 1 - 6.

Schema 2–$cycle[n, m]$ performs a single transposition on d, from $data[n, m]$, producing d'.

2–$cycle[n, m]$

$\Delta data[n, m]$
$i?, j? : 1..m$

$i? \neq j?$

$d' = d \oplus \{i? \mapsto d(j?), j? \mapsto d(i?)\}$

This schema takes two elements in the range of 1 to the number of bells, m, as input. It then performs one transposition on the sequence d. The transposition is defined by the elements $i?$ and $j?$. They must not be the same otherwise the transposition is just the identity permutation. For example to perform the

transposition $c = (3, 4)$, set $i? = 3, j? = 4$. This schema is an important building block when enforcing the constraints, and is therefore included in several other schemas to follow.

If f is any member of S_n then by Steinhaus's Theorem f can be written as a composition of transpositions, and we can now define any cycle of order $n \geq 2$ with the use of 2–$cycle[n, m]$.

Definition 1 (Any cycle) *If* $a : 2\text{–}cycle[1, m]$ *and* $\#\langle n..m\rangle \geq 2$ *then*

$$a^{\langle s..t\rangle} = a[s, \langle s..t\rangle(2)/i?, j?] \fatsemi a^{tail(\langle s..t\rangle)}[a.d/a.d']$$
$$a^{\langle s,t\rangle} = a\,[s, t/i?, j?]$$

From the definition of **any cycle**, and by the use of 2–$cycle[n, m]$, we can generate any permutation from the cycle notation, which is what we need to define working bells. The main result needed from that schema is that d, from $data[1, m]$, is transformed to d' with the use of a cycle of order $m-1$. This cycle is given by $w?$. $j?$ is a number saying how many times $w?$ must be applied to produce d' from d. The first existential quantifier defines the function *perm* as a procedure for calculating a permutation from a cycle, x, of order $m-1$ on the sequence y, of order m. The procedure makes use of the definition of **any cycle** and the schema 2–$cycle[n, m]$. This definition is the result of a theorem stating that any cycle can be decomposed into disjoint sets of transpositions.

Theorem 4 (Cycle Decomposition) *Every permutation is the product of disjoint cyclic permutations.*

The proof for this theorem can be found in [5], page 38.

The four conditions of the second existential quantifier are rules on *perm* defining how $w?$ applied $j?$ times on d compose d'.

working_cycle$[m]$

$$perm : S_{m-1} \times S_m \to S_m$$
$$\Delta data[1, m]$$
$$w? : S_{m-1}[2, m]$$
$$j? : \mathbb{N}$$

$$\exists\, a : 2\text{–}cycle[1, m] \bullet perm(x, y) = a^x[y/a.d]$$
$$\exists\, a : 2\text{–}cycle[1, m] \bullet \forall\, \rho : S_n[1, m] \bullet$$
$$perm(w^0, \rho) = \rho \wedge$$
$$perm(w^{j+1}, \rho) = a^w[\rho/a.d] \fatsemi perm(w^j, a.d') \wedge$$
$$perm(w^j, \rho) = perm(w^{j \bmod m-1}, \rho)$$
$$d' = perm(w^j, d)$$

If you look at the signature of *perm* in this schema, notice that $perm : S_{m-1} \times S_m \to S_m$ is a special case of $perm : S_x \times S_m \to S_m$ where $m \geq x$, and S_x

is then any cycle of order not greater than the sequence it is applied to permute.

We will illustrate on plain bob minimus, how working_cycle[m] uses the definition of **any cycle** and 2–*cycle*$[n, m]$ to generate the permutations $(a^{\langle n..m \rangle})$ on the sequence at the start of a hunt, which must then be the same as the beginning of the next hunt in this method. We know that $w = \langle 2, 3, 4 \rangle$ from [1]. Each hunt begins with $\langle 1, 2, 3, 4 \rangle, \langle 1, 3, 4, 2 \rangle, \langle 1, 4, 2, 3 \rangle$. We should therefore find this sequence by applying w^0, w^1, w^2 to $\langle 1, 2, 3, 4 \rangle$.

To show that $d' = perm(w^j, d)$ defines working bells, let $w? = \langle 2, 3, 4 \rangle$, $d = \langle 1, 2, 3, 4 \rangle$, $d' = \langle 1, 4, 2, 3 \rangle$, $m = 4$, and $j = 2$. Hence $w? : S_n[2, m]$, $d, d' : S_n[1, m]$ holds and we see from table 3.1:

$$
\begin{array}{ll}
\langle 1, 4, 2, 3 \rangle & \\
= & [\text{def } d' = perm(w^j, d)] \\
perm(\langle 2, 3, 4 \rangle^2, d) & \\
= & [\text{def } perm(w^{j+1}, \rho) = a^w[\rho/a.d] \fatsemi perm(w^j, a.d'),\ d = \langle 1, 2, 3, 4 \rangle] \\
a^{\langle 2,3,4 \rangle}[\langle 1, 2, 3, 4 \rangle / a.d] \fatsemi perm(w^1, a.d') & \\
= & [\text{def } a^{\langle n..m \rangle}] \\
a[2, 3, \langle 1, 2, 3, 4 \rangle / i?, j?, a.d] \fatsemi perm(w^1, a.d') & \\
= & [a.d' = \langle 1, 3, 2, 4 \rangle] \\
a[3, 4, \langle 1, 3, 2, 4 \rangle / i?, j?, a.d] \fatsemi perm(w^1, a.d') & \\
= & [\text{def } perm(w^1, a.d') = a^w[a.d'/a.d],\ a.d' = \langle 1, 3, 4, 2 \rangle] \\
a^{\langle 2,3,4 \rangle}[\langle 1, 3, 4, 2 \rangle / a.d] & \\
= & [\text{def } a^{\langle n..m \rangle}] \\
a[2, 3, \langle 1, 3, 4, 2 \rangle / i?, j?, a.d] \fatsemi a^{\langle 3,4 \rangle} & \\
= & [a.d' = \langle 1, 4, 3, 2 \rangle] \\
a[3, 4, \langle 1, 4, 3, 2 \rangle / i?, j?, a.d] & \\
= & [a.d' = \langle 1, 4, 2, 3 \rangle] \\
\langle 1, 4, 2, 3 \rangle. & \square
\end{array}
$$

Table 3.1: A procedure to calculate a permutation given a cycle and a sequence. The calculation is exemplified on the schema *working_cycle*$[m]$.

3.3 Schema - Method[m]

We now define the schema that will enforce the constraints on methods. In this schema, we will see that constraint 1 follows from the definition of M, and constraint 2 is implicit from the definition of M, $M : \text{iseq}(data[1, m]) \frown round$, and $M = ds \frown round$. The remaining constraints will be defined as schemas and will be referred to as rules. These schemas will be discussed in the sections following.

Method[m]

$S_n[1..m]$

$M : \text{iseq}(data[1, m]) \frown round$

$sequence\_of\_transpositions[M]$

$M = ds \frown round$

$M(1) = round$

$\#front(M) = m!$

$Rule3$

$Rule4$

$Rule5$

$Rule6$

For more details on the schema $sequence\_of\_transpositions[M]$, read the section called Rule6. This is also where you will find the applications of this schema.

3.3.1 Rule 3

When defining schema $working\_cycle[n, m]$ we defined the **any cycle** notation. As we saw this notation converted a permutation represented as a cycle into a sequence of transpositions. We now need do define a notation that evaluates the result of performing a sequence of transposition, where the sequence of transpositions is already given. We need this notation as a shorthand, as this is how a change in a row is defined, and this notation is needed because the change between rows may be of different lengths. Then we can state something about a permutation before and after the sequence of compositions.

Definition 2 (sequence of transpositions) *If* $a : 2\text{–}cycle[1, m]$, $\#\langle n..m\rangle \geq 2$, $\#\langle n..m\rangle \in Even(\mathbb{N})$, *and* $\#i \leq \lfloor m/2 \rfloor$ *then*

$$a_{\langle s..t\rangle} = a[s, s{+}1/i?, j?] \fatsemi a^{tail(\langle s..t\rangle)}[a.d/a.d']$$
$$a_{\langle s,t\rangle} = a^{s,t}$$

We can use this to express the change that transforms one row of a method to the next, as we know that it is a product of, sometimes, more than one transposition. This definition has also been set up to enforce constraint 3. Schema $Rule3$ has been split up in two schemas as we need to extract some of the information to produce $Rule3$ because it is needed when defining schema $Rule6$:

adjacent_swap

$$\exists\, i : \text{iseq}(1..m{-}1) \bullet \forall\, x, y : ran(i) \bullet$$
$$\#i \in Even(\mathbb{N}) \land \mid x - y \mid \geq 2 \land\ \#i \leq \lfloor \#m/2 \rfloor$$

We can now define schema $Rule3$ as:

*Rule*3

$$\forall\, k : 1..m! \bullet \exists\, a : 2\text{–}cycle[1, m] \bullet \\ \quad adjacent\_swap[m] \wedge\ a_i[M(k), M(k{+}1/a.d, a.d']$$

Let us exemplify the use of the rule defined here with the use of plain bob minimus again. We know from the previous chapter that the transpositions are $a = (1, 2)(3, 4), b = (2, 3)$ and $c = (3, 4)$. It can be difficult to see what *Rule*3 expresses as it involves a long expression of quantifiers. First k is an index to every row of a method except the last. Then schema *adjacent_swap* sets up the index to define the sequence of transpositions, which is to be used in the definition **sequence of transpositions**, and we must make sure that i is such that when it is applied to a 2-cycle, it satisfies this definition. Finally $a_i[M(k), M(k + 1)]/a.d, a.d']$ is the sequence of transpositions that transforms a row into the next. For example to express the transposition $a = (12)(34)$ on $\langle 1, 2, 3, 4\rangle$, we set $i = \langle 1, 3\rangle$ and $k = 1$, then we get $a_{\langle 1,3\rangle}[\langle 1, 2, 3, 4\rangle, \langle 2, 1, 4, 3\rangle/a.d, a.d']$.

3.3.2 Rule 4

*Rule*4

$$\forall\, j : 1..(m!/size\_hunt) \bullet \exists\, w : S_n[2, m] \bullet \exists\, W : working\_cycle[m] \bullet \\ \quad j{*}size\_hunt \in 1..m! \wedge \\ \quad W\,[round, M(j{*}size\_hunt{+}1), j, w/d, d', j?, w?]$$

To illustrate how rule 4 enforces that every bell except the hunting bell, which is the treble, is doing work, let M be plain bob minimus and let $w = \langle 2, 3, 4\rangle$. Then we must show:

$$W[round, \langle 1, 3, 4, 2\rangle, 1, \langle 2, 3, 4\rangle/the\_work?, the\_work?', j?, w?]$$
$$W[round, \langle 1, 4, 2, 3\rangle, 2, \langle 2, 3, 4\rangle/the\_work?, the\_work?', j?, w?]$$
$$W[round, round, 3, \langle 2, 3, 4\rangle/the\_work?, the\_work?', j?, w?]$$

We see this from the calculation in table 3.1. The two first working permutations has already been shown to hold for plain bob minimus. The last permutation holds by one of the rules, $perm(w^j, \rho) = perm(w^{j \bmod m-1}, \rho)$ and $perm(w^0, \rho) = \rho$.

3.3.3 Rule 5

*Rule*5

$$\forall\, i : 1..m \bullet \forall\, j : 1..m!{+}1 \bullet M(j)(i) \neq M(j{+}2)(i)$$

The fact that plain bob minimus also preserves rule 5, is best illustrated with the help of table 1.3. Go through each row and see that no bell rests in the

same position for more than two consecutive rows. This is easily explained by following the treble. It hunts up and down and only rests for two consecutive rows, when a hunt is ended.

3.3.4 Rule 6

In the next schema we need a definition. The text of constraint 6 refers to the word palindromes. It may be defined with the help of:

$$rev\langle\rangle = \langle\rangle$$
$$rev(\langle x\rangle \frown s) = rev(s) \frown \langle x\rangle$$

Then it is easy to see that a palindrome is:

palindrome[X]

$$string? : seq\ X$$

$$string? = rev(string?)$$

We now need to define the sequence that must be palindromic:

sequence_of_transpositions[M]

$$m = \#M(1)$$
$$derivation : seq\,(seq\,2\text{–}cycle[1, m])$$

$$\#M = m! + 1$$
$$\#derivation = m!$$
$$\forall\, k : 1..m! \bullet \exists\, a : 2\text{–}cycle[1, m] \bullet$$
$$adjacent\_swap[m] \land\ derivation(k) = a_i\,[M(k), M(k+1)/a.d, a.d']$$

This may seem confusing. Let us explain. Let $seq\,(seq\,2\text{–}cycle[1, m]) = ((ab)^3ac)^3$ which produces plain bob minimus. The second sequence comes from the fact that $a = (12)(34)$ is a sequence of two transpositions. Since a is transitive, that is $a = (12) \fatsemi (34) = (34) \fatsemi (12)$, as the two transpositions are disjoint we need to apply them in the same order throughout to get at what we want. That the sequence is palindromic. We know that $\#M = 4! + 1$ as M is the method. Let us see what *derivation* may contain. Let $derivation(k)$ be some $a_i[M(k), M(k+1)/a.d, a.d']$ and let it be the sequence of transpositions $(12) \fatsemi (34)$. Call it a. Then *derivation* becomes the sequence $((ab)^3ac)^3$ if M is plain bob minimus.

