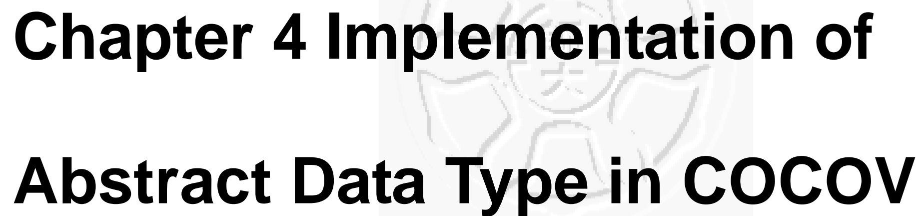


Chapter 4 Implementation of Abstract Data Type in COCOV

In order to add ADTs to rc-Promela, we have to modify the parser and its core code which is developed by [5]. In this chapter, since we do not rewrite it, we describe what is changed or what features we have added.

4.1 Abstract data type of COCOV

In this section, we present our initial work towards to this goal. We implement some frequently used abstract data types (ADTs) into *COCOV*. Here we list two only.

QUEUE

DECLARATION SYNTAX:

exclusive queue qname = [n] **of** {*enumtype*}

METHODS:

void push (*enumtype* val**);**
// to add a value val to the queue

enumtype **pop ();**
// returen and remove the first element of queue

enumtype **front();**
// return the value of first element

SET

DECLARATION SYNTAX:

set qname = [n] **of** {*enumtype*}

METHODS:

enumtype **add (***enumtype* **val);**
// add val to a set

enumtype **remove (***enumtype* **val);**
// remove val from a set

bool **contains(***enumtype* **val);**
// return true if val is in the set

Where *enumtype* is a type defined by **enum** keyword in rc-Promela. A user can use

```
enum client_type = {c1, c2, c3};
```

to define an enumeration type in *rc-Promela*. Note that we add a keyword *exclusive* before *queue* to have the code tell more information about the queue. An exclusive queue requires that values in a queue cannot be duplicated. Modeling a regular queue which can have duplicated values requires more cost in terms of state size. An exclusive queue is adequate for most applications we have dealt with.

Many systems which use array to model process behaviors can be rewritten using the ADTs. The advantages of writing models with ADTs are obvious. For example, the queue-enabled pump can be written as follows:

```
proctype pumpq() {
    exclusive queue q[2];
    do
    ::  op?add, cid     ->
        q.push(cid);
    ::  (q.first() == c1)     ->
        cust?remove, c1;
        q.pop();
        cust?start1;
        cust?finish1;
        op!charge, c1;
    ::  (q.first() == c2)     ->
        cust?remove, c2;
        q.pop();
        cust?start2;
        cust?finish2;
    od
}
```

In this example, ADTs force process behaviors to converge on this one. In addition, modelers need not worry about the impact of an implementation choice to analysis. The rc-Promela translator of COCOV shall choose the best implementation for each ADT, which is transparent to tool users. More importantly, to cases like dispatcher of Chiron, our rc-Promela translator can select the best implementation and guide the refactoring engine to decompose process behaviors based on the ADT used.

4.2 COCOV (Compositional Concurrency Verifier)

Our tool is named *COCOV* which is a concurrency analysis tool. The tool has been continuously built up beginning the work from [5]. COCOV reads the models written in rc-Promela and generates CCS state graphs as *dot* [24] file format. If the models have *refactorby* the keyword of rc-Promela, it will collect all segments and output these segments as files. Our goal is make COCOV can do full-automatic compositional analysis with architecture refactoring.

4.3 The architecture

In Figure 4.1, we illustrate the flow of *COCOV*. Given a model written in rc-Promela, Scanner first translates the input characters into tokens. Second, in Parser stage, tokens are checked to see if they violate the grammar. In third stage, we use AST to generate CCS state graphs for refactoring transformation. In next section we

will describe how we implement every component and build of abstract data type into COCOV. In these stages, the functionality of Scanner and Parser is well-known, so we ignore it in this chapter.

