Bachelor thesis
Computer Science

Radboud University

Generating a Google Go framework from an Uppaal model

Author:
J.J. (Jip) Dekker
s4122100

First supervisor/assessor:
prof. dr. F.W. (Frits) Vaandrager
`F.Vaandrager@cs.ru.nl`

Second assessor:
drs. R.H.A.M. (Rick) Smetsers
`R.Smetsers@cs.ru.nl`

August 21, 2014

Abstract

Dealing with concurrency and parallelism is hard. Using scientific tools, models can be created giving insight into the nature of concurrent structures. Using these models one can sometimes even prove that various conditions hold within the model. To prevent human error when implementing a given model, one might consider a scheme where the source code for the concurrent behavior is generated from the model.

We will demonstrate the possibility of such a scheme, specifically to generate a Google Go code framework from Uppaal models, which allows for the quick testing and correct implementation of the process described in by the model. Furthermore we will show the use of such a scheme using a example. Finally we will informally argue the validity of the algorithm.

Contents

Chapter 1

Introduction

One of the problems that modern computer programmers face is writing concurrent programs [23]. Although a problem may appear to have been easily solved using concurrency and the program seems to pass all tests, a problem could be hiding within the concurrent nature of the program. For example, let us take a look at the following piece of Java code [11]:

```
class MyThread extends Thread {
  private boolean stop = false;

  public void run() {
    while(!stop) {
      doSomeWork();
    }
  }

  public void setStop() {
    this.stop = true;
  }

  public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start();
    try {
      Thread.sleep(5 * 1000);
    } catch(InterruptedException ex) {
      // Something went wrong...
    }
    t.setStop();
  }
}
```

In this example, the main thread creates and starts a new instance of "MyThread" and after five seconds calls the "SetStop()" method when the thread should stop. To most programmers this code seems pretty harmless, but a concurrency and Java expert could detect the fatal flaw. In the definition of a thread. When this thread is created, the run method will run separately from the current running piece of code. And as we can clearly see in the code, this thread will do "doSomework" while the stop variable is not set to "true", for which we even have a method to do this. So what is the problem of this piece of code? The principle is correct, but we make a false assumption. We assume that the variable written by the main thread, is the same as the variable read by the new thread and that the new thread detects the change. This however is only guaranteed in Java when we use the "synchronized" keyword. So using this piece of code could result in doing a few minutes more of "SomeWork", until finally the variable is automatically synchronized. The extra runs of "SomeWork", could result in inconsistent data or other disastrous flaws.

Though a lot of scientists work on both the theoretical and practical sides of concurrency, there is no definite solution for designing and testing a concurrent system, guaranteeing the functionality of the system as a whole. However, there are various approaches to concurrency which improve upon the programmer's toolset to help avoid the problems of concurrent programming. One of the tools in a programmer's toolset is the programming language the programmer chooses. Different programming languages take very different approaches to concurrency. Lately new programming languages with a focus on concurrency try to increase the comprehensibility of the source code and the basic concurrency principles. The comprehensibility of the program is important, because there are various tasks which a parallel programmer has to undertake that are not required of sequential programmer (work partitioning, parallel access control, resource partitioning and replication and direct interactions with hardware) [23]. Due to the difficulties of these tasks, it can be hard to comprehend the source code to its full extend. Concurrency errors in execution of programs can be partly attributed to the limited comprehension of the source code, which also makes it hard to correct these errors.

A solution to these comprehension issues might be fairly evident, the source code should be easier to comprehend. An example of such an approach can be found in the Google Go programming language[1]. The Google Go programming language is a language created out of frustration with the existing programming languages. According to the Google Go website [2]: "Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. . . . Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language.". The initial design of the language was started by Robert Griesemer, Rob Pike and Ken Thompson in 2007 and besides its focus on simplicity, the language is aimed to be modern, with support for networked and multi-core computing. Thus the language supports concurrency at its core level. In November 2009, Go became an public open source project, since then, many people of the community have contributed ideas, discussions and code. The ease of the resulting programming language can be easily demonstrated by the revisal of the earlier Java code:

```
func MyThread(stop chan bool) {
  for {
    select {
    case <-stop:
      return
    default:
      SomeWork()
    }
  }
}

func main() {
  signal := make(chan bool)
  go MyThread(signal)
  time.Sleep(5 * time.Second)
  signal <- true
}
```

One can see that threads can be started using the keyword “go” before the function call and to stop we simply send a message on the stop channel. To further improve on the success of concurrency within the Google Go programming language there are a lot of talks on using concurrency using Google Go [3] and tools like the “Go Race Detector” [19] which helps spot hard to find race conditions. This makes Google Go into a great language to use if you are developing a concurrent program. We will discuss the concurrency primitives in the preliminaries.

Another, more scientific, approach to the problem of preventing concurrency errors is the use of model checkers. Generally these tools let the user create a model of the proposed solution to a specific problem and check this against queries given by the user. A model checker specifically focused on real-time systems is Uppaal. This application can be used to model and validate concurrent systems. Other than a programming language, Uppaal does not just have a textual description of the system, but mainly consists of a graph-like representation of the states and transitions of the different components in the system. The systems can become more and more intricate by adding supporting code and conditional transitions. Once a Uppaal model is constructed, it can be verified using Uppaal’s own query language, given that the model is not too large or too complex. For example to verify if a system is viable for a deadlock (which in Uppaal is when there is no outgoing action transition for all the processes [16]), we only need to enter the following query: “A[] not deadlock”. Although Uppaal can be very expressive, it has its limitations. Uppaal is based on a mathematical theory on concurrency called the Calculus of Communicating Systems. While a lot of conditions can be verified within the system using CSS, when faced with a design that is too intricate, the number of possible reachable states becomes too large that it is almost impossible to compute the necessary answers.

