Model Checking C Programs by Translating C to Promela

Ke Jiang
Linköping University
23 PUBLICATIONS **161** CITATIONS

IT 09 041

Examensarbete 45 hp
September 2009

Model Checking C Programs by Translating C to Promela

Ke Jiang

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Model Checking C Programs by Translating C to Promela

Ke Jiang

Nowadays, the cost of program errors is increasing from day to day, so software reliability becomes a critical problem to the whole world. C is one of the most popular programming languages, and has been widely used for developing all types of software. Hence, the security of software written in C accounts for an important proportion of software reliability.

Spin model checker is the world's most popular tool for detecting software defects in concurrent system designs. However, it could not check C programs directly. This thesis will describe a mediate method of model checking C codes to find potential problems in concurrent programs and parallel systems using Spin. We hope it could initiate more study in this new method for model checking C and other languages.

The translator we developed for this thesis uses syntax-directed translation techniques for doing the translations, and several tools and languages are involved. OCaml is the language we used for programming. CIL is the carrier and does the C syntax analysis for our translator. Spin model checker is the tool for checking the translations. C is the input language, and Promela is the target output language.

Handledare: Bengt Jonsson
Ämnesgranskare: Parosh Abdulla
Examinator: Anders Jansson
IT 09 041
Tryckt av: Reprocentralen ITC

Acknowledgment

My greatest thanks goes to my honorific supervisor, Professor Bengt Jonsson, who led me into the interesting areas of Formal Methods. I sincerely appreciate his infinite patience and foresighted suggestions that guided me through the difficulties. Additionally, I would like to thank Dr. Gerard Holzmann, the developer of Spin Model Checker, who answered my questions and gave me encouragement by emails all along my thesis. I would also like to thank my thesis reviewer, Professor Parosh Abdulla, who approved this thesis project.

I wish to express my appreciations to my lovely girlfriend Li Lian who made my life resplendent, and to my friends in Uppsala. They know who they are.

In the end, I would like to thank my parents for their unselfish love and generous support of my study and living. I would like to dedicate this thesis to them.

Contents

1 Introduction

With the great success of the Digital Revolution, our productivity has grown tremendously in the last two decades, and our lives are becoming more and more convenient at the same time. Now we can access to knowledge or information that was difficult or even impossible to reach previously within just a few seconds. And we can do shopping, working, conferencing, and thousands of other things even without walking out of our bedrooms. All these aspects of our daily lives rely on computers and Internet which have been playing essential roles in numerous aspects of our society for decades, and will definitely be more indispensable to us in the future. In this thesis, we will present a new method of incresing the reliability for some of the computer systems.

As a introduction part, I will present the background, motivation, related works and the structures of the whole thesis.

1.1 Background

In the achievement of computer industry, no one can deny the contribution of software, which acts as the "soul" of computers. It is software that makes the ice-cold machines available to ordinary people instead of becoming computer scientists' costly toys. But since all types of software are written by programmers and compiled by some other human-written programs (the compilers), there would usually be bugs or potential problems existed in them, no matter how proficient and earnest the programmers are. Working programmers may devote more than half of their time on testing and debugging in order to increase reliability [7]. According to a report from The US National Institute of Standards and Technology, programming errors have cost the US economy $60B in 2002 [18]. So improving testing methods raised to be a hot research topic in the recent years. Verification, one of the highly treasured methods, was developed under the purpose of proving or disproving the correctness of the program or system with respect to the formal specification or property. And model checking [7], which is the general topic that will be discussed in this thesis, is an instance of verification.

1.2 Goal

I will present an indirect way of model checking [7] C programs by translating the C code to Promela, which is the input language for Spin [1, 11] and represent the model of corresponding C code. Then we can use Spin to model check the models with our expected purposes. The most significant problems that we want to check are potential data races and overflows of concurrent programs. We decided to handle a subset of C and use several representative algorithms for approaching general concurrent C programs, e.g. the two queue algorithms [14] that Maged Michael and Michael Scott published 13 years ago.

1.3 Motivation

C is one of the most popular programming languages on the planet, and has been widely used for developing all types of systems, from embedded programs to application software. The more C has been used, the more we need to find proper automatic methods for verifying the reliability of the C programs, and find potential errors as early as possible.

Spin model checker is one of the world's most popular model checkers, and arguably one of the world's most powerful tool for detecting software defects in concurrent system designs. The tool has been applied to everything from the verification of complex call processing software that is used in telephone exchanges, to the validation of intricate control software for interplanetary spacecraft [10, 11]. However, it can not check C programs directly, so we have to find some mediate methods for utilizing this powerful tool. In this thesis, we generate Promela codes [1, 11] which represent the models of the C programs by translating the C language into. Then we can analyze the generated models using Spin. Because of the high flexibility and reusability of C, it was impossible to have the whole C language verified in my thesis project, so I bound the target programs into a subset of C containing basic data-types, expressions, statements and functions. Therefore, the C code that will be used later on stands for the subset that I will describe in detail in Section 3.1, and hope the method of model checking C programs presented in this thesis could be the starting point for more extensions and consummations.

The idea of this thesis project came from a paper of Bengt Jonsson, State-Space Exploration for Concurrent Algorithms under Weak Memory Orderings [13], in which he model checked C programs by manually abstracting Promela models from input C codes. Now we would like to present an upgraded method of automatic model extraction from C. I will compare these two methods in chapter 7.

1.4 Related works

1.4.1 Modex

Modex [12] is a tool developed by Gerard Holzmann ten years ago. It is written in ANSI-C, and aims to mechanically extract high-level verification models from ANSI-C codes. It target applications include telephone switching software, distributed operating systems code, protocol implementations, concurrency control methods, and client-server applications [10].

It generates Promela translations from C codes as my project does, and translate ordinary ANSI-standard C programs in a sensible fashion. But it still has obvious limits, namely only makes limited use of Promela, e.g. does not handle pointers and have no procedure calls, that bind the project from precisely simulating some particulars of the original C codes. Since Promela is not as powerful language as C, lots of flexible data-types and ingenious functions can not be straightly translated. So Modex handle these “untranslatable” codes by embedding them directly with the Promela-offered C code embedment facility. The embedded code fragments cannot be checked by SPIN, neither in the parsing phase nor in the verification phase. They are trusted blindly and copied through from the text of the model into the code of the verifier that SPIN generates. In particular, if a piece of embedded C code contains an illegal operation, like a divide by zero operation or a nil-pointer dereference, the result can be a crash of the verifier while it performs the model checking [11]. One of the most significant limits of Modex is the translation for pointers. When handling pointers, it does insert assertions, but only to make sure that the dereferenced pointers is not NULL. Therefore, it will not try to investigate the detail and find out potential problems of any pointers. I will explain my way of handling pointers in Section 4.6.

1.4.2 JPF(Java PathFinder)

When JPF was first implemented in 1999, it had actually been developed as a translator from a subset of Java to Promela, and used Spin to the model check the translation. From the early papers related to JPF, we could find out that it tried to translate Java codes by approaching same functionalities in Promela. Now, it has been developed to be a Java Virtual Machine (JVM), and been used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions [8]. JPF is capable of checking every Java program that does not depend on unsupported native methods.

The translation for Java class is a typical case of how JPF manipulates the translation. A Java class basically consists of data variables and methods. When creating a new object, a new set of data variables are allocated, and the methods of this new object will work on the newly created set of variables. So JPF models the data variables of the class by an array of records, and an index variable points to the next free position of the array. Methods are mapped into macro definitions, and method calls are mapped to applications of macros [8]. This translation of Java data variables and classes is similar to the translation of C pointers and structures in this thesis.

1.4.3 ARINCTester

ARINCTest [6] is a Java-written tool environment for translation and verification of C applications running on top of APEX (APplication EXecutive for space and time partitioning) which is an interface providing a commeon logical environment that enabels the execution of multiple independently developed applications to execute together on the same hardware [17]. It also uses Spin as model checker, and has a graphical interface for Spin. The C to Promela translation function is integrated into the project. When translating, it tries to directly embed the “hard-to-translating” C codes as Modex does which is the main point we would like to improve.

1.4.4 Other related works

Checkfence

The checkfence project automatically translates the C implementation code and the test program into a SAT formula, hands the latter to a standard SAT solver, and constructs counterexample traces if there exist incorrect executions [2, 4]. It provides a sound example of CIL-based model checking implementation for my project.

Bandera

It takes Java code as input and generates a program model in the input language of one of several existing verification tools, like Promela of Spin [5].

1.5 Structure of the thesis

Chapter 2 of this thesis talks about the goals and targets of the whole project. In chapter 3, I will describe the used principles for our C to Promela translator. Chapter 4 presents the syntax-directed translation principle for the translator. Chapter 5 is about the structure of the translator. And the last several chapters list some examples, do evaluations and make conclusions.

2 Goals

2.1 A sample Problem to detect

The purpose of the whole project is to detect potential defects in concurrent C programs, so we give an example in Fig. 1 to illustrate a sample problem.

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

struct queue {
    struct node *head;
    struct node *tail;
};

void initialize(struct queue *Q) {
    struct node *dummy = malloc(sizeof(struct node));
    dummy->next = 0;
    dummy->value = 0;
    Q->head = Q->tail = dummy;
}

void enqueue(struct queue *Q, int data){
    struct node *new = malloc(sizeof(struct node));
    new->next = 0;
    new->value = data;
    Q->tail->next=new;
    Q->tail=new;
}

void dequeue(struct queue *Q){
    struct node *temp;
    temp = Q->head;
    Q->head = temp->next;
    free(temp);
}
```

Figure 1: An enqueue and dequene example

Assume the enqueue and dequeue functions above are running in two independent processes sharing the same memory, and are inserting and removing elements concurrently. Obviously, there would be problems when dequeue function is applied on a queue with only one element (Fig. 2) because of the lack of empty queue checking mechanism. If the only element of the queue has been removed, the queue could not be inserted any new values using enqueue function, since the Q->tail is pointing to an null address, and Q->tail->next would not exist. So there should be an error raised when dequeue function is trying to remove the only element of a queue. This thesis presents a innovative technique to detect such kind of problems by translating and simulating the pointers and their operations in order to maximally preserve the behavior of the original C programs in the target Promela models. The translation and checking result of this example will appear in Chapter 6.

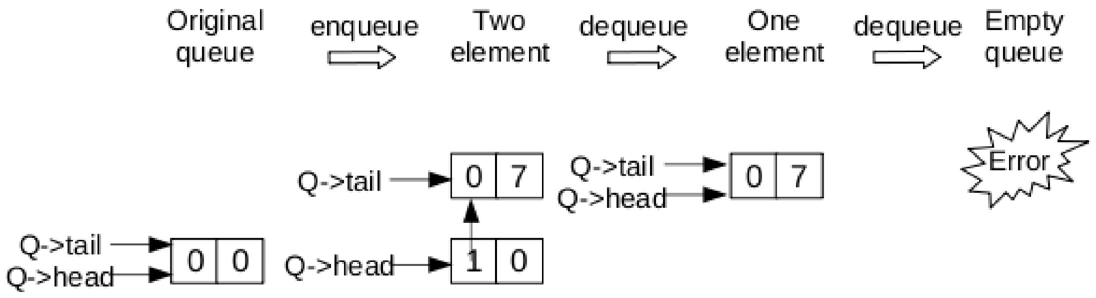

Figure 2: Sample problem

2.2 Running example

An important application in C is concurrent data types, i.e. shared data structures which can be acessed concurrently by several processes. They are often extremely tricky, and easy to cause serious problems if used improperly or guardless. A representative example is concurrent FIFO queues which are widely used both in academic areas for researching purposes, and in industrial fields for implementing parallel applications and various systems. In 1996, Maged Michael and Michael Scott published their concurrent non-blocking and blocking queue algorithms in their paper [14]. The two pseudocode-written queue algorithms are suitable examples for this thesis, so we decided to translate them into C codes, and apply as our examples. Another reason of using these algorithms is to compare the result with the manual Promela translation of the blocking queue algorithm in Bengt Jonsson's paper [13]. With respect to the original algorithm, I changed the return type of dequeue function from boolean to int because there is no boolean type in ANSI C. My translation for the two-lock concurrent queue is as follows, and the Promela abstraction result and analysis can be found in Chapter 7

```
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;
    Q->T_lock = 1;
        Q->Tail->next = node;
        Q->Tail = node;
    Q->T_lock = 0;
}

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

The C translation of the non-blocking queue algorithm can be found in Section 6.3 together with the automatic Promela abstraction result.

