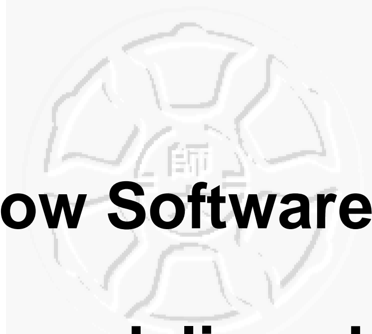

Chapter 3 How Software Analysis is sensitive to modeling choices

In the past, we have successfully refactored several systems. Most of them appear as examples in literatures, such as elevator system [19], furnace system [2], alternating bit protocol [2], etc. The tool support described in the previous section is sufficient for many systems which either have no presence of data values or have few data values to complicate the behavioral patterns. On the other and, we began to encounter systems with behaviors complicated by array, which caused refactoring to fail sometimes. In the attempt to address the problem, we began to realize that the problem is not only a concern to compositional analysis and refactoring but also to global analysis. In this chapter, we use two examples to describe the sensitivity of (compositional) analysis to modeling choices when array is used to implement abstract data types. Later, in next section, we propose our solution to lessen such sensitivity.

3.1 Chiron user interface system

In order to demonstrate the power of refactoring, we challenged refactoring in [5]

with an example called Chiron user interface [15]. It has been analyzed by [1][20]. Chiron user interface system is originally written in Ada. Chiron's design philosophy is to separate application code from user interface code. So, there are user interface agents called *artists* attached to selected data belonging to the application. At runtime, each artist can register events of interests do *dispatcher*. Whenever there is an operation call on the data, the dispatcher intercepts the call and notifies each of the artists associated with that data with the event.

Its Promela model is manually extracted from its Ada source code. The most complicated process in Chiron is a task called *dispatcher*. Dispatcher is responsible for accepting requests to register or unregister an event from an artist. The dispatcher use an array *e1_list*

```
mtype e1_list[no_of_artists];
```

to keep track the artists which have registered on event e1. For an artist to register an event *e1* to dispatcher, the following code fragment is invoked in dispatcher.

```
dispatcher_chan?    register_event,
                    artist_id,
                    event       ->
if
::   (event == e1)  ->
     i = 1;
     do
```

```
:: if
   :: (i > e1_size) ->
      e1_size++;
      e1_list[i-1] = artist_id;
      break;
   :: eles
   fi;
   if
```

The code first receives a command and two parameters from the channel. Two parameters are *artist_id* and *event*. Next, it checks if the *artist_id* is already in the array, using a loop index *i*. If not, the artist_id is appended to the tail of the array.

On the other hand, to unregister event *e1* form dispatcher by an artist, the following code is invoked.

```
dispatcher_chan? register_event,
                 artist_id,
                 event     ->
if
:: (event == e1) ->
   if
   :: (e1_size == 0) -> skip
   :: else ->
      i = 1;
      do
      :: (i > e1_size) -> break;
```

```
                :: eles ->
                   if
                   :: (e1_list[i-1] == artist_id) ->
                      do
                      :: (i >= e1_size) -> beak;
                      :: else ->
                         e1_list[i-1] = e1_list[i];
                         i++;
                      od
                      e1_size--;
                   :: else
                   fi;
                   i++;
                od
                e1_size[e1_size] = 0;
            fi;
        fi;
```

The code first search the array to check if the *artist_id* is in the array. If yes, the element (pointed by *i*) is deleted and all the elements behind *e1_list[i]* is copied to fill the deleted space.

To anyone who knows programming, such implementation is only one of many choices. Typically, if we prefer such implementation, we want to maintain the order of artists by their registration time. That is, an artist which registers *e1* earlier is stored in the front of array. However, in dispatcher task, we found no places where such order is concerned.

Regardless of how peculiar the behavior may be, the copying values between

adjacent elements of the array makes refactoring unable to decompose the dispatcher successfully. Our original hope is to remove segments which are related to artist a_i to a new process but segments related to a_i are entangled with segments related to other artists in a way that causes refactoring to fail.

Later, we fixed the code with another implementation choice – replaceing the array by a bit array.

```
Bit e1_list[no_of_artists];
```

In this implementation, if *e1_list[i]* = 0, it means artist a_i does not register event *e1*. if *e1_list[i]* = 1, it means artist a_i has registered event *e1*. Since there is no copying values between array elements, the behavior can be successfully refactored and bring compositional analysis into fully play.