Another problem that a Uppaal model faces, is that it cannot be used as an actual program in a production environment. Because of the similarities between both

Google Go and Uppaal, generating a concurrency code framework in Google Go from a Uppaal model seems beneficial for Uppaal users who, for testing or implementations purposes, need an actual working program. But this technique could be just as useful for Google Go users who want to validate their code. In this thesis we will formulate an algorithm that perform this specific task. More specific, it will analyze the structure, surrounding logic and initiation of the processes in a Uppaal model and produce Google Go source code which starts goroutines, according to the initiation of the model, that follow the same structure of the processes using a conversion of the surrounding logic. The produced Google Go source code can immediately be compiled using the Google Go compiler and executed. Uppaal models are, however, limited, in most programs we would at least like to see some output, therefore the generated source code is only a framework of an actual computer program and will most likely require some modifications and additions to result in a working computer program. For this reason we will assume that the Uppaal models, for which we generate a code framework, are complete in the logic of the transitions, meaning no external effects can be added to the transitions, but that extra code can be executed while in a location. Note that a process will wait until the user made code is executed before continuing to the next transition. After the specification of the algorithm itself, we will show the use of the algorithm by means of an example. Finally, we will argue the validity of the generated structures by means of informal reasoning.

Chapter 2

Preliminaries

To fully understand the proposed algorithm described in this paper, there are a few basic principles that should be known to the reader. For the reader's convenience we will discuss these principles shortly as an introduction to each of these subjects.

2.1 Model Checker: Uppaal

Model checkers are tools that are widely used within the field of computer system engineering. These tools are used to compute the (in)possibilities of a model. In this paper we specifically use Uppaal. On the Uppaal website[6] the tool is described as follows: "Uppaal is an integrated tool environment for modeling, validation and verification of real- time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).". Meaning that Uppaal is not just way of describing a real-time system in a model, but Uppaal also provides a way to visualize the processes of the system and analyze the model using the Uppaal query language. Using the Uppaal toolset we are able to evaluate systems, programs or critical processes and prove that within the concurrency constructs specific conditions hold.

As the above description suggests, each of the processes (or templates as they are called in the Uppaal documentation) described in a Uppaal model is modeled as an automaton. Each of these automata consist of a directed graph structure where the nodes are called locations and the directed edges are called action transitions (or just transitions). These graphs are extended using extra conditions that should hold when in a specific location or when a transitions is executed. To add extended logic to these conditions, Uppaal has added declarations. These are declarations of variables, functions or types made in a C-like language, while these declarations can be global, and thus available for all templates, each template can also have its own declarations, where variables are local to each instantiation of a template. An example of a Uppaal template can be seen in figure 2.1

Locations have a few attributes which can be changed by the user. In each template,

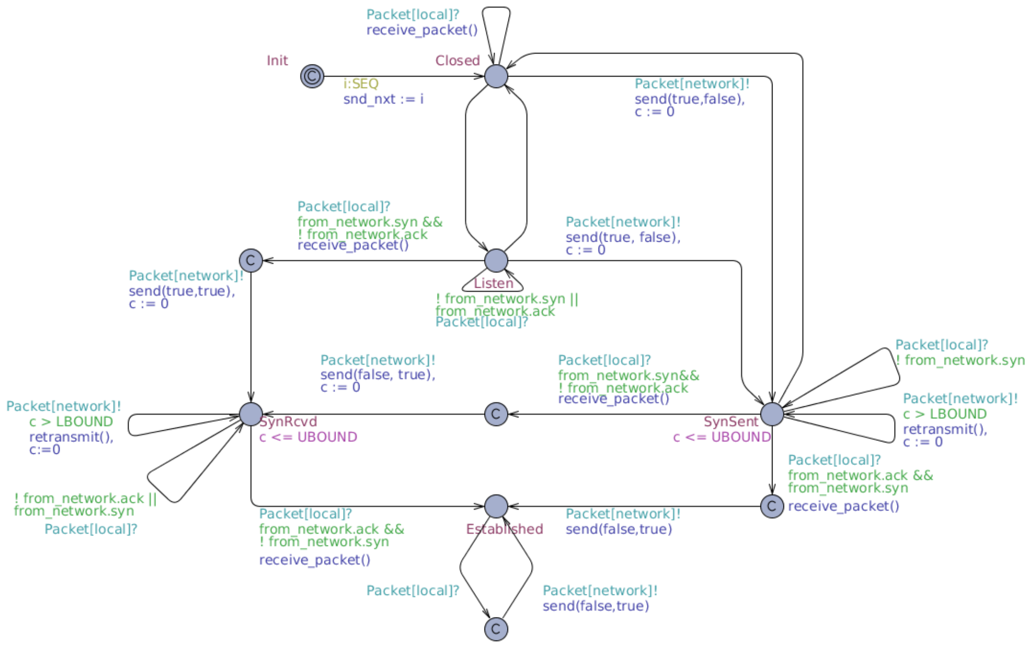


Figure 2.1: An example of a Uppaal template

one of the locations in the template has to be the initial state. When the template is initiated, this location will be active. Although there are other attributes for locations, these won't be considered in this thesis.

When a location is active, it might trigger one of its outgoing transitions. Transitions can include four different statements. The first statement is the seletct statement. In a select statement a value is selected from a range of values in a non-deterministic way. This value can be used in the remaining three statements. Note that when a query is checked each possibility results in a different transition, and when checking a query, Uppaal will check all possible transitions. If a transition has a guard statement, the guard statement should evaluate to "true" for the transitions to be able to trigger. A guard statement is, therefore, an expression. The third type of statement is the synchronization statement. If the synchronization statement is absent, the transition is internal and when triggered there are no other transitions within any of the instantiations of any of the templates can be triggered at the same time. If however the synchronization statement expresses a send action on a default channel, the transition can only be triggered at the same time as a transition which has a synchronization statement which expresses a receive action on the channel (and vice versa). Transitions that include such a synchronization statement are called binary synchronizations. If the synchronization statement on a transition include a send action on a broadcast channel, such a transition can always be triggered (if allowed by the guard statement) and if there are any transitions which include a receive action on the same channel, these will also be triggered. The transitions that include a receive action on a broadcast channel, however, still cannot be triggered if there is no transition that performs a send action on the same channel. Transitions that include such a synchronization statement are called broadcast synchronizations. When a transition is triggered, the last statement will be evaluated. This statement is the update statement, like the other statements, this statement is an expression, but unlike the other statements, the update statement may have side effects which change the state of the state of the system. After executing the transition, the new active location is the location for which the transition is an incoming transition.

Using these constructs, various systems can be expressed in Uppaal. However if the systems in a Uppaal model become too large, evaluating queries on these models will take exponentially more time (due to state space explosion). Therefore Uppaal model usually only contain the most critical processes and information.

