Crafting a Promela Front-End with Abstract Data Types to Mitigate the Sensitivity of (Compositional) Analysis to Implementation Choices

Yung-Pin Cheng

Department of Information and Computer Education
National Taiwan Normal University
Taipei 106, TAIWAN.
ypc@ice.ntnu.edu.tw

Abstract. Recently, an active research topic in software verification is applying model checkers to programs, such as multi-threaded Java code. However, a program typically consists of more behaviors, such as operations on complicated data structures or implementation details which are typically made for some criteria like performance. A brute-force model extraction may produce a poor model for analysis engine. In this paper, we give examples to show how subtle changes in implementation may result in considerable differences in analysis, particularly to compositional analysis. Unfortunately, these implementation choices are made by programmers – people who typically do not possess the knowledge of verification. To mitigate such sensitivity, we advocate that verification tools should recognize and support abstract data types and, in the meantime, prohibit or suppress the use of array. Programming process behaviors with abstract data types can hide and converge the implementation choices. More importantly, abstract data types are informative. They provide essential information for tool automation to select a best implementation for analysis. In this paper, we describe the design and implementation of such a prototype tool which can parse systems written in Promela syntax.

1 Introduction

Automatic verification techniques such as model checking have been viewed as a promising method to ensure the quality of complicated systems. Many hard-to-detect errors, such as deadlocks, can be manifested by these techniques. In past decades, considerable progress has been made in these techniques. Several prototype tools, such as SPIN[15][16], SMV[20], have been built and applied to many software or hardware systems. In this paper, we focus on software verification.

Software typically has more states than hardware. The wide variety of software designs make software verification a more difficult task than hardware. For example, famous Ordered Binary Decision Diagram (OBDD) [3] which is widely used in hardware verification has no obvious merits in software verification (see Corbett's work in [10]). Besides, modeling a software system requires much more efforts, experiences, and human wisdom. In that work [10], Corbett also found subtle modeling differences

for the same system may produce obvious differences in the results of analysis in different tools. However, the objective of Corbett's work is to compare performance of various verification tools, so, finding and eliminating these differences to have a fair comparison of verification tools are his first priority.

Generally, most verification tools share the similar technology in exploring reachable states. They can be used for either software verification or hardware verification. Nevertheless, software verification researchers often prefer one tool over another for its capability of modeling software. For example, Spin [15] provides a model description language (MDL) called Promela, which has syntax close to a high-level programming language. Although it is originally designed for modeling communication protocols, many researchers have chosen it to verify concurrent programs written in Ada, Java and C. However, verifying systems written in these programming languages using Spin can be subtle. The abstraction of programs into Promela models requires human wisdom and experiences and is error-prone. In [17], Holzmann argued that a blindly derived model (for example, either by a naive automatic/manual extraction) is unlikely to work for verification in most cases. In other words, constructing an efficient and correct model requires human wisdom from experienced personals.

Several years ago, some works rose to the challenge. Research tools such as Bandera [11] and Pathfinder[13] have been developed to automatically extract models from Java source code. Their goal is to model-check Java source code by smartly extracting a model from Java code for analysis engines like SPIN. Bandera also introduces slicing techniques to abstract away the program behaviors that do not concern the interested properties (particularly liveness properties) so that the state explosion problem can be alleviated. In our opinions, these progresses mark an important milestone for automatic software verification.

Despite the progress described above, the fundamental barriers of software verification, however, still remain. Verification tools which analyze all processes at once are inevitably limited by the PSPACE lower bound in worst case; that is, the number of reachable states grows exponentially as the number of processes increases. In other words, any attempt to alleviate the state explosion is bound to fail in general but may work for some cases. For example, Bandera uses property to guide the program slicer to slice away the behaviors that are not concerned by the property, particular the liveness properties defined in linear-time temporal logic (LTL) formula. Such approach does not work for property like freedom of deadlocks. Reachable deadlocks must be manifested from all the possible behaviors.