2.3 Expected result

As the result of the project, the following objects should be meet (for the subset of ANSI C that I will point out in section 3.1):

- Reasonable syntax transformation from C to Promela which can preserve pointed structures, function definitions, function calls, etc.
- Sensible approaches to the “untranslatable” aspects of C, such as pointers and structures in previous projects.
- Clean and clear Promela code output with nice formatting.
- The translated Promela codes should be able to be simulated and verified by Spin.

3 Principle of the project

3.1 The subset of ANSI C

To explain the supported subset of C in a more efficient and friendly way, I would like to summarize the covered syntax first as follows.

- A program can work with several other programs using #include. But ANSI standard library is not supported (except the malloc and free functions that have been handled for simulating memory allocations and releases in Promela) currently because their inner functionalities can directly be interpreted by the compiler, while the detail definitions are "invisable" to us. Therefore, it will take long extra time to manually simulate all those functions in Promela, which is not possible within the limits of this thesis project.
- The C programs are not allowed to have input operations, like scanf, since there is no routine in Promela to read input (Promela models must be closed to be verifiable), and the input functions in ANSI standard libray is not supported either.
- Supported data structures are primitive integer, array, structure (allowed to form linked lists) and pointer to integer and structure. And the types of functions are bound to the types above besides void type. Pointer arithmetic is forbidden.
- Expressions can be composed by ordinary operators, e.g. ++, %, >, and +=.
- Assignments, function calls, jumps (goto, break and return), selections and iterations can be used as statements
- Function definitions, declarations, calls and returns are allowed.

These are the brief descriptions for the syntax of the subset. The detailed syntax is listed as follows.

3.1.1 Keywords:

The supported keywords are listed in Table 1.

3.1.2 Data structures:

Primitive data-types

Refer to Table 1. My thesis is focused on abstracting Promela verification models from input C programs, so I must take care of the variables, but not have to investigate much about the data types of primitive data variables, e.g. int or char, because their types will not affect the whole model of the program and there is neither float nor char type in Promela at all. As the reasons above, I decided to bind the data-type of the subset to integers. And all the other int types, e.g. long, signed, will only be considered as the int type.

int	Supported
double, signed, unsigned, long, short	Supported, but will be translated into int types.
void	Supported in void function and argument list statements.
struct	Supported as long as it contains only allowed data-types.
const, static	Only integral values
switch-case-default, if-else	The value of the "choice" expressions should be integers or pointers to integers or structure.
for, do, while	The value of the "guard" expressions should be integers or pointers to integers or structure.
return	The return value should be integers or pointers to integers or structures.
goto	Supported
break	Supported
labeled statements	Supported

Table 1: Supported C keywords

Arrays

- Arrays in Promela are one-dimensional (Although we can work around with this limitation by defining arrays of user-defined structures which have arrays inside). So I decided to limit the arrays in this subset to one dimension. But multi-dimensional arrays might be supported in future works.
- Dynamic array declarations are not supported, e.g.:

```
int *a;
a = malloc(5 * sizeof(int));
a[3] = 10;
```

Structures and Linked lists

Both normal structures (non-pointers) and recursive structures (that point to themselves or other structures) will be taken care of as long as their fields do not contain any unsupported data types. Linked lists are supported as well.

Pointers

Supported types of pointers are int and struct. When defining a pointer with these two types, my program is supposed to translate the definition by simulating the memory in the way I will present in Chapter 4. Functions are allowed to return pointers and have pointer parameters of these two types. "malloc" and "free" operations for these pointers are also supported. Pointer arithmetic and other pointer types, e.g. void pointers, pointers to pointers or pointers to functions, are not allowed currently, but might be supported in future extensions, that would mainly lie in supporting more types of pointers which are actually the “heart” of C. Global variables should not be pointed to (but could be pointers), and the reson will be described in Section 4.6

As an example, the following C code could be translated correctly.

```
struct person{
    int age;
};

int main(){
    struct person p, *ptr, *ptr2;
    ptr = &p;
    ptr2 = malloc(sizeof(struct person));
    p.age = 24;
    ptr->age ++;
    ptr2->age = 18;
    free(ptr);
    return 0;
}
```

3.1.3 Expressions and Operators

Arithmetic	+, -, *, /, %, ++, –
Assignment	=, +=, -=, *=, /=, %=
Logical/Relational	==, !=, >, <, =, <=, >=, ||, &&, !
Others	1. constant values 2. variables 3. sizeof(), &, * (for int or structs) 4. conditional operations: ?, :

Table 2: Supported C expressions and operations

The expressions could be variables and constants or compositions of them using the listed operators in Table 2.

3.1.4 Statements

Supported statements are illustrated in Table 3. The detailed syntax of these statement can be found in the Appendix 1.

Assignment	
Function call	
Jump	goto
	break
	return
Selection	if, else
	switch, case, default
Iteration	while
	for
	do..while

Table 3: Supported statements

3.1.5 Block

A block is a group of statements that is usually used as the body of a function, a branch in a selection statement, or an iteration in a loop structure. A block is supported as long as all its statements are supported.

3.1.6 Functions

Declaration
Function declarations will be ignored.

Definition
Function definitions are supported, and the syntax of the function definition could be found in the Appendix 1.

Return
Return types could be int, struct, void, and pointer to int and struct. The following code could be correctly translated.

```
int* test(){
    int *a = malloc(sizeof(int));
    return a;
}

int main(){
    int *p;
    p = test();
    return 0;
}
```

Argument passing

Arrays are not allowed as arguments in this subset of C, since the process types in Promela cannot have arrays as arguments, and the hidden arrays in structures are not allowed either.

- Passing values

  You can pass the value of a variable to a function. If the function modifies the value, the original variable remains unaltered. The corresponding value passing in the translated Promela codes will not affect the original variables as well. Allowed variable types are integers, structures (non-array).

- Passing pointers

  If we want the function to alter the data, we can pass pointers. The Promela translation will keep this property since the pointers will be simulated. Allowed types are pointers to int and struct.

Function recursion

Function recursion is also supported, and is normally divided into two types, argument recursion and tail recursion, according to their recursion methods. Table 4 presents two typical examples of different recursion types.

`int fib(int n){`	`int gcd(int x, int y) {`
`    if(n<=2) return 1;`	`    if (y == 0) return x;`
`    else`	`    else`
`        return fib(n-1) + fib(n-2);`	`        return gcd(y, x % y);`
`}`	`}`
Argument recursion	Tail recursion

Table 4: Two recursion types

Both of these recursive functions can be correctly handle. But if the recursion depth is n, there will be $\Omega(2^{n-1})$ function calls in the argument recursive functions which will lead to $\Omega(2^{n-1})$ process instantiations in Promela. These instantiations can easily exceed the maximal process number (255) that can be created for one Promela model. So the recursion depth in argument recursive functions should be particularly bewared. Nevertheless the tail recursive functions do not have this problem so severe since they do not require extra space for execution, and only n function calls will be generated for depth n.

3.1.7 Comments

All the comments of the C codes will be ignored as they have nothing to do with the model of the programs.

3.1.8 Macros

Macros in C will be directly replaced by their real values by the translator we developed, as the standard C preprocessors will do when compiling C programs.

3.1.9 Constants

Constants are used in assignments and expressions in C, and their types will not affect the model extraction of the programs which is the main concern of this thesis project. And Promela has neither char nor float type. So I bound the constants to integral values which will simplify the amount of work somewhat.

3.2 Promela syntax

The syntax of Promela is C-like. This section describes the brief Promela syntax that is involved in the translation, and the unused part is not presented here, i.e. some of the optional terms are omitted. (Optional terms are enclosed in "[ ]". The Kleene star * is used to indicate zero or more repetitions of an optional term.)

3.2.1 Keywords

active	assert	atomic	bit	break
chan	do	d_step	else	fi
goto	if	init	int	od
printf	proctype	run	skip	typedef

Table 5: Used Promela keywords

3.2.2 Data structures

Primitive data types

```
typename varname [ initializer ]
```

Possible typenames here are bit, int and user-defined data-types. The initializers, if specified, must be constants, otherwise the variables without initializers will be set to 0 by default.

Structures

```
typedef varname {
    typename varname
    [; typename varname ] *
}
```

typedef is used to declare a new data type with a user-defined structure, e.g.

```
typedef employee{
    int number;
    int salary
}
```

Arrays

```
typename varname '[' const ']'
```

Promela only supports one-dimensional arrays. As in C, the first object of an array always has index zero. The number of objects in the array must be specified with a numeric constant in declaration, i.e. the length cannot be specified with an expression. If an initializer is presented in an array definition whose data-type is not a user-defined structure, the initializer will be evaluated once and all the elements will be initialized to the same resulting value. In the absence of an explicit initializer, all array elements are initialized to zero. However, the arrays of user-defined data-types can not be initialized in this way, e.g.

```
typedef person {
    int age
};
person P[5] = -1;
```

The elements in array P[5] can not be initialized to -1, and the field age will keep the system initialization value 0.

Channels

```
chan varname = '[' const ']' of { typename }
```

A channel in Promela is a data type with two operations, send and receive. Every channel will be associated with a message type; once a channel has been initialized, it can only send and receive messages of its own message type. At most 255 channels can be created [1].

There are two types of channels with different semantics: rendezvous channels of capacity zero, e.g. chan ret_main = [0] of { int }, and buffered channels of capacity greater than zero, e.g. chan comm = [9] of { int }. In this project, the channels will only be used for synchronizing and passing values between processes that simulate conresponding functions, so only the rendezvous channels will be introduced.

3.2.3 Expressions

The operators in Table 6 are used to build expressions within the scope of this project, but expressions could also be just a variable or a value.

+	-	*	/	%
>	>=	<	<=	++
!=	!	&	||	&&
++	−	run		

Table 6: Used Promela operators

The expressions has C-like syntax, e.g. $(a > b) \mid\mid (c > d)$. The keyword “run” is used to instantiate processes.