For a more extensive explanation of Uppaal, one might consider reading “A First Introduction to Uppaal” [25] and “A tutorial on uppaal 4.0” [16]

2.2 Google Go: Concurrency Primitives

As explained in the instroduction, this paper will generate a Google Go code framework out of a Uppaal model. For a basic knowledge of Google Go is required. For a basic understanding of the syntax of the language the reader may follow the Google Go tour [5], which gives one a sense of the language. One can find an overview of the syntax in the Go Programming Language Specification [10]. Though the most important part of the language for this paper are the concurrency primitives.

The first and most important primitive is the “Go statement”. This statement consist of the keyword “go” followed by a function or method call. The function value and parameters are evaluated as usual, but unlike a regular call, the call is executed as an independent concurrent thread of control, also called a “goroutine”, within the same address space. This means that the program execution does not wait for the invoked function to complete, but the function executes independently. When the function terminates, its goroutine also terminates. If the function returns any values, these are discarded upon completion.

Another native Google Go primitive that is important for concurrent programs is the channel structure. Channels act as a synchronized first-in-first-out queue. For example, one goroutine might send values on a channel that are received by another goroutine. A channel is created with a specific data type and only that specific data type can be send and received on the channel. We differentiate two kind of channels, buffered and unbuffered channels. Before communication begins the channel (and in case of a send statement, the expression to be sent) are evaluated. Communication on a channel blocks the goroutine until the action can proceed. On an unbuffered channel, this means that a sender can only proceed if a receiver is ready to receive the data. If however the data is send on a buffered channel, the sender can proceed if there is room in the buffer. If there is data in the buffer, the receiver can proceed immediately as well. The syntax for sending is as follows:

```
channel <- "data"
```

Where channel is a channel for string types. Instead of raw data, variables can also be used as input for the channels. The syntax for receiving data from a channel is as follows:

```
str := <- channel
```

In this case the data received from the channel will be saved in the variable named "str". The values however do not have to be saved, the program will however still block until it receives the data. For such an action the following syntax would be used:

```
<- channel
```

When all data has been send, a channel can be closed. A receive statement on a closed channel will proceed immediately, returning a "zero value" for the specified type. Sending on a blocked channel will cause a "run-time panic". A channel can also be "nil". In case of a "nil" channel, both the send and receive statement block forever.

The last important core concurrency primitive in Google Go is the "Select statement". This statement is similar to the switch statement (in C++ and other programming languages). The difference is that instead of switching on cases for a specific variable, the select statement has cases for different send and receive statements. For all the cases in the statement, the channel expressions are evaluated in a top-to-bottom order, including expressions on the right hand side of send statements. A channel can be "nil", in which case it will be discarded as a possibility, although the expression on the right hand side of a send statement will still be evaluated. If any of the resulting operations can proceed, one is chosen and the associated communication and statements are evaluated. If none of the operations can proceed and there is a default case, it will be executed. If there is no such case, the goroutine will block until one of the communications can complete. Note that if there are no non-"nil" cases, the goroutine will block forever. If multiple cases can proceed, a uniform pseudo-random choice will be made to decide which single communication will be executed. The following piece of code is an example of a correct select statement:

```
var c, c1, c2, c3 chan int
var i1, i2 int
select {
case i1 = <-c1:
        print("received ", i1, " from c1\n")
case c2 <- i2:
        print("sent ", i2, " to c2\n")
case i3, ok := (<-c3):  // same as: i3, ok := <-c3
```

```
                if ok {
                        print("received ", i3, " from c3\n")
                } else {
                        print("c3 is closed\n")
                }
        default:
                print("no communication\n")
        }
```

Although these are the most important concurrency mechanisms Google Go has to offer, there are more. The so called “sync” package [4] for instance offers other structures. In this package we find multiple semaphores, a type they called a “WaitGroup”, which can be used to wait for different goroutines to finish, a structure to enforce an object that can only be used once and a type to implement conditional variables. Although using most of these is discouraged as they are intended to be used by low-level libraries, Google Go offers simple ways to use them.

Chapter 3

The Algorithm

The algorithm itself consists of three steps: Converting the general declarations (the global and template declarations), converting the templates and converting the process initiation (the system declarations). Each of these steps in the algorithm will convert a part of the model. When these conversions are performed in the specified order, the validity of the system can be checked and we can thus guarantee that, if no errors occur in the conversion process, the resulting source code will offer no compile errors and will be in accordance to the provided Uppaal model. Specifically, within the general declarations, we can check if the correct syntax, which we can convert, is used. Using these declarations we can check that all references in the templates themselves are valid. And when the templates are known, we can confirm that the processes, that are started, actually exist.

One should note that Uppaal can deal with nondeterminism. There are, for example, situations when a process can take one of two (or more) transitions. In such a case, Uppaal will consider the results of all transitions that can be taken. In executable code a choice will have to be made, only one transition can be taken. In the generated Google Go code these kind of decisions will be made using a pseudo-random generator or the select statement; in both cases a pseudo- random choice will be made [8]. If this is not the desired behavior, the user should make sure that the outgoing transitions of each location are mutual exclusive, in such a way that at most one outgoing transition is applicable. The possibility to trigger multiple transitions at the same time is another source of nondeterminism. However, as transitions are atomic in Uppaal, there is no possibility for multiple transitions to be triggered at the same time in Uppaal. We will simulate a similar effect in Google Go.

This thesis only covers a subset of Uppaal models. It might be possible to integrate more of Uppaal's (more complex) constructs, but due to the added complexity of these components, these were outside the scope of this thesis. The following constructs that can be used in Uppaal models will not be covered in the following algorithm:

- Time restrictions (including urgent and committed locations).

- Clocks
- Channel Priorities
- Process Priorities
- Broadcast channels

3.1 Converting Global and Template Declarations

The first part of the algorithm consists of the conversion of the declarations of the model. Both global and template declarations consist of a C-like statements. These statements are meant to enable the user to create data structures, operations and channels. In typical declarations we will find type definitions, variable declarations and functions. Each of these can be used in the conditions and statements of the transitions in the templates.

The difference between the global and template declarations are accessibility. Template declarations are accessible only to the instantiation template itself. The global definitions, however, can be accessed from all templates. To ensure that the framework in Google Go exhibits the same behavior as the Uppaal model, the global variables in Google Go must be accessible from the different processes. Luckily Google Go's memory model [7] already guarantees the synchronization of variables between goroutines, the Google Go implementation of processes, given that the reading or writing happens sequentially. Both reading and writing of variables, in both global and local variables, in Uppaal models only happens when action transitions are considered or triggered. The conversion of the declaration can thus be done by converting in the basic structure of the converted transitions in Go 3.2, by guaranteeing that the resulting structure forces sequential read and writes.