To tackle the state explosion problem, a more promising approach is compositional analysis[7, 6, 8, 9, 12]. Compositional analysis avoids state explosion by dividing a whole system into many subsystems. Then, the techniques described above are used to analyze these subsystems. Ideally, the analysis of each subsystem would produce manageable and smaller state space and then each subsystem can be replaced by a simple interface process. The process is continued by combining the analysis of subsystems into a larger subsystems in a hierarchical fashion until the whole system is analyzed. Unfortunately, this ideal scenario seldom happens in practical cases. Compositional analysis is architecture sensitive. In many systems, no feasible hierarchies exist in their

as-built architecture; that is, The power of divide-and-conquer is often limited by the system architecture.

In this paper, we describe the design and implementation of a Promela front-end. This front-end is part of a compositional analysis tool suite which is under developing. The major feature of this front-end includes the new statements for refactoring[1] a process behaviors to overcome the problem of architecture sensitivity of compositional analysis. Another new feature of this front-end is the support of abstract data types. From our past experiences, we discovered a program (either written in Java, C, or Ada) may be written in a way that is poor[2] for analysis engine, particularly when abstract data types are implemented by array. We show two functionally equivalent process behaviors with two implementation choices can produce significant differences in analysis, particularly to compositional analysis with refactoring. To address the problem, we propose an extension of Promela. In this extension, we add abstract data types such as *queue* and *set* to its syntax. We show that encouraging the use of abstract data types and prohibiting the use of array can limit the wild implementation choices a programmer may make, therefore, mitigating the sensitivity of analysis. Furthermore, abstract data types are informative, providing essential information for tools to determine the best implementation for analysis without the need of code analysis.

Note that Spin is a sophisticated piece of software. Our objective is not to rework its features. Our long term goal is the construction of a software verification tool suite which is compositional-oriented. We select Promela as one of our input language because of its syntax simplicity and its popularity. This paper is organized as follows. In section 2, we give an overview of compositional analysis and our refactoring technique. In section 3, we give examples to explain why analysis is sensitive to implementation choices. Section 4 describes our design and implementation of a prototype tool. Finally, we end the paper with discussion, related work, and conclusions in section 5 and 6.

2 An overview of compositonal analysis and refactoring

In this section, we give an overview of two techniques, compositional analysis and model refactoring, so that readers can have a brief idea on the problem we want to address in this paper.

2.1 Compositional analysis

In a compositional analysis, we often have to group a set of processes into a subsystem (or a module). There are two basic criteria of a "good" subsystem. First, the processes inside the subsystem must not generate excessive state space. Second, the subsystem's state space must be able to be replaced by a much simpler *interface process* to represent the subsystem's state space. An interface process can be computed automatically

[1] This technique will be explained later.

[2] Note that a program may be written in a way that is poor for analysis but is good for performance or other measuring criteria.

by hiding internal interactions, minimizing the state space, and exporting the state and transitions (a.k.a interfaces) that will be used by its environment. Note that exporting state and transitions as interfaces can aggregate the state explosion problem if the interfaces are not simple (see [12]). So, simple interface is the key to a "good" subsystem. In other words, an effective subsystem should be *loosely coupled* to its environment so that the chance of having a simple interface process to replace it in compositional analysis is higher. At last, "good" subsystems and processes must produce another larger "good" subsystem in the composition hierarchy until the whole system is analyzed. Unfortunately, this ideal scenario seldom occurs in the compositional analysis of large and complicated systems.

2.2 Model refactoring

In Fig. 1 and Fig. 2, we show the state graphs of three example processes *X,Y,* and *S* in CCS semantics [21] (where synchronization actions are matched in pairs) and their synchronization structure. Such kind of structure, a star-shape structure, appears very often in practice, for example, a stateful server which communicates with clients via separate (or private) channels. Many systems can even have structures of multiple stars.

We say *S* is *tightly coupled* to its environment (which consists of *X* and *Y*) because it has complicated interfaces to its environment. Suppose *S* is a server and *X,Y* are clients. Image the number of clients is increased to a larger number. Any attempt to include *S* as a subsystem is bound to fail because of the complicated interfaces to its environment. That is, no feasible subsystems and composing hierarchies exist in this structure, particularly when client number is large.

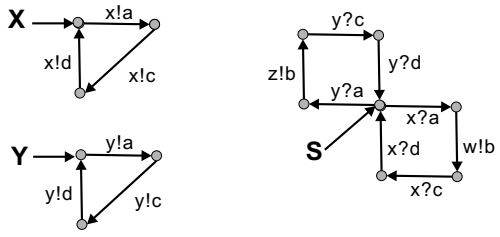

Fig. 1. A simple example with 3 processes X, Y, and S.