*Rule*6

$$\exists\, p : palindrome[2\text{–}cycle[1, m]] \bullet$$
$$(p[derivation \lhd 1..m!{-}1/string?] \lor$$
$$\forall\, k : 0..(m!/size\_hunt){-}1 \bullet$$
$$p[derivation \lhd k{*}size\_hunt..(k{+}1){*}size\_hunt/string?])$$

If we consider the example above the schema, then $palindrome[X]$ ensures that $((ab)^3ac)^2(ab)^3a = rev(((ab)^3ac)^2)(ab)^3a$. If this does not hold then it checks if $(ab)^3a = rev(ab)^3a$. In this case both hold but this must not be the case for all methods. Rule 6 says that either the whole method is palindromic or each hunt is palindromic. As each hunt of plain bob minimus is the same, it is easy to see that every hunt is palindromic too by $(ab)^3a = rev(ab)^3a$.

For background information on the syntax of Z, consult [10].

Chapter 4

CSP development

4.1 Why use FDR (Failure Divergence Refinement)?

We use FDR to implement the method constraints. We will be able to verify that any process producing methods, satisfies the constraints. With CSP it is easy to see the correspondence with Z specifications as placing processes in parallel corresponds to conjoining constraints. As we have seen from the last chapter, a method is defined by a conjunction of constraints and since some methods satisfies stronger constraints (constraints 4-6), and as we have the Z schemas defining a method, CSP becomes a very intuitive approach to method automation.

4.2 A CSP specification of bell ringing

It was thought that the first part of the CSP development should produce an implementation of constraints 1-3, and that part two should implement constraints 4-6. However it turned out that CSP was not feasible to verify any of the constraints 4-6 on any given method with FDR. Later we will see why. Let us now have a look at the implementation of constraints 1-3, and see what assertions FDR can and can not calculate. First we will have a look at how to generate S_n.

4.2.1 Generating S_n

The machine readable CSP has support for functional programming. This is what we need to express S_n as we will express it with the use of recursive functions. A mapping function takes two arguments, a function and a list. It then applies each element of the list to the function. It has the signature:

$$map : (\alpha \rightarrow \beta) \rightarrow \langle\alpha\rangle \rightarrow \langle\beta\rangle$$

where $(\alpha \rightarrow \beta)$ is the function, $\langle\alpha\rangle$ is the second argument, and map produces the list $\langle\beta\rangle$. This mapping function can be defined as:

$$map(f)(\langle\rangle) = \langle\rangle$$

$$map(f)(\langle x \rangle \frown xs) = \langle f(x) \rangle \frown map(f)(xs)$$

An example of a mapping function defined like this may be:

$$map(square)(\langle 9, 3 \rangle) = \langle 81, 9 \rangle$$

where *square* is the function that raises an element to the power of two. The material on mapping function is due to [12].

We will now define a new mapping function to be used when defining a function *perm*. *perm* is CSP defined function and is different from the function encountered in chapter 3.

$$perm : \langle 1..N \rangle \rightarrow \langle\langle 1..N \rangle\rangle$$

Where N defines the number of bells. We will later see that even though all the permutation of sequence will be given by the function $set(perm(\langle 1..N \rangle))$, *perm* satisfies:

$$bag(perm(\langle 1..N \rangle)) = (bag)\ set(perm(\langle 1..N \rangle))$$

We shall now describe the functions composing *perm* and we shall give an example to illustrate why *perm* satisfies this stronger condition.

We need a mapping function, *insNum*, to produce a list of lists from an element, and a list. The idea is to produce unique permutations as lists by inserting the element in front of the list, between every element of the list and at the back of the list. In our definition we need a mapping function that takes a function as its first argument, and the second argument takes an element and a sequence. This signature is defined in CSP as:

$$mapPre(f)(s, t) = \langle f(s, y) \mid y \bullet t \rangle$$

mapPre is a function that will apply f to the two elements s and y where y is an element of t. For our purpose f is the function *prepend* which takes an element and a sequence as argument. The second argument of the mapping function will therefore be an element and a sequence of sequences. The product of *mapPre* is also a sequence of sequences. S_4 is defined by the constant S.

$$N = 4$$
$$S = set(perm(\langle 1\ ..\ N \rangle))$$

$$insNum(n, \langle\rangle) \equiv \langle\langle n \rangle\rangle$$
$$insNum(n, xs) \equiv \langle\langle n \rangle \frown xs\rangle \frown mapPre(prepend)(head(xs), insNum(n, tail(xs)))$$

$$prepend(x, xs) \equiv \langle x \rangle \frown xs$$

These definitions can be found in the file `newMethod3.csp` in appendix A. *set* is a function built into FDR and is defined as the function that converts a sequence into a set. *prepend* is the function that concatenates an element to the beginning of a sequence. $insNum(n, xs)$ inserts n between every element of the sequence xs producing a sequence of $\#xs{+}1$ sequences, each of length $\#xs{+}1$. To illustrate how this recursive definition uses *mapPre* consider table 4.1.

$$
\begin{array}{lcl}
insNum(1, \langle 2,3,4 \rangle) & = & \langle \langle 1,2,3,4 \rangle \rangle \frown mapPre(prepend)(2, insNum(1, \langle 3,4 \rangle)) \\
insNum(1, \langle 3,4 \rangle) & = & \langle \langle 1,3,4 \rangle \rangle \frown mapPre(prepend)(3, insNum(1, \langle 4 \rangle)) \\
insNum(1, \langle 4 \rangle) & = & \langle \langle 1,4 \rangle \rangle \frown mapPre(prepend)(4, insNum(1, \langle \rangle)) \\
insNum(1, \langle \rangle) & = & \langle \langle 1 \rangle \rangle \\
 & & \\
insNum(1, \langle 4 \rangle) & = & \langle \langle 1,4 \rangle \rangle \frown mapPre(prepend)(4, \langle \langle 1 \rangle \rangle) \\
 & = & \langle \langle 1,4 \rangle, \langle 4,1 \rangle \rangle \\
insNum(1, \langle 3,4 \rangle) & = & \langle \langle 1,3,4 \rangle \rangle \frown mapPre(prepend)(3, \langle \langle 1,4 \rangle, \langle 4,1 \rangle \rangle) \\
 & = & \langle \langle 1,3,4 \rangle, \langle 3,1,4 \rangle, \langle 3,4,1 \rangle \rangle \\
insNum(1, \langle 2,3,4 \rangle) & = & \langle \langle 1,2,3,4 \rangle \rangle \frown mapPre(prepend)(2, \langle \langle 1,3,4 \rangle, \langle 3,1,4 \rangle, \langle 3,4,1 \rangle \rangle \\
 & = & \langle \langle 1,2,3,4 \rangle, \langle 2,1,3,4 \rangle, \langle 2,3,1,4 \rangle, \langle 2,3,4,1 \rangle \rangle
\end{array}
$$

Table 4.1: Recursive evaluation of the function $insNum(1, \langle 2,3,4 \rangle)$.

So n has been inserted between each element of the sequence xs. If we have a look at table 4.1 again. At the *rhs* of the last line, before the empty line, we insert the definition which does not contain *insNum* into the very last argument of *mapPre* from the line above, and evaluate the new expression below the empty line. We repeat this process until the definition is given.

All the unique permutations are then produced with the function *perm*.

$$
\begin{array}{l}
perm(xs) \equiv \\
\quad \text{if } \#xs = 2 \text{ then} \\
\quad\quad \langle xs, tail(xs) \frown \langle head(xs) \rangle \rangle \\
\quad \text{else } f(head(xs), perm(tail(xs)))
\end{array}
$$

perm produces a sequence of sequences from argument sequence, and in function f, *xss* is a sequence of sequences.

$$
\begin{array}{l}
f(n, xss) \equiv \\
\quad \text{if } \#xss \neq 1 \text{ then} \\
\quad\quad insNum(n, head(xss)) \frown f(n, tail(xss)) \\
\quad \text{else } insNum(n, head(xss))
\end{array}
$$

To see how *perm* produces all the permutations on a given sequence, consider table 4.2

Each call to *insNum* with $\# = 3$ produces 4 unique sequences. There are 6 unique sequences in the sequence of sequences for which *insNum* is applied to with $n = 1$ as $\#xss = \#perm(\langle 2,3,4 \rangle) = 6$. This will produce $4 * 6 = 24$ unique sequences. By applying the function *Set*, which converts a sequence into a set, $Set(perm(\langle 1..N \rangle)), N = 4$, and we will have S_4.

We will see in the next section why the set, $set(perm(\langle 1..N \rangle))$, is needed as we will make use of the set when implementing constraint 2, see page 20, as a process. This will be described in the section to follow. As we have seen from chapter 2, the methods we will consider are all those where the rows compose the elements of a symmetric group and the operation is $\fatsemi$. Likewise, in our CSP implementation, *perm* generates the elements which can be used to represent

$$
\begin{array}{lcl}
perm(\langle 1,2,3,4\rangle & = & f(1, perm(\langle 2,3,4\rangle \\
perm(\langle 2,3,4\rangle) & = & f(2, perm\langle 3,4\rangle) \\
perm(\langle 3,4\rangle) & = & \langle\langle 3,4\rangle, 4, 3\rangle \\
 & & \\
f(2, \langle\langle 3,4\rangle, 4, 3\rangle) & = & insNum(2, \langle 3,4\rangle) \frown f(2, tail(xss)) \\
insNum(2, \langle 3,4\rangle) & = & \langle\langle 2,3,4\rangle, \langle 3,2,4\rangle, \langle 3,4,2\rangle\rangle \\
f(2, tail(xs)) & = & \langle\langle 2,4,3\rangle, \langle 4,2,3\rangle, \langle 4,3,2\rangle\rangle \\
f(2, perm(\langle 3,4\rangle)) & = & \langle\langle 2,3,4\rangle, \langle 3,2,4\rangle, \langle 3,4,2\rangle, \langle 2,4,3\rangle, \langle 4,2,3\rangle, \langle 4,3,2\rangle\rangle \\
f(1, perm(\langle 2,3,4\rangle) & = & insNum(1, \langle 2,4,3\rangle) \frown tail(1, xss) \\
insNum(1, \langle 2,4,3\rangle) & = & \langle\langle 1,2,4,3\rangle, \langle 2,1,4,3\rangle, \langle 2,4,1,3\rangle, \langle 2,4,3,1\rangle\rangle \\
... & & \\
Set(perm(\langle 1,2,3,4\rangle)) & = & S_4
\end{array}
$$

Table 4.2: Functional definition of S_4.

rows and $set(perm(\langle 1..\rangle))$ will produce all the elements of a symmetric group and the operation will be the same again, $_9$.