In programming, we are accustomed to make implementation choices for different purposes. Similarly, the above two modeling choices produce two functionally equivalent models but unfortunately, result in great difference in analysis. Let the length of array *e1_list* be n, the number of artists. The original Chiron's behaviors can produce states which have growing rate proportional to

$$1+\sum_{i=1}^{n}\binom{n}{i}i!$$

On the other hand, using bit array has a growing rate proportional to 2^n. Although the

two scales are both exponential, the first growing rate is much worse than the second one. Besides, the second one can be refactored and verified compositionally. Slight differences in models can make big difference in analysis. This observation agrees with Holzmann's statement [14] that naively translated or blindly derived models are unlikely to work for analysis in most cases. The observation can be applied to “smart” model extractors like Banera and PathFinder as well. These state-of-art model extractors are unlikely to be able to recognize “bad” modeling choices and then “smartly” replace them by best modeling choices for analysis.

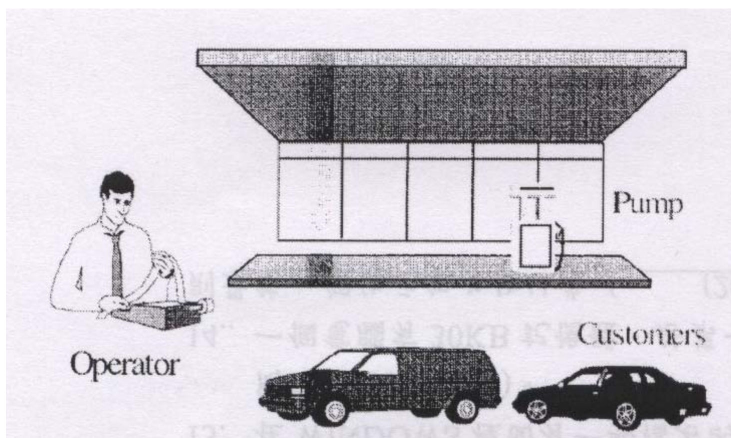

Figure 3.1 A gas station system.

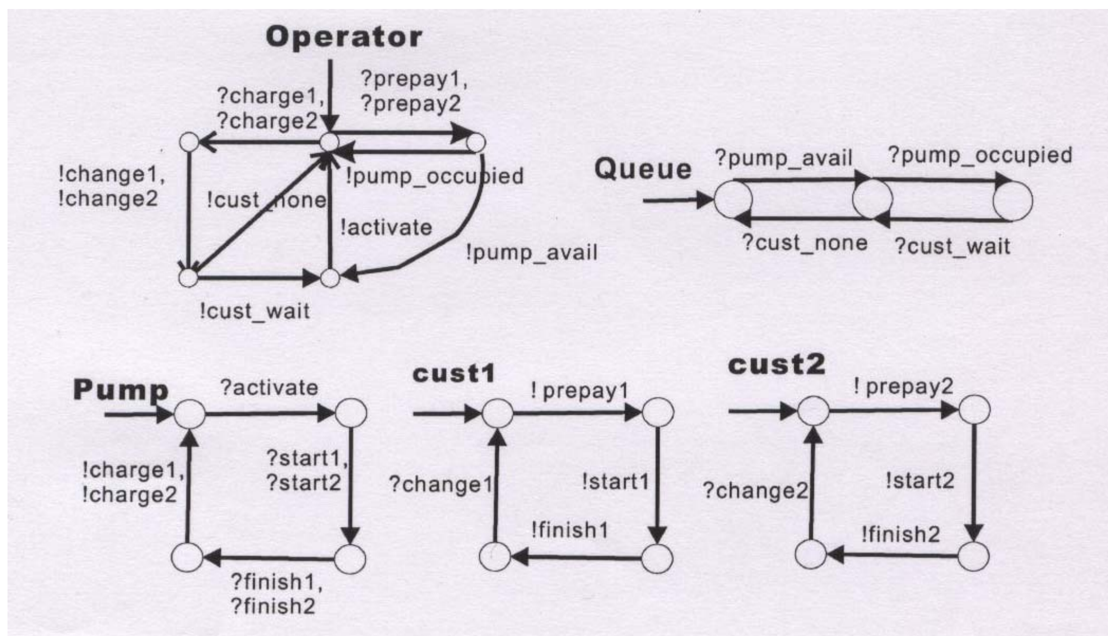

Figure 3.2 The state graphs of gas station system

3.2 Gas Station System

To convince that Chiron's case is not unique, we give another example from a gas station system. Gas station example was originally proposed by Helmbold and Lockham [11] (see Figure 3.1). Since then, the system has become one of the standard examples for software verification. The example models an automated gas-station with an operator, a pump, two customers, and a queue holding customer's requests. In principle, this example can be extended to arbitrary number of customers and pumps.

In Figure 3.2, we show the state graphs of gas station system in CCS semantics, where edge labels are matched in pair. N a typical run, a customer can contact the operator to prepay some money. Once prepaid, the operator activates the pump. After

the pump is activated, the customer can start filling the gasoline. Next, when the customer finishes the pump, the pump counts the volume of pumped gasoline and charges the customer an amount of money by notifying the operator. Operator receives the charged total and returns the changes (if there is any) to the customer.