In [4, 5], we proposed an approach called *model refactoring* to enable compositional analysis for systems which are originally prohibited by their as-built architecture. The refactoring consists a set of transformations. Each transformation maintains the behavioral equivalence (weak bisimulation) of the model. By applying a sequence of transformations, a model P is gradually transformed into a model P' with new structure which is more amenable to compositional analysis. It consists in building a sequence of equivalent models, each obtained by the preceding ones by means of the application of a rule. The rules are aimed for restructuring the as-built structures which are not suitable for

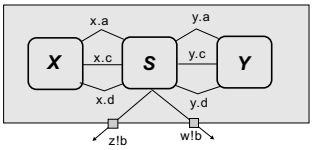

Fig. 2. The structure of the example.

compositional techniques. The goal is to obtain a transformed model whose structure contains loosely coupled components, where processes in each component do not yield state explosion.

The key transformations are to decompose centralized, complicated behaviors of a process into several small new processes while behavioral equivalence is preserved. In [5], we described the basic tool support[3] for refactoring and showed that a refactored elevator system can be analyzed up to hundreds of elevators but global analysis and compositional analysis (without refactoring) can only analyze up to 4 elevators.

For instance, we show the refactored *X,Y,* and *S* in Fig. 3 and the new synchronization structure in Fig. 4. In Fig. 3, the behaviors related to channel *x* (or to process *X*) is removed and wrapped into a new process *Sx.* Similarly, the behaviors related to channel *y* is removed and wrapped into a new process *Sy*. So, the rendezvous of *x!a, x!c,* and *x!d* are now redirected to *Sx*. However, *Sx* and *Sy* are now two individual processes which can execute concurrently, but their original joint behaviors in *S* can not. So, extra synchronizations (*e!lock and e!release*) are inserted to maintain behavioral equivalence; that is, before invoking *x!a* and *y!a, X* and *Y* are forced to invoke *e!lock* first. Then, at the end of *Sx* and *Sy*, *e!release* is used to free *S*.

The idea of refactoring equivalence is easy to explain. Let's image the modified processes (*X,Y*, and *S*) are contained in a black box. Image you are an external observer of the black box. The external behaviors of the black box are defined by *z!b* and *w!b*. In Fig. 2, the black box (which we call it B1) is implemented by 3 processes. The black box (we call it B2) in Fig. 4, on the other hand, is implemented by 5 processes. The external behaviors are also defined by *x!b* and *y!b*. Our refactoring must ensure the external behaviors are equivalent before and after a transformation. Intuitively, B1's external behaviors can be viewed as an specification. Then, we choose to implement the specification with 5 processes. Since we use 5 processes to do the same work which was originally done by 3 processes, extra communications for process coordination are inevitable. As long as the extra synchronizations are restricted inside the black box, the two black boxes behave equivalently to an external observer.

2.3 Tool support

From the above example, it may looks like identifying the behaviors and decomposing them can be done at the finite-state representation. In practice, automatic refactoring

[3] The tool support can successfully refactor many systems in an automated fashion, particularly the behavioral patterns which do not involve complicated data structures.

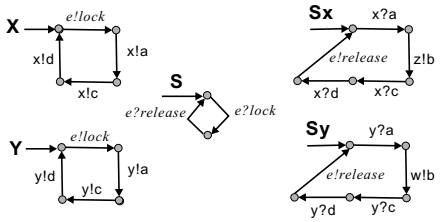

Fig. 3. The refactored example system.

