Potential Approaches for the Project

- Using LLMs to convert C code to Promela code.
- Using Lex for lexical analysis and Yacc for parsing to convert the C code to the intermediate code.

1 Data Types in Promela

Promela (Process Meta Language) provides several data types similar to C but with some differences in their usage and constraints.

Basic Data Types

1. **bit**: Stores 0 or 1 (1-bit storage).
   bit flag = 1;
2. **bool**: Stores true (1) or false (0).
   bool status = true;
3. **byte**: Stores values from 0 to 255 (8-bit storage).
   byte counter = 0;
4. **short**: Stores values from -32,768 to 32,767 (16-bit storage).
   short value = -100;
5. **int**: Stores values from -2,147,483,648 to 2,147,483,647 (32-bit storage).
   int number = 1000;
6. **unsigned**: Stores only positive values.
   unsigned x = 50;
7. **mtype**: Enumerated user-defined types.
   mtype = { READY, RUNNING, WAITING };
   mtype state = READY;

Data Structures

Arrays

```
byte arr[5];  // Array of 5 bytes
arr[0] = 10;
```

Channels

```
chan ch = [2] of { int };  // Channel storing int values
```

```
ch!5;  // Sending data to channel
ch?x;  // Receiving data from channel
```

2 Writing and Running Promela Code

Writing Promela Code

1. Define processes using proctype.
2. Use init to start execution.
3. Use run to create process instances.

Array

C	Promela
`int a[10];`	`int a[10];`

Structure

C	Promela

`struct node{`
`  int value;`
`};`
`struct node n[10];`
	`typedef node{`
	`  int value;`
	`}`
	`node n[10];`

Conditional Expressions

C	Promela
a = (b > c) ? b : c;	if :: (b > c) -> a = b :: else -> a = c fi

Break

C	Promela

while(1){
  if(a>b) break;
  else a++;
}
	do
	:: (a>b) -> break
	:: else -> a++
	od

If

C	Promela
if(a>b) b++; else a++;	if :: (a>b) -> b++ :: else -> a++ fi

Switch

C	Promela

switch(x){
  case 0:
  a++;
  b++;
  case 1:
  a- -;
  b- -;
  default:
  break;
}
	if
	:: (x ==0) -> a++; b++
	:: (x == 1) -> a- -; b- -
	:: else -> break
	fi

Loops

int i = 1;
for(; i< n; i++) {
  a = a + i;
}
	int i;
	i = 1;
	do
	:: !(i < n) -> break
	:: else -> a = a + i; i++
	od

int i = 1;
while(1){
  if(i > n) break;
  i++;
}
	i = 1;
	do
	:: true -> i ++; if :: (i > 5) -> break :: else fi
	od;

C	Promela

Recursive Functions

C	Promela

int gcd(int x, int y){
if(y==0) return x;
else
return gcd(y, x % y);
}
	proctype gcd(chan in_gcd; int x; int y){
	chan ret_gcd = [0] of { int };
	int tmp;
	if
	:: (y == 0) ->in_gcd ! x;
	goto end
	:: else ->run gcd(ret_gcd, y, (x % y));
	ret_gcd ? tmp;
	in_gcd ! tmp;
	goto end
	fi;
	end :
	printf ("End of gcd")
	}

Functions

```
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);
}
```

```
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
}
```

C	Promela

Pointers

C	Promela

struct node {
  struct node *next;
  int value;
};
	typedef node {
	  int next;
	  int value;
	}
	node node_mem[9];
	int node_valid[9];

Pointers Assignment & Reference

C	Promela

struct node {
  struct node *next;
  int value;
};
struct node *tail;
 	
void test(struct node *tmp){
  tail->next = tmp;
  tail = tmp;
}
	typedef node {
	  int next;
	  int value;
	}
	node node_mem[9];
	int node_valid[9];
	int tail;
	
	proctype test(chan in_test; int tmp){
	  node_mem[tail].next = tmp;
	  tail = tmp;
	  in_test ! 0;
	  goto end;
	end :
	  printf ("End of test")
	}

