Chapter 5 Adding Refactoring Statements into rc-Promela

In chapter 3, we have described the process of refactoring transformation. This chapter we describe that the new statements for refactoring automation.

5.1. The semantics and grammar of *Refactorby*

In Figure 5.1 is the syntax of the *refactorby* statement. The enclosed block is called *r-block*. In theory, *refactorby* can enclose arbitrary codes. In that case, refactoring transformation shall do what as told. However, in practice, the codes inside an *r-block* often begin with a statement which cause branches in control flow, such as *do* block, *if* statement or statement that receive external messages. *expr_list* behind keyword **refactorby** are grouping parameter list. *expr_list* are a list of *exp* and each exp can be a channel name or variable name. We use *exp* as a criterion to divide segments. The order of *exp* in *expr_list* determines the order of division. We describe it in next paragraph.

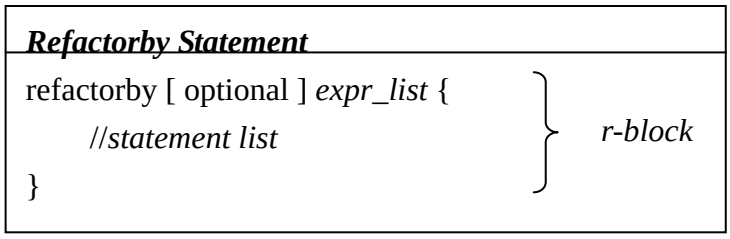

Figure 5.1. Refactorby statement definition.

The common *expr_list* are explained in the following five cases:

I. refactorby *var1,var2… {…..}*

This option first divides the segments into groups using possible values of *var1* and next divides these groups again using possible values of *var2*. For example, if possible values of *var1* are {0, 1} and *var2* may be {1, 2, 3}, segments will be divided into six groups. The value of *var1* and *var2* in this six groups are (0,1), (0,2), (0,3), (1,1), (1,2), and (1,3). These groups are wrapped into six new processes after refactoring transformation.

II. refactorby *ch1, ch2… {…..}*

In this case, ch1 and ch2 are the names of *Channel*. This option first divides the segments into two groups, one containing segments with *ch1* in their edge label and the other containing no *ch1* in edge labels. Next, we use *ch2* to divide the two groups. Therefore, the possible groups at most will be divided into four groups: with *ch1*, with *ch2*, with *ch1* and *ch2*, and none of the *Channel* in the group.

III. refactorby *ch1,var1…{…..}*

This option first divide these segment into two groups according to the channel name. Next, refactoring use possible values of *var1* to divide the two groups into smaller groups.

IV. refactorby *var1 = num1,var2 = num2,…{….}*

In this option, each *exp* in *expr_list* can be an assignment. The option first divide segments into two groups: one using *var1* with value equivalent to *num1* and one using *var2* with value not equivalent to *num1*. Next, divides the two

groups again using *var2 = num2*.

V. refactorby [optional] *expr_list* {…..}

To allow more flexible refactoring decisions to be made, we add an optional field to the statement. This option allows further manual refinement of refactoring strategy. For example, in Figure 5.2, we list a fragment of codes of Chiron [11] example. This example will be explained in Chapter 7. In this do loop, every time *Dispatcher* receives an event which is equivalent to *e1*, it will enter a nested *do* loop to check the value in every element of *e1_lst*. If *e1_lst[i-1]* is true, the process output a message *"notify,e1"* via channel a_i*_chan*. According to refactoring transformation, this code generates segments as in Figure 5.3, which is in the same group.

```
mtype = {e1,e2}
mtype event;
process Dispatcher( ){
    refactorby [delayedeval = e1_lst] event{
    do
    :: in ? notify ,event→
      if
      :: (event == e1)→ i = 1;
         do
         :: (i > 2 ) → break;
         ::else→
           if
           :: ( e1_lst[i-1] == 1) → ……...……..(＊)
                if
                :: ( i-1 == 0)→ a1_chan ! notify, 1;
                :: ( i-1 == 1)→ a2_chan ! notify, 2;
                fi
           :: else→i++;
           fi
         od
      fi
    od
    }
}
```

Figure 5.2 A fragment code of rc-Promela.

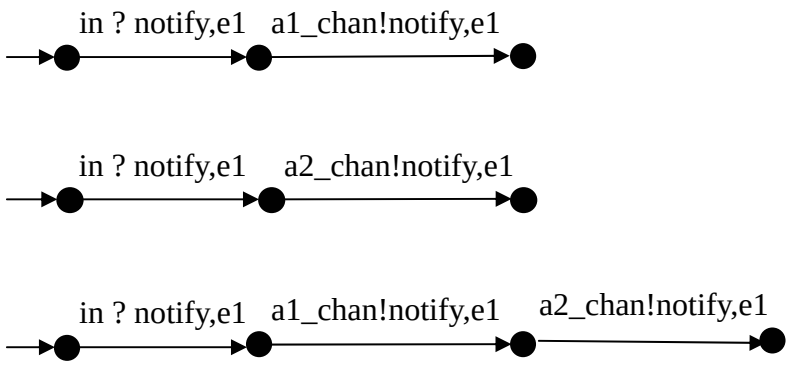

Figure 5.3. Three segments generated from Figure 5.2 which are in the same group.

