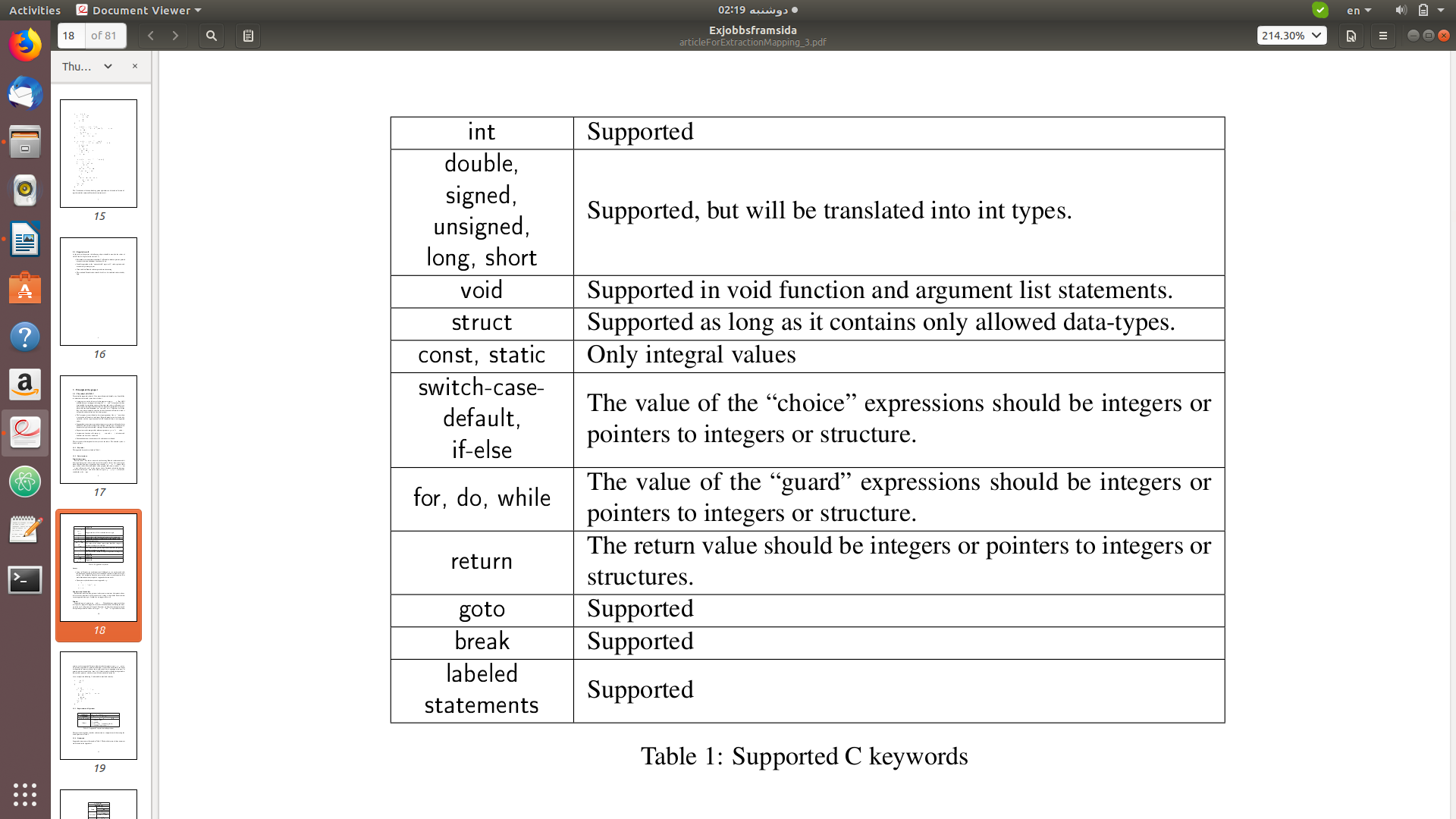
* syntax-directed translation techniques,
* several **tools and languages:**
* Ocaml = is the language we used for programming
* CIL = is the carrier and does the C syntax analysis for our translator
* Spin model checker = is the tool for checking the translations
* C = is the input language, and
* Promela = is the target output language.
* The most significant problems that we want to **check** are potential data races and overflows of concurrent programs.
* They 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.