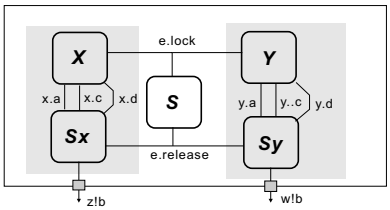

Fig. 4. The refactored structure of the example system.

does not work at this level of representation. Some important information needed by refactoring engine is lost at this level. The refactoring automation must be made when a CCS state graph is created. So, a refactoring statement is added to Promela's syntax. We chose a subset of Promela syntax and built a parser to translate Promela code into CCS [21] state graph. For example, *S* in Fig. 1 can be written in Promela as follows:

```
mtype = { a,b,c,d } ;
chan x = [0] of mtype ; chan y = [0] of mtype ;
chan z = [0] of mtype ; chan w = [0] of mtype ;
proctype S() {
   do (0)
   refactorby x, y {
   :: x?a (1) ->
      z!b (2); x?c (3); x?d (4);
   :: y?a (5) ->
        w!b (6); y?c (7); y?d (8);
   }
   od
}
```

Note that in the example, we mark a statement with "*(addr)*" as the address of each statement. To translate a Promela code into a CCS state graph is trivial. First, we collect the values of local variables and the address of current statement to make a tuple like $(v_1, v_2, ..v_n, addr)$, where v_i are values of local variables and $addr$ is the current statement address. In process *S*, there are no local variables, so the only element in the tuple

is statement address *(addr)*. Since the process starts at statement address 0, so we use (0) as initial state. We begin parsing the abstract syntax tree (AST) of the code. When we parse a channel statement which sends or receives a message, we create a new outgoing transition to a new state. The new state has its $addr$ updated to next statement address and the outgoing transition has label in the form of "*ch?msg*" or "*ch!msg.*" The traversal is continued until no more new states are explored.

To activate refactoring, tool users can add keyword *refactorby* to enclose a block of statements they want to refactor. For example, to obtain the result in Fig. 3 and Fig. 4, we use *refactorby* to notify refactoring to separate the behaviors by channel name. In general, process behaviors can be distinguished by channel name, variable's values, etc.

When refactoring mode is activated, we translate Promela code into segments of behaviors. For example, the sequence of transitions beginning from *x?a* and ending with *x?d* is called a segment. These segments are grouped according to group options, the parameters behind keyword ***refactorby***. Next, segments are wrapped into a new process such as *Sx* in Fig. 3 by a unified transformation.

Note that, in principle, it is impossible for our transformation to decompose any process behaviors and make compositional analysis work in general, otherwise, we would have solved the notorious state explosion problem. So, it is easy for a malicious tool user to write a peculiar process behaviors which makes refactoring fail. However, under normal circumstances, most process behaviors are written in common patterns. Our ultimate goal is to make refactoring work for most behavioral patterns.

3 Sensitivity of (compositional) analysis

In the past, we have successfully refactored several systems. Most of them appear as examples in literatures, such as elevator system[22], furnace system[23], alternating bit protocol[2], etc. The tool support described in the previous section is sufficient for many systems whose process behaviors either have no presence of data values or only have simple data values to enrich its behavioral patterns. On the other hand, we began to encounter systems with behaviors complicated by array. When process behaviors are complicated by array, segmented behaviors may be interwined and tangled and refactoring transformations are no longer feasible. We described some of the behaviors as follows.

3.1 Chiron user interface system

The first example is called Chiron user interface [18]. It has been analyzed by [1, 24]. 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[4] belonging to the applications. At runtime, each artist can register *events* of interests to *dispatcher.* Whenever there is an operation call

[4] You can consider the data as an object and the object's values (or attributes) is linked to a visualization tool called artists. In other words, an artist can be viewed as a graphic drawing unit for the data.

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*. When an artist registers an event *e1* to dispatcher, the following code fragment is executed 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 ;
        :: else
        fi;
        if
        :: (e1_list[i-1] == artist_id) ->
           break ;
        :: else
        fi;
        i++ ;
     od
```

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* from dispatcher by an artist, the following code is executed.

```
dispatcher_chan? unregister_event, artist_id, event ->
if
:: (event == e1) ->
    if
    :: (e1_size == 0) -> skip
    :: else ->
       i = 1 ;
       do
       :: (i> e1_size) -> break ;
       :: else ->
          if
          :: (e1_list[i-1] == artist_id ) ->
             do
             :: (i>= e1_size) -> break ;
             :: 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. In other words, the elements in *e1_list* are shifted. To anyone who know programming, such implementation is only one of many choices. Typically, if we prefer such kind of 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 clues where such order is concerned.

3.2 Implementation alternative

Since the order of registration is not a concern to dispatcher, a better implementation choice is using 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 on event *e1*. If *e1_list[i] = 1*, it means artist $a_i$has registered on event *e1*.