The conversion of the Uppaal declarations to Google Go source code is done by a compiler. As both languages are derivatives of C, most statements in a Uppaal syntax can be expressed in a similar manner in Google Go. The main difference between the Uppaal syntax, other C-like languages and Google Go is that Google Go hassled with the order of most statements and the absence of semicolon. For example where C-like languages begin a declaration of a variable with its type, Google Go will explicitly state that it is a variable declaration and end the statement with the type after the name of the variable. So to declare a integer "i", in Uppaal we would state "`int i;`", but in Google Go the same will be accomplished with "`var i int`". We will assume that similar statements in both languages are evaluated in the same way. For all statements that are valid in the Uppaal Syntax [8], we will informally state what the compiler should do with the Uppaal statements to result in valid Google Go statements.

3.1.1 Converting Variable Declarations

The first type of statement in the Uppaal syntax is the variable declaration. Uppaal uses the following syntax for the declaration of variables:

```
VariableDecl ::= Type VariableID (',' VariableID)* ';'
VariableID ::= ID ArrayDecl* [ '=' Initialiser ]
Initialiser ::= Expression | '{' Initialiser (',' Initialiser)* '}'
Type ::= Prefix TypeId
Prefix ::= 'urgent' | 'broadcast' | 'meta' | 'const'
TypeId ::= ID | 'int' | 'clock' | 'chan' | 'bool'
   | 'int' '[' Expression ',' Expression ']'
   | 'scalar' '[' Expression ']'
   | 'struct' '{' FieldDecl (FieldDecl)* '}'
FieldDecl ::= Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
ArrayDecl ::= '[' Expression ']'
   | '[' Type ']'
```

Restriction: As of this time, Google Go does not have support for range types. We will implicitly convert these types to integers.

Restriction: Array declarations cannot have a Type between the square brackets.

To convert the syntax from Uppaal to Google Go, we have to make various changes. First we split the Type into its two parts. The Prefix will remain in front of the VariableID's and is simplified: It is either "const" matching the prefix used in the previous syntax and in all other cases the prefix will be replaced by "var" or a prefix "var" will be added. The "meta" prefix does not add any functionality, as it is used to exempt a variable from the state space. Therefore this prefix can be ignored. The other prefixes are on the list of Uppaal constructs that are not covered in this thesis. The other part of the Type will be placed after the VariableID's.

The TypeId's themselves will also need to endure some changes. At this time, the algorithm doesn't have any support for clock variables and it should not be used when using this algorithm. In accordance to the Google Go philosophy the array declaration is part of the type. This means that the array declaration cannot be changed for specific ID's within the same variable declaration. This will impact multiple variable declaration of the same type on one line, if the array declaration isn't the same, these will be split into two statements in the resulting Google Go statements. Also, the field declarations for the structure type will have a slightly different format. The Type will be moved to the end of the line, while multiple IDs can be stated in succession split by commas instead of semicolons. And again, as the array declaration is part of the type in Google Go, if multiple variable declarations of the same type have different array declarations, these will have to be split.

These changes also impact the VariableDecl in general, first all ID are stated split by commas and then all variables can be initialized. Lastly, all semicolons will be replaced by a newline. The resulting variable declaration will match the following syntax:

```
VariableDecl ::= Prefix ID (',' ID)* ['=' Initializer
 (',' Initializer)*] ArrayDecl* TypeId '\n'
Initializer ::= Expression | '{' Initializer (',' Initializer)* '}'
Prefix ::= 'var' | 'const'
TypeId ::= ID | 'int' | 'chan' | 'bool'
   | 'struct' '{' FieldDecl (FieldDecl)* '}'
FieldDecl ::= ID (',' ID)* ArrayDecl* Type '\n'
ArrayDecl ::= '[' Expression ']'
```

Exception: According to the Google Go philosophy, array declarations are part of the Type. Due to this fact a statement containing a multiple variable declarations with different "ArrayDecl" will have to be split into multiple statements.

Exception: In Google Go, channels have to be initialized, therefore for each declared unbuffered channel we will add a initialization to the start of the main function of the package. For example, a channel named "c1" would be initialized using the following expression: "`c1 = make(chan chan bool)`"

3.1.2 Converting Type Declaration

Once we know how to convert variable declarations, the other conversions are a lot easier as they contain some of the same rules. The following syntax is used for Type Declarations:

```
TypeDecls ::= 'typedef' Type ID ArrayDecl* (',' ID ArrayDecl*)* ';'
```

Restriction: It is not possible to create a type definition for purely constant values in Google Go. As the other prefixes were already ignored, all type prefixes are ignored in type definitions.

The only other changes we have to make are changing the keyword "typedef" to Google Go's keyword "type", moving the array declaration and type to the end of the line and replacing the semicolon with a newline. The resulting type definitions will match the following syntax:

```
TypeDecls ::= 'type' ID ArrayDecl* TypeId '\n'
```

Exception: Google Go does not allow multiple type definitions in the same statement. If applicable the statement would be split into multiple statements with the same type for each of the type definitions.

3.1.3 Converting Function Definitions

The most extensive of the main syntax rules are the rules concerning function definitions. The following syntax is used for function definitions:

```
Declarations ::= (VariableDecl | TypeDecl | Function | ChanPriority)*
Function ::= Type ID '(' Parameters ')' Block
Block  ::= '{' Declarations Statement* '}'
Statement ::= Block
 | ';'
 | Expression ';'
 | ForLoop
 | Iteration
 | WhileLoop
 | DoWhileLoop
 | IfStatement
 | ReturnStatement
ForLoop  ::= 'for' '(' Expression ';' Expression ';' Expression ')' Statement
Iteration ::= 'for' '(' ID ':' Type ')' Statement
WhileLoop ::= 'while' '(' Expression ')' Statement
DoWhile ::= 'do' Statement 'while' '(' Expression ')' ';'
IfStatment ::= 'if' '(' Expression ')' Statement [ 'else' Statement ]
ReturnStatement ::= 'return' [ Expression ] ';'
Parameters ::= [ Parameter (',' Parameter)* ]
Parameter ::= Type [ '&' ] ID ArrayDecl*
```

Restriction: It is not possible to create a function that returns constant values or takes constant values as parameters in Google Go. Therefore all type prefixes are ignored in return types and parameters.