* They bound the target programs into a **subset of C** containing: basic data-types, expressions, statements and functions.
* Manual:
* 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.

***C:***

* **Keywords:**

****

* **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.

* Arrays:
* one-dimensional (Although we can work around with this limitation by defining arrays of user-defined structures which have arrays inside)
* 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 ++;

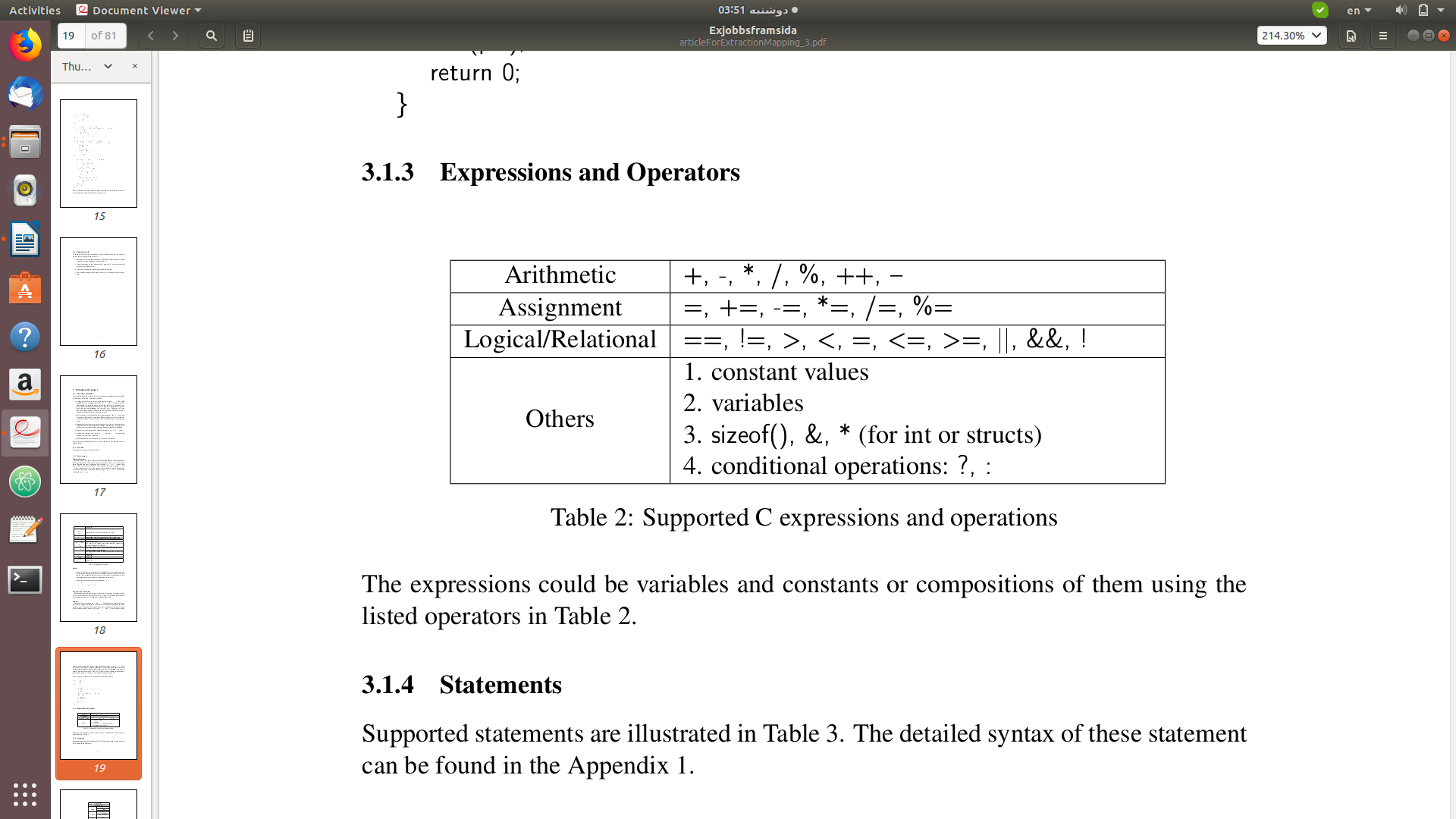
ptr2->age = 18;

free(ptr);