Malloc & Free

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

void test(){
    struct node *new =
malloc(sizeof(struct node));
    free(new);
}
```

```
typedef node {
    int next;
    int value;
}
node node_mem[9];
int node_valid[9];

proctype test(chan in_test){
    int malloc_node_c;
    int new;
    int tmp;
    atomic {
        malloc_node_c = 1;
        do
        :: (malloc_node_c >= 9) -> break
        :: else ->
            if
            :: (node_valid[malloc_node_c] == 0) ->
                node_valid[malloc_node_c] = 1;
                break
            :: else -> malloc_node_c ++
            fi
        od;
        assert (malloc_node_c < 9);
        tmp = malloc_node_c
    };
    new = tmp;
    d_step {
        node_valid[new] = 0;
        node_mem[new].next = 0;
        node_mem[new].value = 0
    };
    in_test ! 0;
    goto end;
end :
    printf ("End of test")
}
```

C	Promela

3 Key Concepts in Promela

proctype

A proctype defines a concurrent process in Promela. It encapsulates a sequence of actions that can be executed in parallel with other processes.

Uses:

- Helps in modeling concurrent systems.
- Allows defining independent execution units.
- Can be instantiated multiple times using run.

Features:

- Supports communication via channels.
- Can contain control structures like loops and conditionals.
- Can interact with other processes through shared variables and synchronization mechanisms.

```
proctype exampleProcess() {
  byte x = 0;
  do
  :: x < 5 -> x++;
  :: else -> break;
  od;
}

init {
  run exampleProcess();
}
```

inline

The inline construct is used to define reusable code blocks similar to macros in C. It helps to avoid code repetition and improve readability.

Uses:

- Reduces redundancy by creating reusable code blocks.
- Enhances code modularity and readability.
- Helps in defining frequently used instructions.

Features:

- Works like a macro with direct substitution.
- Cannot contain process instantiations (run).
- Improves maintainability by centralizing repetitive logic.

```
inline square(x) {
  x = x * x;
}

init {
  int num = 5;
  square(num);
  printf("Square: %d\n", num);
}
// OUTPUT:
```

```
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
        Square: 25
 1 process created
yogesh@yogesh:~/Desktop/test$
```

typedef

The typedef construct is used to define complex data structures, similar to structs in C.

Uses:

- Helps in defining structured data types.
- Improves code organization and readability.
- Allows grouping of multiple related variables.

Features:

- Supports multiple fields of different types.
- Provides better abstraction for complex data handling.
- Can be used to create arrays of structured data.

```
typedef Node {
  int value;
  int next;
}

Node node_mem[2];
int head = 0;

init {
  atomic {
    node_mem[0].value = 10;
    node_mem[0].next = 1;

    node_mem[1].value = 20;
```

```
    node_mem[1].next = -1;

    int temp = head;
    printf("Linked List: ");
    do
    :: (temp != -1) ->
       printf("%d -> ", node_mem[temp].value);
       temp = node_mem[temp].next;
    :: (temp == -1) -> break;
    od;
    printf("NULL\n");
  }
}
// OUTPUT:
```

```
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
       Linked List:       10 ->       20 ->
 1 process created
yogesh@yogesh:~/Desktop/test$
```

4 Loops

C	Promela

```c
#include <stdio.h>
void example() {
  unsigned char x = 0;

  while (1) {
    if (x < 5){
      if (x % 2 == 0) {
        printf("Even: %d\n", x);
      } else {
        printf("Odd: %d\n", x);
      }
      x++;
    } else {
      break;
    }
  }
}
int main() {
  example();
  return 0;
}
```

```
proctype example() {
  byte x = 0;

  do
  :: (x < 5) ->
    if
    :: (x % 2 == 0) -> printf("Even: %d\n", x);
    :: (x % 2 == 1) -> printf("Odd: %d\n", x);
    fi;
    x++;
  :: else -> break;
  od;
}

init {
  run example();
}
//OUTPUT:
```

// OUTPUT:

```
yogesh@yogesh:~/Desktop/test$ cd "
Even: 0
Odd: 1
Even: 2
Odd: 3
Even: 4
yogesh@yogesh:~/Desktop/test$
```

```
yogesh@yogesh:~/Desktop/test$ spin
            Even: 0
            Odd: 1
            Even: 2
            Odd: 3
            Even: 4
  2 processes created
yogesh@yogesh:~/Desktop/test$
```

```
for (int i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

```
byte i = 0;

do
:: (i < 5) ->
  printf("%d\n", i);
  i++;
:: else -> break;  /* Stops when i >= 5 */
od;
```

5 Random Choice in Promela

```
byte x;
proctype randomChoice() {
  do
  :: x = 1; printf("Randomly chose: %d\n", x);
  :: x = 2; printf("Randomly chose: %d\n", x);
  :: x = 3; printf("Randomly chose: %d\n", x);
  ::break;
  od;
}
init {
  run randomChoice();
}
```

//OUTPUT:

```
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
            Randomly chose: 3
            Randomly chose: 1
            Randomly chose: 1
            Randomly chose: 2
  2 processes created
```

How It Works:

- The do...od loop contains multiple choices (::).
- One of the choices is **randomly** selected during execution.
- The loop breaks immediately after one selection to simulate a **one-time random choice**.
- If we remove break;, the process can keep making random choices indefinitely.

Alternative Using if...fi

- if...fi behaves like switch-case in C but picks **one** of the choices **non-deterministically**.

```
byte x;
proctype randomChoice() {
  if
  :: x = 1; printf("Randomly chose: %d\n", x);
  :: x = 2; printf("Randomly chose: %d\n", x);
  :: x = 3; printf("Randomly chose: %d\n", x);
  fi;
}

init {
  run randomChoice();
}
```

//OUTPUT:

```
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
            Randomly chose: 1
  2 processes created
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
            Randomly chose: 3
  2 processes created
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml
            Randomly chose: 1
  2 processes created
yogesh@yogesh:~/Desktop/test$
```

6 Channels in Promela

Channels in Promela are used for process communication. They allow message passing between processes, either synchronously (no buffer) or asynchronously (buffered).

Declaring Channels

Syntax:

chan channel_name = [buffer_size] of { type1, type2, ... };

- buffer_size: Defines the capacity (0 for synchronous, >0 for asynchronous).
- { type1, type2, ... }: Specifies the data types sent through the channel.

Example Declarations

```
chan ch1 = [0] of { byte };  // Synchronous channel (no buffer)
chan ch2 = [3] of { int };   // Asynchronous channel (buffer size = 3)
chan ch3 = [2] of { int, bool }; // Channel storing (int, bool) pairs
```

Sending & Receiving Messages

Syntax

- Send Message: channel ! value1, value2, ...
- Receive Message: channel ? var1, var2, ...

Example

```
han ch = [2] of { byte };
proctype Sender() {
  ch!5;  // Sending value 5
}
proctype Receiver() {
  byte x;
  ch?x;  // Receiving into x
  printf("Received: %d\n", x);
}
init {
  run Sender();
  run Receiver();
}
```

- The Sender process sends 5 to ch.
- The Receiver process receives 5 and prints it.

Buffer Size & Sequential Execution

- Buffer size [0] (Synchronous) → Forces sequential execution
    - The sender must wait until the receiver is ready.
    - Acts like a function call where the sender blocks until the receiver picks up the message.
- Buffer size [N] (Asynchronous) → Allows independent execution
    - The sender can continue execution even if the receiver is not ready.
    - The receiver picks messages later, if needed.