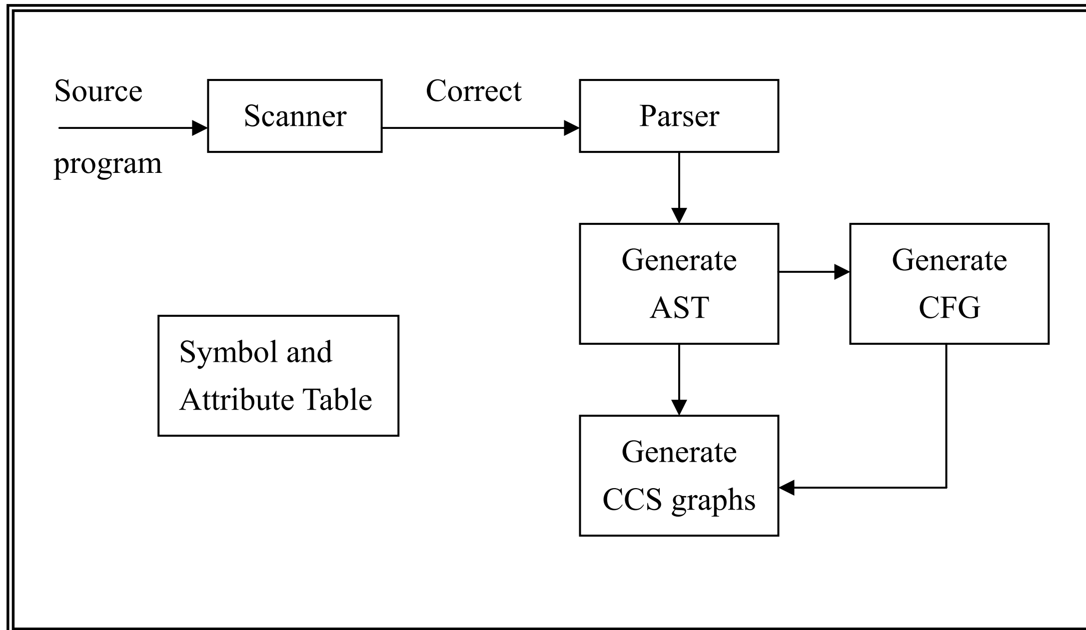


Figure 4.1 Structures of COCOV .

4.4 Grammar of abstract data type

Originally, the Promela does not support ADTs, except "array". In order to make COCOV to support other ADTs, we add new production rules into rc-Promela's grammar. The whole grammar is listed in Appendix.

Because the original grammar of rc-Promela has plenty of production rules, adding new production rules produces shift-reducce/reduce-reduce conflicts. How to avoid these conflicts becomes a challenge during our implementation. For example, some of the original grammars of rc-Promela:

```
one_decl: vis TYPE var_list
        | vis UNAME var_list
        | ENAME var_list
        | vis TYPE asgn '{' nlst '}'
        ;
var_list: ivar
        | ivar ',' var_list
        ;
ivar : vardcl
        | vardcl ASGN expr
        | vardcl ASGN ch_init
        ;
ch_init : '[' CONST ']' OF '{' typ_list '}'
        ;
```

Here we do not list the rule of “vis”, “asgn”, “nlst”, “vardcl”, and “type_list”. “TYPE” stands for basic data type supported by rc-Promela, such as “int”, “or bit”, etc. “UNAME” and “ENAME” stands for the variables that are declared in the source code. “ASGN” stands for the symbol “=”. “CONST” stands for the constant value. At first attempt, we let the declaration of QUEUE looks like CHANNEL and we use the rule of “ch_init” for expending the declaration of QUEUE, too. However we have two problems, one is that new production rule and rule “ch_init” are conflicted. The other one is the semantic of “ivar” is confused. To overcome these two problems, we decide to make QUEUE an independent production, isolated from rules for parsing CHANNEL. After trying some possible type of rules, for example

```
one_decl: vis TYPE var_list
        | vis EXCLUSIVE ATYPE var_list2
        | vis UNAME var_list
        | ENAME var_list
        | vis TYPE asgn '{' nlst '}'
var_list: ivar
        | ivar ',' var_list
        ;
var_list2: ivar2
        | ivar2 ',' var_list2
        ;
ivar : vardcl
        | vardcl ASGN expr
        | vardcl ASGN ch_init
        ;
ivar2    : vardcl
        | vardcl ASGN expr
        | vardcl ASGN adt_init
        ;
ch_init : '[' CONST ']' OF '{' typ_list '}'
        ;
adt_init : '[' CONST ']' OF ENAME
        |     '[' CONST ']' OF '{' typ_list '}'
        ;
```

Such solution can avoid the two practical difficulties we describe above.