4.2.2 The processes

We now embark on defining the process that produce methods. A process to model that only adjacent bells can swap, will look like figure 4.1

Figure 4.1: Positions as processes
Assuming only four bells are used. The circles represent a bell. The numbers indicate positions a bell may be in

Each bell is modelled as a process that can do two things:

1. Synchronize on α, β or γ with a neighbor bell to swap place.
2. Do a *noop* event.

The channels used by the processes of this program are:

$channel\ addS, isinS : S$
$channel\ yes, no$
$channel\ a, b, g$
$channel\ row$
$channel\ noop$
$channel\ p\_st, p\_nd : T$
$channel\ q\_st, q\_nd : T$
$channel\ r\_st, r\_nd : T$
$channel\ assign : T.T$
$channel\ sync$

All bells must finish one event before starting on the next event. This is why we have a *noop* event, simply to give a representation of each row in a

method. The row like structure is then supported by the event *row*. Each bell is modelled by the processes:

$$
\begin{aligned}
x(1,i) = \; & row \rightarrow \\
& (\alpha \rightarrow p\_st!i \rightarrow p\_nd?j \rightarrow sync \rightarrow assign!1!j \rightarrow x(1,j) \; \Box \\
& noop \rightarrow sync \rightarrow assign!1!i \rightarrow x(1,i) \;) \\
x(2,j) = \; & row \rightarrow \\
& (\alpha \rightarrow p\_st?i \rightarrow p\_nd!j \rightarrow sync \rightarrow assign!2!i \rightarrow x(2,i) \; \Box \\
& \beta \rightarrow q\_st!j \rightarrow q\_nd?k \rightarrow sync \rightarrow assign!2!k \rightarrow x(2,k) \; \Box \\
& noop \rightarrow sync \rightarrow assign!2!j \rightarrow x(2,j) \;) \\
x(3,k) = \; & row \rightarrow \\
& (\beta \rightarrow q\_st?j \rightarrow q\_nd!k \rightarrow sync \rightarrow assign!3!j \rightarrow x(3,j) \; \Box \\
& \gamma \rightarrow r\_st!k \rightarrow r\_nd?l \rightarrow sync \rightarrow assign!3!l \rightarrow x(3,l) \; \Box \\
& noop \rightarrow sync \rightarrow assign!3!k \rightarrow x(3,k) \;) \\
x(4,l) = \; & row \rightarrow \\
& (\gamma \rightarrow r\_st?k \rightarrow r\_nd!l \rightarrow sync \rightarrow assign!4!k \rightarrow x(4,k) \; \Box \\
& noop \rightarrow sync \rightarrow assign!4!l \rightarrow x(4,l) \;)
\end{aligned}
$$

Now we are ready to compose a process that satisfies constraint 3, page 20, this is done by the process:

$$
\begin{aligned}
start = (\; (\; (\; & x(1,1) \; [\![\, \{\!|\; a, row, sync, p\_st, p\_nd \;|\!\} \,]\!] \; x(2,2) \;) \\
& [\![\, \{\!|\; b, row, sync, q\_st, q\_nd \;|\!\} \,]\!] \; x(3,3) \;) \\
& [\![\, \{\!|\; g, row, sync, r\_st, r\_nd \;|\!\} \,]\!] \; x(4,4) \;)
\end{aligned}
$$

This is a very liberal process that only insists that adjacent bells swap order. We do yet not have a notion of methods as constraints 1 and 2 has not been implemented. It is still feasible to talk about this process as producing methods because given the right set of transpositions, α, β, and γ, this process may preserve constraints 1 and 2 as well.

To explain, the process identity $x(1,i)$, for example should be read; bell i is in position 1. Intuitively this is best understood by reading each process $x(*,*)$ as a process in possession of a fixed position, and where bells with numberings i, j, k and l that might visit these positions.

If a bell moves to the left it will first communicate its current position to the environment. This can be seen from $x(i,1)$ which can only send its bell to the left. Therefore it does the events

$$p\_st!i \rightarrow p\_nd?j$$

which can be read; bell i would like to move to someone that synchronizes with $p\_st!i$, and I - the $x(1,i)$ process, will hold the bell j, for which someone will synchronize with $\{\!|\; p\_nd \;|\!\}$. $\{\!|\; p\_nd \;|\!\}$ is more liberal than for example $p\_nd!j$, as it can take on any event, i, j, k or l, from channel p_nd. The new value $x(*,*)$ takes on, is indeed j, as can be seen from the recursive call $x(1,j)$ at the end of the chain of events from the process with identifier $x(1,i)$.

p_st stands for the first, and *p_nd* for the second of the '*p*' events. Likewise on process $x(2,j)$, we see that bells that would like to move to the right must do

the opposite. They must first communicate the bell they would like to receive. We can see this from:

$$x(2,j) = ... \rightarrow p\_st?i \rightarrow p\_nd!j \rightarrow ...$$

This way, $x(1,i)$ and $x(2,j)$, can synchronize on the channel 'p'. This structure is implemented with each position. A bell in the middle position can synchronize on two different channels for each of its two neighbors. This enforces constraint 3, and justifies viewing the process *start* as a line with processes as depicted in figure 4.1. The events α, β and γ are not required, but they provide a nice abstraction for the operators $[p/q/r]\_st \rightarrow [p/q/r]\_nd$ as they link to the events given in this figure. They are also used by other processes, and makes refinement checks more succinct and intuitive to understand.

A *noop* event can be performed by a bell when it does not wish to change its current position. If the process does a *noop* event it is readily seen that this indeed does not change the position of a bell because the recursive calls in all the processes are the same as the first call.

The event *sync* is needed so that all swapping of places has been resolved before the event $\{|\ assign\ |\}$ takes effect. The event $\{|\ assign\ |\}$ has the effect of recording all the bells positions. Hence recording a row. This is done before the next *row* event may take effect.

A refinement:

$$\begin{array}{l}\textsf{assert}\\ \quad start \setminus \\ \qquad \{|\ p\_st, p\_nd, q\_st, q\_nd, r\_st, r\_nd, noop, sync, assign\ |\} \\ \qquad\quad \sqsubseteq_T\ pBobMini\end{array}$$

validates that *start* can indeed produce methods, as *start* is clearly more liberal than process *pBobMini*. Think of this process as the one that produces method plain bob minimus, only. The process will be outlined in more detail later in this chapter.

Constraint 2

To model constraint 2, 'No other change is repeated', we must record each change before a row and add this to a set. We start with the empty set and then add elements. If the element is already in the set, the process deadlocks because this can not be the chiming of a method.

The elements that are allowed in the set are all elements of the symmetric group, S_n. This is why we produced the symmetric group with *perm* at the beginning of this chapter. It now only remains to record the change before each row event, which is done by storing each row as an element of the sequence, $\langle i,j,k,l\rangle$, where each of the variables corresponds to the bell held by the processes $x(1,i), x(2,j), x(3,k)$ and $x(4,l)$, (in general by the processes

$(x(1,*)..x(N,*))$. This is done with the events $\{|\ assign\ |\}$ which we met when discussing the process *start* above.

A process to model a set with the limited *add* and *test−for−membership* operations looks like:

$$SSet(X) = addS?x \rightarrow SSet(union(X,x)) \ \Box \\ isinS?x \rightarrow (\text{if } member(x,X) \text{ then } yes \text{ else } no) \rightarrow SSet(X)$$

To get a row of data we have the process:

$$get = row \rightarrow sync \rightarrow assign!1?i \rightarrow assign!2?j \rightarrow assign!3?k \rightarrow assign!4?l \rightarrow \\ (isinS.\langle i,j,k,l\rangle \rightarrow ((no \rightarrow addS.\langle i,j,k,l\rangle \rightarrow get) \ \Box \\ (yes \rightarrow STOP)))$$

and in order to validate the uniqueness of each row is unique we must store this data, which is done by:

$$gget = SSet(\{\}) \ [\![\{|\ isinS, addS, yes, no\ |\}]\!] \ get$$

This must all be applied to the method we will produce, therefore:

$$constraint2 = gget \ [\![\{|\ row, assign, sync\ |\}]\!] \ start$$

The observant reader might have noticed that in the processes $x(*,*)$ only the new bell positioning is recorded. For example the process $x(1,i)$ does an $assign!1!j$, but not an $assign!1!i$. Hence the initial positioning of the bells will not be recorded. However, constraint 1 states that the first and last change shall be the same. This way we record the round to the *SSet* at the very end of a method, ensuring that the process $constraint2$ preserves constraint 1.

4.2.3 process *pBobMini*

We will now discuss the process *pBobMini* which implements $((ab)^3ac)^3$ by doing the events α , β and γ.

$$A = \langle 0,1,3\rangle \\ B = \langle 0,2\rangle \\ C = \langle 0,3\rangle$$

Permutation a is a cross-change. Remember $a = (12)(3,4)$. We need to do event α which does permutation (12), then event γ which does permutation $(3,4)$. Hence we have the mappings $a = \alpha \fatsemi \gamma$, $b = \beta$ and $c = \gamma$. What sequences A, B, and C does is mapping these permutations to the corresponding events. The mapping of the sequence to the events is done by process h. It also takes a second argument saying how many times the sequence composed should be applied:

$$pBobMini \equiv h(A \frown B,3) \fatsemi h(A \frown C,1) \fatsemi h(A \frown B,3) \fatsemi \\ h(A \frown C,1) \fatsemi h(A \frown B,3) \fatsemi h(A,1) \fatsemi STOP$$

$$h(xs,m) \equiv h1(xs,m,0)$$

$h1(xs, m, j) \equiv$
if $(m \neq 0)$ then
if $(j! = \#\, xs)$ then
$(nth(j, xs) = 0)$ & $row \rightarrow h1(xs, m, j{+}1)$ $\Box$
$(nth(j, xs) = 1)$ & $\alpha \rightarrow h1(xs, m, j{+}1)$ $\Box$
$(nth(j, xs) = 2)$ & $\beta \rightarrow h1(xs, m, j{+}1)$ $\Box$
$(nth(j, xs) = 3)$ & $\gamma \rightarrow h1(xs, m, j{+}1)$
else $h1(xs, m{-}1, 0)$
else $SKIP$

$nth(u, xs) \equiv$
if $u = 0$ then
$head(xs)$
else $nth(u{-}1, tail(xs))$

There exists an alternative more succinct implementation of plain bob minimus called *newBobMini*, which can be found in appendix A, file `newMethod3.csp`. However, I have decided to keep this implementation as it both makes use of the $\fatsemi$ operator and the process $STOP$. The process h makes use of the function nth which can be found in the reference material of [11].

Finally we should be able to verify whether a process describing a method preserves all the constraints 1-3. If our process is *pBobMini* we may express this by:

```
assert
  constraint2 \
      {| p_st, p_nd, q_st, q_nd, r_st, r_nd, noop, sync, assign |}
          [=_T pBobMini
```

There are over 4!! methods and trying to verify this with FDR requires more than 14 million transitions. It takes days, possible weeks to evaluate this expression. However I have had more success with ProBE(Process Behavior Explorer). There is a branch from the process *constraint2* which behaves just like plain bob minimus. This proves the assertion.

The complete implementation can be found in appendix A.

4.3 An attempt at implementing constraint 4

At the beginning of last chapter we stated that the goal of the CSP development was to implement all the constraints and verify them with FDR. We will now discuss why the implementation of constraints 4-6 has not been successful. Firstly, let us consider an extension to the development of the last section which verifies that the first hunt of plain bob minimus is $w = \langle 2, 3, 4 \rangle$. Only the updated and new code will be presented.

$ROWS = \{1..26\}$

This set should more generally be defined $\{1..N+2\}$, as the set should be as small as possible because it is the main source of trouble for FDR. From the name you can see that this set will represent row numbers in a method, and the row event takes the new type:

$$channel\ row : ROWS$$

Since we are working with 4 bells, methods contain 25 rows (4!+1). Some counter examples of constraint 2 will not be possible if we only allow for 25 rows, as all processes of methods should deadlock after 25 rows. If it is not a method it might continue beyond 25 rows, and if it does, we immediately understand that this process does not represent a method, so 26 rows is enough. The new change to channel *row* means that whenever FDR needs to calculate a refinement, it must go through the whole search tree on the particular number of bells. This means an upper bound of 4!! leaf nodes.

$$w1 = \quad \langle 1,3,4,2 \rangle$$

$$\begin{aligned} c4 = {} & row?any \rightarrow c4 \ \Box \\ & assign!1!1 \rightarrow c4 \ \Box \\ & assign!2!3 \rightarrow c4 \ \Box \\ & assign!3!4 \rightarrow c4 \ \Box \\ & assign!4!2 \rightarrow c4 \ \Box \\ & row!8 \rightarrow assign!1!1 \rightarrow assign!2!3 \rightarrow assign!3!4 \rightarrow assign!4!2 \rightarrow c4 \end{aligned}$$

$$\begin{aligned} & working\_bell = \\ & \quad c4 \ [\![\{\!| \ row, assign.1.1, assign.2.3, assign.3.4, assign.4.2 \ |\!\}]\!] \ constraint2 \end{aligned}$$

This development has the additional assertion:

```
assert
  working_bell
  {| p_st, p_nd, q_st, q_nd, r_st, r_nd, noop, sync, assign |}
      [=T row!9 -> STOP
```

The assertion only checks that the 8'th row is the change $\langle 1,3,4,2 \rangle$. We need a more general implementation saying that for any method which preserve constraint 4, see page 20, so there exists a w, it will not deadlock with this process. Let us call it *constraint*4. The philosophy is that you should enter this w to the program and *constraint*4 will verify that the permutation w is satisfied for every hunt in any methods the process produces. This approach is much the same as we have previously implemented methods on 4 bells. For example it is easy to extend the previous development to work for 5 or 6 bells by setting $N = 5$ and adding another process $x(5, m)$, and then adding this process to the processes *start* and *constraint*2.

This development compiles, but FDR cannot verify the trace refinement even if we set $Row = \{1..10\}$ ($10! = 3628800$), as it again takes more than 14 million transpositions to calculate. The development can be found in appendix A with

filename `newMethod3b.csp`.