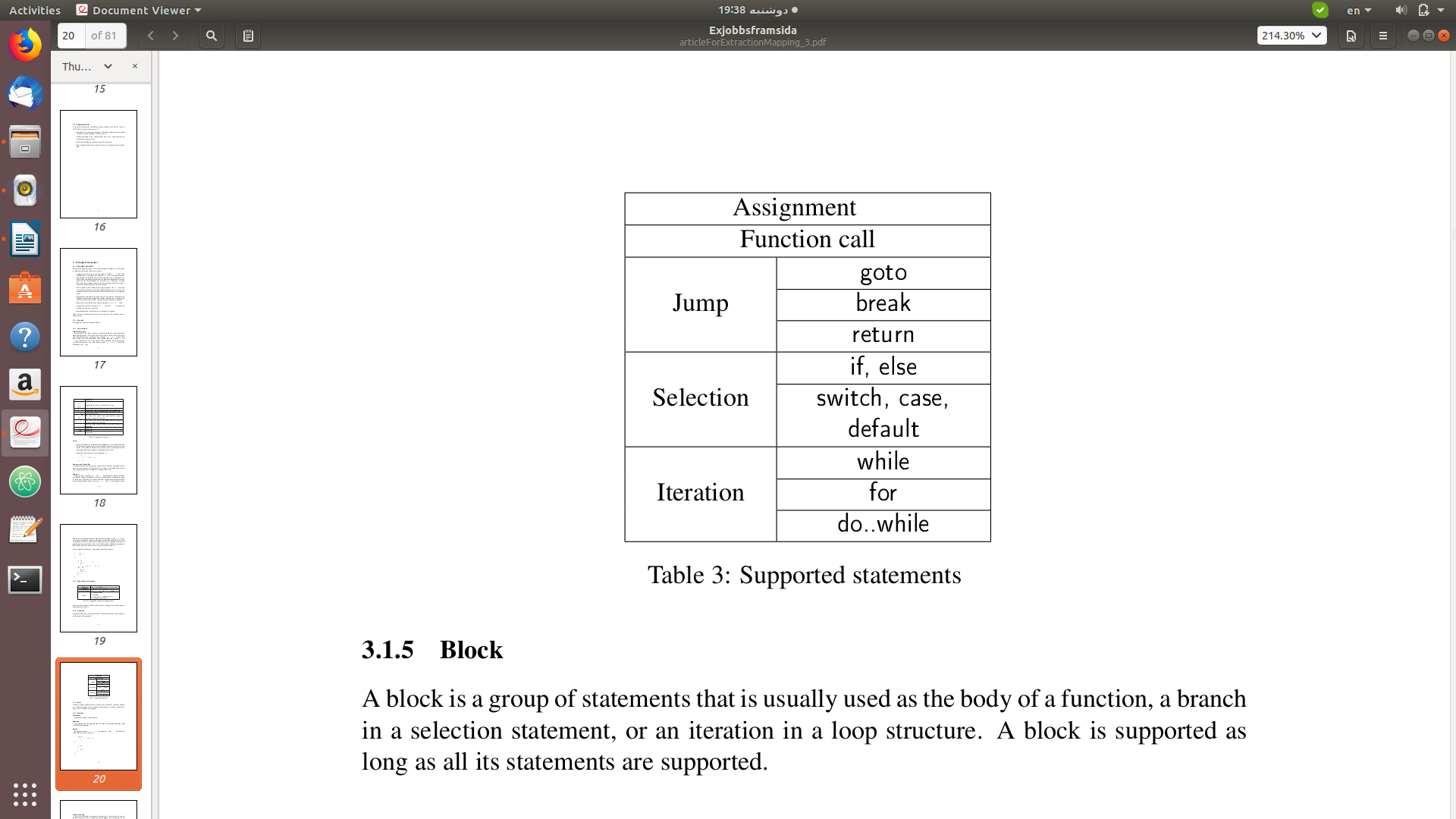
return 0;

}

* **Expressions and Operators:**



* **Statements**:



* **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.

* **Functions**:
* Declaration:

Function declarations will be ignored.

* Definition:

Function definitions are supported.