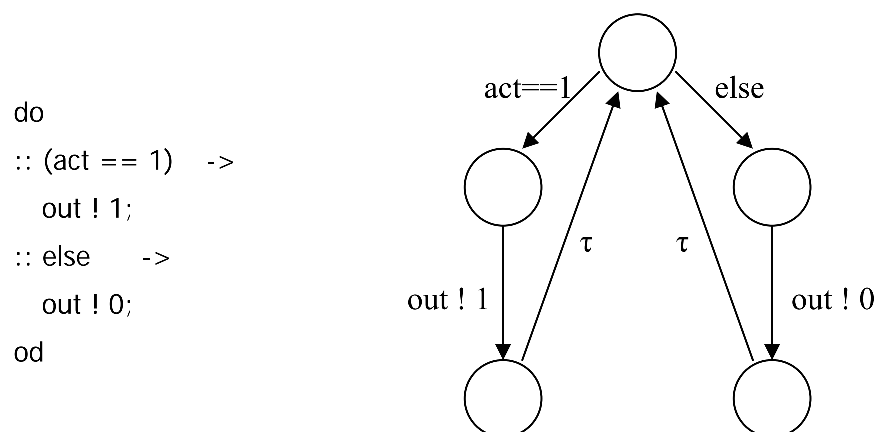
4.5 Building abstract syntax tree

While parsing and checking input tokens, the parser builds *abstract syntax tree* (AST) at the same time. The advantage of building abstract syntax trees is that we can traverse the tree to get the information that we need. There is no need to change

semantic routine behind the grammar production rules, so that, the grammar file will not be modified often. Such advantage is well-known in crafting a compiler. We add new AST node type in the AST data structure, for example, we add “QUEUE” AST node into original node data structure and add proper semantic routine for the new ADT production rules in the grammar.

4.6 Building control flow graph

In the past, we traverse AST to produce CCS state graph and process refactoring directives. However, our past experience shows that AST is not a convenient intermediate representation (IR) to generate CCS graphs. For example, when a model has do-loop or if-statement, for *COCOV*, it is difficult to detect the previous state which is created while traversing AST. For example, the code fragment and its CCS graph generated by COCOV is illustrated below.



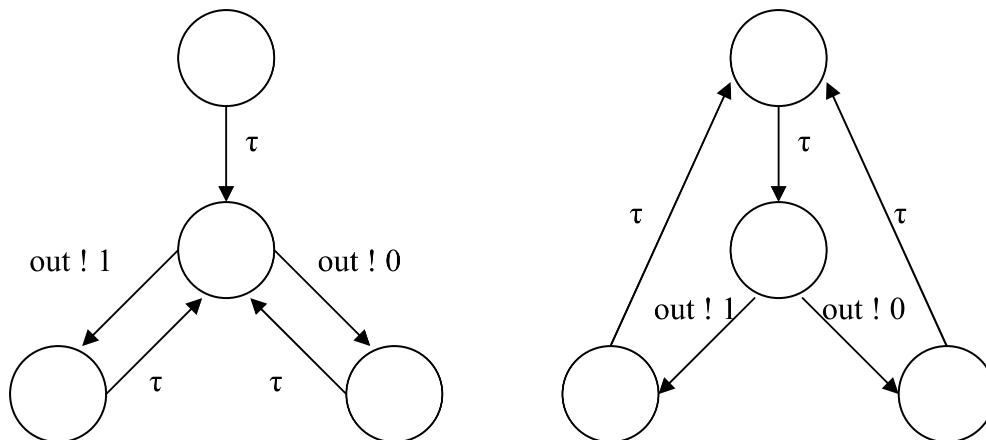
The symbol “out” is a channel.

It’s easy to generate the graph using AST, but if there are nested do-loop in the

source cod. For example,

```
do
:: (act == 1)    ->
   do
   :: out ! 1;
   :: out ! 0;
   od;
od
```

Here we list two CCS graphs:

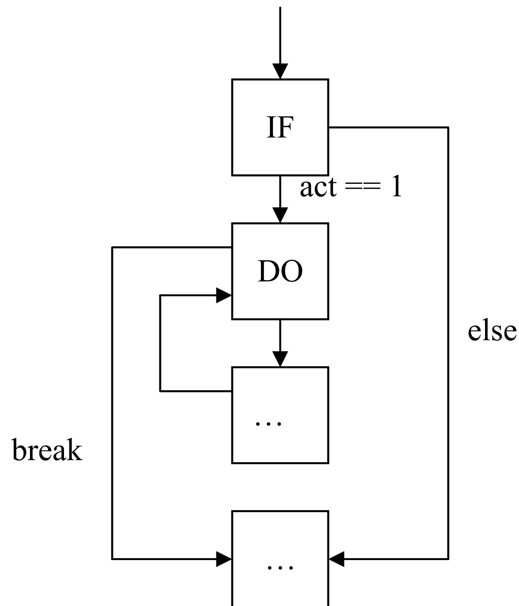


The left-hand graph is the correct one. While we only traverse AST to generate CCS graph, we probably generate the right-hand one if we miss the program flow. Because the program flow is not obviously obtained in AST, we have to record it in COCOV before generate CCS graph. Such problem also happens in refactoring.