3.3 Analysis of implementation choices

In programming, we are accustomed to make implementation choices for some reasons, perhaps for performance or maintenance. Similarly, the above two implementation 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. We call the array of original dispatcher as queue array. The original dispatcher'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 for global analysis.

To compositional analysis, the implementation with queue array produce interwined and tangled behaviors which cannot be refactored effectively. It can be only analyzed up to 2 artists. On the other hand, the behaviors with bit array can be refactored effectively into loosely coupled components. Its refactored structure can fully take the advantage of divide-and-conquer. It can be analyzed up to 14 artists. Note that, in Chiron, increasing an artist means adding a new process to the system.

In Fig. 5(a), we show the tangled behaviors of the queue array implementation with two artists. In the figure, a registration event is abbreviated into "?Rx" where x is the

type of event. An unregistration event is abbreviated into "?Ux." Beside each state, we print the contents of array *e1_lst[]*. On the other hand, Fig. 5(b) shows the behaviors using bit array, which presents some form of symmetry. This behavior can be effectively transformed by refactoring (It uses value processes to model value change for each array element *e1_lst[i]*. Readers who are interested in these technical details, please refer to [5]).

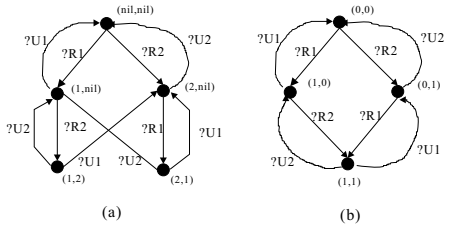

Fig. 5. The behavioral patterns of two implementations.

This observation agrees with Holzmann's statement [17] that naively translated or blindly derived models are unlikely to work for analysis in most cases. The key points of our observation are:

- **Programmers often make implementation choices for performance or other criteria, but not for analysis. A model which is directly extracted from a program inherits the program's implementation choices, which can be "poor" for analysis but makes no significant difference in runtime execution.**
- **Subtle changes in implementation may produce significant differences in (compositional) analysis. Analysis tends to magnify small and slight implementation changes.**

So, with these observations and the state explosion problem, we believe model checking programs is just a beginning. We should be cautious and conservative on the the general applicability and practicability of these tools.

3.4 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 [14](see Fig. 6).[5] 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.

[5] The figure is borrowed from Cheung and Krammer [8].

In Fig. 7, we show the state graphs[6] of gas station system in CCS semantics, where edge labels are matched in pair. In 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.

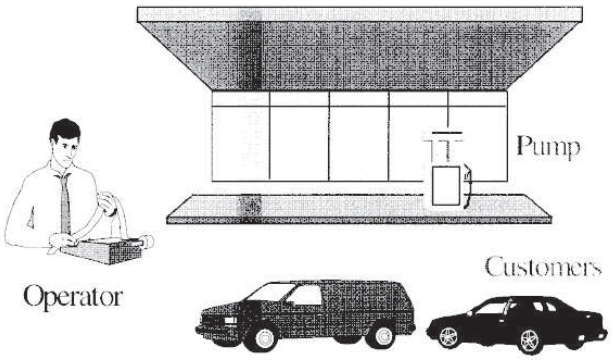

Fig. 6. A gas station system.

Fig. 6. A gas station system.

In this version of gas station (see state graphs in Fig. 7), 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 started, 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
```

[6] The state graphs are borrowed from Cheung and Krammer [8] but we rename its edge labels into pairwise notations.

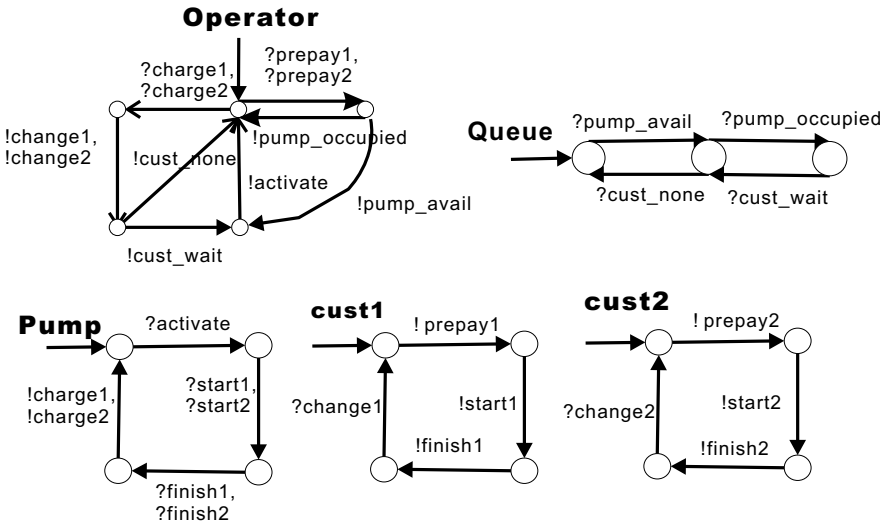

Fig. 7. The state graphs of gas station system