Under normal setting, these three segments can be combined like *RRS0* that we described it in Chapter 3. However, this refactoring strategy becomes awkward as the number of artist increases. So, we allow user to use an option "*[delayedeval = e1_lst]*" to fine tune the refactoring. The statement marked with (＊) will not be evaluated to generate segment like Figure 5.3. Instead, it generates segment like Figure 5.5 where the value of *e1_lst* is evaluated by *VP*, rather by parser. We call this *delayed evaluation*. That is, for some refactoring strategy, we rather delay the evaluation until parallel composition. There might be other options for refactoring strategy refinement in the future.

According to those five cases as discussed above, Figure 5.4 is the grammar for refactorby statement. The codes in refactorby statement may have many statements. *Optional* that describe for other options for refactoring may have many conditions, now we have only two options. The *group_var* describes group parameter lists for grouping option; it can be a series of *expr*. Each *expr* can be an assignment, variable, or channel name.

```
Stmt :
        | REFACTORBY optional group_var ‘{‘ stmts ‘}’
       ….
       ;
optional:
        |     ‘[‘ roptions ‘]’
        ;
roptions: roption
        | roption ‘,’ roptions
        ;
roption: INDEPFORLOOP
        | DELAYEDEVAL ASGN var
       ;
group_var: expr
        | expr ‘,’ group_var
       ;
```

Figure 5.4. Grammar for refactorby statement.

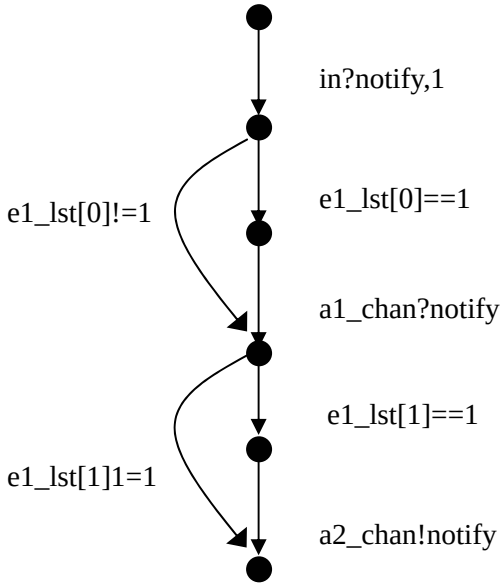

Figure 5.5. The process use *delayedeval* option.

5.2. Restriction of refactorby statement

Note that when using refactorby statement with a nested *do*, such as in the left-hand side in Figure 5.6, it is equivalent to the right-side statement..

```
refactorby expr_list {
    do
    :: condition1
    :: condition2
    od
}
```

```
do
::  refactorby expr_list{ condition1 }
::  refactorby expr_list { condition2 }
od
```

Figure 5.6. An example of r-block with nested do.

However, if users write refactorby statement as in Figure 5.7, there are *do* statement and other statements in *r-block*. We will return a warning message that this type of refactorby statement may not be suitable for refactoring.

```
refactorby expr_list {
    do
    ::
    ::
    od
    statement1;
    statement2;
    ……
}
```

Other statements

Figure 5.7. A wrong example of refactorby statement.

5.3. Adding enumeration in *rc-Promela*

If a variable's type is not numeric types, such as bit, byte, or int, it is *mtype* in Promela. You can declare a set of symbolic constants for *mtype*. Symbolic constants are like enumeration in many other programming. However, Promela allows only one declaration of symbolic constants. For example, in Figure 5.8, the symbolic constant of variable *var1* will be "unrolled" in finite state verification. Thus, *var1* generate six transitions according to the number of *mtype*. However, some symbolic constants defined in *mtype* may never be used by some channel. It may produce unused transition/state in CCS state graphs. For this reason, if we can define several enumerations for symbolic constants, such as enumeration type in C, many states/transition may be reduced.

```
mtype = {one,two,three,four,five,six}
mtype var1;
chan in_chan = [0] of {mtype};
proctype a( ) {
    in_chan ? var1 ;
    ……
}
```

Figure 5.8 A fragment code of Promela.

We add enumeration constants into *rc-Promela* to solve this problem. Declaration of enumeration is like C language. Enumeration just can be defined in global, like the definition of *mtype*. Enumeration declaration is as follows:

```
enum name = { NAME1,NAME2,NAME3,…}
```

Only at most 255 symbolic constants can be declared in enumeration type. The advantage of adding enumeration type into *rc-Promela* is we precisely compute the number of transitions for symbolic expansion.

Having enumeration in *rc-Promela*, we can write Promela code as in Figure 5.9. If we don't have enumeration type, we will declare seven constants in mtype. Therefore, in symbolic expansion we'll expand all the seven possible values for that *mtype* variable. However, if we declare enumeration type, we just need to expand all the possible values for the enumeration variable.

```
enum elevator_id    = {elevator1, elevator2};
enum direction = {up_or_down, idle};
enum user_command = {call ,request, quit};
user_command command ;
proctype elevator_model( ){
      in ? command;
      ……..
}
```

Figure 5.9. An enumeration type definition in *rc-Promela.*

To implement enumeration type, it has two steps to do:

I. Specify grammar as in Figure 5.10:

Figure 5.10 is grammar for enumeration. First, the **NAME** in etype is new declaration type name; second, the **nlst** in etype is a series of constants which is content of **NAME**. We construct AST for this type and save it in symbol table.