In this version of gas station (see state graphs in Figure 3.2), the following scenario could happen: *Customer1* prepays the money and has the pump activated. Under normal circumstances, *customer1* is supposed to start pumping the gas. However, suppose he goes to restroom. In the meantime, *customer2* enters the station, prepays the money, and starts pumping (already activated) the gas immediately. After *customer2* finishes, the pump charges *customer1* for the volume of gas pumped by *customer2*. So, although there is a process called *queue* in this version, it does not actually enqueue the customer *id* to serialize the order of service.

In an attempt to remedy this problem, we modify the pump to use a queue to store the customer *id*. The queue-enabled pump accepts a customer *id* from the operator. Before the pump can be stared, the customer *id* must be verified. Image that in real scenario, after prepay, the operator gives customers an *id* to enter to computer at the pump station. A customer must enter a correct *id* (which should be the same as the front *id* in the queue) to start pumping the gas, In a first attempt, we modified the pump to make it queue-enabled using a circular queue. The queue-enabled process is

called *pumpq1*.

```
#define QLEN 3
enum client_t = {NIL, c1, c2};
proctype pumpq1() {
      client_t buf[QLEN];
      byte head = 0;
      byte tail = 0;
      client_t cid;
      do
      :: op?add, cid   ->
         /* accept id from operator */
         buf[tail] = cid;
         tail = (tail+1) % QLEN;
         /* insert id to queue */
      :: (head != tail && buf[head] == c1)       ->
         cust?remove, c1;
         /* accept id validation from customer */
         buf[head] = NIL;
         /* remove id from queue */
         head = (head+1) % QLEN;
         cust? start1;
         cust? finish1;
         op! charge, c1;
      :: (head != tail && buf[head] == c2)       ->
         /* accept id validation from customer */
         buf[head] = NIL;
         /* remove id from queue */
         head = (head+1) % QLEN;
         cust? start2;
         cust? finish2;
      od
}
```

In the example, *pumpq1* receives a customer id, *cid*, from *operator* via channel op and stores in array *buf* when an operator receives a prepay from customer *cid* and enters it to pump's computer. Later, only the customer who has *id* same as the *id* in the front of the queue can start pumping the gas. Other customer who does not have the correct id will block on his call to *cust!remove, ci*. Note that array *buf* has length 3 to avoid checking the boundary conditions.

We had rc-Promela translator to parse the code, but immediately found the modeling choice produces redundant states, which aggregate the state explosion problem. We fixed the code into the following *pumpq2*.

```
#define QLEN 2
enum client_t = {NIL, c1, c2};
proctype pumpq2() {
      client_t buf[QLEN];
      byte i;
      byte size = 0;
      client_t cid;
      do
      :: op?add, cid  ->
         /* accept id from operator */
         /* insert id to queue */
         buf[size] = cid; size++;
      :: (size !=0 && buf[0] == c1)  ->
         /* accept id validation from customer */
         /* remove id from queue */
```

```
        i = 1;
        do
        :: (i >= size) ->
           buf[i] = NIL; break;
        :: else  ->
           buf[i-1] = buf[i];
           i++;
        od;
        size--;
        cust? start1;
        cust? finish1;
        op! charge, c1;
     :: (head != tail && buf[head] == c2)      ->
        cust?remove, c2;
        /* accept id validation from customer */
        /* remove id from queue */
        i = 1;
        do
        :: (i >= size) ->   buf[i] = NUL; break;
        :: else  ->
        do
        :: (i >= size) ->   buf[i] = NUL; break;
        :: else  ->
           buf[i-1] = buf[i];
            i++;
        od;
        size--;
        cust? start2;
        cust? finish2;
        op! charge, c2;
     od
}
```

Just like the original dispatcher's behaviors in Chiron, this solution copies values behind the first elements to fill the empty space when the first element of the array is removed. Most programmers would agree that this solution is an inferior implementation compared to the *pumpq1* – low performance when queue length is large but surprisingly, *pumpq2* is a more effective model than *pumpq1* for the verification.

For *pump1*, rc-Promela translator selects (*buf[0],buf[1],buf[2],head, tail, addr*) as the tuple for traversing AST to produce state graph. So, if an id *c1* is in the queue, that state could be one of

(c1, nil, nil, 0, 1, addr)
(nil, c1, nil, 1, 2, addr)
(nil, nil, c1, 2, 0, addr)

depending on the values of head and tail. On the other hand, for *pumpq2*, rc-Promela translator selects (buf[0],buf[1], size, addr) as the tuple to traverse the code. The state can only be represented by (*c1, nil, 1, addr*). In other words, if the queue length is *n*, *pumpq1* will generate a state graph with size *n* + 1 times than *pumpq2* – a considerable impact to analysis.