```
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) ->
     cust?remove,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)->
     cust?remove,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] = NIL;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 *pumpq1*, rc-Promela translator selects *(buf[0],buf[1],buf[3], 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.

4 Using abstract data types to mitigate sensitivity of analysis

The problems described in the previous section can be the tip of the iceberg. Analysis tools have always been geared towards being adopted by industry to assure high software quality. However, if an analysis tool must depend on the virtue of the code or limit itself to trained experts, the fruit of software verification research will always be limited in research community.

To address the problem, the first approach we tried is attempting to analyze the array usage in the code and gather useful informations for refactoring automation. Suppose we can analyze and understand the semantics of Chiron's dispatcher task mechanically, we can replace it with bit array to produce best analysis results. Unfortunately, that is hard and impractical. We re-analyze the essence of the problem and have three observations, which are:

1. Array is the very basic blocks for constructing abstract data types (ADTs). Most process behaviors with array operations can be summed up to some kinds of ADT operations.
2. An ADT may have several implementation choices. However, these implementation choices can be hidden by ADT interfaces.
3. The process of using an ADT for a task encourages precise and high-level thinking.

These observations are the basis of this work. Image an extreme scenario where analysis tools are integrated into a programming environment. A programmer is responsible for a critical task which requires concurrent programming. Under this condition, array is prohibited by the environment because the code must be analyzable. A programmer would be forced to select appropriate ADTs to complete his work. In the case of Chiron's dispatcher, he would select *set* as the most appropriate ADT for the job.

In the scenario, the usage of *set* provides explicit directives for tool automation. Selecting best implementation (i.e., bit array in this case) becomes straightforward. There is no need to incorporate other static analysis techniques for program comprehension. Consequently, sensitivity to implementation choices is controlled and mitigated. In this paper, we implement two frequently used ADTs into Promela to demonstrate our idea. They are:

QUEUE

```
DECLARATION SYNTAX:
  queue qname = [n] of {enumtype}
METHODS:
  void push(enumtype val); // to add a value val to the queue
  enumtype pop();   // return and remove the frist element of queue
  enumtype front();   // return the value of first element
```

SET

```
DECLARATION SYNTAX:
  set sname = [n] of {enumtype};
METHODS:
  insert(enumtype val);  // add val to a set
  erase(enumtype val);   // remove val from a set
  int find(enumtype val); // return the index of the value
```

Where *enumtype* is a type defined by **enum** keyword, another new function we add to Promela to extend **mtype** of Promela. A user can use

$$enum\ \ clien\_type = \{c1, c2, c3\};$$

to define an enumeration type in Promela. Both the ADTs are exclusive; that is, values in these containers can not be duplicated. The implementation of containers which allow duplicated elements can be quite different from the exclusive ones. Currently, exclusive ones can satisfy our need.

Using the new ADTs, the dispatcher can be rewritten into

```
enum artist_type = {a1,a2};
set e1_lst = [2] of {artist_type};
.....
dispatcher_chan? register_event, artist_id, event ->
if
:: (event == e1) ->  e1_lst.insert(artist_id);
:: (event == e2) ->  e2_lst.insert(artist_id);
fi
.....
```

In this example, not only the process behaviors are concise and easy to understand, but also process behaviors are forced to "converge" on this one. Our tool automatically select the best implementation choice for (compositional) analysis, which is transparent to tool users. With the prevalence of object-oriented programming languages nowadays, the constraint (to prohibit or suppress the use of array) may not be strong as it looks but the merits are manifold.