The purpose of CFG is simple, it is served as a more convenient intermediate representation to generate CCS graph with AST. The main information which CFG keeps is the flow of a program. CFG contains similar information as AST but

represents a program as a flow of blocks.. For example,

```
if
:: (act == 1)    ->
   do
   :: out ! 1;
   :: out ! 0;
   :: break;
   od;
:: else ->
   in ? v1;
fi
```



Graph in right-hand side is the CFG of the code fragment at left-hand. In the CFG graph, each block has a pointer to point to its AST node. By this strategy, while traversing CFG, it is also traversing AST.

In [25][26], they listed the basic rules to generate CCS graph. We do not describe them again, but we focus on the ADTs. Take QUEUE for example, a tool user may declare

exclusive queue qpump = [2] of {client}

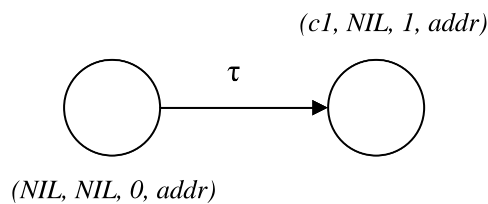
the *client* is enumtype which is {C1, C2}. COCOV will create a data structure of QUEUE in the memory. The structure of QUEUE includes an array. Here, we list some important elements of the data structure.

```
char* name
int* entry
int size;
int head
int tail
```

The *name* is “qpump”, the *entry* is an array of size two, and the *size* is record the filled size of *entry*. The *head* and *tail* will be used when QUEUE is block by “***refactorby***”. Because COCOV uses tuple *(var1,var2, ... addr)* to represent each state. Intuitively, we may use *qpump* to constitute a tuple like *(qpump, addr)*, but *qpump* is a structure of QUEUE and its *entry* and *size* element are the true primitive variables which truly affects the behavior of QUEUE. So, the actual tuple used to generate state graph is *(entry[0],entry[1], size, addr)*. For example, when COCOV traverse the code

qpump.push(C1);

