# Using SPIN to Model Check Concurrent Algorithms, using a translation from C to Promela

Ke Jiang, Bengt Jonsson
Department of Computer Systems, Uppsala University, Sweden
keji4095@student.uu.se, bengt@it.uu.se

## Abstract

*This paper addresses the problem of automatically verifying correctness of concurrent algorithms, e.g., as found in concurrent implementations of common data structures, using model checking. In order to use a model checker to analyze programs in, e.g., C, one must first translate programs to the input language of the model checker. Since our aim is to use SPIN, we present an automated translation from a subset of C to Promela. This translation is able to handle features not covered by previous such translations, notable pointer structures and function calls. We illustrate the application of our translation to a concurrent queue algorithm by Michael and Scott.*

## 1 Introduction

In order to utilize multicore processors, increased attention is given to algorithms that achieve maximal concurrency to achieve performance, often resulting in complex algorithms. Concurrency libraries are an important example of this trend: the Intel Threading Building Blocks, the `java.util.concurrent` package, and other libraries support the programmer by providing concurrent implementations of familiar data abstractions such as queues, sets, or maps. Implementations of such libraries typically do not follow locking disciplines, and instead use lock-free synchronization for gaining performance (e.g., [9, 14]). Such implementation are typically hard to get correct, as witnessed by many bugs found in published algorithms (e.g., [2, 10]).

This paper addresses the problem of automatically verifying correctness of, or finding bugs in, complicated but not too large, concurrent algorithms, e.g., as found in concurrent implementations of common data structures. For such algorithms, standard race-detection tools (e.g., [3, 13]) are of limited use. An analysis of correctness must consider the detailed operation of the algorithm implementation. This is an application, where model checking techniques can be helpful. There are several powerful model checkers that can be used to analyze correctness of concurrent algorithms. Perhaps the most popular is the model checker SPIN [7], which analyzes algorithms expressed in the Promela modeling language. However, SPIN can not analyze programs written in, e.g., C, directly: first the programs must be translated into Promela.

The goal of our work is to use SPIN to automatically analyze concurrent algorithms written in (a significant subset of) C. This paper describes a step in this direction: an automated translation from a subset of C to Promela. Several such translations exist, but are not adequate for our purposes (see the next section). The translation of control structures is relatively straight-forward. In this paper we will describe our techniques for translating pointer structures and functions, which are not supported by Promela. Features that are not yet supported in our implementation are a completely faithful representation of atomicity in basic C statements (e.g., when they concern two global variables), and consideration of weak memory models which become relevant when standard locking disciplines are not obeyed.

We illustrate our translation by applying it to a concurrent blocking queue algorithm by Michael and Scott [11].

**Some Related Work** There are several implemented translations from C to Promela, developed for different purposes. ModEx [5] is a tool intended for model checking ANSI-C code by mechanically extracting Promela models from C program. There are several features of C, most notably functions and pointers, that are not handled by ModEx. To increase its applicability, it is possible to embed C code directly into Promela statements [4, Sec. 10]. However, for model checking, one must manually describe how to manipulate the associated state: it is unclear whether this can be done for pointer structures.

This work is inspired by our previous work [8], where we used a *manual* modeling in Promela to analyze concurrent algorithms under a particular weak memory model. This work is a step towards mechanising that approach, where we handle important aspects of C, such as functions and

pointers. In this paper, we still do not consider the impact of weak memory models, but use sequential consistency.