3.2.4 Statements

Table 7 shows the involved statement types.

assert	assignment	atomic	break
declaration	else	expression	goto
send/receive	selection	skip	repetition

Table 7: Used Promela statement types

3.2.5 Control flow:

sequence

Sequence is used to enclose an arbitrary block of code. Four allowed types in sequence are statements, variable declarations and exclusive send and receive operations on channels. The steps in the sequence are separated by either a semi-colon (;) or, equivalently, an arrow (->). The arrow is sometimes used to indicate a causal relation between successive statements and also in selection and repetition statements to separate the guard from its succeeding statements.

else

```
else
```

This special guard can be used (only once) in every selection or repetition structure and is enabled precisely when all the other guards are blocked.

selection

```
if
    :: sequence
    [ :: sequence ]*
 fi
```

A selection starts with an if keyword, and ends up with a fi. A sequence can be selected only if its “guard”, namely the first statement, is approachable. This selection structure in Promela will be used for translating the if and switch statements in C. An example of Promela selection is as follows.

```
if
:: (a>b) ->
   c = 0
:: else ->
   c = 1
fi
```

repetition

```
do
    :: sequence
    [ :: sequence ]*
 od
```

Repetition has similar syntax as selection except the starting and ending keywords. By default, the end of each option sequence leads back to the first control state to achieve repeated execution. A transition to the control state that follows the repetition construct can be defined with a break.

break

```
break
```

The break statements are used to jump to the end of the innermost repetition structure as the “break” in C.

goto and labels

- goto labelname

  goto is normally used to determine the target control state for the immediately preceding statement, and should be used together with labels.

- labelname : statement

  This is used to identify a unique control state in a proctype declaration.

atomic

```
atomic { sequence }
```

Introduces a sequence of statements that is to be executed indivisibly.

d_step

```
d_step { sequence }
```

The execution of the sequence inside d_step is indivisible as in atomic. But the statements of the sequence in d_step should be deterministic. If non-determinism is nonetheless present, it is resolved in a fixed and deterministic way: i.e. nondeterminism is resolved deterministically in favor of the first alternative [1]. Because of the determinism, the d_step structure is more efficient that atomic, and generates much less states in verification procedure.

3.2.6 Processes

Processes serve like the “function units” of a Promela model, and cannot have arrays or structures with arrays inside as formal parameters. Maximally, 255 processes could be created for one Promela model [11]. There are three types of processes in Promela, namely basic, active and initial types, but only the basic and initial processes will be used in this thesis.

(a) Basic processes

```
proctype procname ( typename varname
        [; typename varname ] * )
 { sequence }
```

Such processes should be instantiated by the run statement.

(b) Initial process

```
init { sequence }
```

This process, if present (at most one in a Promela model), is instantiated once, and is often used to prepare the true initial state of a system by initializing variables and running the appropriate process-instances.

4 Syntax directed translation

The CIL software, which severs as the carrier of our translator, simplifies the input C codes and handles many of the ugly corners of C that are not in the scope of this thesis for us. So before my program performs syntax tree transformations, it will utilize CIL to process the input C codes first to get the CIL (a highly-structured, “clean” subset of C) abstract syntax trees, and then transform these “purified sources” to Promela syntax trees. Therefore, I will divide the abstract syntax of C in a CIL-defined manner.

There will be two syntax, namely the CIL and the transformed Promela syntax, presented in each subsection, and one example added for every necessary transformation. The translations for most supported keywords will be presented in this chapter, so I will not describe them one by one redundantly. (Optional terms are enclosed in “[ ]”. The Kleene star * is used to indicate zero or more repetitions of an optional term.)

4.1 Data structures

CIL keeps the name, type, and other information of a variable inside an element varinfo throughout the syntax tree. Two fields of varinfo which we need to investigate and can be used to differentiate variables are vname and vtype that keeps the name and type respectively. However, Promela has no specific syntax for recording the type of a variable as CIL. The type is determined at the time of a variable definition. And when using a variable, we can directly invoke its name (and the offset if needed). The transformation for variable expressions will be described in Section 4.2. Thus, I will focus on depicting the declaration aspect for data structures in Promela.

CIL separates the one-line variable definition and initialization into two single steps, i.e. a definition and an assignment. So I will skip the initializer potentiality of variable definition in Promela since the assignment statments will be described specifically in Section 4.3.

4.1.1 Primitive data types

CIL: varinfo.vname: string
Promela: one_decl(INT, name)

Only integral primitive varibales will be considered. The name of the CIL varible vname will be reserved in Promela name, and the type are defined as INT. An example is shown in Table 8.

int a;	int a;
C	Promela

Table 8: Translation example for primitive data types

4.1.2 Arrays

CIL: varinfo.vname: string
    varinfo.vtype: TArray(typ, exp, attributes)
Promela: one_decl(typename, name, const)

The typ and exp in varinfo.vtype indicate the base type and the length of the array, and vname stores the name of the array. They will be transformed to the typename, const and name terms respectively. An example is shown in Table 9.

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

Table 9: Translation example for array declaration

4.1.3 Structures and linked lists

- structure type definition

  CIL: compinfo.cname: compname
      compinfo.cfields: fieldinfo list
  Promela: TYPEDEF(name, decl_lst)

  A structure in CIL is defined as a global and kept in compinfo. The cname field records the name, and cfields stores all the fields of the structure which are transformed into a list of variable declarations decl_lst under the user-defined typename name in Promela. An example is shown in Table 10.

struct node{
    int value;
};
	typedef node{
	    int value;
	}

C	Promela

Table 10: Translation example for structure definition

- variable definition of structure type

  CIL: varinfo.vname: string
      varinfo.vtype: TComp(compinfo, attributes)
  Promela: one_decl(TNAME(name), name)

  The first term TNAME(name) is the user-defined structure type, and the second name indicates the name of the variable. Structures could also be used for defining arrays, that has already been implied in the array transformation above. An example is shown in Table 11.

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

C	Promela

Table 11: Translation example for structure variable definitions

The structures are allowed for making linked lists in this C subset, and the transformation of the linked lists will be presented Section 4.6.

4.2 Expressions and operations

4.2.1 constants

CIL: Const(constant)
Promela: any_expr(CONST(const))

Only the integral constants will be kept in the transformed Promela syntax, and the real value in CIL constant is reserved in Promela const.

4.2.2 variables and pointers

CIL: Lval(lval)
Promela: any_expr(VARREF(varref))

The information, such as the name and offset (an array index offset or a field offset), of a variable that is used as an expression in CIL will be retained in the "varref". Pointers could also be used as expressions, that will be described in Section 4.6.

4.2.3 unary operators

CIL: UnOp(unop, exp, typ)
Promela: any_expr(OPE(unarop, any_expr))

The exp in this expression which holds the information without the unary operator will be transformed to the second any_expr in Promela. The unop operator will be fully kept without modifying in the unarop, e.g. !a will be directly translated into !a.

4.2.4 binary operators

CIL: BinOp(binop, exp, exp, typ)
Promela: any_expr(EOPE(any_expr, binarop, any_expr))

The two exps in the CIL syntax will be transformed into the second and the third any_expr, and supported binary operations binop will be kept in the binarop element.

An example is shown in Table 12.

(a % b) != c;	(a % b) != c;
C	Promela

Table 12: Translation example for binary operators

4.2.5 conditional expressions

CIL: If(exp, block, block, location)
Promela: IF((step, step list), sequence list)

The conditional expressions in C will be transformed into if structures by CIL as shown in the syntax above. Although Promela has C-like conditional expressions, the syntax tree branches that my program gets for conditional expressions are actually the trees for if structure. So the conditional operations in C will not be reserved in exactly the same fashion. The transformation for if structure is described in Section 4.3.2. An example is shown in Table 13.

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

C	Promela

Table 13: Translation example for conditional expressions

4.3 Statements

4.3.1 Instruction

- assignment

  CIL: Set(lval, exp, location)

  Promela: ASS(varref, any_expr)

  or: INCRE(varref)

  or: DECRE(varref)

  The syntax of assignments in CIL has no other variants. Variable increments and decrements are also stored in the same CIL syntax above. But our translator will estimate whether it is a variable increment or decrement when coming to an assignment. If it is, transform it in the later two types.

The location represents the position of the assignment in a source file, e.g. line number and file name, and is not useful for our model abstractions. So I will skip all the posterior location elements. An example is shown in Table 14.

C	Promela
a = b;	a = b;
a++;	a++;
a–;	a–;
a = a + 1;	a++;

Table 14: Translation example for assignments

- function calls

  Function calls will be described in Section 4.5

4.3.2 Control structures

- Jump

  1. goto

     CIL: goto(stmt ref, location)

     Promela: GOTO(name)

     In Promela, the goto statement do not care about the actual statement to jump to, but only the "label" instead. So the label will be fetched out from the stmt ref item in a CIL goto statement, and reserved as the target for Promela goto statement.

  2. labeled statements

     CIL: stmt.labels: label list

     Promela: stmnt(LABELED(name, stmnt))

     A label is kept as an attribute of a statement in CIL. While in Promela, labeled statement is defined as a special statement kind. So before performing the transformation, we detect whether the statement is labeled or not first. If it is, transform it to a LABELED statement. Otherwise, treat it as normal. The case and default labels will not be considered when transforming a statement as they will be taken care of while transforming the switch structure seperately. An example for goto and lebeled statements is shown in Table 15.

enter:
    a++;
    goto enter;
	enter:
	    a++;
	    goto enter;

C	Promela

Table 15: Translation example for goto and labeled statement

3. break

   CIL: Break(location)

   Promela: BREAK

   Both of the break statements in C and Promela are used to jump to the end of the innermost repetition structure. An example is shown in Table 16.

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

C	Promela

Table 16: Translation example for break

4. return

   See section 4.5.

- Selection

  1. if

     CIL: If(exp, block, block, location)

     Promela: IF ((step, step list), sequence list)

     The exp, which is the "guard" for CIL-if, will be merged together with the first block into the (step, step list) iterm. The step is transformed from exp, and the step list is obtained from the first block. The second block will be processed into the only element in the sequence list with an ELSE statement attached in front denoting the selectable condition. If the second block in CIL syntax contains an empty statement list, namely no else branch for this if structure, the sequence list in Promela will be an empty list. An example is shown in Table 17.

  2. switch

     CIL: Switch(exp, block, stmt list, location)

     Promela: IF ((step, step list), sequence list)

C	Promela

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

Table 17: Translation example for if structure

All the statements of a switch structure is stored in a statement list, and the "cases" are kept as labels of the statements. Meanwhile, several statements might share the same "case", in other words, they compose a branch of the case as a whole, but only the first statement is labeled. An example of a typical switch structure and its translation is presented in Table 18.

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

Table 18: Translation example for switch structure

The five statements inside this switch structure are stored in a statement list, and the case 0, case 1 and default entries are kept as the lable for statement a++, a- - and break. So the statements b++ and b- - do not have their own labels, but share with the statements a++ and a- -. So we have to gather the statements sharing the same label together into a independent list first, and transform this new list to a sequence in Promela's if structure. The statement reorganization function for switch structure can be found in the Appendix 2.

- Iteration

  CIL: Loop(block, location, [stmt], [stmt])

  Promela: DO((step, (STMNT(BREAK))), sequence list)

  or: DO((STMNT(SEXPR(CONST(TRUE))), step list), [])

The guard of the loop is implicated in the block directly, or the termination is accomplished by a break statement inisde the block if no exact guard represented. So we have to locate the guard and the real loop block before transforming. The code for locating the guard and the block can be found in the Appendix 3.