* 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:

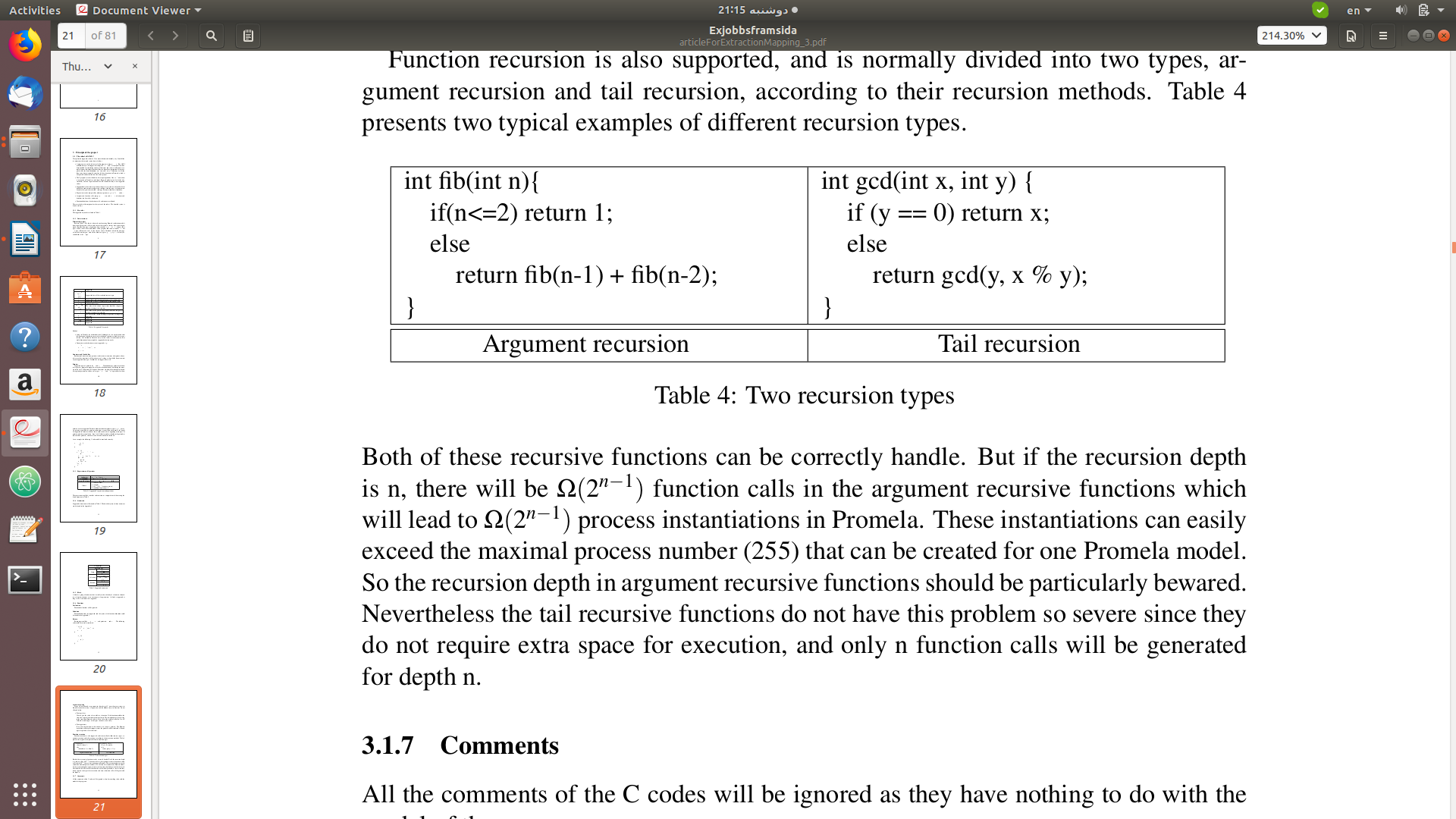
You can pass the value of a variable to a function. If the function modifies the value, the original variable remains unaltered. The corresponding value passing in the translated Promela codes will not affect the original variables as well. Allowed variable types are integers, structures (non-array).

* 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.

* 

Both of these recursive functions can be correctly handle. But if the recursion depth is n, there will be Ω(2 n−1 ) function calls in the argument recursive functions which will lead to Ω(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.