## 2 A Motivating Example

Our ambition is to automatically perform model checking on algorithms written in a subset of C, such as the below concurrent queue algorithm, translated from the pseudocode code published by Michael and Scott [11]. Algorithms like this are often used to avoid making highly used data structures bottlenecks in concurrent programs. It is also the original algorithm of our previous manual translation, which we would like to compare with. With respect to the original algorithm, we changed the type of `dequeue` function from `boolean` to `int` since there is no boolean type in ANSI C. The `lock` and `unlock` functions represent the special locking and unlocking functions which need to be treated specially. The C code of the algorithm is as follows.

```
struct node_t {
  struct node_t *next;
  int value;
};

struct queue_t {
  struct node_t *Head;
  struct node_t *Tail;
  int H_lock;
  int T_lock;
};

void initialize(struct queue_t *Q)
{
  struct node_t *dummy =
    malloc(sizeof(struct node_t));
  dummy->next = 0;
  dummy->value = 0;
  Q->Head = Q->Tail = dummy;
  Q->H_lock = Q->T_lock = 0;
}

void enqueue(struct queue_t *Q, int val)
{
  struct node_t *node =
    malloc(sizeof(struct node_t));
  node->value = val;
  node->next = 0;
  lock(&Q->T_lock);
    Q->Tail->next = node;
    Q->Tail = node;
  unlock(&Q->T_lock);
}

int dequeue (struct queue_t *Q, int *pvalue
    )
{
  struct node_t *node;
  struct node_t *new_head;
  lock(&Q->H_lock);
    node = Q->Head;
    new_head = node->next;
    if (new_head == 0) {
      unlock(&Q->H_lock);
      return 0;
    }
    *pvalue = new_head->value;
    Q->Head = new_head;
  unlock(&Q->H_lock);
  free(node);
  return 1;
}
```

Our ambition is to analyze this program by translating it to a Promela model, which can be analyzed using SPIN. The translation must faithfully represent the control structure of the above operations, as well as the dynamic memory operations. It is also desirable to preserve the structuring into functions, which can be called in different ways by different test harnesses. In the next section, we consider how this can be accomodated in Promela.

## 3 Promela

The modeling language Promela is designed for state space exploration of finite-state models of systems of communicating processes. The main structuring unit is processes, which communicate via channels or global variables. In order that models have finite and not too large state spaces, individual processes should typically have a small set of local variables, each with a not too big domain. Thus, dynamic allocation of data, or pointer structures, can not be represented, and there is a small set of control flow primitives.

In order to use SPIN, we must solve some problems of representation in Promela.

- Promela does not support dynamically allocated data structures. To overcome this, we must explicitly model dynamically allocated memory, e.g., by arrays.
- Promela does not support functions. We must therefore emulate function calls, e.g., by creation of processes, and represent the parameter passing and function return by appropriate synchronizations.
- We must consider the issue of creating suitable test harnesses to exercise different configurations of concurrent processes.

## 4 Syntax-directed translation

In this section, we give an overview of our translation from (our subset of) C to Promela.

### 4.1 Data Structures and Statements

Promela has C-like syntax (actually it is compiled to C), but supports limited data-types. Our current translator handles a subset of ANSI C. Integer and array variables will be preserved in the same way as in C, while pointers and structures (and linked lists) will be handled separately. The translation of pointers is described in Section 4.2.

Simple statements are translated straight-forwardly because of the similarity between statements in C and Promela. Control structures, including jumps (except `return` which is described in Section 4.3), selection, and iteration are translated to corresponding structures in Promela. We illustrate these aspects by the following extract, which is the bubble sort algorithm on `a[n]`.

```
#define n 5
struct record{
  int min;
  int max;
};
...
  int x, y;
  struct record r;
  for(x=0; x<n; x++)
    for(y=0; y<n-1; y++)
      if(a[y]>a[y+1]) {
        int temp = a[y+1];
        a[y+1] = a[y];
        a[y] = temp;
      }
  r.min = a[0];
  r.max = a[n-1];
...
```

The C code fragment will be translated as follows.

```
typedef record {
  int min;
  int max;
}

...
  int x; int y;
  record r;
  int temp;
  x = 0;
  do
  :: (!(x < 5)) -> break
  :: else ->
     y = 0;
     do
     :: (!(y < 4)) -> break
     :: else ->
        if
        :: (a[y] > a[(y + 1)]) ->
              temp = a[(y + 1)];
              a[(y + 1)] = a[y];
                    a[y] = temp
        :: else
        fi;
        y ++
     od;
     x ++
  od;
  r.min = a[0];
  r.max = a[4];
...
```