4.1 Object-oriented tool design and implementation

Crafting the experimental parser described in this paper requires a lot of works. This pilot prototype[7] has been worked towards to a new framework illustrated in Fig. 8. In the figure, boxes colored in grey are tools which have been developed or under developing. White boxes are tools which can be developed by other parties or will be developed by us in the future. Finally, boxes decorated with grey stripes are tools constructed by others. DOT is graph visualization tool from AT&T. Fc2tool [19] is a tool suite from INRIA, France, which consists of tools to enumerate and minimize CCS state graphs.

In the framework, our ADT-promela parser reads a Promela file and produces several *cfg* (control flow state graph) files and a symbol table[8], where each process has a *cfg* file. A control flow state graph is like Fig.9. Each state can be interpreted as the address of a statement in the program. Each edge is then attributed by a statement. The statement is stored in the form AST (abstract syntax tree) which can be evaluated by a postfix traversal algorithm. The *cfg* file format uses tags which can be parsed and read easily. It can be modified to XML syntax if necessary.

The spirit of our design is to use files to communicate among tools. This design avoids building a monolithic tool which can be harder to modify or evolve. The framework also encourages cooperative work. Tools with a language front-end typically do not use flow graph of a program for data exchange because it is language dependent. However, we found that control flow state graph is where many tools start with. For instance, our state graph translator traverses it to generate communicating finite state machines (such as CCS or CSP state graphs). Simulation tools can use it to exercise traces. Other tools such as program slicers can work on this representation as well. These reasons make us to design it into a format which can be shared by language front-ends and analysis back-ends.

To deal with control flow state graph, we design a set of object-oriented CFG classes (shown in Fig. 10), which can parse a *cfg* file to construct a control flow state graph.

[7] The old prototype (without ADT) described in [5] has been torn apart and restructured towards the structure in Fig. 8.

[8] Spin is capable of outputting control flow state graphs and symbol tables. However, that output is not designed for the purpose like ours.

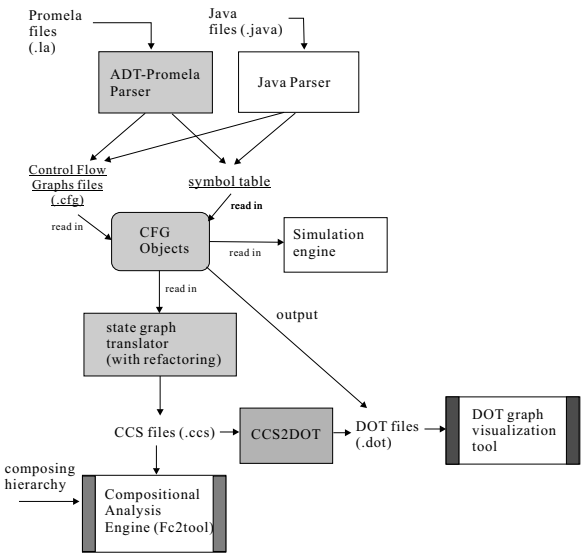

Fig. 8. The framework of tool implementation.

We introduce an inheritance hierarchy to separate what is language dependent from what is language independent. A language front-end can implement their own AST to store a statement. Next, it should implement an overidden *eval()* method (see class *CFG_edge_promela*). Other tools, such as a simulation engine, will only invoke *eval()* to evaluate the AST of a statement and update variable values in the symbol table. The details of AST (which is language dependent) are transparent to other tools. By this design, we can implement a language-independent state graph translator or a simulation engine.

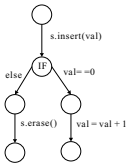

Fig. 9. A control flow state graph.

The implementation of an ADT method is actually done in the overridden *eval()* of class *CFG_edge_promela*. For example, when a *set* is defined, we create a bit array in the symbol table. Later, when a statement *s.insert(i)* is evaluated, *eval()* sets the *ith* element in the bit array. In our state graph translator, bit array will be included as a tuple (see section 2.3) to traverse the control flow state graph to generate CCS state graphs.