We shall now embark on a generalization of this development. Again I shall only include and discuss those lines new to the development. This program can also be found in appendix A with filename `newMethod4.csp`.

$$W = set(perm(\langle 2..N \rangle))$$

$$tmp = fact(N)/(2 * N)$$
$$II = \langle 1..tmp \rangle$$

From the example of file `newMethod3b.csp` we saw that the user will have to enter a specific w to have process $c4$ produce methods satisfying constraint 4, and with hunts of this particular w, only. To satisfy constraint 4 in the most general form, process $c4$ should accept any permutation w as long as it is a $(n-1) - cycle$. We use W for this, as it contains the set of all $(n-1) - cycle$ represented as sequences. We also need the sequence containing the sequence from 1 to the number of hunts in a method. So we can check that every hunt preserves the permutation w. This is given by the set II. Because the expression $II = \langle 1..(fact(N)/2N) \rangle$ gives a parse error in FDR the variable $tmp = fact(N)/2N$ is needed. $fact$ is the normal factorial function:

$$fact(1) = 1$$
$$fact(x) = x * fact(x - 1)$$

II is always a sequence because it is an invariant that $n! \mid 2n$.

We have seen that the event row takes a new type, and $x(1, i)$ now looks like:

$$x(1, i) = row?ct \rightarrow (a \rightarrow ...$$

where ct is defined by the process:

$$countRow(ct) = row!ct \rightarrow countRow(ct + 1)$$

This results in the following update to the process $start$

$$\begin{aligned} start = (\,(\,(\,x(1,1) \,[\![\, \{\!|\, a, row, sync, p\_st, p\_nd \,|\!\} \,]\!]\, x(2,2)\,) \\ [\![\, \{\!|\, b, row, sync, q\_st, q\_nd \,|\!\} \,]\!]\, x(3,3)\,) \\ [\![\, \{\!|\, g, row, sync, r\_st, r\_nd \,|\!\} \,]\!]\, x(4,4)\,) \\ [\![\, \{\!|\, row \,|\!\} \,]\!]\, countRow(1) \end{aligned}$$

This means that every time the a row event is performed it increments the counter ct, which starts from 1.

$$\begin{aligned} &tmp2(xx) = xx * 2 * n \\ &PP(ii) = row!(tmp2(head(ii))) \rightarrow ...algorithm - to - get - (n-1) - cycle... \rightarrow \\ &\qquad isinS.\langle y, z, x \rangle \rightarrow (yes \rightarrow PP(tail(ii)) \,\Box \\ &\qquad\qquad\qquad no \rightarrow STOP) \end{aligned}$$

Process $PP(ii)$ should get the 2N+1'th row and calculate the $(n-1) - cycle$ from the start of the previous hunt. It was hoped that the same approach of

naming a temporary variable would solve the problem of having expressions with expression. However FDR and ProBE does not like variable $tmp2$ which is needed in process $PP(ii)$.

Where this to work we could enforce constraint 4 by:

$$working\_bell = SSet(W) \,[\![\, \{|\, isinS, yes, no \,|\} \,]\!]\, PP(II)$$

$$constraint4 = constraint2 \,[\![\, \{|\, isinS, assign \,|\} \,]\!]\, working\_bell$$

It should be noted that constraint 5 and 6 also needs to make use of the new row event so there is no reason for attempting further developments of these.

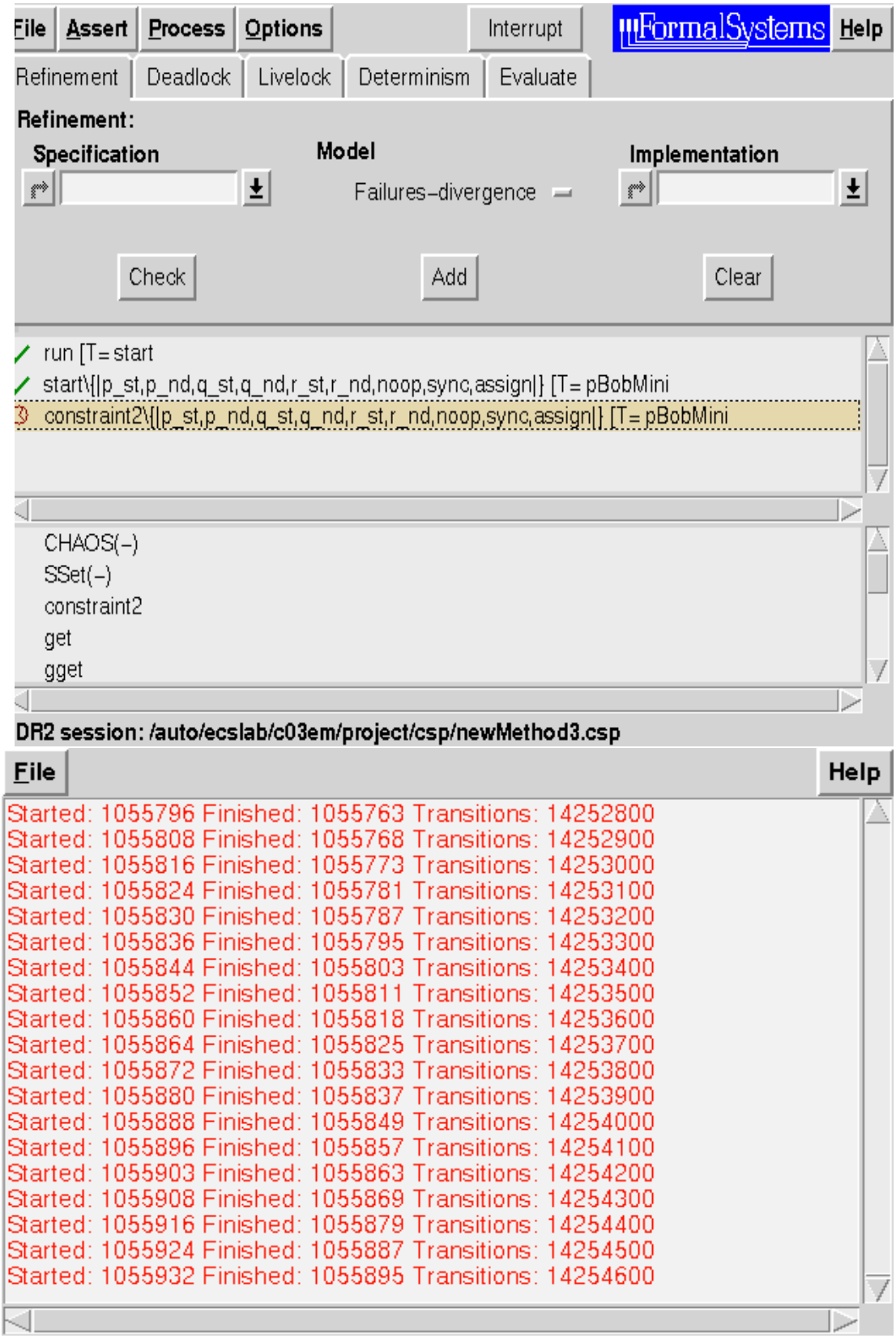


Figure 4.2: `newMethod3b.csp` in FDR

Chapter 5

Conclusion and findings

I started the project by reading to find out what bell-ringing was. As I read, I had an idea of a Java like program that might produce all methods on up to 8 bells. After all there is only 2 methods on 3 bells, so the number of methods on 8 bells should not be infeasible, but as it stands we have only generated all the methods on 4 bells. It is a fault of the project that it only talks about methods on 4 bells. It does so because methods on 4 bells are the most convenient to write out, only 25 rows, as 5 bells would take 121 rows to write out.

In the project description, unfortunately not available at the University of Oxford computer lab web page (http://www.comlab.ox.ac.uk) any longer, it states; 'it is the purpose that each bell be modelled by a process, $Bell(i, xs)$, ... the first part of the project consists of determining the sequence xs for those [Bells] ... '. I believe the philosophy behind this statement is to support the idea of a distributed algorithm. This sequence is what we have expressed when for example we have stated $I = ((ab)^3ac)^3$, even though the sequence takes a different, more general form than that expressed in the project description. Also I believe the answer to this question is completely answered by the 6 constraints stated in chapter 3. A lot of the project work has evolved around these constraints. However, it was not so clear to me at the beginning of the project that this should be the direction to go. Much time has been spent on trying to find out how bell-ringers communicate methods between themselves, how they think about methods and what vocabulary they use to express methods. The idea a bell-ringer has of how a method is rung is best described as a line following one bell through a table representing a method as depicted in figure 5.1.

This is the most natural way for a bell-ringer to think of a method because he/she needs to know at what position to call the bell at any time. This supports a distributed way of thinking about bell-ringing as the bell ringer can easily be represented as a process.

An example of the vocabulary used by bell-ringers, is the term bob and single. These are called out by a the person in charge of the method. This happens at the end of some hunts. An attempt of expressing this formally was given up when I realized that they had slightly different meanings in different methods. This was also the point at which the bell ringers idea of representing methods

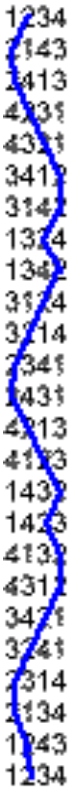


Figure 5.1: The line through the rows are the positions a bell-ringer must call his bell.

was abandoned in advantage to the more group theoretic. So the project has eventually become to specify in Z and implement in CSP the 6 constraints describing methods.

The second part of the project was to verify that the constraints held for the CSP implementation with the help of FDR. This part has largely been unsuccessful. None of the constraints, except constraint 3, has been verified by FDR. This is because the search space is too large. I believe implementing the constraints in a third generation programming language like Java, as has been done with the Java program searching for methods described at the end of chapter 2. I think a generalization to this program may allow for any number of bells up to 7 or 8 should be feasible. It would be nice to find out how may bells such a program would be able to calculate under constraints 1-3 first then under them all. At some point during the project work, my supervisor Dr Jeff Sanders and I tried to calculate the number of methods from the number of bells. We still only know the upper bound, $n!!$, where n is the number of bells. I believe the data from this program would be very helpful in getting closer at finding an expression for the number of possible methods on any given set of bells.

The project description states that the third and last part of the project should investigate whether the constraints could be met in more novel ways and what new styles of methods are possible. This is to some degree answered by the fact that constraints 4-6 are sometimes optional. Hence the set of methods under constraints 1-3 are all the methods there exists. However this part has not been completed, or even been described in any depth. The papers by White give an introduction to this part of the project. Even though in chapter 2 we talk about decomposing methods into cosets, we have not applied this to methods in any way. This is what I imagined would be needed in getting closer at an answer.

This report has described a few bell ringing terms and described them in a formal way. It has got a little closer in providing a vocabulary to describe bell ringing in a formal language.

Bibliography

[1] A. T. White, Fabian Stedman: The First Group Theorist?, The American Mathematical Monthly, nov. 1996, p 771-778

[2] A. T. White, Ringing the Cosets, The American Mathematical Monthley, oct 1987, p 721-746

[3] A. T. White and R. Wilson, The Hunting Group, Mathematical Gazette, 1995, p 5-16

[4] G. Birkhoff and S. MacLane, A survey of modern algebra, macmillan 1961

[5] D. Joyner, Adventures in Group Theory, The Johns Hopkins University press, 2002, p 41-46

[6] K. Mørken, Kompendium med Prosjektoppgaver for MAT100b, University of Oslo, http://heim.ifi.uio.no/ knutm/MAT100B/h2002/, p 83

[7] E.S. and M. Powell, The Ringers' Handbook", Whitehead & Miller Ltd, Leeds

[8] R. Robertson, The Minimus handbook, 1978

[9] Technical Info, Visit the Swan Bells, http://www.swanbells.com.au/technical_info_bells.htm

[10] J. Woodcock and J. Davies, Using Z specification, refinement, and proof, Prentice Hall, 1996

[11] B. Roscoe, The Theory and Practice of CSP, Prentice Hall, 1998

[12] R. Bird, Introduction to Functional Programming usin Haskell, Prentice Hall, 1998