If a real guard is found, the loop will be transformed in the upper mannar. The guard is indicated in the step, and STMNT(BREAK)) implies that the termination is accomplished when the guard is violated. Otherwise, the loop will likely be terminated by a break statement inside the loop block, and transformed in the second way. Therefore, the head of the sequence is the "true" guard, and the rest is transformed from the CIL loop block where the termination is possiblely implicated. An example illustating these two loop termination is shown in Table 19.

C	Promela

```
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;
	```

Table 19: Translation example for for loops

4.4 Block

The block can be used in four different structures, namely as ordinary block, else block, loop block and switch block, in the supported C subset. And these four different blocks will be transformed to four Promela structures respectively. The ordinary block will be transformed into a sequence, and the else block will be transformed into a sequence as well but with an ELSE guard attached in front. The transformation for the loop block and switch block have already been described in the transformation for interation and switch. And the inner statements manipulation is described in Sec. 4.3.

4.5 Function

The translation for functions, which are the task units of a program, is critical to the correctness of the whole project. We manage to handle the ordinary C functions as well as the recursive ones in this translator. However, there exists big gaps between the syntax of C and Promela, i.e. a function in C is an unit of the whole program which is written to achieve specific tasks, but there is no function concept in Promela. So how to maintain the behaviors while achieving "checkable" translations of function definitions was a problem in the beginning of the implementation procedure.

The aim of this project is to model check concurrent C programs with the assistance of Spin. So we decided to use Promela processes for undertaking the C functions. Each C function definition will be translated to a process type declaration in Promela. Local variables will be kept in the corresponding process definitions, and will be valid within the scope of its instantiations. The parameters in function calls are translated into arguments in the process instantiation statement. The details for translating functions are as follows.

4.5.1 Function definition

The function definitions are kept in GFun(fundec, location) as globals in CIL, and will be transformed to PROCTYPE(proctype) structures which preserve the functionalities.

The function calls in C can be considered as statements, and the secceeding statements will not be excecuted until the called function finished their tasks. Meanwhile, proctype instances in Promela are to invoke new processes by the run-operator, and after instantiated, the process will run independently from the main process, and the following statements will be executed without waiting the child-process to finish. To overcome this gap, we add a channel to every proctype (except two special function "e" and "d" which I will explain in Chapter 7) for synchronizing. In other words, every proctype will be given a new channel in its parameter list, and use this channel for synchronizing with the main process. In the end of the sequence, a labeled printf statement will be added for being the exit symbol of the return statements.

The translation for function calls and return is describe in the following two subsections.

4.5.2 function call

A function call will be interpreted into several steps. First, define a channel for the proctype instantiation in the front of the main proctype. If a proctype was instantiated several times within the same proctype, only one channel would be defined to prevent channel redefinitions. Second, instantiate the proctype by passing the predefined channel as an argument. After that, wait for termination of the child process on that channel.

malloc and free calls of int or struct types are supported, and will be translated into lists of statements that are used to simulate the memory allocation and release operations. I will talk more about these two functions in Section 4.6.4.

4.5.3 Return

```
CIL: Return([exp,] location)
Promela: STMNT(SEND(EXCLA(varref, send_args)));
         STMNT(GOTO("end"))
```

A return statement in a C function is used to return a value to its caller and terminate the function. So we have to simulate both of these two properties of return. After a channel communication which imitates the value transmission, a goto end statement will be appended to terminate the process (adding the end label has been mentioned previously). By doing this in the translation for return, the process will jump to the end of the proctype, and terminate spontaneously, that is the same as the return in C.

If a void is returned, a dummy value 0 will be sent through the channel which means that the channel is only used for synchronization. If a real value is returned, it will be sent through the channel to its instantiator.

4.5.4 Parameter passing

```
CIL: fundec.sformals
Promela: (one_decl, one_decl list)
```

The parameter list in the function definition will be transformed to the one_decl list with a channel declaration attached in front. The channel is used for undertaking the synchronization channel, and does not specify the message type. The message type of the channel will be inherited from the channel that is used in the main process for instantiating this proctype. The pointer parameters will be translated to positions of memory simulation arrays (see Section 4.6).

4.5.5 Recursive functions

The recursive mechanism of Promela processes varies a lot from that of C functions. In C, the recursion is achieved by calling the function itself directly or indirectly, but in Promela, the recursion is carried out by instantiating the proctype itself inside its body. However, the instantiated child process will execute asynchronously with the main process, in other words, the newly created process will start executing concurrently with its creator. So in order to conduct the recursion execution stepwise, we force the creator waiting for the termination of its child process through channel synchronizations. The channel defined in each level are independent from each other, so there will not be collisions for channel definitions in the recursion procedure. Table 20 shows how a tail recursive function is translated, and Fig. 3 illustrates the execution flow of the Promela gcd translation.

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")
	}

C	Promela

Table 20: Translation example for tail recursive functions

4.5.6 An example

The code in Table 21 gives a demonstration for the function translation.

4.6 Pointers and memory operations

Pointer is the heart of C programming language, but yet there is no pointer-like datatypes in Promela. So we have to find appropriate approaches for simulating pointer definitions, assignments, references and other operations. All the previous works of automatic C to Promela translation mentioned in Section 1.3 did not manage to translate the pointer related codes at all, but directly embedded them instead which can only make sure the pointers would not refer to NULL. In this project, we implemented a series of methods for simulating the pointers and operations, which were entirely

```
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

Table 21: Translation example for functions

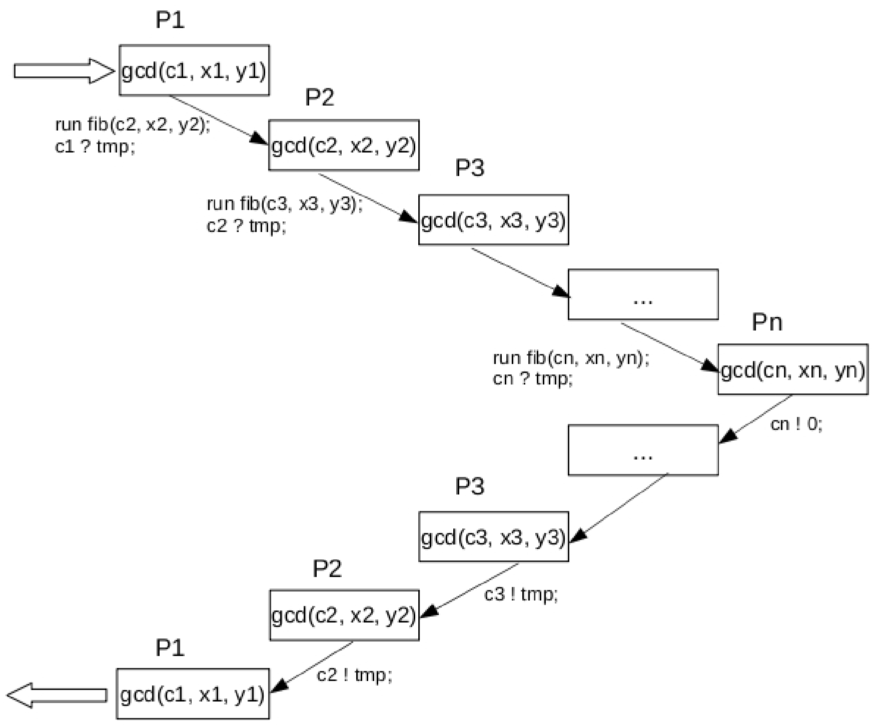

Figure 3: The recursive instantiation of proctype fib

innovative comparing to the previous works. Hence, this part became the core portion of this thesis without doubt. The methods were inspired by Bengt Jonsson's paper [13], in which he manually translated some pointers and operations to Promela.

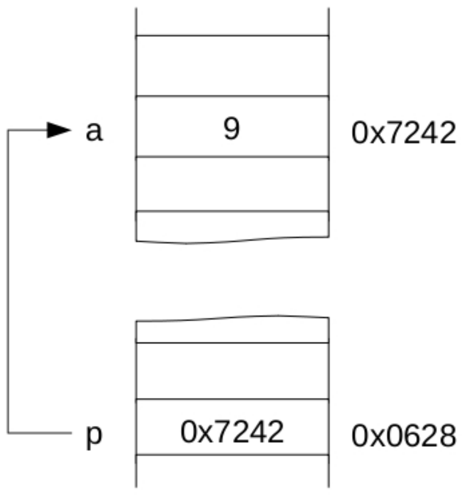

Figure 4: A pointer in C

A pointer in C contains an address that points to data (Fig. 4). So finding address approaches is the starting point for this whole section.

From Fig. 4 which demonstrates the operational principle of the pointer and memory, we can consider the memory as a large sequential array where all types of variables are stored. The difference between a pointer and a normal variable is the information they hold. The preceding one stores the “position” of the varible that it points to. And the second one holds a real data. Based on this abstraction, we decide to simulate the memory by defining arrays for specific data-types, and the pointers by transforming them to positions in the corresponding arrays. Different data-types can not share the same array, since they might have different structures and fields. Therefore, every data-type that need to be memory simulated will induce an independent array definition, and the pointers to this type will be transformed to the positions of the array.

The global variables should not be pointed to, since C will initialize the global variables right after their definitions. But if we want to simulate the pointers and pointed variables in Promela, we have to initialize them by running several assignment statements (for initialization purpose) before using. This procedure cannot be done to global variables in Promela, although can be worked around by initializing them in the starting process init of Promela. So we decide not to take them into account currently, but this could be overtaken in future extensions.

4.6.1 Pointer definition