* **Comments**:

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

* **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.

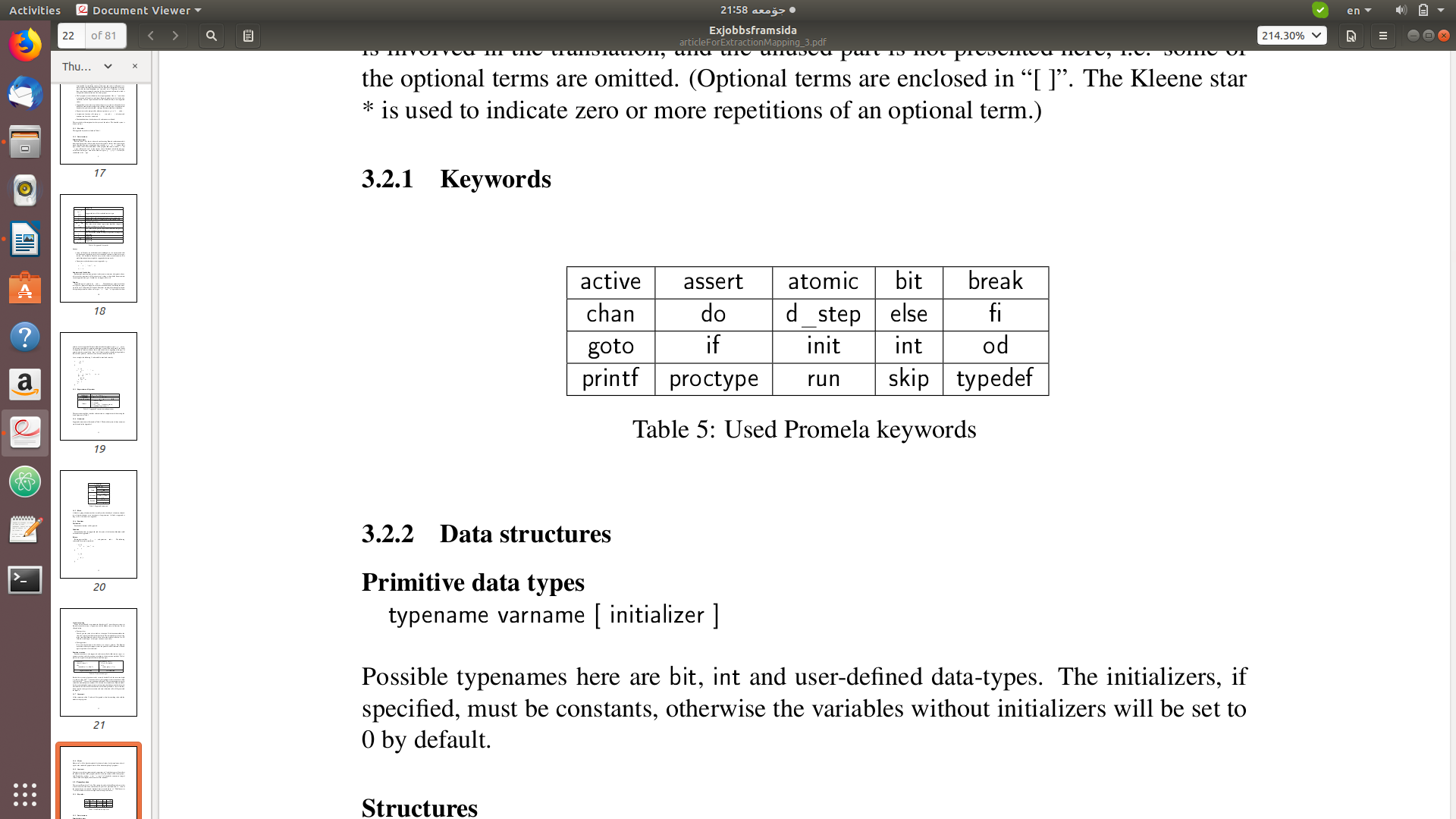
* **Constants**:

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.

***Promela:***

* **Keywords**:



* **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.