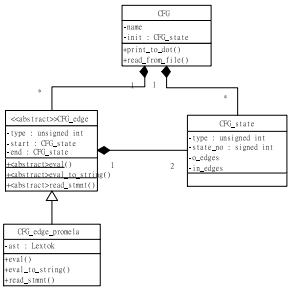

Fig. 10. The inheritance structure of CFG objects.

The tools described in this paper will be gradually released at URL link *http://www.ice.ntnu.edu.tw/~ypc/ArCats.htm.*

5 Related works and discussion

From the best of our knowledge, Java model extractor Pathfinder has not supported abstract data types from Java standard library. In other words, it assumes the behavior of a Java thread does not involve ADTs. Bandera can process code which uses *vector*. However, *vector* is just another safe-to-use array. Supporting ADTs requires a great amount of efforts and works. The reasons they do not support ADTs from standard library are simply an issue of cost [13]. They do not address the sensitivity problem we described in this work. Besides, they focus on global analysis, whereas compositional analysis and refactoring are our major concerns.

Supporting abstract data types in Promela can be done in another way. For example, we can use CPP's macros to implement ADTs. This approach may be easier to maintain and implement. There is no need to modify Promela's grammar. However, it is more difficult to cope with object-oriented syntax nowadays. For example, many ADT objects may have methods all named *add()*. Solving naming conflicts in macro programming is more difficult. It is also more difficult to cope with our design framework in previous section.

6 Conlusions

In this paper, we describe a special phenomenon of software verification – analysis (particularly compositional analysis) is sensitive to implementation choices when array is used to implement complicated data structures. We give examples to show that an implementation choice which is the definite choice in the view of programming may be a poor choice for analysis. On the other hand, a poor implementation choice from the view of programming can be a good choice for analysis. We show that such sensitivity

can be mitigated if ADTs are supported and the usage of array are suppressed or prohibited. Two ADTs SET and QUEUE are implemented in a prototype tool. Models rewritten with ADTs have obvious advantages. First, using ADTs forces process behaviors to converge. Programmers have less room to make their own implementation choices to endanger analysis. Second, the ADTs provide useful automation information. Selecting the best implementations for ADTs behaviors becomes straightforward. There is no need to incorporate any static analysis or program comprehension techniques.

References

1. G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical Report UM-CS-1999-069, Dept. of CS, University of Massachusetts, November 1999. (in preparation).
2. K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of ACM*, 12(5):260–261, 1969.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
4. Y. Cheng. Refactoring design models for inductive verification. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA2002)*, pages 164–168, Rome, Italy, July 2002.
5. Y.-P. Cheng, M. Young, C.-L. Huang, and C.-Y. Pan. Towards scalable compositional analysis by refactoring design models. In *Proceedings of the ACM SIGSOFT 2003 Symposium on the Foundations of Software Engineering*, pages 247–256, 2003.
6. S. C. Cheung, D. Giannakopoulou, and J. Kramer. Verification of liveness properties using compositional reachability analysis. In *5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 227–243, Zurich, Switzerland, September 1997.
7. S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–377, October 1996.
8. S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 8:49–78, January 1999.
9. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of 4th IEEE Symposium on Logic in Computer Sciences*, pages 353–362. IEEE Computer Society Press, 1989.
10. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 2(3):161–180, March 1996.
11. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
12. S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proceedings of the 2nd International Conference of Computer-Aided Verification*, pages 186–204, 1990.
13. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
14. D. P. Helmbold and D. C. Luckham. Debugging Ada tasking programs. In *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pages 96–105, St. Paul, October 1984.
15. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.
16. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

17. G. J. Holzmann. Designing executable abstractions. In *Proceedings of the second workshop on formal methods in software practice*, pages 103–108, Clearwater Beach, Florida USA, March 1998.
18. R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.
19. E. Madelaine and R. de Simone. *The FC2 Reference Manual*. Available by ftp from cma.cma.fr:pub/verif as file fc2refman.ps, INRIA.
20. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
21. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.
22. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, May 1992.
23. W. J. Yeh and M. Young. Re-designing tasking structure of Ada programs for analysis:A case study. *Software Testing, Verification, and Reliability*, 4:223–253, 1994.
24. M. Young, R. Taylor, D. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for Ada programs: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, Jan 1995.