```
C: varinfo.vname: string
   varinfo.vtype: TPrt(typ, attributes)
Promela: (None, INT, (name, None, None), [ ])
```

All the pointer definitions will be transformed to integer declarations, only the names of the pointers are reserved.

4.6.2 Simulated types

Our preprocessor will tranverse the whole input CIL syntax tree first, and find out all the data-types that need to be memory simulated. Then there will be array definitions added for all these types, e.g. if a structure node is found to be pointed, there will be an unique array defined after the declaration of the structure. Two types are allowed in the lastest translator, namely int and user-defined struct.

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

C	Promela

Table 22: Translation example for simulated pointers

To simulate the memory, we define two new arrays that have different functions for the data-type, and the name of the arrays are composed by the original type name and two different suffixs, namely _mem and _valid. As we can see from Table 22, the pointer next for creating linked lists is translated to an int variable, and will be used to point to a position in the array node_mem[9] (9 is determined as the length for defining memory simulation arrays in our translator). The extra array node_valid[9] imitates the mechanism of how C manipulates address availabilities by recording whether corresponding positions in memory array node_mem[9] are free. If node_mem[p] is available, the value of node_valid[p] will be 0. Otherwise, node_valid[p] will be 1 meaning the position has been occupied.

The linked lists in C is achieved using user-defined “recursive” structures, e.g. the node structure in Table 22 can be used for creating linked lists by pointing the inner pointer next to a subsequent element. The joints of the lists are the pointers, like the node above, to the structure itself directly or indirectly. There is no pointer types in Promela at all, so we have to find approaching methods to succeed the linked list facility from C. As I just mentioned, all the pointed data-types will cause array definitions. So the structures for making lists will surely lead to new array definitions, and the pointers inside the structures will become int values of specific positions. This machinery could be used to create static lists. Fig. 5 illustates how a linked list is formalized in Promela using the node presented in Table 22.

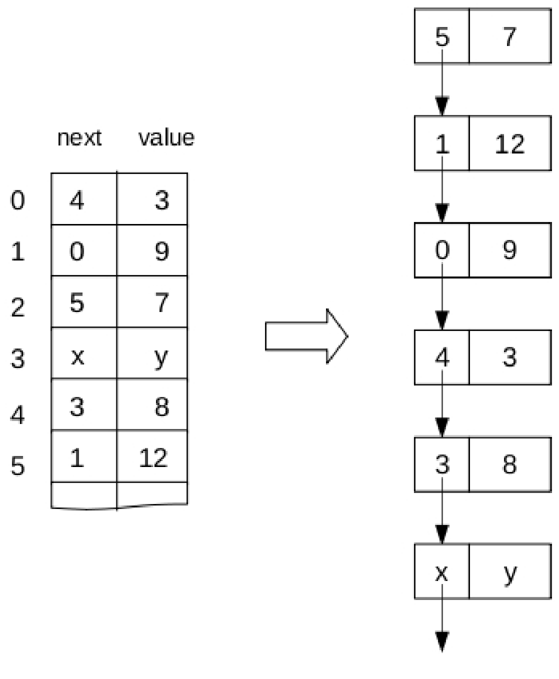

Figure 5: The linked list approach in Promela (the memory array)

The left table in Fig. 5 shows the actual storage of the data structure, and the right portion gives a more dramatic image which could be considered as a linked list.

4.6.3 Pointer assignment and reference

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")
	}

C	Promela

Table 23: Translation example for malloc and free

All the pointers in C become “position recorders” in Promela, and the pointed variables become elements in arrays, so the assignments between pointers will be translated to normal integer assignments. Table 23 is an example showing how pointer assignments and references are manipulated, i.e. the pointer tail, next and tmp become integers. The element that tail points to is node_mem[tail], so the pointer assignment tail->next = tmp; is translated into node_mem[tail].next = tmp;. Pointers could also be used as parameters, which will be translated to int variable declarations as the parameter tmp of function test in Table 23.

4.6.4 Pointer initialization and release

The preprocessor is composed by several functions that record all the “malloced” and “freed” types (possibly int and struct), and define the translation for mallocing and freeing involved types. When requesting a new address using malloc, the malloc-like statements will find out the first available position in array type_valid[9], and mark it as occupied. When releasing an address by free, the free-like statements will set the position in type_valid[9] and the element in type_mem[9] back to 0. This malloc and free translations is to simulate the mechanism in C. Fig. 6 shows how a new position is obtained, and Table 24 demonstates the translation for malloc and free calls.

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

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

C

```promela
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")
}
```

Promela

Table 24: Translation example for malloc and free

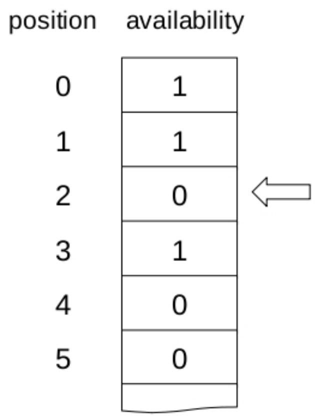

Figure 6: Position allocation (availability array)

5 Implementation

5.1 Involved tools and languages

5.1.1 CIL (C Intermediate Language)

CIL [15], which oughts to be considered as the "carrier" of this thesis, can compile valid C programs into a few core constructs with a very clean semantics to make the result program easier to analyze. For example, all looping constructs, `while`, `for` and `do..while`, in the input C programs are reduced to a single form `while`; all function bodies are given explicit return statements; and a group of involved files are merged into one. CIL also can be guided to perform user defined transformation like the translator module we developed. The C to Promela translator is actually a new module which can be used by CIL, and will use the CIL-analyzed syntax trees as the input to perform translations on.

5.1.2 OCaml

OCaml [9, 16] is a powerful functional programming language and the most popular variant of the Caml language. It extends the core Caml language with a fully edged object-oriented layer, as well as a strong module system. It is the language in which CIL is developed and the language we used for developing the translator.

5.1.3 Spin

Spin [1, 11] is a general tool for verifying the correctness of distributed software (software design) in a rigorous and mostly automated fashion, and it takes Promela codes as input modules for checking. We use Spin to check the translated Promela models so that the input C programs are model checked mediately.

5.1.4 C and Promela

C is the input language that we will translate, and Promela is the targeting language that is expected to represent the model of the input C codes.

5.2 The structure of the module

Our translator is divided into five portions that collaborate as a whole, and can also work individually.

5.2.1 c2promela.ml

This is the entry program for the whole C to Promela translator module. It contains a feature declaration (the exclusive symbol of the module) that can be detected, compiled and used (as a command-line argument) by CIL.

```
open Cil

module P = Prec2p
module R = Printpmlabs
module T = Transc2pabs

let initc2p (f:file) =
    let _ = P.preprocessc2p f in
    let pmltree = T.transc2p f in
    R.printpml pmltree

let feature : featureDescr = {
    fd_name = "c2promela";
    fd_enabled = ref false;
    fd_description = "Generation of Promela from C code";
    fd_extraopt = [];
    fd_doit = initc2p;
    fd_post_check = true;
}
```

The first four lines can be understood as the "#include" facility in C. They declare the necessary external modules for utilizing. The initc2p function has three sequential steps that call three different portions of the module respectively. First, call the preprocessc2p function in prec2p.ml to initialize the translation. Second, transform the C syntax tree to Promela's using the transc2p function in transc2pabs.ml. Last, print out Promela code by calling the printpml function in printpmlabs.ml on the transformed Promela syntax tree.

The name of the file's entry point for CIL is feature, and is named by the field fd_name which associated the command-line argument. The fd_enabled variable shows that this feature is not enabled by default in CIL. The fd_description depicts the feature. And the fd_doit line defines the operation to work on the input C codes, and will be executed when this feature is enabled.

When the user passes the –doc2promela command-line option to CIL, the variable associated with the fd_enabled flag is set and the initc2p function is called on the file which represents the merger of all involved C files.

5.2.2 pmlabs.ml

This file defines the abstract syntax of Promela that will be used by the others.

5.2.3 prec2p.ml

Behaves as the "preprocessor" of the translator. It mainly manipulates the following four tasks. a) Records all the data-types that need to be memory simulated. b) Records all the functions calls inside each function for possible local channel definitions. c) Defines the translation of malloc and free operations for different data-types. d) Add an init proctype for being the originating point of the module.

5.2.4 transc2pabs.ml

The core part of the project which handles the syntax tree transformation from C to Promela. The translation could be effectively considered as syntax tree transformation which have been described in the Chapter 4.

5.2.5 printpmlabs.ml

This program is in charge of outputting target Promela code by traversing the whole transformed Promela syntax tree.

5.3 Translation flow

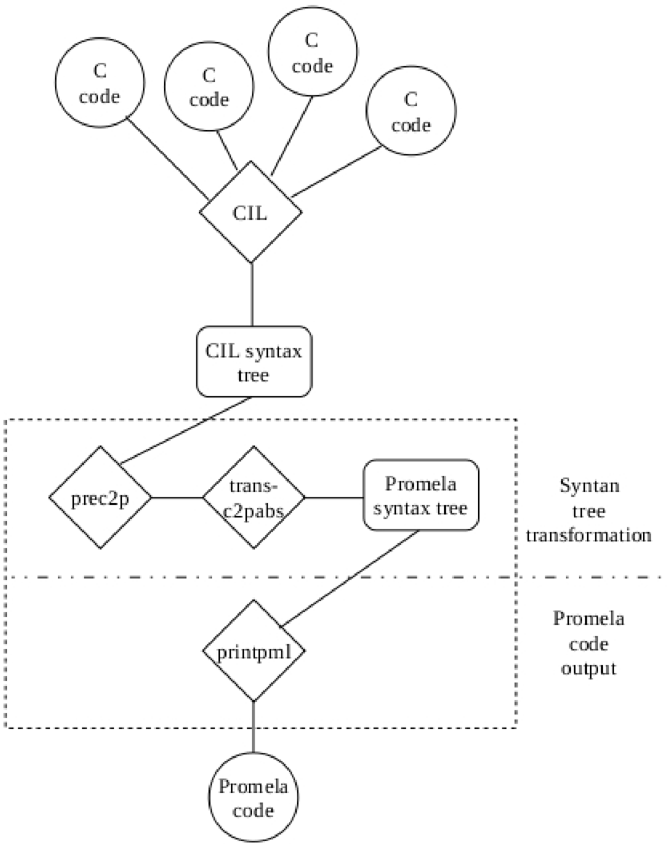

Figure 7: The whole translation flow

6 Examples

6.1 The sample problem in section 2.1

6.1.1 C code

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

struct queue {
    struct node *head;
    struct node *tail;
};

void initialize(struct queue *Q) {
    struct node *dummy = malloc(sizeof(struct node));
    dummy->next = 0;
    dummy->value = 0;
    Q->head = Q->tail = dummy;
}

void enqueue(struct queue *Q, int data){
    struct node *new = malloc(sizeof(struct node));
    new->next = 0;
    new->value = data;
    Q->tail->next=new;
    Q->tail=new;
}

void dequeue(struct queue *Q){
    struct node *temp;
    temp = Q->head;
    Q->head = temp->next;
    free(temp);
}

void e(struct queue_t *Q) {
    int val = 1;
    while(1) enqueue(Q, val++);
}

void d(struct queue_t *Q) {
    while(1) dequeue(Q);
}
```

```
int main(){
    struct queue *qu = malloc(sizeof(struct queue));
    initialize(qu);
    e(qu);
    d(qu);
}
```

6.1.2 Promela code translation

```
typedef node {
    int next;
    int value;
}
node node_mem[9];
int node_valid[9];
typedef queue {
    int head;
    int tail;
}
queue queue_mem[9];
int queue_valid[9];

proctype initialize(chan in_initialize; int Q){
    int malloc_node_c;
    int dummy;
    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
    };
    dummy = tmp;
    node_mem[dummy].next = 0;
    node_mem[dummy].value = 0;
    queue_mem[Q].tail = dummy;
    queue_mem[Q].head = queue_mem[Q].tail;
    in_initialize ! 0;
    goto end;
end :
    printf ("End of initialize")
}

proctype enqueue(chan in_enqueue; int Q; int data){
    int malloc_node_c;
    int new;
    int tmp;
    int mem_5;
    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;
    node_mem[new].next = 0;
    node_mem[new].value = data;
    mem_5 = queue_mem[Q].tail;
    node_mem[mem_5].next = new;
    queue_mem[Q].tail = new;
    in_enqueue ! 0;
    goto end;
end :
    printf ("End of enqueue")
}

proctype dequeue(chan in_dequeue; int Q){
    int temp;
    temp = queue_mem[Q].head;
    queue_mem[Q].head = node_mem[temp].next;
    d_step {
        node_valid[temp] = 0;
        node_mem[temp].next = 0;
        node_mem[temp].value = 0
    };
    in_dequeue ! 0;
    goto end;
end :
    printf ("End of dequeue")
}

proctype e(int Q){
    chan ret_enqueue = [0] of { bit };
    int val;
    int tmp;
    val = 1;
    do
    :: true ->
        tmp = val;
        val ++;
        run enqueue(ret_enqueue, Q, tmp);
        ret_enqueue ? 0
    od;
end :
    printf ("End of e")
}

proctype d(int Q){
    chan ret_dequeue = [0] of { bit };
    do
    :: true ->
        run dequeue(ret_dequeue, Q);
        ret_dequeue ? 0
    od;
end :
    printf ("End of d")
}

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

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

6.1.3 Check the problem

We can check the problem that there might be an dequeue process excecuting on an empty queue by using the LTL formula:

```
[] (p -> [] p)
```

where p is defined as

```
#define p (queue_mem[1].head != 0)
```

The formula means that, so long as the head of the queue is set to a non-zero value, namely the queue has been intialized, the head of the queue will always points to a real element. This checks the same property as that a dequeue function will not work on an empty queue. The automatically generated never statement of Spin is:

```
never { /* !([] (p -> [] p)) */
T0_init:
    if
    :: ((p)) ->
        goto T0_S4
    :: (1) ->
        goto T0_init
    fi;
T0_S4:
    if
    :: (! ((p))) ->
        goto accept_all
    :: (1) ->
        goto T0_S4
    fi;
accept_all:
    skip
}
```

When verifying the translated Promela code with this LTL formula, an error will be found which is the shortest error tracing as Fig. 2. The simulation result is as follows:

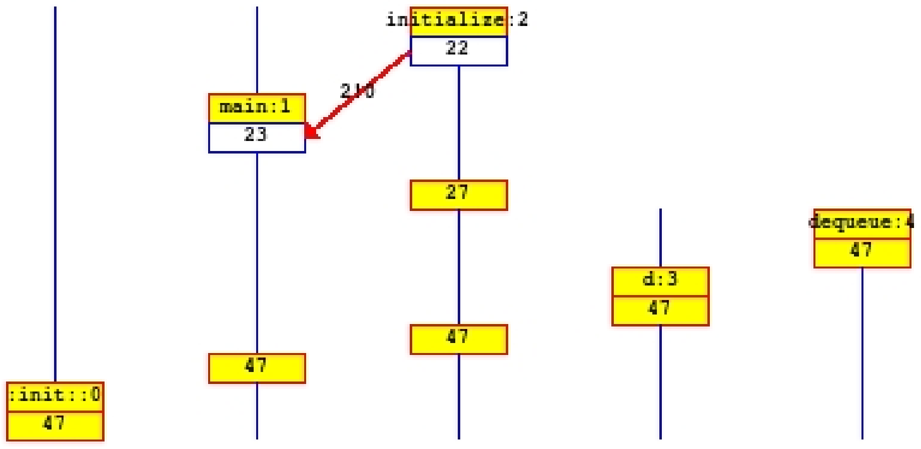

Figure 8: The error tracing of the problem in section 2.1

We can see that, after initialization of the queue, dequeue process directly worked on the one-element queue. Then an error is found.

6.2 An argument recursive function

6.2.1 C code

```
int fib(int n){
    if(n<=2) return 1;
    else
        return fib(n-1) + fib(n-2);
}

int main(){
    int result;
    result = fib(5);
    return 0;
}
```

6.2.2 Promela code translation

```
proctype fib(chan in_fib; int n){
    chan ret_fib = [0] of { int };
    int tmp;
    int tmp___0;
    if
    :: (n <= 2) ->
        in_fib ! 1;
        goto end
    :: else ->
        run fib(ret_fib, (n - 1));
        ret_fib ? tmp;
        run fib(ret_fib, (n - 2));
        ret_fib ? tmp___0;
        in_fib ! (tmp + tmp___0);
        goto end
    fi;
end :
    printf ("End of fib")
}

proctype main(chan in_main){
    chan ret_fib = [0] of { int };
    int result;
    run fib(ret_fib, 5);
    ret_fib ? result;
    in_main ! 0;
    goto end;
```

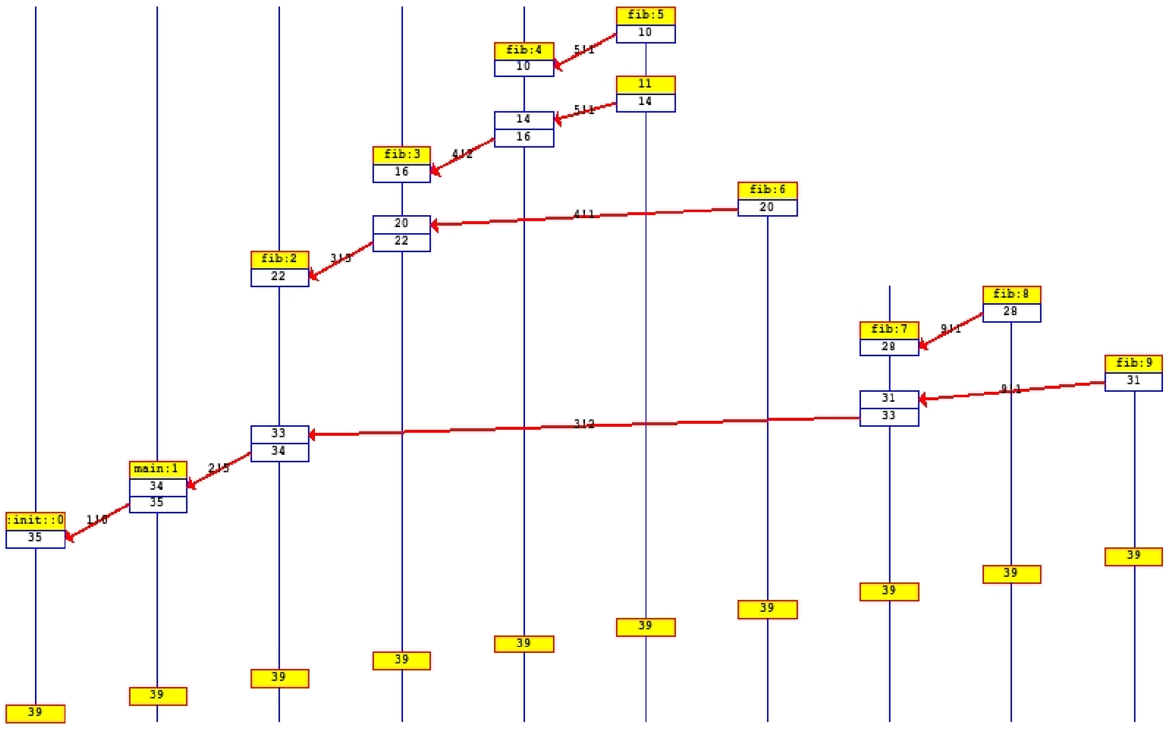

Figure 9: Execution of the argument recursive function

```
end :
    printf ("End of main")
}

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

6.2.3 Verification and simulation

The translation can be successfully verified by Spin, and the simulation graph is shown in Fig. 9. We can find that the correct result 5 is sent back by the first instantiated fib process (fib:2) to process main. The branches in the graph indicate the inner proctype instantiation procedures, and are the same as the function call processes of the original C programs.

6.3 Non-blocking queue algorithm

6.3.1 C translation

With respect, I modified the non-blocking queue algorithm in Maged Michael and Michael Scott's paper [14] a little to satisfy my requirement:

- Merge the two structure definitions, pointer_t and node_t, into one node_t, and these two ways of defining the "node" in the queue will be the same for my project
- Change the CAS operations with if-statements and assignment operations
- Change the type of dequeue function from boolean to int for simplification

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

struct queue_t {
    struct node_t *Head;
    struct node_t *Tail;
};

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

void enqueue(struct queue_t *Q, int value) {
    struct node_t *node, *tail, *next;
    node = malloc(sizeof(struct node_t));
    node->value = value;
    node->next = 0;
    while (1) {
        tail = Q->Tail;
        next = tail->next;
        if (tail == Q->Tail)
            if (next == 0) {
                if (&tail->next == next) {
                    tail->next = node;
                    break;
                }
            }
            else
                if(&Q->Tail == tail)
                    Q->Tail = next;
    }
    if(&Q->Tail == tail)
    Q->Tail = node;
}

int dequeue(struct queue_t *Q, int *pvalue) {
    struct node_t *head, *tail, *next;
    while (1) {
        head = Q->Head;
        tail = Q->Tail;
        next = head->next;
        if (head == Q->Head) {
            if (head == tail) {
                if (next == 0)
                    return 0;
                if(&Q->Tail == tail)
                    Q->Tail = next;
            }
            else {
                *pvalue = next->value;
                if (&Q->Head == head) {
                    Q->Head = next;
                    break;
                }
            }
        }
    }
    return 1;
}
```

6.3.2 Promela code translation

```
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;
```

```
}
queue_t queue_t_mem[9];
int queue_t_valid[9];

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

proctype enqueue(chan in_enqueue; int Q; int value){
  int malloc_node_t_c;
  int node;
  int tail;
  int next;
  int tmp;
  atomic {
    malloc_node_t_c = 1;
    do
    :: (malloc_node_t_c >= 9) -> break
    :: else ->
```

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

proctype dequeue(chan in_dequeue; int Q; int pvalue){
  int head;
  int tail;
  int next;
  do
  :: true ->
    head = queue_t_mem[Q].Head;
    tail = queue_t_mem[Q].Tail;
    next = node_t_mem[head].next;
    if
    :: (head == queue_t_mem[Q].Head) ->
      if
      :: (head == tail) ->
        if
        :: (next == 0) ->
          in_dequeue ! 0;
          goto end
        :: else
        fi;
        if
        :: (queue_t_mem[Q].Tail == tail) ->
          queue_t_mem[Q].Tail = next
        :: else
        fi
      :: else ->
        int_mem[pvalue] = node_t_mem[next].value;
        if
        :: (queue_t_mem[Q].Head == head) ->
          queue_t_mem[Q].Head = next;
          break
        :: else
        fi
      fi
    :: else
    fi
  od;
  in_dequeue ! 1;
  goto end;
end :
  printf ("End of dequeue")
}
```

6.3.3 Verification and simulation

If we test the translation with a simple case, namely enqueue two elements and dequeue one, the translation will be successfully verified by Spin and the execution graph is shown in Fig. 10. From the whole simulation process we can see that our translation behaves precisely like the original C programs, especially the execution procedure of pointers, function calls and returns.

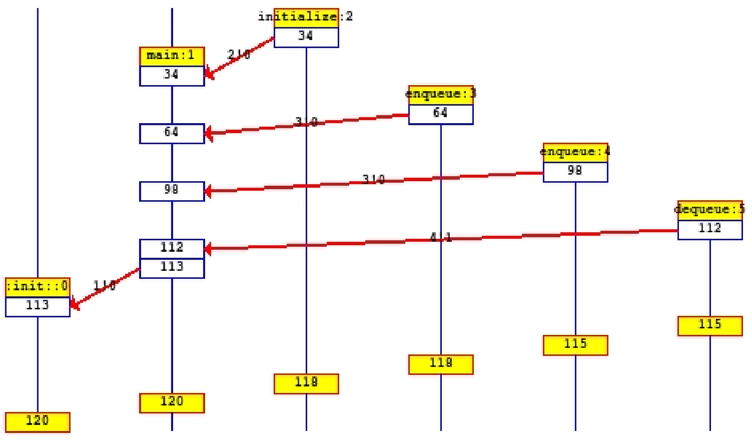

Figure 10: Simulation of non-blocking queue algorithm

6.4 Other examples

An array-implemented queue algorithm can be found in the Appendix 4, and the example in section 2.3 will be described minutely in Chapter 7.