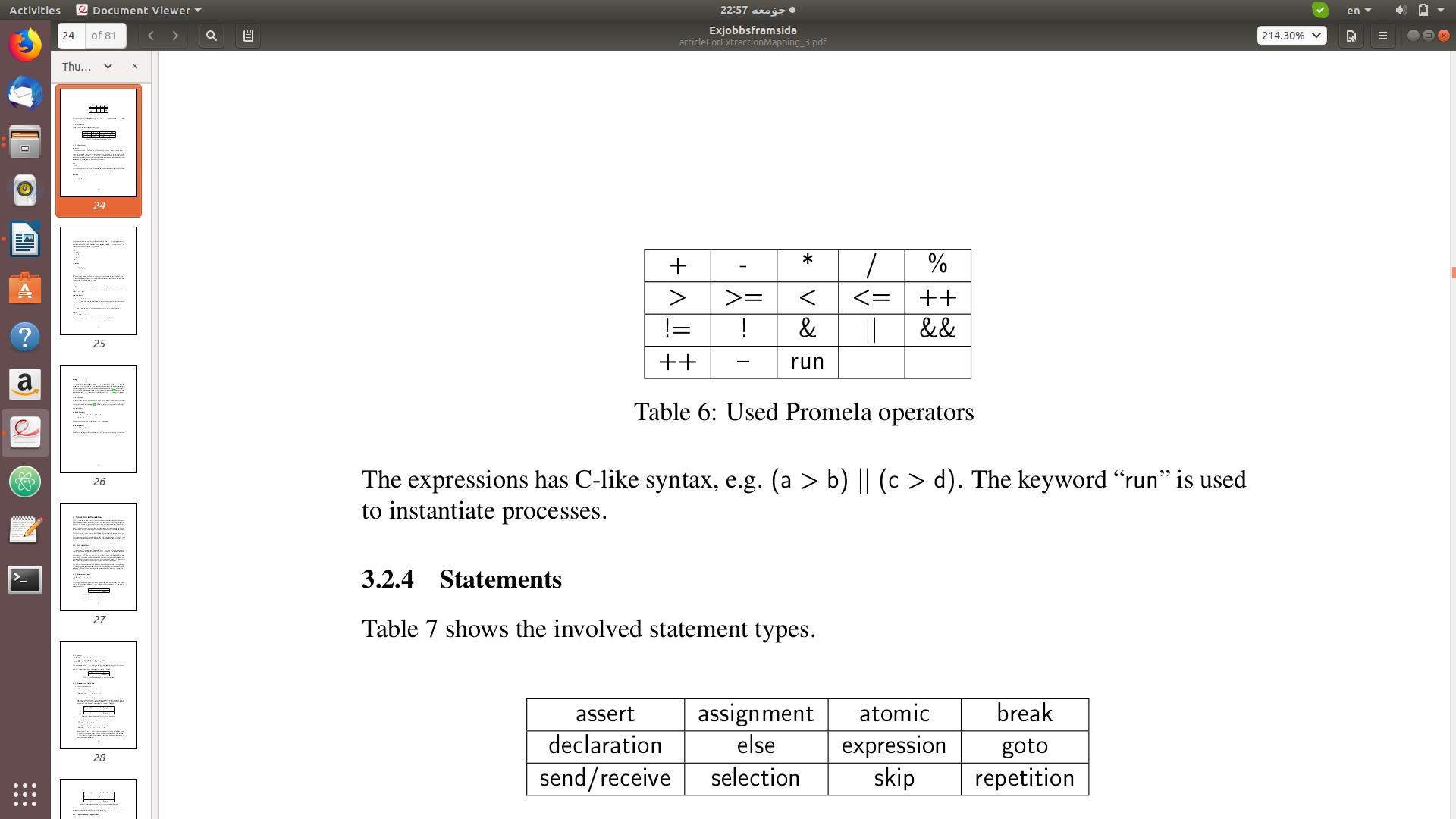
* Channels:

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.