Luckily for us most of these most of these statements are similar to the Google Go syntax. For the function definition itself a keyword “func” will be added at in the front of the line, the return type will be placed just before the block. For the parameters the order in which each parameter is stated will be different, like in previous cases. A parameter will be of the form: “ID [´*´]ArrayDecl* TypeId”. Note that instead of a reference, we use a pointer, this should be taken into account in the general conversions (see section 3.1.4). Instead of the “while” keyword, Google Go uses the “for” keyword, we will replace the instances of “while”. Like the other syntaxes, again the semicolons will be replaced by newlines. Due to this fact, Google Go does not accept if/else and for statements (which includes converted while statements) without braces surrounding to which it applies. If these are not already present we’ll add the necessary braces. The resulting function definitions will match the following syntax:

```
Declarations ::= (VariableDecl | TypeDecl | Function | ChanPriority)*
```

```
Function ::= 'func' ID '(' Parameters ')' TypeId Block
Block  ::= '{' Declarations Statement* '}'
Statement ::= Block
 | '\n'
 | Expression '\n'
 | ForLoop
 | WhileLoop
 | IfStatement
 | ReturnStatement
ForLoop  ::= 'for' '(' Expression ';' Expression ';' Expression ')' Statement
WhileLoop ::= 'for' '(' Expression ')' Block
IfStatment ::= 'if' '(' Expression ')' Block [ 'else' Block ]
ReturnStatement ::= 'return' [ Expression ] '\n'
Parameters ::= [ Parameter (',' Parameter)* ]
Parameter ::= ID [ '*' ]ArrayDecl* TypeId
```

Exception: There is no similar structure in Google Go to the "DoWhile" control structure. We can however transform this structure to a structure where we first execute the structure once and then begin a "WhileLoop" with the same statement and expression. This way the statement can be translated.

Exception: Likewise, there is no similar structure in Google Go to represent the "Iteration" control structure. We can however perform an analysis of the given type and generate a for-loop which spans these values.

3.1.4 General Conversions

Each of these different kind of statements in the Uppaal syntax depend on the last building block: the basic expressions. In Uppaal these expressions are formulated using the following syntax:

```
Expression ::= ID
| NAT
| Expression '[' Expression ']'
| '(' Expression ')'
| Expression '++' | '++' Expression
| Expression '--' | '--' Expression
| Expression Assign Expression
| Unary Expression
| Expression Binary Expression
| Expression '?' Expression ':' Expression
| Expression '.' ID
 | Expression '(' Arguments ')'
 | 'forall' '(' ID ':' Type ')' Expression
 | 'exists' '(' ID ':' Type ')' Expression
| 'deadlock' | 'true' | 'false'

Arguments ::= [ Expression ( ',' Expression )* ]

Assign ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
| '|=' | '&=' | '^=' | '<<=' | '>>='
Unary ::= '+' | '-' | '!' | 'not'
Binary ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
| '+' | '-' | '*' | '/' | '%' | '&'
| '|' | '^' | '<<' | '>>' | '&&' | '||'
| '<?' | '>?' | 'or' | 'and' | 'imply'
```

Restriction: The following operators are not supported in Google Go and should not be used: '>?', '<?' and '?:'.

Restriction: The following statements are restricted to Uppaal usage and should not be used in the declarations: "'forall' '(' ID ':' Type ')' Expression", "'exists' '(' ID ':' Type ')' Expression", "NAT" and "deadlock".

Other than the constraints given by the restrictions, only three minor changes must be conversions have to be made to the expressions. First we replace every occurrence of ":=" in the assignments with "=". In Uppaal these operators accomplish the same feat, in Google Go however, the first one can only be used for semi-dynamic variable declarations. The second conversion is only applicable for expressions within a function. As we use pointers instead of references (as stated in section 3.1.3), all the ID's of parameters that were passed by reference in the Uppaal definition should be converted to "&ID". This ensures us that we'll use the actual value instead of the pointer value. Lastly the operators 'or', 'and', 'imply' and 'not' must be converted to their logical equivalents, as they can't be expressed in this way in Google Go. The resulting expressions will match the following syntax:

```
Expression ::= ID
| '&'ID
| Expression '[' Expression ']'
| '(' Expression ')'
| Expression '++' | '++' Expression
| Expression '--' | '--' Expression
| Expression Assign Expression
| Unary Expression
| Expression Binary Expression
| Expression '.' ID
| Expression '(' Arguments ')'
| 'true' | 'false'
```

```
Arguments ::= [ Expression ( ',' Expression )* ]

Assign ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
| '|=' | '&=' | '^=' | '<<=' | '>>='
Unary ::= '+' | '-' | '!'
Binary ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
| '+' | '-' | '*' | '/' | '%' | '&'
| '|' | '^' | '<<' | '>>' | '&&' | '||'
```

3.1.5 Placement of Statements

Now that we know how to convert Uppaal statements to the Google Go syntax, we need to discuss were these statements should be placed. In Google Go, code is structured using packages, each packages can contain multiple files containing source code within the same folder. A package should consist of code working towards a specific goal and shares the same variable space. In our case our converted model will consist of one package. For the global declarations, this means that the converted statements can be added to the package without any repercussions. The usage of these declarations will be exactly the same, all goroutines instantiations are able to access the globally defined variables and functions, furthermore the type definitions can be used throughout all other statements within the package.

Placing the converted statements from the declarations for the instantiations of the templates is more complicated. Placing these statements in the global scope of the package will result in multiple problems. Besides the fact that all template instantiations would be able to access the variables, there would only one instance of each of the variables, where in Uppaal each of the instantiations of a template has access to its own set of instances of the variables declared in the template declarations. Furthermore, there can be naming conflicts between the template and the global declarations or the declarations of different templates. The last problems could have been solved by splitting each template into its own package. But while it solves the naming conflict, access to the global variables wouldn't be synchronized between the different packages and it doesn't solve the other problem. The solution lies in the structure in which the templates will be converted, by moving all variables defined in the template declarations into the function governing the template structure, as described in the section on template conversion (section 3.2), when a new instantiation of the template is called, new instances for all of the variables will be created and there cannot be any naming conflicts. Named functions and types, however, must be created in the global scope. To solve the possibility of naming conflicts we prefix the names of each of these functions and types with the name of the template for which it is declared. Because the variables are not declared in the global scope, some functions that made use of these variables are now unable to access the them. Therefore the functions declared in the template declarations are analyzed for the use of variables that are now out of scope and we add a pointer for these variables to the parameters of the function. By replacing the occurrences of these variables to a dereferencing of the passed parameters, the same result can be achieved. Within the structure that governs the template structure, special care is taken to pass these locally defined variables to the function calls where necessary.