7 Evaluation

7.1 Promela translation for the example in Sec. 2.3

```
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 malloc_node_t_c;
    int dummy;
    int tmp;
    atomic {
        malloc_node_t_c = 1;
        do
        :: (malloc_node_t_c >= 9) ->
            break
        :: else ->
            if
            :: (node_t_valid[malloc_node_t_c] == 0) ->
                node_t_valid[malloc_node_t_c] = 1;
                break
            :: else ->
                malloc_node_t_c ++
            fi
        od;
        assert (malloc_node_t_c < 9);
        tmp = malloc_node_t_c
    };
    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 malloc_node_t_c;
    int node;
    int tmp;
    int mem_5;
    atomic {
        malloc_node_t_c = 1;
        do
        :: (malloc_node_t_c >= 9) ->
            break
        :: else ->
            if
            :: (node_t_valid[malloc_node_t_c] == 0) ->
                node_t_valid[malloc_node_t_c] = 1;
                break
            :: else ->
                malloc_node_t_c ++
            fi
        od;
        assert (malloc_node_t_c < 9);
        tmp = malloc_node_t_c
    };
    node = tmp;
    node_t_mem[node].value = val;
    node_t_mem[node].next = 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;
   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")
}
```

7.2 Experiments

In order to evaluate the translator we developed, we decided to use some of the test harnesses that Bengt Jonsson introduced in his paper [13] on the translation above. Then we could evaluate the automatic translation comparatively with the manual translation in Bengt's paper. We denote test harnesses in a condensed notation (following [3]), 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, one with two enqueue operations, and one with two dequeue operations [13].

When translating function definitions, the two functions, "e" and "d", will be treated specially as they will be used for simulating two independent processes that our enqueue and dequeue processes are running in. So when translating them, there will not be channel synchronizations. 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 seperately.

7.2.1 Test harness 1

First, we performed a simple test (E | D), which can pass the verification successfully. There were 602 states generated and 2.696Mb memory used within one second in the verification procedure. The simulation graph is shown in Fig. 11.

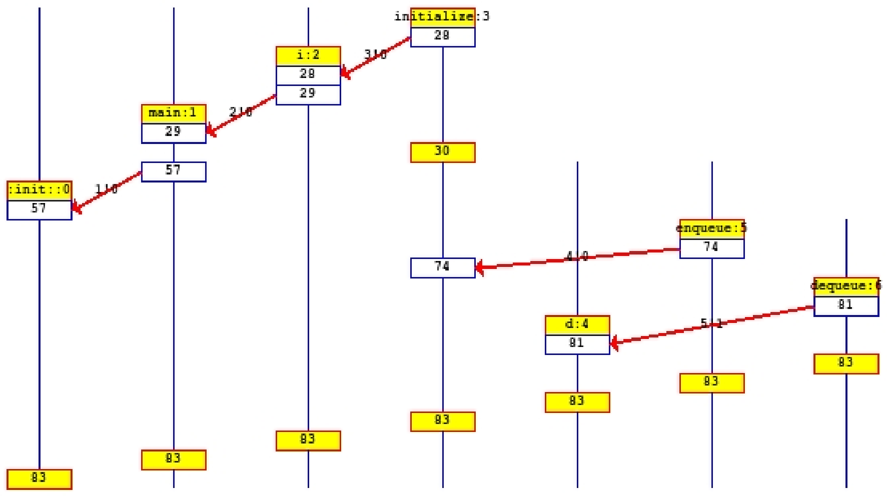

Figure 11: Test harness 1: (E | D)

The C code implementation of this test harness is in the Appendix 5, and the implementations for the posterior test harnesses are similar.

7.2.2 Test harness 2

Then we tested the translation with a more advanced one (EEEEE | DDDDD). Verifying this consumed a great amout of memory which exceeded the physical memory of the computer we worked on (1.5Gb memory), so we used the Supertrace/Bitstate mode. There were 2791002 states generated which was equivalent to 1746.080Mb memory usage during the 18 seconds' verification procedure. The values that dequeue processes got (the value of *pvalue) is shown on the right side of Table 25, and the complete simulation process is demonstrated in Fig. 12.

Enqueued values	Dequeued values
1	0
2	1
3	2
4	3
5	4

Table 25: Result of test harness 2

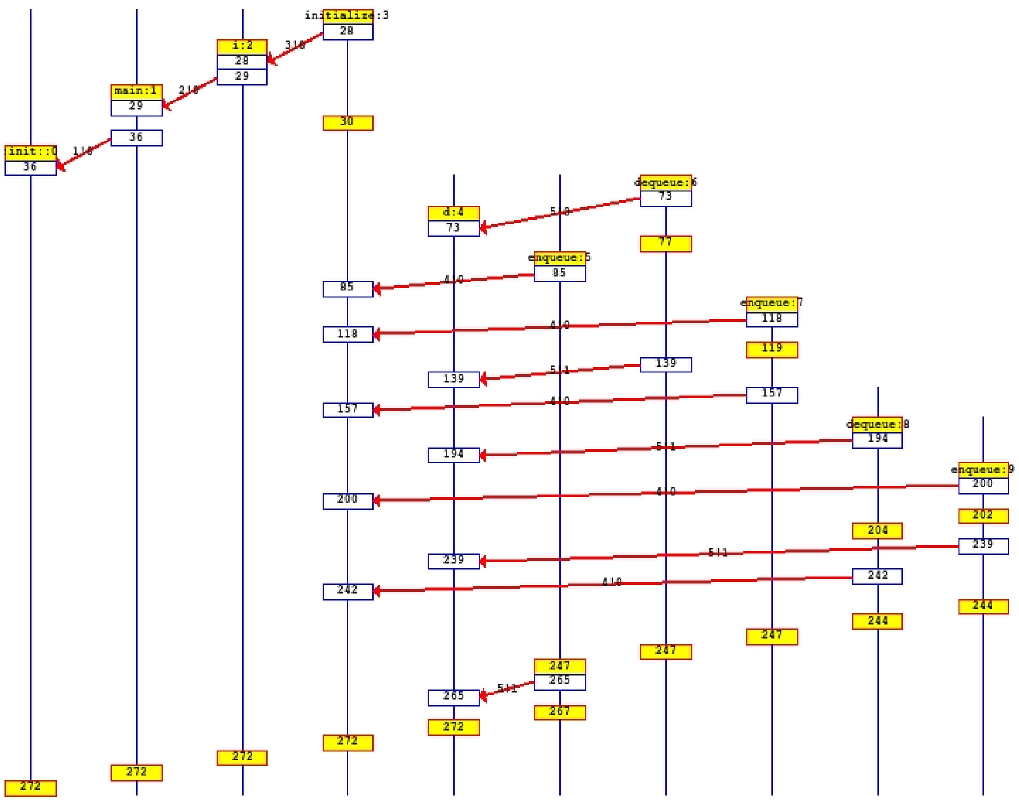
Figure 12: Test harness 2: (EEEEE | DDDDD)

We can see that the first value inserted was 1, but the first value that dequeue process got was 0. This is because that there was a dequeue process finished execution before any new elements were inserted, which can be reflected from the process dequeue:6 in Fig. 11. There were five processes instantiated for enqueue and dequeue proctypes respectively. However, Spin will reclaim the processes that have finished their execution, and make the process ids available again. So some of them shared the same process id, and that is why there might be several channel synchronizations comming out from the same process in Fig. 12.

7.2.3 Test harness 3

After that, we perfomed an even complicate one, (E | E | E | E | D | D). We used the Supertrace/Bitstate mode as in the previous test. There were 27558506 states generated which was equivalent to 15874.231Mb memory usage during the 205 seconds' verification process. The complete simulation process is shown in Fig. 13.

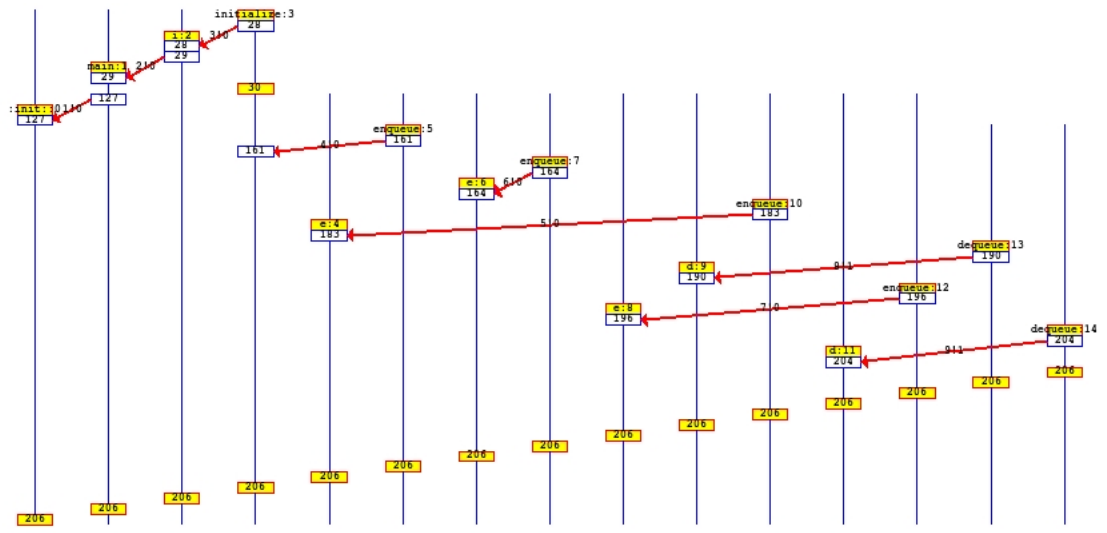

Figure 13: Test harness 3: (E | E | E | E | D | D)

7.2.4 Test harness 4

In the end, we would like to verify a complex case of multiple independent processes using the Exhaustive mode, namely perform an Exhaustive verification on a test harness under the physical memory limitation of our computer (1.5Gb). The harness we performed was (E | E | D | D). The whole verification procedure generated 2752997 states, and consumed 1471.243Mb memory in 376 seconds. The complete simulation process is shown in Fig. 14.