### 4.2 Dynamically Allocated Memory

Promela does not support dynamically allocated data structures or pointers, hence we must emulate them. Since the memory can be seen as a large sequential array containing all program data, one option could be to define a global array, indexed by addresses, that simulates all memory. This option obviously is too space-consuming for analysis by model checking. A less costly approach is to allocate arrays which hold only objects that are dynamically allocated by the program. We must allocate one array for each data type, since different data-types have different structures and fields. For each data type, which has an object that is accessed by a pointer in the program, we allocate an array, and the pointers to this data type will be translated into indices in the array. With this approach, we obviously cannot handle pointer arithmetic; only type-respecting pointer operations are permitted. Our current translator has implemented this solution for two data types: `int` and user-defined `struct`.

Since the array for a data type must be allocated statically at the beginning of program execution, we must also record which cells are allocated and which are not. This can done by a special "unallocated" value, or by a separate array of booleans (as we have done).

Figure 1 illustrates how a linked list is formalized in Promela. The left part in Figure 1 shows the Promela array, and the right part shows a linked list that is represents.

**Translating Pointer operations** From the previous description, it follows that assignments to pointers is translated to normal integer assignments. Allocation of a new object in the C program must be done using `malloc`, and a new object will be allocated to the first available position in the corresponding array (by convention named `type_valid[9]`). Freeing an object is analogous. A small example of the translation of a code snipped follows.

```
struct person{
  int age;
};
...
  struct person *ptr;
  ptr = malloc(sizeof(struct person));
```

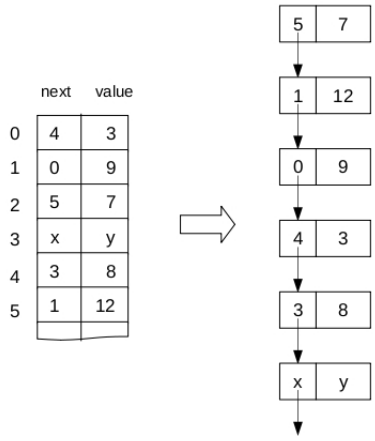


**Figure 1. Array representation in Promela of a linked list**

```
  ptr->age = 24;
  free(ptr);
...
```

Its translation is

```
typedef person {
  int age;
}
person person_mem[9];
int person_valid[9];

...
  int person_ct;
  int ptr; int tmp;
  atomic {
    person_ct = 1;
    do
    :: (person_ct >= 9) -> break
    :: else ->
       if
       :: (person_valid[person_ct] == 0) ->
             person_valid[person_ct] = 1;
             break
       :: else -> person_ct ++
       fi
    od;
    assert (person_ct < 9);
    tmp = person_ct;
    person_ct = 1
  };
  ptr = tmp;
  person_mem[ptr].age = 24;
  d_step {
    person_valid[ptr] = 0;
    person_mem[ptr].age = 0
  };
...
```

### 4.3 Translating functions

The translation of functions and function calls is essential for translation of C programs, e.g., so that we can define test harnesses that call functions that manipulate shared data, or translate recursively defined functions. Since there is no function concept in Promela, they must be emulated. The only structuring concept available in Promela is the process, whence we will use them to emulate functions. Each function definition will be translated to a Promela process type declaration which uses the corresponding local variables. A function call will be translated into the creation of a process. Parameters that are passed in the function call will be translated into parameters that are instantiated when the function is created. Return values will be passed by passing a message back with the corresponding value as parameter to the calling context. This idea was originally proposed by Holzmann [6, Ch. 5.9], and we have altered it slightly. We present a more detailed account in the following paragraphs.

**Function definitions** in C will be translated into process type declarations in Promela, whose parameters will be the formal parameters of the function, plus an additional channel parameter that is used to communicate the return value. The body of the function is translated "as usual". A `return` statement in C is translated into the transmission of a return message, after which process should be terminated immediately.