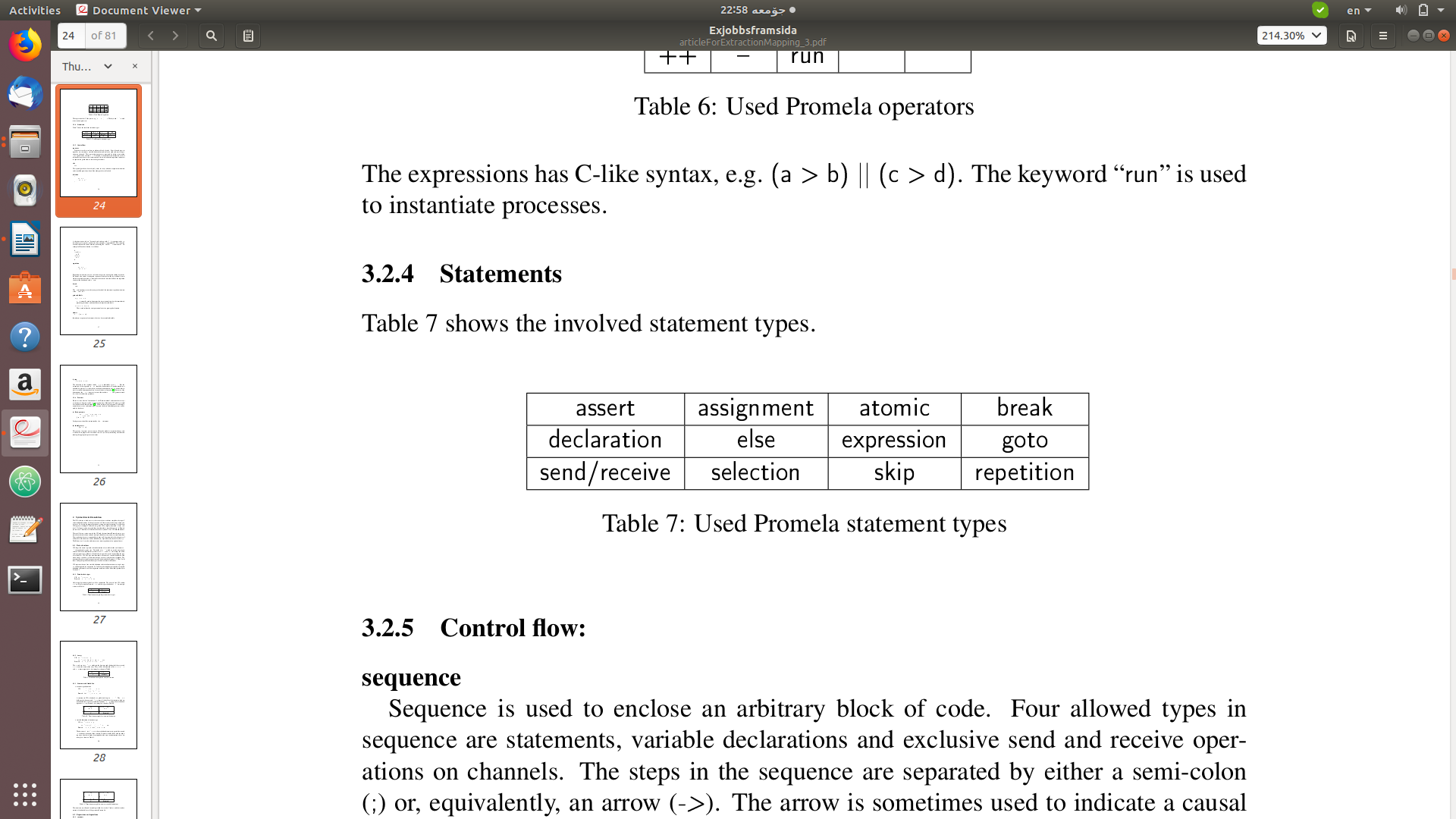
There are two types of channels with different semantics: rendezvous channels of capacity zero, e.g. chan ret\_main = [0] of { int }, and buffered channels of capacity greater than zero, e.g. chan comm = [9] of { int } . In this project, the channels will only be used for synchronizing and passing values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

* **Expressions**:



expressions could also be just a variable or a value.

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

*  **Statements**:
* **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 oper-

ations 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:

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 # . 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:

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. Because of the determinism, the d\_step structure is more efficient that atomic, and generates much less states in verification procedure

* **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.

***Syntax directed translation***

***CIL***

* part 4
* 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 .

The methods were inspired by Bengt Jonsson’s paper [13], in which he manually translated some pointers and operations to Promela.

* Comparison with 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.