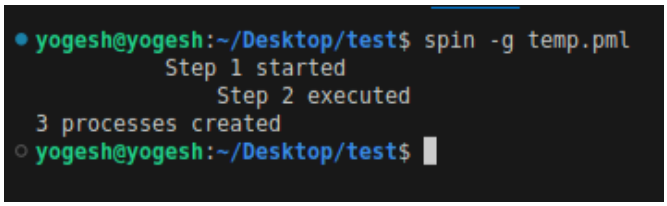
Achieving Sequential Execution

To enforce sequential execution using channels, use a synchronous channel ([0] buffer).

```
chan sync_ch = [0] of { byte };
proctype Step1() {
  printf("Step 1 started\n");
  sync_ch!1;  // Waits until received
}
proctype Step2() {
  byte msg;
  sync_ch?msg;  // Receives and allows next step
  printf("Step 2 executed\n");
```

```
}
init {
  run Step1();
  run Step2();
}
```

//OUTPUT:



How It Works

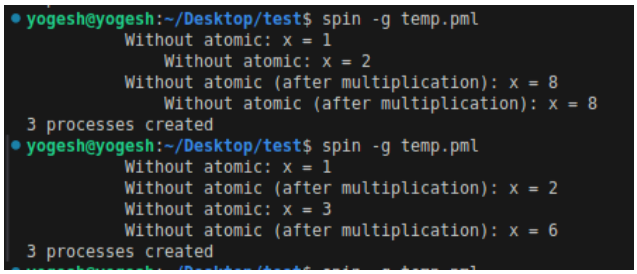
1. Step1 cannot proceed until Step2 picks up the message.
2. This ensures strict sequential execution (Step1 → Step2).

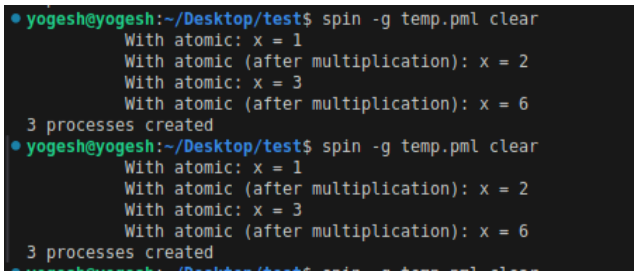
7 Atomic in Promela

The atomic block ensures that all enclosed statements execute **without interruption** by other processes.

```
byte x = 0;
proctype Example() {
  x = x + 1;
  printf("Without atomic: x = %d\n", x);
  x = x * 2;
  printf("Without atomic (after multiplication): x = %d\n", x);
}
init {
  run Example();
  run Example();
}
```

```
byte x = 0;
proctype Example() {
  atomic {
    x = x + 1;
    printf("With atomic: x = %d\n", x);
    x = x * 2;
    printf("With atomic (after multiplication): x = %d\n", x);
  }
}
init {
  run Example();
  run Example();
}
```





Key Difference:

- Without atomic: Different interleavings may cause inconsistent values.
- With atomic: Execution is sequential, preventing interference.

Handling Return Type in Promela

Unlike C, **Promela does not support explicit function return types** because it is designed for modeling concurrent systems rather than traditional programming. However, we often need to **simulate return values** in processes or inline functions.

How to Handle Return Values in Promela?

1. Using Global Variables (Shared State)

```
byte result;
inline add(a, b) {
  result = a + b;
}
init {
  add(3, 4);
  printf("Sum: %d\n", result);
}
//OUTPUT:
```

```
yogesh@yogesh:~/Desktop/test$ spin -g temp.pml clear
        Sum: 7
  1 process created
yogesh@yogesh:~/Desktop/test$
```

Pros: Simple to use.
Cons: Not thread-safe in concurrent execution.

2. Using Channels (Safe for Parallel Execution)

```
byte a = 3, b = 4, result; // Variable passing is not allowed
proctype add() {
  result = a + b;
}
init {
  run add();
  printf("Sum: %d\n", result);
}
```

Pros: Avoids race conditions, better for concurrency.
Cons: Requires process synchronization.

Best Practice:

- Use **global variables** for simple cases.
- Use **channels** when handling multiple processes concurrently.