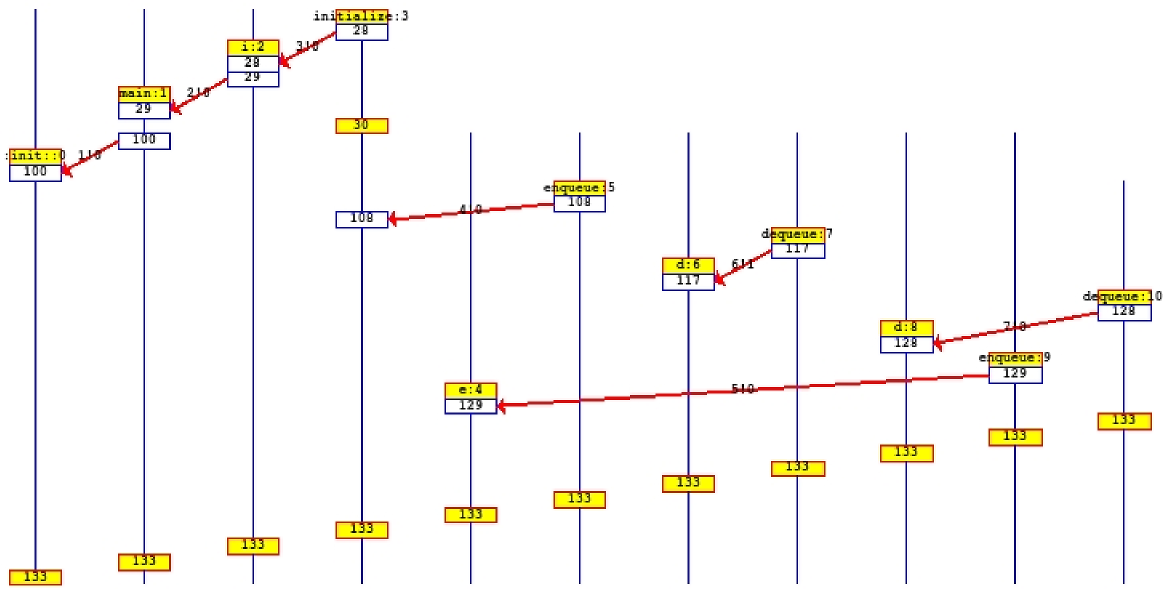

Figure 14: Test harness 4: (E | E | D | D)

7.3 Comparison

7.3.1 Bengt's translation

Comparing with Bengt's manual translation in his paper [13], this C to Promela translator is more efficient and intelligent which can do the translation automatically from given C programs. That makes the users who are not proficient in Promela can Spin model check C programs as well.

The program structures of the translations are much more similar with the original C codes, e.g. pointers, linked lists and function calls, which makes program analysis easier, especially the pointer translations. Meanwhile, since the translator works automatically, it cannot investigate specific properties as accurate as Bengt's translation, in which he organized the statements accordingly.

7.3.2 Modex

This translator employs a new translation method which is different from the Modex project. The improvements mostly consist in the translation for pointers, structures, function calls and many other aspects. This allows more precise verification and simulation of the target C programs.

8 Future works

8.1 Pointed global variables

The current version of our translator does not support pointed global variables due to initialization reasons. But this could be resolved in future extensions by enclosing batchs of statements, as initializing local pointed variables, in the init process which is the starting point of the whole model.

8.2 Multi-dimensional array supports

As I mentioned previously, Promela has limits of supporting multi-dimensional arrays, but luckily we can work around with this by defining structure arrays in which there are array fields, e.g.

```
typedef rowt {
    int col[9];
}
rowt row[9];
```

Here, the variable row[9] can be considered as a two-dimensional array. When using a specific node of the array, we can invoke row[x].col[y] where a data can be stored.

This method can be used for simulating multi-dimensional arrays, although a bit inconvenient since the inner dimensions should be created by defining arrays in a new structure and be called as a field. But multi-dimensional array supports might be able to appear in future works.

8.3 More pointer types

Pointers are the core of C. So supporting more pointer types, e.g. void pointer, pointer to pointers and pointer to functions, becomes the fundamental task for future works. Take pointer to pointers for example.

```
int val, *p, **t;
val = 1;
p = &val;
t = p;
```

To simulate t above, we could define an array for all the pointers. In other words, all the pointers will have their own “addresses”. When a pointer to pointer is assigned with a pointer, assign it with the position of that pointer in the array. And when using a pointer to pointer to reach the orginal data, we need the aid of the pointer that it points to to keep track of the real data, i.e.

```
**t = 2;
```

Assume the pointer p has a position in the pointers' memory array pointer_mem[9] which has been recorded by t, and the variable val is kept in the array int_mem[9], then the assignment above could be translated as follows.

```
int tempc = pointer_mem[t];
int_mem[tempc] = 2;
```

References

[1] Mordechai Ben-Ari. *Principles of the Spin Model Checker.* Springer, 2008.

[2] Sebastian Burckhardt. *Memory Model Sensitive Analysis of Concurrent Data Types.* PhD thesis, August 2007.

[3] Sebastian Burckhardt. Memory model sensitive analysis of concurrent data types. *PhD Thesis*, August 2007.

[4] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2007*, pages 12–21, 2007.

[5] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. *In Proc. of the 22nd Int. conference on Software engineering*, pages 439–448, 2000.

[6] Pedro de la Camara, Maria del Mar Gallardo, and Pedro Merino. Abstract matching for software model checking. *In 13th Int. Workshop on Model Checking of Software (SPIN06)*, pages 182–200, 2006.

[7] E. Allen Emerson. The beginning of model checking: A personal perspective. *25 Years of Model Checking 2008*, 27-45, 2008.

[8] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal of Software Tools for Technology Transfer*, pages 366–381, 2000.

[9] Jason Hickey. *Introduction to Objective Caml.* Cambridge University Press, January 2008.

[10] Gerard J. Holzmann. Logic verification of ansi-c code with spin. *7th International SPIN Workshop*, pages 131–147, 2000.

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

[12] Gerard J. Holzmann and Margaret H. Smit. Software model checking: Extracting verification models from source code. *Formal Methods for Protocol Engineering and Distributed Systems*, (Conference Proceedings FORTE/PSTV99):481–497, 1999.

[13] Bengt Jonsson. State-space exploration for concurrent algorithms under weak memory orderings. *First Swedish Workshop on Multi-Core Computing (MCC-08)*, pages 82–88, 2008.

[14] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *In Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[15] 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.

[16] Joshua B. Smith. *Practical OCaml*. Apress, 2006.

[17] Joyce L. Tokar. Space & time partitioning with arinc 653 and pragma profile. *Proceedings of the 12th international workshop on Real-time Ada*, XXIII(4):52–54, 2003.

[18] National Institute of Standards and Technology US Department of Commerce. Software errors cost u.s. economy $59.5 billion annually. *NIST News Release*, June 2002.

Appendix

1. Syntax of the C subset (partial)

   - Statements
     Refer to table 26 ([ ] means optional terms).

Assignment	variable = <expression>;	
Function call	[variable =] functionName(<parameter-list>);	
Jump	goto	goto <label>;
	break	break;
	return	return <expression>;

Selection	if, else	if (<expression>)
		    <block1>
		else
		    <block2>
		or:
		if (<expression>)
		    <block>

	switch, case, default	switch (<expression>) {
		case <label1> :
		<statements 1>
		    case <label2> :
		<statements 2>
		    ...
		    default:
		        <statements n>
		}

Iteration	while	while (<expression>) <block>
	for	for (<expression> ; <expression> ; <expression>) <block>

	do..while	do
		    <block>
		while (<expression>) ;

Table 26: Supported C statements

   - Function definition

```
<return-type> functionName( <parameter-list> )
   {
      <statements>
      return <expression of type return-type>;
```

```
        }
```

The return statement is optional.

2. Statement reorganization function for switch structure

```
let temp = ref [] in
let postbl = ref [] in
let rec findpostbl tbl =
    match tbl with
    | [] -> List.append !postbl [!temp]
    | (th::tt) ->
        if(th.labels = []) then
            temp := List.append !temp [th]
        else if(!temp = []) then temp := [th]
            else begin
                postbl := List.append !postbl [!temp];
                temp := [th];
            end;
        findpostbl tt
```

After execution, the function will return a list of statement list that have gathered the statements sharing the same label into indipendent lists.

3. The function for locating the guard and the block in interation

(*lb is the block in the CIL loop syntax*)

```
    let cond, blk =
        let rec skipEmpty = function
            | [] -> []
            | {skind=Instr [];labels=[]} :: rest -> skipEmpty rest
            | x -> x
        in
        match skipEmpty lb.bstmts with
        (*Match the first statement after skipping all the empty elements*)
        | {skind=If(e,tb,fb,_); labels=[]} :: rest -> begin
            (*In case that the first statement is an if*)
            match skipEmpty tb.bstmts, skipEmpty fb.bstmts with
                | [], {skind=Break _; labels=[]} :: _ ->
                        (*The loop terminates if the guard e is violated,*)
```

```
                    (*so negate the expression e*)
                    UnOp(LNot, e, intType), rest
                | {skind=Break _; labels=[]} :: _, [] ->
                    (*The loop terminates if the guard e is satisfied,*)
                    (*then use e directly*)
                    e, rest
                | _ -> raise Not_found
            end
        | _ -> raise Not_found
```

4. Queue implemented in array

   C:

```
#define MAXLEN 8
#define MASK MAXLEN

int writePosQ = 0, readPosQ = 0, countItemQ = 0;
int Buffer[MAXLEN];

int isQueueEmpty() {
    int retval = 1;
    if(countItemQ != 0)
        retval = 0;
    return retval;
}

void insertDataQueue(int d) {
    if(countItemQ != MAXLEN)
    {
        Buffer[writePosQ] = d;
        writePosQ++;
        writePosQ%=MASK;
        countItemQ++;
    }
}

int extractDataQueue(void) {
    int retval = -1;
    retval = Buffer[readPosQ];
    readPosQ++;
    readPosQ %= MASK;
    countItemQ–;
    return retval;
}

void main(void) {
    int queuedata;
    while(1) {
        insertDataQueue(3);
        queuedata = 0;
        while(!isQueueEmpty()) {
            queuedata = extractDataQueue();
        }
    }
}
```

Promela:

```
int writePosQ = 0;
int readPosQ = 0;
int countItemQ = 0;
int Buffer[8];

proctype isQueueEmpty(chan in_isQueueEmpty){
  int retval;
  retval = 1;
  if
  :: (countItemQ != 0) ->
    retval = 0
  :: else
  fi;
  in_isQueueEmpty ! retval;
  goto end;
```

```
end :
  printf ("End of isQueueEmpty")
}

proctype insertDataQueue(chan in_insertDataQueue; int d){
  if
  :: (countItemQ != 8) ->
    Buffer[writePosQ] = d;
    writePosQ ++;
    writePosQ = (writePosQ % 8);
    countItemQ ++
  :: else
  fi;
  in_insertDataQueue ! 0;
  goto end;
end :
  printf ("End of insertDataQueue")
}

proctype extractDataQueue(chan in_extractDataQueue){
  int retval;
  retval = -1;
  retval = Buffer[readPosQ];
  readPosQ ++;
  readPosQ = (readPosQ % 8);
  countItemQ –;
  in_extractDataQueue ! retval;
  goto end;
end :
  printf ("End of extractDataQueue")
}

proctype main(chan in_main){
  chan ret_insertDataQueue = [0] of { bit };
  chan ret_isQueueEmpty = [0] of { int };
  chan ret_extractDataQueue = [0] of { int };
  int queuedata;
  int tmp;
  do
  :: true ->
    run insertDataQueue(ret_insertDataQueue, 3);
    ret_insertDataQueue ? 0;
    queuedata = 0;
    do
    :: true ->
      run isQueueEmpty(ret_isQueueEmpty);
      ret_isQueueEmpty ? tmp;
      if
      :: tmp ->
        break
      :: else
      fi;
      run extractDataQueue(ret_extractDataQueue);
      ret_extractDataQueue ? queuedata
    od
  od;
end :
  printf ("End of main")
}

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

5. Test harness implementation for (E | D)

```
void e(struct queue_t *Q) {
    int val = 1;
    enqueue(Q, val);
}


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


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