3.2 Converting the Templates

Probably the most important part of the conversion process is the conversion of the templates. The Uppaal templates are the backbone of both the model and our generated application. Each template in the Uppaal model represents a process in the application or, more specific to Google Go, a goroutine. Each of these templates consists of locations and action transitions. The locations, or nodes, and action transitions, or edges, form a directed graph. For our application each of the locations will represent a different state of one of the processes and each of the action transitions describes a transition between two such states.

3.2.1 Representing Locations

To enforce the structure of the Uppaal models in the generated application we have to be mindful of the transitions within the models. In a Uppaal model the specific locations only enforces a specific invariant, which can be used to enforce a time constraint, which is outside the scope of our algorithm. While the edges of the template provide us with the means of selection, synchronization, control flow and variable manipulation and function execution. Therefore the locations will be the place where the user can add his or her own input within the code framework, while the more critical transitions should be complete in the Uppaal model and the generated code should be left untouched by the user.

To accomplish this, each location in the model will be translated into a function in the source code. The name for the function can be derived from the names of the locations, if the location is anonymous (the location has no name), the unique location id, as used in the XML save file, can be used. To avoid naming conflicts, each name is prefixed by the template name. In the most basic form of this algorithm, these functions representing the locations will have no arguments or return type and the contents of these location functions will thus have to be filled by the user. As it is not possible to extrapolate the contents of such a function from the Uppaal model. So for example, the function representing the first location in figure 3.1 would look like this in the generated source code:

```
func template1Location1() {
        //INSERT EXECUTABLE CODE
}
```

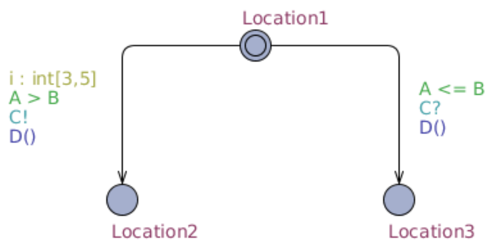


Figure 3.1: A small sample template

Note that in this case the location has no access to the template variables. If one considers it essential, read access could be provided to the locations, for example by passing the variables by value. However, to guarantee validity a location should never write to either the template or global variables. When shared variables are read, their values can't be guaranteed, another processes can change them at any time.

3.2.2 The General Template Structure

The easiest way to accommodate the structure of the program would be to string together the functions representing locations with the functions representing edges. This solution will result in errors when the program has a undetermined amount of transitions. Although the stack in which stacked function calls will be placed currently has a maximum size of 1GB on a 64 bit system[9], eventually it will run out of memory, if our program keeps making new function calls. A better approach is explicitly using the Automata-based programming paradigm. In this programming paradigm, the programmer regards the program as a model of formal automatons. When using this paradigm, the programmer should explicitly express the state in which the program is in. In our case we can accomplish this by defining an enumeration of all states and execute the right location function using a switch statement. The next state of the program is decided by a transitions function representing all outgoing states of a location. For functionality we add another value, None, to the enumeration. None will be used when implementing the transition functions. Note that all constructs in template definitions might suffer from naming conflicts and are thus prefixed by the template name. So the function for the template as a whole, using an example with three locations as shown in figure 3.1, will be represented in the following manner:

```
type State int

const (
        Template1None State = iota
        Template1Location1
        Template1Location2
        Template1Location3
        Template1Exit
)

func Template1() {
        state := Template1Location1 // UPPAAL INITIAL STATE
        for state != Template1Exit {
                switch(state) {
                        case Template1Location1:
                                template1Location1()
                                state = template1Location1Transitions()
                        case Template1Location2:
                                template1Location2()
                                state = Template1Exit
                        case Template1Location3:
                                templateLocation3()
                                state = Template1Exit
                }
        }
}
```

Note that if the template has any parameters, these will be arguments to this main function and passed as pointers to the states which use them. Likewise, the variable declarations will be added to the beginning of the template function. If these variables are used by any outgoing transition of a location, pointers to these variables are passed to the transition functions as parameters.

3.2.3 Converting Transitions

Converting the transitions of the Uppaal model is a more extensive task. In our application each of the transitions will represent an change between two states. Other than providing this change, a transition in Uppaal holds four statements stating the conditions for the transitions and the code that should be executed while transitioning. For each of these statements we will first look at a way to translate the individual statements and then combine the different statements to find a general translation of the transitions.

Converting different kind of transition statements

The first statement is the select statement. This statement provides the means to non-deterministically bind a value to a given variable from a provided range. As Google Go is deterministically sound, there is no one way in which the select statement can be converted. There are however two ways in which the select statement is used while modeling. The first way is when a variable should be initiated with a pseudo-random number. In this case, we can use the default library “rand” in Google Go to choose a random number from the same range as provided in the select statement. The second case in which the select statement is used is when the resulting value can’t be determined in the model, meaning that it might require further analysis than can be provided in the model. In this the user could add the necessary analysis to the location function and return the variable. This variable can then be passed to the transition function of that location as an extra parameter. Notice that analysis should always provide us with a value within the given range of the select statements for the model to remain valid. In our algorithm we will assume that a value of the select statement variable is chosen by the pseudo-random generator by default, but the user might change this into an analysis. This means that in the transition from Location1 to Location2 in figure 3.1 the select statement would be translated into:

```
var i int = (5-3) + rand.Intn(5-3)
```

The next kind of statement is the guard statement. This statement provides the means to make an transition conditional. Guard statements are expressions that evaluate to an boolean value, if the value equals “true”, the transition is applicable and the process can execute it. In most programming languages we would represent this by the use of an if/else structure. In Google Go, however, we can use the switch statement without an argument, which makes for a more clear syntax. For example the selection of transitions based on the guard statement of the first location in figure 3.1 would be translated into:

```
state := Template1None
switch {
        case A > B:
                state = Template1Location2
        case A <= B:
                state = Template1Location3
}
```