C to Promela

```
proctype proc1() {
  printf("Process 1 running\n");
}

proctype proc2() {
  printf("Process 2 running\n");
}

init {
  run proc1();
  run proc2();
}
```

```
#include <stdio.h>

void proc1() {
  printf("Process 1 running\n");
}

void proc2() {
  printf("Process 2 running\n");
}

int main() {
  proc1();
  proc2();
  return 0;
}
```

If else

```
int x = 10;

proctype check() {
  if
  :: (x > 5) -> printf("x is greater than 5\n")
  :: (x <= 5) -> printf("x is 5 or less\n")
  fi;
}
```

```
#include <stdio.h>

int x = 10;

void check() {
  if (x > 5)
     printf("x is greater than 5\n");
  else
```

```
init {
  run check();
}
```

```
    printf("x is 5 or less\n");
}

int main() {
  check();
  return 0;
}
```

Loop

```
int i = 0;

proctype loop_example() {
  do
  :: (i < 5) -> printf("i: %d\n", i); i = i + 1;
  :: (i >= 5) -> break;
  od;
}
init {
  run loop_example();
}
```

```
#include <stdio.h>

void loop_example() {
  int i;
  for (i = 0; i < 5; i++) {
    printf("i: %d\n", i);
  }
}

int main() {
  loop_example();
  return 0;
}
```

```
int balance = 5000;

proctype deposit(int amount) {

  balance = balance + amount;
  printf("Updated Balance: %d \n", balance);
}

init {
  run deposit(1000);
  run deposit(1000);
}
```

```
#include <stdio.h>

int balance = 5000;

void deposit(int amount) {
  balance = balance + amount;
  printf("Updated Balance : %d\n", balance);
}

int main() {
  deposit(1000);
  deposit(1000);
```

	return 0; }

Array

int arr[5];
proctype initialize() {
  arr[0] = 10;
  arr[1] = 20;
  arr[2] = 30;
  arr[3] = 40;
  arr[4] = 50;
}
 	
proctype print_array() {
  int i = 0;
  do
  :: (i < 5) -> printf("arr[%d] = %d\n", i, arr[i]); i = i + 1;
  :: (i >= 5) -> break;
  od;
}
 	
init {
  run initialize();
  run print_array();
}
	#include <stdio.h>
	
	int arr[5];
	
	void initialize() {
	  arr[0] = 10;
	  arr[1] = 20;
	  arr[2] = 30;
	  arr[3] = 40;
	  arr[4] = 50;
	}
	
	void print_array() {
	  int i;
	  for (i = 0; i < 5; i++) {
	  printf("arr[%d] = %d\n", i, arr[i]);
	  }
	}
	
	int main() {
	  initialize();
	  print_array();
	  return 0;
	}

Switch case

int option = 2;
proctype switch_case() {
  if
  :: option == 1 -> printf("Case 1: Option is 1\n");
	#include <stdio.h>
	
	void switch_case(int option) {
	  switch (option) {

```
    :: option == 2 -> printf("Case 2: Option is 2\n");
    :: option == 3 -> printf("Case 3: Option is 3\n");
    :: else -> printf("Default Case: Option is not 1, 2, or 3\n");
  fi;
}

init {
  run switch_case();
}
```

```
    case 1:
      printf("Case 1: Option is 1\n");
      break;
    case 2:
      printf("Case 2: Option is 2\n");
      break;
    case 3:
      printf("Case 3: Option is 3\n");
      break;
    default:
      printf("Default Case: Option is not 1, 2, or 3\n");
      break;
  }
}
int main() {
  int option = 2;
  switch_case(option);
  return 0;
}
```

Break

```
int i = 0;

proctype break_example() {
  do
    :: (i >= 5) -> printf("Breaking at i = %d\n", i); break;
    :: printf("i = %d\n", i); i++;
  od;
}

init {
  run break_example();
}
```

```
#include <iostream>
using namespace std;
```

```
void break_example() {
  for (int i = 0; i < 10; i++) {
    if (i >= 5) {
      cout << "Breaking at i = " << i << endl;
      break;
    }
    cout << "i = " << i << endl;
  }
}

int main() {
  break_example();
  return 0;
}
```