Appendix A

CSP development

A.1 CSP program for constraint 1-3 - newMethod3.csp

```
-- created by Endre 7.7.04

mapPre(f)(s,t) = <f(s,y) | y <- t>

N = 4
T = {1..N}
S = set(perm(<1..N>))

insNum(n,<>) = <<n>>
insNum(n, xs) = <<n>^xs>^mapPre(prepend)(head(xs), insNum(n, tail(xs)))
prepend(x, xs) = <x>^xs

-- Produces a recursive definition of the symmetric
-- group S_N
perm(xs) = if #xs == 2 then
     <xs,tail(xs)^<head(xs)>>
        else f(head(xs), perm(tail(xs)))

f(n, xss) = if #xss != 1 then insNum(n, head(xss))^f(n, tail(xss))
    else insNum(n, head(xss))

channel addS, isinS : S
channel yes,no
channel a, b, g
channel row
channel noop
channel p_st, p_nd : T
channel q_st, q_nd : T
channel r_st, r_nd : T
channel assign : T.T
channel sync
```

```
--constraint 3. only adjecent bells swap
x(1,i) = row -> (a -> p_st!i -> p_nd?j -> sync -> assign!1!j -> x(1,j) []
noop -> sync -> assign!1!i -> x(1,i))
x(2,j) = row -> (a -> p_st?i -> p_nd!j -> sync -> assign!2!i -> x(2,i) []
 b -> q_st!j -> q_nd?k -> sync -> assign!2!k -> x(2,k) []
noop -> sync -> assign!2!j -> x(2,j))
x(3,k) = row -> (b -> q_st?j -> q_nd!k -> sync -> assign!3!j -> x(3,j) []
 g -> r_st!k -> r_nd?l -> sync -> assign!3!l -> x(3,l) []
noop -> sync -> assign!3!k -> x(3,k))
x(4,l) = row -> (g -> r_st?k -> r_nd!l -> sync -> assign!4!k -> x(4,k) []
noop -> sync -> assign!4!l -> x(4,l))

-- simpelest csp process to produce methods.
start = (( (x(1,1) [|{|a,row,sync,p_st, p_nd|}|] x(2,2) )
[|{|b,row,sync,q_st, q_nd|}|] x(3,3))
[|{|g,row,sync,r_st, r_nd|}|] x(4,4))

SSet(X) = addS?x -> SSet(union(X,{x})) []
  isinS?x -> (if member(x,X) then yes else no) -> SSet(X)

get = row -> sync -> assign!1?i -> assign!2?j ->
 assign!3?k -> assign!4?l ->
     (isinS.<i,j,k,l> ->((no -> addS.<i,j,k,l> -> get) []
                         (yes -> STOP)))

-- record a sequance of assign processes and
-- store the result in SSet if it is unique
-- otherwise process get will deadlock.
gget = SSet({}) [|{|isinS, addS, yes, no|}|] get

-- no change is repeated more than once except for rounds
constraint2 = gget [|{|row, assign, sync|}|] start


-- To follow is an implementation of plain bob minimus
-- as CSP processes
A = <0,1,3>
B = <0,2>
C = <0,3>

pBobMini = h(A ^ B, 3) ; h(A ^ C, 1) ; h(A ^ B, 3) ;
   h(A ^ C, 1) ; h(A ^ B, 3) ; h(A, 1) ; STOP

newBobMini = h(append(append(A^B,3)^A^C, 3),1)

append(xs, m) = if(m != 1) then xs ^ append(xs, m-1)
        else xs

h(xs, m) = h1(xs, m, 0)
```

```
h1(xs, m, j) = if( m != 0) then
         if(j != #xs) then
   (nth(j,xs) == 0) & row -> h1(xs, m, j+1) []
              (nth(j, xs) == 1) & a -> h1(xs, m, j+1) []
     (nth(j, xs) == 2) & b -> h1(xs, m, j+1) []
     (nth(j, xs) == 3) & g -> h1(xs, m, j+1)
           else
     h1(xs, m-1, 0)
       else
         SKIP

run = a -> run []
      p_st?i -> run []
      p_nd?i -> run []
      b -> run []
      q_st?j -> run []
      q_nd?j -> run []
      g -> run []
      r_st?k -> run []
      r_nd?k -> run []
      noop -> run []
      assign?i?j -> run []
      sync -> run []
      row -> run

assert  run [T= start

-- solitaire.csp
-- Bill Roscoe, modified Endre
-- counts from 0 .. n-1
nth(u,xs) = if u==0 then head(xs)
                    else nth(u-1,tail(xs))

-- Verify that start can produce methods by
-- checking that it can produce one.
assert start\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
       noop,sync, assign|} [T= pBobMini

--verify that plain bob minimus preserve constraint 2.
-- This takes more than 14 million transitions to verify in fdr
-- A trace has been made in probe to see that this is
-- true. Fdr is not able to do the same.
assert constraint2\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
     noop,sync, assign|} [T= pBobMini
```

A.2 CSP program to examplify constraint 4 - newMethod3b.csp

```
-- created by Endre 17.6.04

mapPre(f)(s,t) = <f(s,y) | y <- t>

ROWS = {1..26}
N = 4
T = {1..N}
S = set(perm(<1..N>))

insNum(n,<>) = <<n>>
insNum(n, xs) = <<n>^xs>^mapPre(prepend)(head(xs), insNum(n, tail(xs)))
prepend(x, xs) = <x>^xs

perm(xs) = if #xs == 2 then
     <xs,tail(xs)^<head(xs)>>
        else f(head(xs), perm(tail(xs)))

f(n, xss) = if #xss != 1 then insNum(n, head(xss))^f(n, tail(xss))
    else insNum(n, head(xss))

channel addS, isinS : S
channel yes,no
channel a, b, g
channel row : ROWS
channel noop
channel p_st, p_nd : T
channel q_st, q_nd : T
channel r_st, r_nd : T
channel assign : T.T
channel sync

--constraint 3. only adjecent bells swap
x(1,i) = row?ct -> (a -> p_st!i -> p_nd?j -> sync -> assign!1!j -> x(1,j) []
noop -> sync -> assign!1!i -> x(1,i))
x(2,j) = row?ct -> (a -> p_st?i -> p_nd!j -> sync -> assign!2!i -> x(2,i) []
 b -> q_st!j -> q_nd?k -> sync -> assign!2!k -> x(2,k) []
noop -> sync -> assign!2!j -> x(2,j))
x(3,k) = row?ct -> (b -> q_st?j -> q_nd!k -> sync -> assign!3!j -> x(3,j) []
 g -> r_st!k -> r_nd?l -> sync -> assign!3!l -> x(3,l) []
noop -> sync -> assign!3!k -> x(3,k))
x(4,l) = row?ct -> (g -> r_st?k -> r_nd!l -> sync -> assign!4!k -> x(4,k) []
noop -> sync -> assign!4!l -> x(4,l))

countRow(ct) = row!ct -> countRow(ct+1)
```

```
start = ((( (x(1,1) [|{|a,row,sync,p_st, p_nd|}|] x(2,2) )
[|{|b,row,sync,q_st, q_nd|}|] x(3,3))
[|{|g,row,sync,r_st, r_nd|}|] x(4,4))
[|{|row|}|] countRow(1) )

SSet(X) = addS?x -> SSet(union(X,{x})) []
  isinS?x -> (if member(x,X) then yes else no) -> SSet(X)

get = row?any -> sync -> assign!1?i -> assign!2?j ->
 assign!3?k -> assign!4?l ->
     (isinS.<i,j,k,l> ->((no -> addS.<i,j,k,l> -> get) []
                         (yes -> STOP)))

gget = SSet({}) [|{|isinS, addS, yes, no|}|] get

constraint2 = gget [|{|row, assign, sync|}|] start

w1 = <1,3,4,2>
c4 = row?any -> c4 []
-- should say "any\\ {8\}"
     assign!1!1 -> c4 []
     assign!2!3 -> c4 []
     assign!3!4 -> c4 []
     assign!4!2 -> c4 []
row!8 -> assign!1!1 -> assign!2!3 -> assign!3!4 -> assign!4!2 -> c4

working_bell = c4 [|{|row, assign.1.1, assign.2.3,
      assign.3.4,assign.4.2|}|] constraint2

A = <0,1,3>
B = <0,2>
C = <0,3>

pBobMini = h(A ^ B, 3) ; h(A ^ C, 1) ; h(A ^ B, 3) ;
   h(A ^ C, 1) ; h(A ^ B, 3) ; h(A, 1) ; STOP

newBobMini = h(append(append(A^B,3)^A^C, 3),1)

append(xs, m) = if(m != 1) then xs ^ append(xs, m-1)
        else xs

h(xs, m) = h1(xs, m, 0)

h1(xs, m, j) = if( m != 0) then
         if(j != #xs) then
   (nth(j,xs) == 0) & row?any -> h1(xs, m, j+1) []
            (nth(j, xs) == 1) & a -> h1(xs, m, j+1) []
     (nth(j, xs) == 2) & b -> h1(xs, m, j+1) []
     (nth(j, xs) == 3) & g -> h1(xs, m, j+1)
           else
```

```
     h1(xs, m-1, 0)
       else
         SKIP

run = a -> run []
      p_st?i -> run []
      p_nd?i -> run []
      b -> run []
      q_st?j -> run []
      q_nd?j -> run []
      g -> run []
      r_st?k -> run []
      r_nd?k -> run []
      noop -> run []
      assign?i?j -> run []
      sync -> run []
      row?any -> run
-- problem here refinement checks are more diffecult

aWork = row!9 -> STOP

assert  run [T= start

-- solitaire.csp
-- Bill Roscoe, modified Endre
-- counts from 0 .. n-1
nth(u,xs) = if u==0 then head(xs)
                    else nth(u-1,tail(xs))

assert start\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
       noop,sync, assign|} [T= pBobMini

assert constraint2\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
      noop,sync, assign|} [T= pBobMini

assert working_bell\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
      noop,sync,assign|} [T= row!9 -> STOP
```

A.3 CSP program for constraint 4 - newMethod4.csp

```
-- created by Endre 17.6.04

mapPre(f)(s,t) = <f(s,y) | y <- t>

ROWS = {1..26} -- could be set to fact(n+2)
N = 4
T = {1..N}
S = set(perm(<1..N>))

W = set(perm(<2..N>))

tmp = fact(N) / (2*N)
II = <1 .. tmp>
-- parse error
-- II = <1 .. (fact(N) / 2N)>
--invariant fact(n) | 2N = 0

insNum(n,<>) = <<n>>
insNum(n, xs) = <<n>^xs>^mapPre(prepend)(head(xs), insNum(n, tail(xs)))
prepend(x, xs) = <x>^xs

perm(xs) = if #xs == 2 then
     <xs,tail(xs)^<head(xs)>>
        else f(head(xs), perm(tail(xs)))

f(n, xss) = if #xss != 1 then insNum(n, head(xss))^f(n, tail(xss))
    else insNum(n, head(xss))

fact(1) = 1
fact(x) = x * fact(x-1)

channel addS, isinS : S
channel yes,no
channel a, b, g
channel row : ROWS
channel noop
channel p_st, p_nd : T
channel q_st, q_nd : T
channel r_st, r_nd : T
channel assign : T.T
channel sync

--constraint 3. only adjecent bells swap
x(1,i) = row?ct -> (a -> p_st!i -> p_nd?j -> sync -> assign!1!j -> x(1,j) []
noop -> sync -> assign!1!i -> x(1,i))
x(2,j) = row?ct -> (a -> p_st?i -> p_nd!j -> sync -> assign!2!i -> x(2,i) []
 b -> q_st!j -> q_nd?k -> sync -> assign!2!k -> x(2,k) []
```

```
noop -> sync -> assign!2!j -> x(2,j))
x(3,k) = row?ct -> (b -> q_st?j -> q_nd!k -> sync -> assign!3!j -> x(3,j) []
 g -> r_st!k -> r_nd?l -> sync -> assign!3!l -> x(3,l) []
noop -> sync -> assign!3!k -> x(3,k))
x(4,l) = row?ct -> (g -> r_st?k -> r_nd!l -> sync -> assign!4!k -> x(4,k) []
noop -> sync -> assign!4!l -> x(4,l))

countRow(ct) = row!ct -> countRow(ct+1)

start = ((( (x(1,1) [|{|a,row,sync,p_st, p_nd|}|] x(2,2) )
[|{|b,row,sync,q_st, q_nd|}|] x(3,3))
[|{|g,row,sync,r_st, r_nd|}|] x(4,4))
[|{|row|}|] countRow(1) )

SSet(X) = addS?x -> SSet(union(X,{x})) []
  isinS?x -> (if member(x,X) then yes else no) -> SSet(X)

get = row?any -> sync -> assign!1?i -> assign!2?j ->
 assign!3?k -> assign!4?l ->
     (isinS.<i,j,k,l> ->((no -> addS.<i,j,k,l> -> get) []
                         (yes -> STOP)))

gget = SSet({}) [|{|isinS, addS, yes, no|}|] get

constraint2 = gget [|{|row, assign, sync|}|] start

--it should say row!(head(ii)*2*n) but does not parse

tmp2(xx) = xx*2*n
PP(ii) = row!(tmp2(head(ii))) -> assign!2?x -> assign!3?y -> assign!4?z ->
      isinS.<x,y,z> -> (yes -> PP(tail(ii)) []
        no -> STOP)

startPP = PP(II)
working_bell = SSet(W) [|{|isinS, yes,no|}|] PP(II)

constraint4 = constraint2 [|{|isinS,assign|}|] working_bell

A = <0,1,3>
B = <0,2>
C = <0,3>

pBobMini = h(A ^ B, 3) ; h(A ^ C, 1) ; h(A ^ B, 3) ;
   h(A ^ C, 1) ; h(A ^ B, 3) ; h(A, 1) ; STOP

newBobMini = h(append(append(A^B,3)^A^C, 3),1)

append(xs, m) = if(m != 1) then xs ^ append(xs, m-1)
        else xs
```

```
h(xs, m) = h1(xs, m, 0)

h1(xs, m, j) = if( m != 0) then
         if(j != #xs) then
   (nth(j,xs) == 0) & row?any -> h1(xs, m, j+1) []
             (nth(j, xs) == 1) & a -> h1(xs, m, j+1) []
     (nth(j, xs) == 2) & b -> h1(xs, m, j+1) []
     (nth(j, xs) == 3) & g -> h1(xs, m, j+1)
           else
     h1(xs, m-1, 0)
       else
         SKIP

run = a -> run []
      p_st?i -> run []
      p_nd?i -> run []
      b -> run []
      q_st?j -> run []
      q_nd?j -> run []
      g -> run []
      r_st?k -> run []
      r_nd?k -> run []
      noop -> run []
      assign?i?j -> run []
      sync -> run []
      row?any -> run
-- problem here refinement checks are more diffecult

aWork = row!9 -> STOP

assert  run [T= start

-- solitaire.csp
-- Bill Roscoe, modified Endre
-- counts from 0 .. n-1
nth(u,xs) = if u==0 then head(xs)
                    else nth(u-1,tail(xs))

assert start\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
       noop,sync, assign|} [T= pBobMini

assert constraint2\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
     noop,sync, assign|} [T= pBobMini

assert working_bell\ {|p_st,p_nd,q_st,q_nd,r_st,r_nd,
       noop,sync,assign|} [T= row!9 -> STOP
```