Like the second statement, the third statement, the synchronization statement, provides a condition to a transition. In this statement the user can provide a channel on which to send(!) or receive(?) a signal. Upon reaching the transition, a receiver may only proceed if there is a sender sending a signal on the same channel. Likewise a sender on a normal channel needs to wait until there is a receiver on the same channel to receive the signal. This wouldn't be the case if the sender would send a signal on a broadcast channel. However, broadcast channels are outside of the scope of this algorithm. In Google Go the selection of transitions based on synchronization statements can be done by means of a select statement. In this case, the channels used in Uppaal can be translated to unbuffered boolean channels in Google Go and the transition function of a location contains a select statement where either “true” is send or received on a channel according to the Uppaal model. In our example template, figure 3.1, the statement to select the right transition would be:

```
state := Template1None
select {
        case C <- true:
                state = Template1Location2
        case <- C:
                state = Template1Location3
}
```

The last statement that can be included in a transition is the update statement. This statement is evaluated when a transition is triggered. The expression evaluated in the update statement is the only of these four statements which may have side effects which will be saved to the environment. The update statement itself is expressed using the Uppaal C-like syntax. It can be converted to Google Go code using the process as described in section 3.1. If two transitions are synchronized, the update statements are evaluated sequentially, first the update statement of the sender, then the update statement of the receiver. To ensure that only one transition can write the variables at the same time, we have to use a semaphore. Furthermore to facilitate the sequential update statement of synchronized transitions, a unbuffered channel can be used to synchronize the two goroutines. So to accommodate the update statements, the following declarations should be made somewhere in the package:

```
import "sync"
var lock sync.Mutex
seq_update := make(chan bool)
```

For the transition from Location1 to Location2, in our example template shown in figure 3.1, could be accommodated by the following statement:

```
lock.Lock()
D()
seq_update <- true
<- seq_update
Lock.Unlock()
```

Note that if the transition wasn't synchronized, the statements regarding the "seq_update" should be omitted. The receiving transition between Location1 and Location3 in the same model would then be expressed as:

```
<- seq_update
D()
seq_update <- true
```

So as the sender passes a message on the channel after it is done with its update statement, we force that the update statement of the sender is executed before the update statement of the receiver, as the receiver waits for this message. This however only works if the synchronization statements only enables the update statements of the transitions that are triggered, otherwise, the sequence in which the update statements are executed is lost, as the channel by which the processes synchronize could be used by other processes at the same time.

Forming a full transition

Now that we have a way to translate each of the statements, one would expect that finding a general way to convert a whole transition into a transition function would be rather easy. However, combining these statements is no trivial task. The only statement that can easily be combined with any of the other statements is the select statement, as it's just an assignment that can be done at the beginning of the transition, which all other statements can then use. Combining the other statements, we run into various problems. These problems stem from the fact that within the Uppaal model, transitions are atomic operations, while one transition is being executed, no other (parts of) transitions can be executed. But as we have pointed out, in Google Go the transitions will consists of multiple operations and are thus not atomic.

The first problem we encounter is that if an edge has both a guard condition and a synchronization condition both must hold at the same time before a edge can be triggered. If however we first check if a signal has been (or can be) send on a specific channel, once the message is passed, it is removed from the channel and can thus only be checked once. So if we then check the guard statement and it evaluates to "false", we would have to cancel the transition, risking a endless loop, each time canceling the transition tried. So for a solution to this problem we will have to turn it around and check the guard statements before we check the channels. In Google Go, this is syntactically tricky as Google Go has no integrated form of a guarded select statement. However, Russ Cox, a Google employee, proposed a simple solution[17]. Instead of wrapping the channel used in the select structure in an if-statements, which isn't possible in Google Go, we add a new function which only returns the evaluated channel if a boolean expression evaluates to "true". This function can of course be used for channel with various forms of data. For our purposes, we will uses channels with boolean values, the function would then look like this:

```
func when(b bool, c chan bool) chan bool {
        if !b {
                return nil
        }
        return c
}
```

This still leaves us with a problem that when two processes are looking for a transition at the same time, both can be triggered. If, in this case, the update statement of the first transition changes the state in such a way that the the guard statement of the second transition now evaluates to "false" then the second transition should not take place. One might consider a Read/Write-mutual exclusion lock, where there is either one or more readers or one writer, but this would not solve the problem. In this case the second process still has the possibility to read before the first process is able to write. Using Uppaal as an example is also of little use. We might simulate the Uppaal transitioning system in which no time passes while the transitions are being evaluated. This would however require us to pause all goroutines while a transition is taking place, which would be terribly inefficient. A better solution would be to simulate the same effect by only executing the first applicable transition (or synchronized transition) and "canceling" all other transitions in progress, so these statements are evaluated again. To accomplish this canceling effect we create a governing structure that works similar to a Uppaal broadcasting channel. Upon entering the transition function, the transition can request a cancellation channel. Once one transition has been executed, the cancellation channels will close and all other transition functions will have to reset themselves. The cancellation structure will look like this:

```
import "sync"

type Cancellor struct {
        channels []chan bool
        mutex    sync.Mutex
}

func (c *Cancellor) GetChannel() <-chan bool {
        b := make(chan bool, 1)
        c.mutex.Lock()
        c.channels = append(c.channels, b)
        c.mutex.Unlock()
        return b
}

func (c *Cancellor) cancellation() {
        for _, b := range c.channels {
                close(b)
        }
}

func (c *Cancellor) CallUpdate() {
        c.mutex.Lock()
        c.cancellation()

        c.channels = make([]chan bool, 0)
        c.mutex.Unlock()
}
```

Using this implementation we can cancel all transitions that needs to be reevaluated. The final piece of the puzzle is getting the synchronization of the transition of two processes right. As previously stated, when a synchronized transition is applied, first the update statement of the sender is executed, then the update statement of the receiving transition will be applied. To ensure this order we need to make use of another channel. However the only available channel (used for the original synchronization) might also be listened to by other processes. Therefore, instead of passing a simple boolean statements, the synchronization channels will signal by passing a boolean channel themselves. So we are using channels that send and receive boolean channels, then these channels can be used to synchronize the update statement. To achieve synchronization, the receiver will wait for a message on the passed channel by the sender. If the message is positive, the update statement will we executed and a message will be passed back. Otherwise the receiver resets the transition.