**Function call** will be translated into a sequence of statements in Promela. Before calling a function, a channel must be defined for the transmission of return values. If the same function is called several times within the same context, only one channel is necessary. The actual call is translated into the creation of a process (using the `run` statement of Promela), to which the actual parameters, and the recently defined channel are passed. The created process will execute asynchronously. Meanwhile, the calling context will be blocked on its next statement, which is a (blocking) receive of the return value on the return channel. The two functions, `lock` and `unlock`, which are used to represent the locking and unlocking mechanisms in C, will not be translated into process creations but into an `atomic` structure simulation.

**Return statements** A `return` statement in C passes a value to its caller and terminates the function. So we have to emulate both of these two properties. The first property is emulated by transmitting a return value over the synchronous return channel. If a `void` is returned, a dummy value `0` will be sent. The second property is emulated by a `goto end` statement immediately after the transmission.

**Recursive functions** The above scheme for translating function definitions and calls automatically handles recursion. The main reason is that the declared return channels are local to the calling process, and that therefore multiple independent local copies of the return channels are created by the scoping mechanism of Promela. Promela allows at most 255 processes to be created in an execution, so this puts a limit on the recursion depth: this limitation is not significant for the analyses that we envisage.

**An example** The code below demonstrates the function translation.

```
int test(int a, int b){
  if(a >= b) return a;
  else return b;
}

int main(){
  int x = 2;
  int y = 3;
  return test(x, y);
}
```

The Promela translation is as follows.

```
proctype test(chan in_test; int a; int b){
  if
  :: (a >= b) -> in_test ! a; goto end
  :: else -> in_test ! b; goto end
  fi;
end :
  printf ("End of test")
}

proctype main(chan in_main){
  chan ret_test = [0] of { int };
  int x; int y; int tmp;
  x = 2;
  y = 3;
  run test(ret_test, x, y);
  ret_test ? tmp;
  in_main ! tmp;
  goto end;
end :
  printf ("End of main")
}

init {
  chan ret_main = [0] of { bit };
  run main(ret_main);
  ret_main ? 0
}
```

## 5 Implementation

We have implemented the described translation with the aid of CIL [12]. CIL handles the parsing and semantic analysis for C programs, relieving us from a lot of work. CIL compiles valid C programs into a few core constructs with a clean semantics to make the result program easier to analyze. For example, all looping constructs, i.e., `while`, `for` and `do..while`, in the input C programs are transformed to a single form `while`; all function bodies are given explicit `return` statements; and a group of involved files are merged into one. CIL can also be guided to perform user defined transformation like the translator module we developed. Our C to Promela translator is actually realized as a new module which can be employed by CIL, and will use the CIL-analyzed syntax trees as its input format. After obtaining the syntax trees of a C program, our translator will preprocess the C syntax tree, transform it into Promela syntax tree and generate Promela code. The whole translation procedure is illustrated in Figure 2.

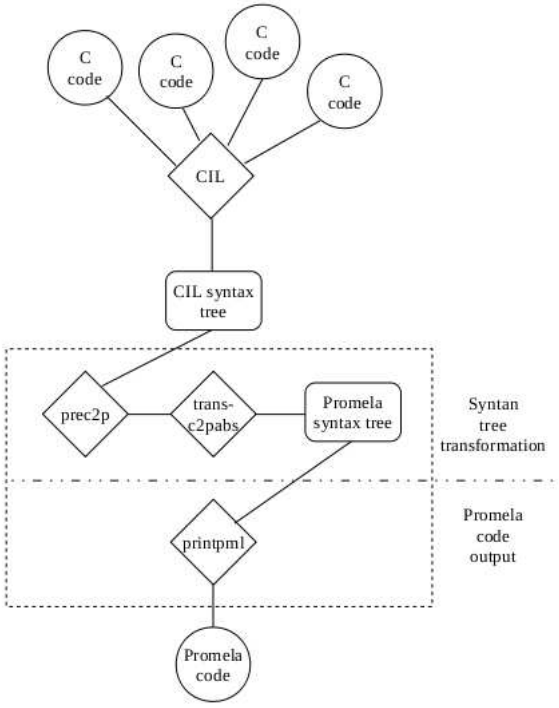


**Figure 2. Translation flow**

## 6 Experiments

In order to evaluate the translator we developed, we decided to perform some test harnesses including the ones we previously introduced [8] on the translation of blocking queue algorithm. Then we can evaluate the automatic translation comparatively with our previous manual translation. We denote test harnesses in a condensed notation (following [1]), using a sequence of `E` (for `enqueue`) and `D` (for