Appendix B

Java development

B.1 Illustrative program

B.1.1 AccessLine.java

```
import java.lang.*;
import javax.sound.sampled.*;
import java.io.*;


public class AccessLine extends Thread{
    private int[] intSound = null;
    private byte[] data = null;

    //format 44,41,37,33,29,29,28,25,22
    private SourceDataLine line = null;
    private boolean playflag = false;
    private int theBell = -1;

    AccessLine(int theBell){
        System.out.println("setting up line for bell: "+theBell);
        this.theBell = theBell;
        if(theBell > 8){
            System.out.println("Only sound line support for bell 1..8");
            System.exit(1);
        }

        line = AccessMixer.getLine(theBell);
        intSound = new int[8000*10];
    }

    public void setSound(float[] floatSound){
        for(int j=0; j < floatSound.length; j++){
            intSound[j] = (int)Math.round(32768*floatSound[j]);
        }
```

```
    System.out.println("setting sound");
    data = new byte[2*intSound.length];

    for(int i=0; i < intSound.length; i++){
        if(intSound[i] <= -32767){
            data[2*i] = (byte)128;
            data[2*i+1] = 1;
        }else if(intSound[i] >= 32767){
            data[2*i] = 127;
            data[2*i+1] = (byte)255;
        }else{
            data[2*i] = (byte)(intSound[i]>>8);
            data[2*i+1] = (byte)(intSound[i]&255);
        }
    }
}

public void playSound(){
    long currentTime = System.currentTimeMillis();
    int numBytesRemaining = 2*intSound.length;
    while (numBytesRemaining > 0) {
        if(((numBytesRemaining / 2000) == 0) && //stop bell after 1.5 sec
           (System.currentTimeMillis() - currentTime) > 1500){
            line.flush();
            return;
        }else{
            numBytesRemaining
                -= line.write(data, 0, numBytesRemaining);
        }
    }
}

public void playSoundLong(){
    long currentTime = System.currentTimeMillis();
    int numBytesRemaining = 2*intSound.length;
    while (numBytesRemaining > 0)
        numBytesRemaining
            -= line.write(data, 0, numBytesRemaining);

}

public void setPlayflag(){playflag = true;}


public void run(){
    while(true){
        if(playflag == true){
            playSound();
            playflag = false;
        }
```

```
        }
    }
}
```

B.1.2 AccessMixer.java

```
import java.lang.*;
import javax.sound.sampled.*;
import java.io.*;


//set up lines to the mixer, with sampling
//like the A-major scale
public class AccessMixer{
    //format 44,41,37,33,29,29,28,25,22
    private static AudioFormat format1 = null;
    private static DataLine.Info info1 = null;
    private static SourceDataLine[] line = null;

    public static SourceDataLine getLine(int theBell){
        if(theBell > 8){
            System.out.println("Only sound line support forup to 8 bells");
            System.exit(1);
        }
        line = new SourceDataLine[8];

        if(theBell == 0){ //format 44
            format1 = new AudioFormat(44100.0f,16,1,true,true);
            info1 = new DataLine.Info(SourceDataLine.class, format1);
            try{
                line[0] = (SourceDataLine) AudioSystem.getLine(info1);
                line[0].open(format1, 44100);
            }catch (LineUnavailableException ex){
                System.out.println("Unable to open the line: "+ex);
                System.exit(1);
            }
            line[0].start();
        }

        if(theBell == 1){ //format 41
            format1 = new AudioFormat(41500.0f,16,1,true,true);
            info1 = new DataLine.Info(SourceDataLine.class, format1);

            try{
                line[1] = (SourceDataLine) AudioSystem.getLine(info1);
                line[1].open(format1, 41500);
            }catch (LineUnavailableException ex){
```

```
        System.out.println("Unable to open the line:(41kHz) G# "+ex);
        System.exit(1);
    }
    line[1].start();
}

if(theBell == 2){ // format 36
    format1 = new AudioFormat(36667.0f,16,1,true,true);
    info1 = new DataLine.Info(SourceDataLine.class, format1);

    try{
        line[2] = (SourceDataLine) AudioSystem.getLine(info1);
        line[2].open(format1, 36667);
    }catch (LineUnavailableException ex){
        System.out.println("Unable to open the line:(37kHz) F# "+ex);
        System.exit(1);
    }
line[2].start();
}

if(theBell == 3){ //format 33
    format1 = new AudioFormat(33000.0f,16,1,true,true);
    info1 = new DataLine.Info(SourceDataLine.class, format1);

    try{
        line[3] = (SourceDataLine) AudioSystem.getLine(info1);
        line[3].open(format1, 33000);
    }catch (LineUnavailableException ex){
        System.out.println("Unable to open the line:(33kHz) E " + ex);
    System.exit(1);
    }
    line[3].start();
}

if(theBell == 4){ // format 29
    format1 = new AudioFormat(29333.0f,16,1,true,true);
    info1 = new DataLine.Info(SourceDataLine.class, format1);

    try{
        line[4] = (SourceDataLine) AudioSystem.getLine(info1);
        line[4].open(format1, 29333);
    }catch (LineUnavailableException ex){
        System.out.println("Unable to open the line:(29kHz) D " + ex);
        System.exit(1);
    }
    line[4].start();
}

if(theBell == 5){
    format1 = new AudioFormat(27500.0f,16,1,true,true);
```

```
            info1 = new DataLine.Info(SourceDataLine.class, format1);

            try{
                line[5] = (SourceDataLine) AudioSystem.getLine(info1);
                line[5].open(format1, 27500);
            }catch (LineUnavailableException ex){
                System.out.println("Unable to open the line:(28kHz) C# "+ex);
                System.exit(1);
            }
            line[5].start();
        }
        if(theBell == 6){
            format1 = new AudioFormat(24750.0f,16,1,true,true);
            info1 = new DataLine.Info(SourceDataLine.class, format1);

            try{
                line[6] = (SourceDataLine) AudioSystem.getLine(info1);
                line[6].open(format1, 24750);
            }catch (LineUnavailableException ex){
                System.out.println("Unable to open the line:(25kHz) H " + ex);
                System.exit(1);
            }
            line[6].start();
        }

        if(theBell == 7){
            format1 = new AudioFormat(22050.0f,16,1,true,true);
            info1 = new DataLine.Info(SourceDataLine.class, format1);

            try{
                line[7] = (SourceDataLine) AudioSystem.getLine(info1);
                line[7].open(format1, 22050);
            }catch (LineUnavailableException ex){
                System.out.println("Unable to open the line:(22kHz) A " + ex);
                System.exit(1);
            }
            line[7].start();
        }
        return line[theBell];
    }
}
```

B.1.3 Bells.java

```
import java.lang.*;
```

```
//hardcoded different implementations
//of the Karlplus-strong algorithm
//to sound a bell in different keys
public class Bells{
    //constructor
   //BELLS
    public static float[] ren_e(float sound[]){
        float a[] = new float[8000*10];
        System.out.println("returning e");
        for(int t=0; t<(8000*10);t++){
            a[t] = (float)(Math.exp(-0.06*(t/10000.0))*
                           Math.sin((2*Math.PI*80/8000*t)+4*
                                    Math.exp(-0.09*(t/10000.0))*
                                    Math.sin(2*Math.PI*112/8000*t)));
        }
        return a;
    }

    public static float [] ren_a(float sound[]){
        float a[] = new float[8000*10];

        for(int t=0; t < (8000*10); t++){
            a[t] = (float)(Math.exp(-0.06*(t/10000.0))*
                           Math.sin((2*Math.PI*85/8000*t)+4*
                                    Math.exp(-0.09*(t/10000.0))*
                                    Math.sin(2*Math.PI*112/8000*t)));
        }
        return a;
    }

    public static float [] ren_d(float sound[]){
        float a[] = new float[8000*10];

        for(int t=0; t < (8000*10); t++){
            a[t] = (float)(Math.exp(-0.06*(t/10000.0))*
                           Math.sin((2*Math.PI*90/8000*t)+4*
                                    Math.exp(-0.09*(t/10000.0))*
                                    Math.sin(2*Math.PI*112/8000*t)));
        }
        return a;
    }
    public static float [] ren_g(float sound[]){
        float a[] = new float[8000*10];

        for(int t=0; t < (8000*10); t++){
            a[t] = (float)(Math.exp(-0.06*(t/10000.0))*
                           Math.sin((2*Math.PI*95/8000*t)+4*
                                    Math.exp(-0.09*(t/10000.0))*
                                    Math.sin(2*Math.PI*112/8000*t)));
        }
```

```
        return a;
    }
}
```

B.1.4 GrandsireDouble.java

```
import java.lang.*;

//An implementation of the Steinhaus
//algorithm to sound on particular
//instance of grandsire double on 5 bells
public class GrandsireDoubles{
    private int n = 0;
    private int maxsize = 0;  //maxsize = n!
    private int[] repMethod = null;

    private int bell5[] = {1, 2, 3, 4, 5};
    private int a1[] = {1, 2};  // represent transpositions
    private int a2[] = {4, 5};
    private int b1[] = {2,3};
    private int b2[] = a2;
    private int c1[] = a1;
    private int c2[] = {3, 4};
    private int d[] = a2;

    private boolean bobFlag = false;
    private boolean singleFlag = false;
    private boolean flag_odd = false;
    private boolean verbosity = true;//to print, set to true;

    //a = (1,2),(4,5)
    //b    int index = 0; = (2,3),(4,5)
    //c = (1,2),(3,4)
    //d = (4,5)
    //e = (a((bc)4 ba(bc)3 (ba)2)2 (ba)4 ba(bc)3 bad)3
    public int[] grandsireDoublesA(){ //version (bob, bob, single
        n = bell5.length;
        maxsize = MethodUtil.fact(n);
        repMethod = new int[n*maxsize];

        int i=0;
        System.arraycopy(bell5,0,repMethod,i*n,n);
        if(verbosity){
            System.out.print("GrandsireDoubles\n");
            MethodUtil.printChange(bell5);
        }
        for(; i < maxsize; i++){
```

```
            if(((i+1) % 60) == 0){//transpose d, done in 2 singles
                if(verbosity){
                    System.out.print("d ");
                    singleFlag = true;
                }
                MethodUtil.transposition(bell5, d);
                flag_odd = true;
            }else if((((i) % 10) == 0 || (((i+2) % 20) == 0)) ){
                //use i in; i mod 10, as we also wish to do this first time
                //'a' has do to bobs and singles
                if(verbosity){
                if((i % 10 == 0)) System.out.print("\n");
                else  bobFlag = true;
                System.out.print("a ");
                }
                MethodUtil.transposition(bell5, a1);
                //happens 1.- and very:11. time
                MethodUtil.transposition(bell5, a2);
                flag_odd = false;
            }else if(flag_odd == true){
                if(verbosity)
                    System.out.print("c ");

                MethodUtil.transposition(bell5, c1);
                MethodUtil.transposition(bell5, c2);
                flag_odd = false;
            }else if(flag_odd == false){
                if(verbosity)
                    System.out.print("b ");

                MethodUtil.transposition(bell5, b1);
                MethodUtil.transposition(bell5, b2);
                flag_odd = true;
            }
            System.arraycopy(bell5,0,repMethod,i*n,n);
            if(verbosity)
                MethodUtil.printChange(bell5);
        }
        return repMethod;
    }

    public void printMethod(int aMethod[]){
        if( bobFlag == true) {
            System.out.print(" B 2 up");
            bobFlag = false;
        }
        if( singleFlag == true){
            System.out.print(" S 3 up");
            singleFlag = false;
        }
```

```
        System.out.println();
        return;
    }
}
```