Continue

```
int i = 0;

proctype continue_example() {
  do
    :: (i == 2) -> i++; printf("Skipping i = 2\n");
    :: (i >= 5) -> break;
    :: printf("i = %d\n", i); i++;
  od;
}

init {
  run continue_example();
}
```

```
#include <iostream>
using namespace std;
```

```
void continue_example() {
  for (int i = 0; i < 10; i++) {
      if (i == 2) {
          cout << "Skipping i = 2" << endl;
          continue;
      }
      cout << "i = " << i << endl;
  }
}

int main() {
  continue_example();
  return 0;
}
```

```
proctype func1() {

      printf("Step 1: Executing function 1, statement 1\n");
      printf("Step 2: Executing function 1, statement 2\n");
      printf("Step 3: Executing function 1, statement 3\n");
      printf("Step 4: Executing function 1, statement 4\n");
      printf("Step 5: Executing function 1, statement 5\n");
}

proctype func2() {
      printf("Step 6: Executing function 2, statement 1\n");
      printf("Step 7: Executing function 2, statement 2\n");
      printf("Step 8: Executing function 2, statement 3\n");
      printf("Step 9: Executing function 2, statement 4\n");
      printf("Step 10: Executing function 2, statement 5\n");
}

proctype func3() {
  printf("Step 11: Executing function 3, statement 1\n");
  printf("Step 12: Executing function 3, statement 2\n");
  printf("Step 13: Executing function 3, statement 3\n");
  printf("Step 14: Executing function 3, statement 4\n");
  printf("Step 15: Executing function 3, statement 5\n");
```

```
}

proctype func4() {
  printf("Step 16: Executing function 4, statement 1\n");
  printf("Step 17: Executing function 4, statement 2\n");
  printf("Step 18: Executing function 4, statement 3\n");
  printf("Step 19: Executing function 4, statement 4\n");
  printf("Step 20: Executing function 4, statement 5\n");
}

proctype func5() {
  printf("Step 21: Executing function 5, statement 1\n");
  printf("Step 22: Executing function 5, statement 2\n");
  printf("Step 23: Executing function 5, statement 3\n");
  printf("Step 24: Executing function 5, statement 4\n");
  printf("Step 25: Executing function 5, statement 5\n");
}



init {
  run func1();
  run func2();
  run func3();
  run func4();
  run func5();
}
```

```
#include <stdio.h>

void func1() {
  printf("Step 1: Executing function 1, statement 1\n");
  printf("Step 2: Executing function 1, statement 2\n");
  printf("Step 3: Executing function 1, statement 3\n");
  printf("Step 4: Executing function 1, statement 4\n");
  printf("Step 5: Executing function 1, statement 5\n");
}

void func2() {
```

```
    printf("Step 6: Executing function 2, statement 1\n");
    printf("Step 7: Executing function 2, statement 2\n");
    printf("Step 8: Executing function 2, statement 3\n");
    printf("Step 9: Executing function 2, statement 4\n");
    printf("Step 10: Executing function 2, statement 5\n");
}

void func3() {
    printf("Step 11: Executing function 3, statement 1\n");
    printf("Step 12: Executing function 3, statement 2\n");
    printf("Step 13: Executing function 3, statement 3\n");
    printf("Step 14: Executing function 3, statement 4\n");
    printf("Step 15: Executing function 3, statement 5\n");
}

void func4() {
    printf("Step 16: Executing function 4, statement 1\n");
    printf("Step 17: Executing function 4, statement 2\n");
    printf("Step 18: Executing function 4, statement 3\n");
    printf("Step 19: Executing function 4, statement 4\n");
    printf("Step 20: Executing function 4, statement 5\n");
}

void func5() {
    printf("Step 21: Executing function 5, statement 1\n");
    printf("Step 22: Executing function 5, statement 2\n");
    printf("Step 23: Executing function 5, statement 3\n");
    printf("Step 24: Executing function 5, statement 4\n");
    printf("Step 25: Executing function 5, statement 5\n");
}

int main() {
    func1();
    func2();
    func3();
    func4();
    func5();
    return 0;
```

```
}
```

References:

1. https://www.diva-portal.org/smash/get/diva2:235718/FULLTEXT01.pdf