dequeue) in each thread, and separating threads by |. For example, the test (EE|DD) has two threads, which respectively calls two enqueue/dequeue operations sequentially [8]. We created two functions, "e" and "d", for simulating two independent threads where our enqueue and dequeue processes are running. Therefore, when translating these two functions, they will not be synchronized with their calling process. The reason is an obvious one, that the channel synchronization will keep the executions in sequence, but we need these two processes running simultaneously, i.e. the test (EE|DD) will cause process e and d to instantiate enqueue and dequeue processes twice independently.

First, we performed a simple test (E|D), in which 851 states were generated and 5.044Mb memory in the rapid verification procedure (almost 0 second consumed). Then we tested the translation with a number of more advanced ones, and present the results in Table 1. The C code implementation of the test harness (EEE|DDD) is presented in the Appendix, and the codes for the others are similar. The automatic Promela translation for the test harness is also presented, in which we manually inserted the checking mechanism for dequeued values, which checks that the dequeued values is a prefix of the enqueued values, possibly interleaved with dequeue operations that return "empty queue".

| Test harness | States | Mem. | Time |
|---|---|---|---|
| E\|D | 851 | 5.044 | ~0 |
| EE\|DD | 14,467 | 11.587 | 0.13 |
| EE\|DDD | 29,506 | 19.009 | 0.26 |
| EEE\|DDD | 138,751 | 74.575 | 1.33 |
| EEEE\|DD | 128,611 | 69.399 | 1.26 |
| EEEE\|DDDD | 1,101,416 | 583.583 | 11.5 |
| EEEEE\|DDDD | 3,181,607 | 1727.196 | 34.4 |
| EEEEE\|DDDDD | 7,894,946 | 4403.891 | 87.5 |

**Table 1. Verification results**

We also considered the most complicated test harness from [8], (E|E|E|E|D|D). Verifying this would consume a large amount of memory, exceeding the physical memory of our computer (8Gb). So, we analyzed it non-exhaustively, using the *Supertrace/Bitstate* mode. There were 28,131,899 states generated which was equivalent to 18887.383Mb memory usage.

Of all the experiments we performed on the *Exhaustive* mode, two were the same with our previous work [8], namely (E|D) and (EEEEE|DDDDD). For the first one, the manual translation generated 900 states comparing to the 851 states of the automatic translation. For the second experiment, there were about 100,000 states generated for our manual work, while there were about 7,900,000 states generated by the automatic translation.

# 7 Conclusion

We have presented an implementation of a translation from a subset of C to Promela, motivated by the intention to automatically model check concurrent programs, e.g., implementing shared data structures. Nontrivial aspects of the translation include dynamically allocated data structures and function calls. In our previous work [8], we performed such a translation manually. Compared with that translation, our current implementation still does not have the same capabilities: it still does not adequately divide statements into atomic parts (e.g., a statement like x++ is not broken up into atomic parts), and does not consider the impact of weak memory models, but translates into sequential consistency: this means that errors related to these features may not be detected in subsequent model checking analyses. When comparing the manual and automated translations with regard to efficiency, we see that for corresponding test runs, the automated translation generated about an order of magnitude more states. Considering that the manual translation also takes into account complications caused by weak memory models, there appears to be room to optimize the translation for more efficient model checking.

Although the C code we processed is still a subset, we can already foresee that many complex portions of C could also be handled using the methods we proposed, e.g. multi-dimensional arrays and pointers to pointers. We also plan to add the impact of weak memory models. Future work should also include optimizing the current translation (e.g., making sequences of statements without global effects atomic). An interesting extension would also be to include garbage collection (as is done, e.g., Promela models considered in [15]).