B.1.5 MethodUtil.java

```
import java.lang.*;

//a utility class for common operations
//performed by the method class
public class MethodUtil{
   public static int[] transposition(int[] bells,int[] transpose){
        int tmp;

        tmp = bells[transpose[0]-1];
        bells[transpose[0]-1] = bells[transpose[1]-1];
        bells[transpose[1]-1] = tmp;

        return bells;
    }

    public static int fact(int n){
        if(n <= 1) return n;
        else return fact(n-1) * n;
    }

    public static void printChange(int[] b){
        for(int i=0; i < b.length; i++){
            System.out.print(b[i]+" ");
        }
        System.out.print("\n");
    }

    /*
    public boolean verify_ii(Permutation[] method){
        int check = 0;
        System.out.println("method length: " + method.length);
        while(check < method.length-1){
            for(int i=check+1; i < method.length-1; i++){
                if(method[check].isEqual(method[i].toArray()) == false) {
                    return false;
                }
            }
            check++;
        }
        return true;
```

```
    }
    */
}
```

B.1.6 MyFrame.java

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

//fire this class to run the application.
//This class starts the GUI and the background
//processes to ring bells
public class MyFrame{
    private JFrame jframe = new JFrame("Sound!");
    private Container container = jframe.getContentPane();

    private JButton stopMethod = new JButton("stop method");

    private String[] methods = {"plain bob minimus","plain bob double",
                                "grandsire double", "round"};
    private JList listMethods = new JList(methods);
    private JScrollPane scrollPane = new JScrollPane(listMethods);

    private JMenuBar mb = new JMenuBar();
    private JMenu MenuBell = new JMenu("Bells");
    private JMenuItem Mbell44 = new JMenuItem("8: A  44 Hz");
    private JMenuItem Mbell41 = new JMenuItem("7: G# 41 Hz");
    private JMenuItem Mbell37 = new JMenuItem("6: F# 37 Hz");
    private JMenuItem Mbell33 = new JMenuItem("5: E  33 Hz");
    private JMenuItem Mbell29 = new JMenuItem("4: D  29 Hz");
    private JMenuItem Mbell28 = new JMenuItem("3: C# 28 Hz");
    private JMenuItem Mbell25 = new JMenuItem("2: H  25 Hz");
    private JMenuItem Mbell22 = new JMenuItem("1: A  22 Hz");

    private JMenu MenuKey = new JMenu("Keys");
    private ButtonGroup group = new ButtonGroup();
    private JRadioButtonMenuItem renE = new JRadioButtonMenuItem("$ E");
    private JRadioButtonMenuItem renA = new JRadioButtonMenuItem("$ A");
    private JRadioButtonMenuItem renD = new JRadioButtonMenuItem("$ D");
    private JRadioButtonMenuItem renG = new JRadioButtonMenuItem("$ G");
    private JMenuItem EADG = new JMenuItem("E;A;D;G");

    private SimpleSoundGraph graph = new SimpleSoundGraph();;

    private String currentMethod = null;
```

```
private SoundMethods theSound = new SoundMethods();
private Thread batch = new Thread(theSound);

private int[] intSoundUI = null;
private float[] floatSoundUI = null;

public MyFrame(){
    batch.start();

    WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.out.println("Window closing - exiting");
                System.exit(0);}
        };
    jframe.addWindowListener(l);

    stopMethod.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                theSound.stopMethod();
            }
        });


    MouseListener mouseListener = new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                if (e.getClickCount() == 2) {
                    currentMethod = (String)listMethods.
                        getSelectedValue();
                    theSound.setStartMethod(true);
                    theSound.setRingMethod(currentMethod);
                }
            }
        };
    listMethods.addMouseListener(mouseListener);

    renE.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){

                floatSoundUI = Bells.ren_e(null);
                intSoundUI = new int[floatSoundUI.length];
                for(int j=0; j < floatSoundUI.length; j++){
                    intSoundUI[j]=(int)Math.round(32768*floatSoundUI[j]);
                }
                graph.setSound(intSoundUI);

                graph.updateUI();
            }
        });

    renA.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e){

            floatSoundUI = Bells.ren_a(null);
            intSoundUI = new int[floatSoundUI.length];
            for(int j=0; j < floatSoundUI.length; j++){
                intSoundUI[j]=(int)Math.round(32768*floatSoundUI[j]);
            }
            graph.setSound(intSoundUI);

            graph.updateUI();
        }
    });

renD.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){

            floatSoundUI = Bells.ren_d(null);
            intSoundUI = new int[floatSoundUI.length];
            for(int j=0; j < floatSoundUI.length; j++){
                intSoundUI[j]=(int)Math.round(32768*floatSoundUI[j]);
            }
            graph.setSound(intSoundUI);

            graph.updateUI();
        }
    });

renG.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){

            floatSoundUI = Bells.ren_g(null);
            intSoundUI = new int[floatSoundUI.length];
            for(int j=0; j < floatSoundUI.length; j++){
                intSoundUI[j]=(int)Math.round(32768*floatSoundUI[j]);
            }
            graph.setSound(intSoundUI);

            graph.updateUI();
        }
    });

EADG.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(Bells.ren_e(null),0);
            theSound.playSoundLong(0);
            theSound.setSoundToPlay(Bells.ren_a(null),0);
            theSound.playSoundLong(0);
            theSound.setSoundToPlay(Bells.ren_d(null),0);
            theSound.playSoundLong(0);
            theSound.setSoundToPlay(Bells.ren_g(null),0);
```

```
            theSound.playSoundLong(0);
        }
    });

Mbell44.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 7);
            theSound.playSoundLong(7);
        }
    });
Mbell41.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 6);
            theSound.playSoundLong(6);
        }
    });

Mbell37.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 5);
            theSound.playSoundLong(5);
        }
    });
Mbell33.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 4);
            theSound.playSoundLong(4);
        }
    });
Mbell29.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 3);
            theSound.playSoundLong(3);
        }
    });
Mbell28.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 2);
            theSound.playSoundLong(2);
        }
    });
Mbell25.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 1);
            theSound.playSoundLong(1);
        }
    });
Mbell22.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            theSound.setSoundToPlay(floatSoundUI, 0);
```

```
            theSound.playSoundLong(0);
        }
    });

MenuBell.add(Mbell44);
MenuBell.add(Mbell41);
MenuBell.add(Mbell37);
MenuBell.add(Mbell33);
MenuBell.add(Mbell29);
MenuBell.add(Mbell28);
MenuBell.add(Mbell25);
MenuBell.add(Mbell22);
mb.add(MenuBell);

group.add(renE);
group.add(renA);
group.add(renD);
group.add(renG);
MenuKey.add(renE);
MenuKey.add(renA);
MenuKey.add(renD);
MenuKey.add(renG);
MenuKey.add(EADG);
mb.add(MenuKey);
jframe.setJMenuBar(mb);

renA.setSelected(true);

theSound.mpane =
    new JScrollPane(theSound.mmethodTxt,
                    JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                    JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

container.setLayout(null);

//setbounds(x,y,width, height)
graph.setBounds(10,10,597,260);
theSound.mpane.setBounds(10,300,200,300);
scrollPane.setBounds(300,300,250,250);
stopMethod.setBounds(350,560,150,50);

container.add(stopMethod);
container.add(graph);
container.add(scrollPane);
container.add(theSound.mpane);

theSound.mmethodTxt.setEditable(false);
jframe.setSize(650, 700);
jframe.setLocation(100, 100);
jframe.setVisible(true);
```

```
        floatSoundUI = Bells.ren_a(null);
        intSoundUI = new int[floatSoundUI.length];
        for(int j=0; j < floatSoundUI.length; j++){
            intSoundUI[j] = (int)Math.round(32768*floatSoundUI[j]);
        }
        graph.setSound(intSoundUI);

        graph.updateUI();
    }

    static public void main(String[] args) {
        new MyFrame();
    }
}
```

B.1.7 PlainBob.java

```
import java.lang.*;

//Two implementations of methods called
//plain bob, one on 4, and one on 5 bells
public class PlainBob{
    private int n = 0;
    private int maxsize = 0; //n!
    private int[] repMethod = null;

    private int bell4[] = {1, 2, 3, 4};
    private int bell5[] = {1, 2, 3, 4, 5};

    private int[] a1 = {1, 2};
    private int[] a2 = {3, 4};
    private int[] b = {2, 3};
    private int[] c = {3, 4};

    private boolean flag_odd = true;
    private boolean verbosity = true;

    public int[] plainBobMinimus(){
        n = bell4.length;
        maxsize = MethodUtil.fact(n);
        repMethod = new int[n*maxsize];

        int i = 0;
        System.arraycopy(bell4,0,repMethod,i*n,n);
        if(verbosity){
            System.out.print("Plain bob minimus\n");
```

```
        MethodUtil.printChange(bell4);
    }
    for(; i < maxsize; i++){
        if( ((i+1) % 2) == 1){ // i : odd(NAT)
            MethodUtil.transposition(bell4, a1);
            //transposition is commutative if trans. disjoint
            MethodUtil.transposition(bell4, a2);
        }else{
            if( ((i+1) % 8) != 0){
                MethodUtil.transposition(bell4, b);
            }else{
                if(verbosity) System.out.print("\n");
                MethodUtil.transposition(bell4, c);
            }
        }
        System.arraycopy(bell4,0,repMethod,i*n,n);
        if(verbosity)
            MethodUtil.printChange(bell4);
    }
    return repMethod;
}

private int b2[] = {4,5};
private int d[] = b2;
//a = (1,2)(3,4)
//b = (2,3)(4,5)
//c = (3,4)
//d = (4,5)
//plainBobDoubles = [((ab)^4 ac)^3 (ab)^4 ad]^3 = e
public int[] plainBobDoubles(){
    n = bell5.length;
    maxsize = MethodUtil.fact(n);
    repMethod = new int[n*maxsize];

    int i=0;
    System.arraycopy(bell5,0,repMethod,i*n,n);
    if(verbosity){
        System.out.print("plainBobDoubles\n");
        MethodUtil.printChange(bell5);
    }
    for(; i < maxsize; i++){
        if(((i+1) % 40) == 0){
            if(verbosity) System.out.println("d ");
            MethodUtil.transposition(bell5, d);
            flag_odd = true;
        }else if(((i+1) % 10) == 0){
            if(verbosity) System.out.println("c ");
            MethodUtil.transposition(bell5, c);
            flag_odd = true;
        }else if(flag_odd == false){
```

```
                if(verbosity) System.out.print("b ");
                MethodUtil.transposition(bell5, b);
                MethodUtil.transposition(bell5, b2);
                flag_odd = true;
            }else if(flag_odd == true){
                if(verbosity) System.out.print("a ");
                MethodUtil.transposition(bell5, a1);
                MethodUtil.transposition(bell5, a2);
                flag_odd = false;
            }
            System.arraycopy(bell5,0,repMethod,i*n,n);
            if(verbosity)
                MethodUtil.printChange(bell5);
        }
        return repMethod;
    }
}
```

B.1.8 SimpleSoundGraph.java

```
import javax.swing.*;
import java.awt.*;

//Class to display a graph of a given
//array. In this program the graph is
//the numerical samplings to produce the
//bell like sound.
class SimpleSoundGraph extends JPanel
{
  static final int MAXWIDTH=600;
  Image theSoundImage;
  int graphWidth=256;

  SimpleSoundGraph()
  {
    //  setPreferredSize(new Dimension(256,256));
  }

  public void setSound(int newSound[]) // newSound must not be null
  {
    Graphics theGraphics;
    int scale=1,samples,newWidth;

    if(newSound==null)
      {
        samples=0;
        newWidth=256;
```

```
    }
  else
    {
      samples=newSound.length;

      scale=(samples+MAXWIDTH-1)/MAXWIDTH;
      newWidth=samples/scale;
    }

  if(graphWidth != newWidth)
    {
      Dimension newDimension;

      newDimension = new Dimension(newWidth,256);
      System.out.println("1 MAXWIDTH: "+MAXWIDTH);
      System.out.println("2 theSoundImage: "+theSoundImage);
      System.out.println("3 graphWidth: "+graphWidth);
      System.out.println("4 newSound: "+newSound);
      //System.out.println("theGraphics"+theGraphics);
      System.out.println("5 scale: "+scale);
      System.out.println("6 samples: "+samples);
      System.out.println("7 newWidth: "+newWidth);
      System.out.println("8 newDimension: "+newDimension);
      theSoundImage = createImage(newWidth,256);
      System.out.println("GraphicsEnvironment.isHeadless()"+
                         GraphicsEnvironment.isHeadless());
      System.out.println(" theSoundImage: "+theSoundImage);
      setPreferredSize(newDimension);
    }
  graphWidth=newWidth;

  theGraphics=theSoundImage.getGraphics();
  theGraphics.setColor(Color.white);
  theGraphics.fillRect(0,0,newWidth, 256);
  theGraphics.setColor(Color.black);

  for(int i=0;i<samples;i++){
      int c=(newSound[i]+32767)/256;

      theGraphics.drawLine(i/scale,c,i/scale,c);
  }
  repaint();
}

public void paintComponent(Graphics g)
{
  if(theSoundImage != null) g.drawImage(theSoundImage,0,0,this);
}

  public static void main(String[] args){
```

```
        SimpleSoundGraph s = new SimpleSoundGraph();
        s.setSound(null);
    }
}
```

B.1.9 SoundMethods.java