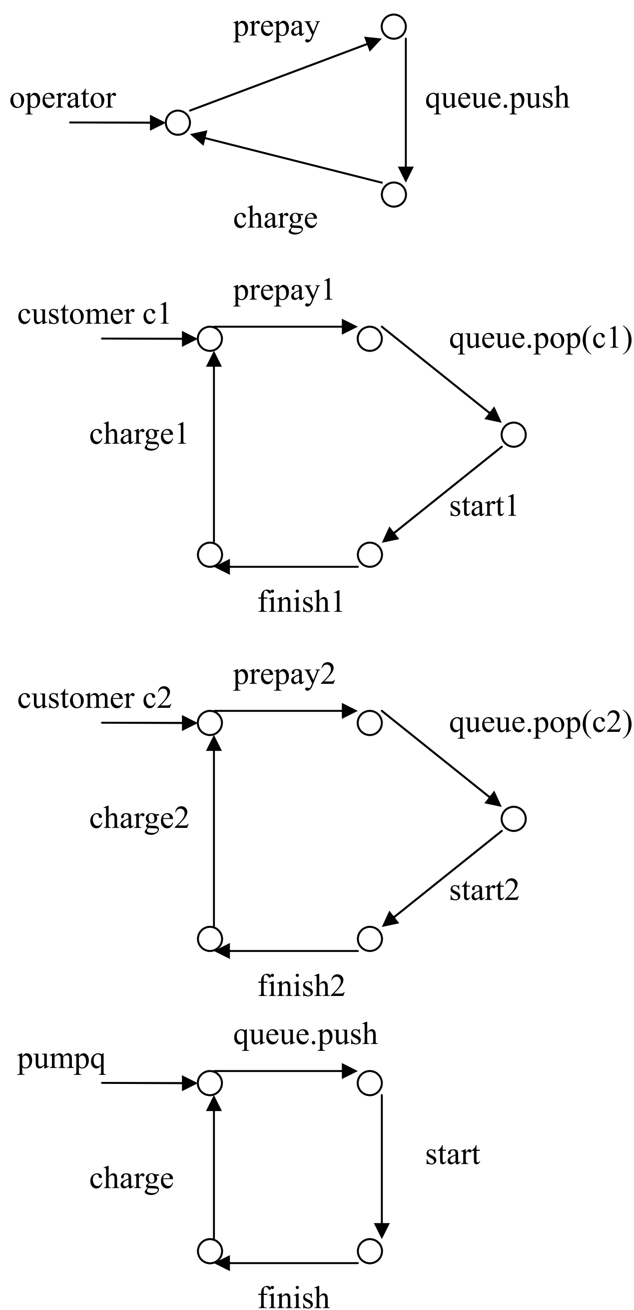
it will generate a CCS graph like



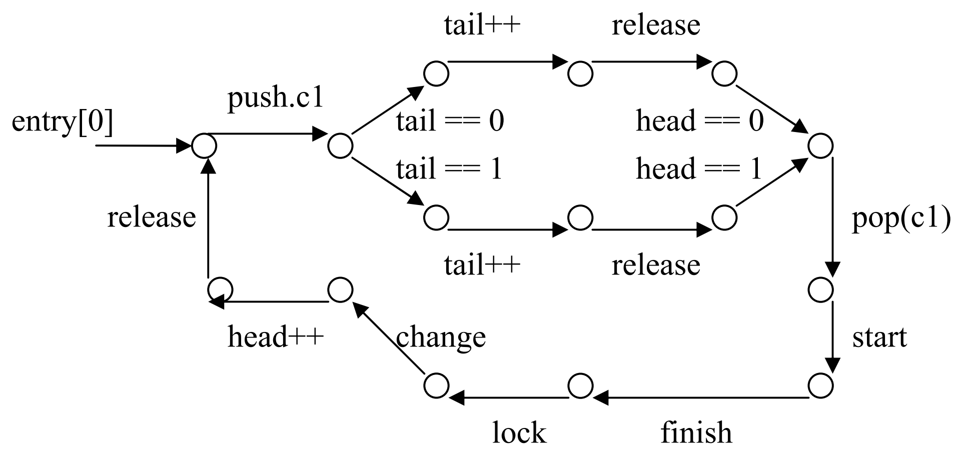
The method “*pop*” is dealt similary.

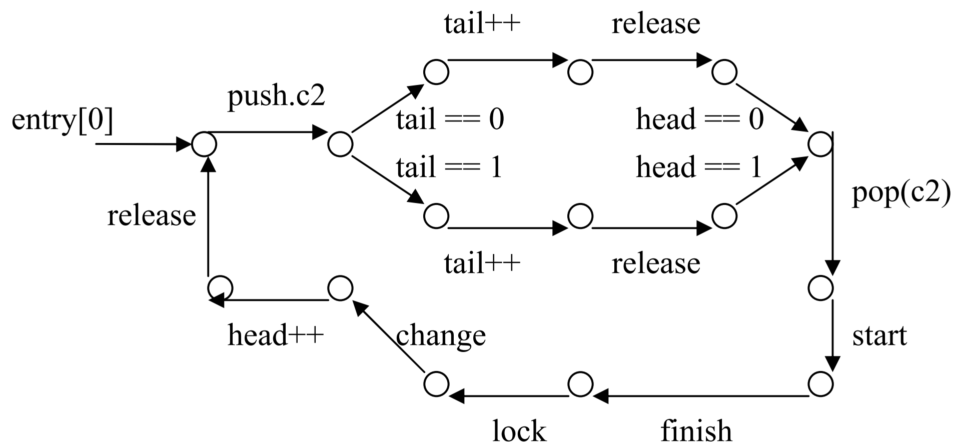
4.7 Model refactoring example using QUEUE

Since our design language supports ADTs for tool users, one of benefits is that we can choose the best implementation for the ADT. Such advantage can be applied to refactoring as well. In this section we continue using QUEUE example in refactoring. We change gas station to queue version in CCS format (the size of pumpq is 2)



When parsing code, COCOV knows *pumpq* is a variable of QUEUE type. At present, we know to refactor QUEUE is to refactor its *entry* element and the element of *head* and *tail* will be used for helping us to refactor. After refactoring, each element of entry is translated into a new process, for example a new process *entry[0]* is illustrated blow





From now on, the behavior of *QUEUE* is decomposed, but it is still equivalence to the original QUEUE. In a word, a tool user uses ADTs which are supported by COCOV and then COCOV can choose the best refactoring strategy for the ADT.