# References


[1] S. Burckhardt. *Memory Model Sensitive Analysis of Concurrent Data Types*. PhD thesis, Univ. of Pennsylvania, 2007.

[2] S. Doherty, D. Detlefs, L. Groves, C.H. Flood, V. Luchangco, P.A. Martin, M. Moir, N. Shavit, and G.L. Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain*, pages 216–224, 2004.

[3] C. Flanagan and S.N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.

[4] Gerard J. Holzmann. *Spin Model Checker, The Primer and Reference Manual*. Addison Wesley, September 2003.

[5] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Softw. Test., Verif. Reliab.*, 11(2):65–79, 2001.

[6] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[7] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[8] Bengt Jonsson. State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). *SIGARCH Comput. Archit. News*, 36(5):65–71, 2008.

[9] M.M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI 2004*, pages 35–46, 2004.

[10] M.M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[11] M.M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.

[12] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 14(4):391–411, Nov. 1997.

[14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

[15] T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *Proc. 16th Int. SPIN Workshop*, volume 5578 of *Lecture Notes in Computer Science*, pages 261–278, Grenoble, France, 2009. Springer.

# Appendix

- Translation for the concurrent queue algorithm

```
int int_mem[9];
int int_valid[9];
typedef node_t {
  int next;
  int value;
}
node_t node_t_mem[9];
int node_t_valid[9];
typedef queue_t {
  int Head;
  int Tail;
  int H_lock;
  int T_lock;
}
queue_t queue_t_mem[9];
int queue_t_valid[9];

proctype initialize(chan in_initialize;
                    int Q){
  int node_t_ct; int dummy; int tmp;
  atomic {
    node_t_ct = 1;
    do
    :: (node_t_ct >= 9) -> break
    :: else ->
       if
       :: (node_t_valid[node_t_ct] == 0) ->
             node_t_valid[node_t_ct] = 1;
             break
       :: else -> node_t_ct ++
       fi
    od;
    assert (node_t_ct < 9);
    tmp = node_t_ct;
    node_t_ct = 1
  };
  dummy = tmp;
  node_t_mem[dummy].next = 0;
  node_t_mem[dummy].value = 0;
  queue_t_mem[Q].Tail = dummy;
 queue_t_mem[Q].Head = queue_t_mem[Q].Tail;
  queue_t_mem[Q].T_lock = 0;
  queue_t_mem[Q].H_lock =
               queue_t_mem[Q].T_lock;
  in_initialize ! 0;
  goto end;
end :
  printf ("End of initialize")
}

proctype enqueue(chan in_enqueue; int Q;
    int val){
  int node_t_ct; int node;
  int tmp; int mem_5;
```

```
  atomic {
    node_t_ct = 1;
    do
    :: (node_t_ct >= 9) -> break
    :: else ->
       if
       :: (node_t_valid[node_t_ct] == 0) ->
             node_t_valid[node_t_ct] = 1;
             break
       :: else -> node_t_ct ++
       fi
    od;
    assert (node_t_ct < 9);
    tmp = node_t_ct;
    node_t_ct = 1
  };
  node = tmp;
  node_t_mem[node].value = val;
  node_t_mem[node].next = 0;
  atomic {
    (queue_t_mem[Q].T_lock == 0) ->
        queue_t_mem[Q].T_lock = 1
  };
  mem_5 = queue_t_mem[Q].Tail;
  node_t_mem[mem_5].next = node;
  queue_t_mem[Q].Tail = node;
  queue_t_mem[Q].T_lock = 0;
  in_enqueue ! 0;
  goto end;
end :
  printf ("End of enqueue")
}

proctype dequeue(chan in_dequeue; int Q;
    int pvalue){
  int node; int new_head;
  atomic {
    (queue_t_mem[Q].H_lock == 0) ->
        queue_t_mem[Q].H_lock = 1
  };
  node = queue_t_mem[Q].Head;
  new_head = node_t_mem[node].next;
  if
  :: (new_head == 0) ->
        queue_t_mem[Q].H_lock = 0;
        in_dequeue ! 0;
        goto end
  :: else
  fi;
  int_mem[pvalue] = node_t_mem[new_head].
      value;
  queue_t_mem[Q].Head = new_head;
  queue_t_mem[Q].H_lock = 0;
  d_step {
    node_t_valid[node] = 0;
    node_t_mem[node].next = 0;
    node_t_mem[node].value = 0
  };
  in_dequeue ! 1;
  goto end;
end :
  printf ("End of dequeue")
}
```