```
import java.lang.*;
import javax.swing.*;

//Starts NUM_BELLS number of threads be sounded
//by the accessLine class
public class SoundMethods extends Thread{
    private AccessLine[] l = null;
    private GrandsireDoubles sire = new GrandsireDoubles();
    private PlainBob plainbob = new PlainBob();
    private int[] sequence = null;
    private Thread[] t = null;
    private final int NUM_BELLS = 8;
    private int NUM_CUR_BELLS = 0;
    private String ringMethod = null;
    private boolean startMethod = false;
    private boolean stopTheMethod = false;

    public JTextArea mmethodTxt = new JTextArea();
    public JScrollPane mpane = null;

    public void setRingMethod(String m){ ringMethod = m;}
    public void setStartMethod(boolean f){startMethod = f;}

    public SoundMethods(){
        t = new Thread[NUM_BELLS];
        l = new AccessLine[NUM_BELLS];
        for(int i=0; i<NUM_BELLS;i++){
            l[i] = new AccessLine(i);

            //default bell
            l[i].setSound(Bells.ren_a(null));

            t[i] = new Thread(l[i]);
        }
    }

    public void  stopMethod(){
        System.out.println("Stopping a method with "+NUM_CUR_BELLS+" bells");
        mmethodTxt.append("\n\n");
        stopTheMethod = true;
```

```
}


public void run(){
    while(true){
        if(startMethod == true){
            startMethod = false;
            ringTheMethod();
        }

    }
}
public void ringTheMethod(){
    ringMethod.trim();
    ringMethod.toLowerCase();

    if(ringMethod.equals("plain bob minimus")){
        sequence = plainbob.plainBobMinimus();
        NUM_CUR_BELLS = 4;
        startThreads();
    }
    if(ringMethod.equals("plain bob double")){
        sequence = plainbob.plainBobDoubles();
        NUM_CUR_BELLS = 5;
        startThreads();
    }
    if(ringMethod.equals("grandsire double")){
        sequence = sire.grandsireDoublesA();
        NUM_CUR_BELLS = 5;
        startThreads();
    }
    if(ringMethod.equals("round")){
        sequence = new int[NUM_CUR_BELLS];
        for(int k=0; k< NUM_CUR_BELLS; k++){
            sequence[k] = k+1;
        }
        startThreads();
    }
}

private void startThreads(){
    for(int j=0;j < NUM_CUR_BELLS; j++){
        if(t[j].isAlive() == false){
            t[j].start();
            System.out.println("started line "+j);
        }
    }

    for(int i=0;i < sequence.length;i++){
        if(stopTheMethod == true) i = sequence.length;
```

```
            else{
                mmethodTxt.append(new Integer(sequence[i]).toString());
                if((i+1) % (NUM_CUR_BELLS) == 0)
                    mmethodTxt.append("\n");
                l[sequence[i]-1].setPlayflag();
                try{
                    Thread.sleep(500); //sleep 0.5 sec
                }catch(Exception e){e.printStackTrace();}
            }
        }
        stopTheMethod = false;
        //for(int j=0;j < NUM_CUR_BELLS; j++)
        //t[j].sleep();
    }

    public void setSoundToPlay(float[] f, int i){
        l[i].setSound(f);
    }

    public void playSoundLong(int i){
        l[i].playSoundLong();
    }


    public void setKey(int i){

    }

}
```

Appendix C

Program to search all methods

C.1 TransMinimusOld.java

```
import java.lang.*;
import java.util.*;


//the beginning to a program to generate
//all the permutations on four bells
public class TransMinimusOld{

    // all the transpositions
    private int a1[] = {1,2};
    private int a2[] = {3,4};
    private int b[] = {1,2};
    private int c[] = {3,4};
    private int d[] = {2,3};
    private int round[] = {1,2,3,4};

    //The four(!) possible transpositions after the initial perm.
    private int rA[] = {2,1,4,3};
    private int rB[] = {2,1,3,4};
    private int rC[] = {1,2,4,3};
    private int rD[] = {1,3,2,4};

    //the four first transpositions are hardcoded
    private     TreeNode mother = new TreeNode(round, null,0, 0);
    private     TreeNode treeA = new TreeNode(rA, mother,1,1);
    private     TreeNode treeB = new TreeNode(rB, mother,2,1);
    private     TreeNode treeC = new TreeNode(rC, mother,3,1);
    private     TreeNode treeD = new TreeNode(rD, mother,4,1);
```

```
//counting from 0..23, 24 branches
private final int maxDepth = 17;
private int methodCounter = 0;

//set if you wish print out on the screen
private boolean verbosity = false;
private boolean verbosity2 = false;

public void getAllTranspositions(){
    expandOneBranch(treeA, 2);
    expandOneBranch(treeB, 2);
    expandOneBranch(treeC, 2);
    expandOneBranch(treeD, 2);
    System.out.println("methodcounter: "+methodCounter);
}

//explores all unique branches until maxDept. This is
//a recursive function. Uses prunning to delete explored branches.
//haveDone is the current depth of the tree
public void expandOneBranch(TreeNode bottomA, int haveDone){
    int didLast; //the permutation that a Node did last
    int[] newTransA = clone((int[])bottomA.element);

    didLast = bottomA.didLast;
    if(verbosity)
        System.out.println("didlast, depth: "+didLast+", "+haveDone);

    if(didLast != 1){
        newTransA = transposition(newTransA, a1);
        newTransA = transposition(newTransA, a2);
    if(verbosity){
        print(newTransA);System.out.print(" A");}

    //delete a node if it is not unique - else add it
        if(isUnique(newTransA, bottomA) == false){
            if(verbosity){
                System.out.print(" del A ");print(newTransA);
                System.out.print(", ");print((int[])bottomA.element);
                System.out.print(", "+bottomA.depth);
            }

            bottomA.left1 = null;
        }else{
            //prune1 if all nodes below has been explored
            if(haveDone == maxDepth){
                bottomA.left1 = null;
                methodCounter++;
                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
```

```
                System.out.println();
            }

        }else
            bottomA.left1 = new TreeNode(newTransA, bottomA, 1,haveDone);
    }
if(verbosity)
    System.out.println();
}
if(didLast != 2){
    newTransA = clone((int[])bottomA.element);
    newTransA = transposition(newTransA, b);
    if(verbosity){
        print(newTransA);System.out.print(" B");}

    //as well as deciding if to keep a node,
    //decide which branch it should go to.
    boolean flag = isUnique(newTransA, bottomA);
    if(didLast == 1){
        if(flag == false){
            if(verbosity)
                System.out.print(" del B");

            bottomA.left1 = null;
        }else{
            if(haveDone == maxDepth){
                bottomA.left1 = null;
                methodCounter++;
                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
                    System.out.println();
                }

            }else
                bottomA.left1 = new TreeNode(newTransA,bottomA,2,haveDone);
        }
    }else{
        if(flag == false){
            if(verbosity)
                System.out.print(" del B");

            bottomA.left2 = null;
        }else{
            if(haveDone == maxDepth){
                bottomA.left2 = null;
                methodCounter++;
                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
```

```
                    System.out.println();
                }
            }else
                bottomA.left2=new TreeNode(newTransA,bottomA,2,haveDone);
        }
    }
    if(verbosity)
        System.out.println();
}
if(didLast != 3){
    newTransA = clone((int[])bottomA.element);
    newTransA = transposition(newTransA, c);
    if(verbosity){
        print(newTransA);System.out.print(" C");}

    boolean flag = isUnique(newTransA, bottomA);
    if(didLast <= 2){
        if(flag == false) {
            if(verbosity)
                System.out.print(" del C");

            bottomA.left2 = null;
        }else{
            if(haveDone == maxDepth){
                bottomA.left2 = null;
                methodCounter++;

                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
                    System.out.println();
                }

            }else
                bottomA.left2 = new TreeNode(newTransA,bottomA,3,haveDone);
        }
    }else if(didLast == 4){
        if(flag == false){
            if(verbosity)
                System.out.print(" del C");

            bottomA.right = null;
        }else{
            if(haveDone == maxDepth){
                bottomA.right = null;
                methodCounter++;
                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
                    System.out.println();
```

```
                    }

                }else
                    bottomA.right=new TreeNode(newTransA,bottomA,3,haveDone);
            }
        }
        if(verbosity)
            System.out.println();
    }
    if(didLast != 4){
        newTransA = clone((int[])bottomA.element);
        newTransA = transposition(newTransA, d);
    if(verbosity){
        print(newTransA);System.out.print(" D");}

        boolean flag = isUnique(newTransA, bottomA);
        if(flag == false){
            if(verbosity)
                System.out.print(" del D");

            bottomA.right = null;
        }else{
            if(haveDone == maxDepth){
                bottomA.left1 = null;
                methodCounter++;
                if(verbosity2){
                    System.out.print(" Added perm:");
                    print(newTransA);
                    System.out.println();
                }
            }else
                bottomA.right = new TreeNode(newTransA, bottomA, 4,haveDone);
        }
        if(verbosity)
            System.out.println();
    }

    //recursively prune all empty nodes
    prune2(bottomA);

    if(haveDone >= maxDepth) return;
    else{
        if(bottomA.left1 != null)
        expandOneBranch(bottomA.left1, haveDone+1);
        if(bottomA.left2 != null)
        expandOneBranch(bottomA.left2, haveDone+1);
        if(bottomA.right != null)
        expandOneBranch(bottomA.right, haveDone+1);
    }
}
```

```
private void prune2(TreeNode currNode){
    if(currNode.left1 == null && currNode.left2 == null &&
       currNode.right == null){
        if(verbosity){
            System.out.print("\t Deleting perm ");
            print((int [])currNode.element);
            System.out.println(" depth "+currNode.depth);
        }

        TreeNode itsParent = currNode.parent;
        if(itsParent.left1 == currNode){
            itsParent.left1 = null;
            prune2(itsParent);
        }else if(itsParent.left2 == currNode){
            itsParent.left2 = null;
            prune2(itsParent);
        }else if(itsParent.right == currNode){
            itsParent.right = null;
            prune2(itsParent);
        }
    }
}

//Test if the permutation is represented in any of
//the parent nodes of the Tree goUp
private boolean isUnique(int[] newPerm, TreeNode goUp){
    int[] upPerm = null;
    while(goUp != null){
        upPerm = (int[])goUp.element;
        if(compareTo(newPerm, upPerm)== true) return false;
        else goUp = goUp.parent;
    }
    return true;
}

private void print(int[] el){
    for(int i=0;i<el.length;i++)
        System.out.print(el[i] + " ");
}

private int[] transposition(int[] perm, int[] trans){
    int tmp = perm[trans[0]-1];
    perm[trans[0]-1] = perm[trans[1]-1];
    perm[trans[1]-1] = tmp;
    return perm;
}

private boolean compareTo(int[] perm1, int[] perm2){
    //error condition not dealt with
```

```
        //      if(perm1.length != perm2.length) return true;

        for(int i=0; i<perm1.length; i++){
            if(perm1[i] != perm2[i])
                return false;
        }
        return true;
    }

    TransMinimusOld(){
        getAllTranspositions();
    }

    private int[] clone(int a[]){
        int[] b = new int[a.length];
        for(int i=0;i<a.length;i++)
            b[i] = a[i];
        return b;
    }

    public static void main(String[] args){
        new TransMinimusOld();
    }

    private void testClone_transposition(){
        int[] tmp = new int[rA.length];
        tmp = clone(rA);
        print(rA);
        System.out.println("rA");
        print(tmp);
        System.out.print(" ");
        print(transposition(tmp, b));
        System.out.println();

        tmp = clone(rA);
        print(tmp); System.out.print(" ");
        print(transposition(tmp, c));
        System.out.println();

        tmp = clone(rA);
        print(tmp); System.out.print(" ");
        print(transposition(tmp, d));
        System.out.println();
    }
}
```

C.2 TreeNode.java

```
import java.lang.*;

// Basic node stored in a linked tree
class TreeNode{
    // Constructors
    TreeNode( Object theElement, TreeNode parent, int last, int d){
        this( theElement, null, null, null, parent, last, d );
    }

    TreeNode( Object theElement,
              TreeNode l1, TreeNode l2, TreeNode r, TreeNode p,
              int l, int d){
        element = theElement;
        left1 = l1;
        left2 = l2;
        right = r;
        parent = p;
        didLast = l;
        depth = d;
    }

    // Friendly data; accessible by other package routines
    TreeNode parent;
    Object   element;
    TreeNode left1;
    TreeNode left2;
    TreeNode right;
    int didLast;
    int depth;
}
```