- Test harness implementation for (`EEE|DDD`)

```
int ins = 1;

void i(struct queue_t *Q)  {
  initialize(Q);
}

void e(struct queue_t *Q) {
  enqueue(Q, ins);
  ins ++;
  enqueue(Q, ins);
  ins ++;
  enqueue(Q, ins);
  ins ++;
}

void d(struct queue_t *Q) {
  int res;
  int *val = malloc(sizeof(int));
  res = dequeue(Q, val);
  res = dequeue(Q, val);
  res = dequeue(Q, val);
}

int main(){
  struct queue_t *queue;
  queue = malloc(sizeof(struct queue_t));
  i(queue);
  e(queue);
  d(queue);
  return 0;
}
```

- Translation for test harness (`EEE|DDD`). The manually added codes for checking purpose were the second line and the three `if` structures at the end of process `d`.

```
int ins = 1;
int check;

proctype i(chan in_i; int Q){
  chan ret_initialize = [0] of { bit };
  run initialize(ret_initialize, Q);
  ret_initialize ? 0;
  in_i ! 0;
  goto end;
end :
  printf ("End of i")
}
```

```
proctype e(int Q){
  chan ret_enqueue = [0] of { bit };
  run enqueue(ret_enqueue, Q, ins);
  ret_enqueue ? 0; ins ++;
  run enqueue(ret_enqueue, Q, ins);
  ret_enqueue ? 0; ins ++;
  run enqueue(ret_enqueue, Q, ins);
  ret_enqueue ? 0; ins ++;
end :
  printf ("End of e")
}

proctype d(int Q){
  chan ret_dequeue = [0] of { int };
  int int_ct;  int res;  int val;  int tmp;
  atomic {
    int_ct = 1;
    do
    :: (int_ct >= 9) -> break
    :: else ->
       if
       :: (int_valid[int_ct] == 0) ->
             int_valid[int_ct] = 1;
             break
       :: else -> int_ct ++
       fi
    od;
    assert (int_ct < 9);
    tmp = int_ct;
    int_ct = 1
  };
  val = tmp;
  run dequeue(ret_dequeue, Q, val);
  ret_dequeue ? res;
  if
  :: (int_mem[val] == check) -> check ++
  :: else -> assert(res == 0)
  fi;
  run dequeue(ret_dequeue, Q, val);
  ret_dequeue ? res;
  if
  :: (int_mem[val] == check) -> check ++
  :: else -> assert(res == 0)
  fi;
  run dequeue(ret_dequeue, Q, val);
  ret_dequeue ? res;
  if
  :: (int_mem[val] == check) -> check ++
  :: else -> assert(res == 0)
  fi;
end :
  printf ("End of d")
}

proctype main(chan in_main){
  chan ret_i = [0] of { bit };
  int queue_t_ct;   int queue;   int tmp;
  atomic {
    queue_t_ct = 1;
    do
    :: (queue_t_ct >= 9) -> break
    :: else ->
       if
       :: (queue_t_valid[queue_t_ct] == 0) ->
             queue_t_valid[queue_t_ct] = 1;
             break
       :: else -> queue_t_ct ++
       fi
    od;
    assert (queue_t_ct < 9);
    tmp = queue_t_ct;
    queue_t_ct = 1
  };
  queue = tmp;
  run i(ret_i, queue);
  ret_i ? 0;
  run e(queue);
  run d(queue);
  in_main ! 0;
  goto end;
end :
  printf ("End of main")
}

init {
  chan ret_main = [0] of { bit };
  run main(ret_main);
  ret_main ? 0
}
```