Putting all these puzzle pieces together we can then finally create a function that combines all four statements. Using our "when" function, the cancellation governor and the channel that passes boolean channels, we can form a full transition in Google Go. The transition function of Location1 from our example template, as shown in figure 3.1, will look like this:

```
var state State = Template1None
```

```
for state == Template1None {
        var i int = (5-3) + rand.Intn(5-3)

        cancel := cancellor.GetChannel()
        c := make(chan bool)
        select {
        case when(A > B, C) <- c:
                Lock.Lock()
                select {
                case <- cancel:
                        c <- false
                        state = Template1None
                default:
                        D()
                        state = Template1Location2
                        c <- true
                        <-c
                        cancellor.callUpdate()
                }
                Lock.Unlock()
        case x:= <- when(A <= B, C):
                execute := <- x
                if execute {
                        D()
                        state = Template1Location3
                        x <- true
                } else {
                        state = Template1None
                }
        case <- cancel:
                state = Template1None
        }
}
return state
```

Note the difference between the cases of the sending and the receiving edge in the select structure. We also note that if a location has more possible transitions, we need to append these as new cases to the top level select structure. Translations of Uppaal select statements can be appended at the beginning of the for loop.

It will often occur that one or more of the four statements is not present on an edge. In this case the structure of the case will change depending on which statements are missing. If the select statement is missing, it can be omitted without any problems. The same applies to the update statement, we can easily omit the update statement. When a transition is missing the guard statement, we don't need the "when" function, instead we can just use the channel as mentioned in the synchronization statement. If however the synchronization statement is omitted, a lot will change. If a location contains one or more outgoing transition without a synchronization statement, we declare a new boolean channel in the beginning of the transition function and immediately close the channel. For the case in the select structure we will then read from this channel, as reading from a closed channels requires no synchronization. The contents of the case itself will be like contents of the sender in a binary synchronization, except for the synchronization itself. For example the left transition from our example template, as shown in figure 3.1, without synchronization and guard statement will be represented by the following transition function:

```
var state State = Template1None
pass := make(chan bool)
close(pass)
for state == Template1None {
        var i int = (5-3) + rand.Intn(5-3)

        cancel := cancellor.GetChannel()
        c := make(chan bool)
        select {
        case <- pass:
                Lock.Lock()
                select {
                case <- cancel:
                        state = Template1None
                default:
                        D()
                        state = Template1Location2
                        cancellor.callUpdate()
                }
                Lock.Unlock()
        case <- cancel:
                state = Template1None
        }
}
return state
```

3.3 Converting the System Declarations

To start all the instantiations of templates as goroutines in Google Go, we need to convert the system declaration. This piece of the Uppaal Model states which processes are started. The only statement in the system declarations that will be converted and put into the main function is the “system” statement. If however variables are declared in the system declaration and these are used within the system statement, these are also converted using the rules given in section 3.1.1. We evaluate this statement to start the different goroutines. If we for example take the following system definition:

```
int v, u;
int x = 5;
const struct { int a, b, c; } data[2] = { { 1, 2, 3 }, { 4, 5, 6 } };

Q(int &x, const int i) = P(x, data[i].a, data[i].b, 2 * data[i].c);
Q1 = Q(v, 0);
Q2 = Q(u, 1);
```

```
system Q1, Q2,  R(x, 1);
```

In this case, the first three statements can be converted using the default conversion methods, as discussed in section 3.1.1. The other statement however are template initiations. To convert these to Google Go goroutines we need to need a different set of rules. The idea is quite simple, for each of the the template initiation we will track it to where the actual declaration of the process. When we have found the actual declaration, we can convert this declaration to a function call to the main function of the converted template, with the right parameters.

In our example the simplest case is "R(x, 1)". We know R() is a known template (as we would have converted it in the previous step of our algorithm), therefore we we only need to add a similar function call to our Go implementation: "go R(x, 1)".

Q1 and Q2 are harder cases, these will track back to a call to a template named "Q". Other than in the previous case, we have not converted a template named "Q". However, we do find a rename statement. We will use this to generate the actual initiations of the templates. By replacing the parameter of the call of the rename statement in the actual statement, we can find the next declaration. In this case there is only one rename statement, but this process could be iterated. By using the declarations of the rename statements, we end up with the actual template which should be initiated and by replacing the parameters of the rename statements by the values by which the rename statement is called we resolve the statement to a declaration that we can use. As the template "P" is a known template, we find the calls which we again can use to start a goroutine: "go P(&v, data[0].a, data[0].b, 2 * data[0].c)" and "go P(&u, data[1].a, data[1].b, 2 * data[1].c)". Note that the parameters in the call are converted in the same way as with function calls.

At the end of the main function we add an empty select statement. By doing this we lock the main thread. This ensures that all goroutines can run until they are deadlocked themselves, which ends the program. Because if the main thread is finished, the whole program would quit. This behavior is similar to the behavior of Uppaal, were a trace only ends when it becomes deadlocked. The resulting source code starting the processes would be:

```
package main

func main() {
        var v, u int
        var x int = 5
        var data [2]struct {
                a, b, c int
        } = [2]struct {
                a, b, c int
        }{struct {
                a, b, c int
        }{1, 2, 3}, struct {
```

```
                        a, b, c int
        }{4, 5, 6}}

        go R(x, 1)
        go P(&v, data[0].a, data[0].b, 2*data[0].c)
        go P(&u, data[1].a, data[1].b, 2*data[1].c)

        select {}
}
```

Chapter 4

Example: Generating a web-crawler

We will demonstrate the usefulness of the given method by showing an example case where the given method can be of value. The Uppaal model that we are converting is a model representing a simple web crawler.

The model, as shown in figures 4.1, 4.2 and 4.3, consists of three components which are regularly found in a web crawler. The Core handles all correspondences between the downloaders and the crawlers, it dispenses the request to the downloaders and the responses to the crawlers. The results of the other components are also returned to the core. This makes the core the governor of the other components. The downloaders have a simple task, when they receive a request, they will try to acquire the information requested and return the information as a response. One of the crawlers will then process the response and hopefully return some useful information. Note that there should be only one core, but there can be as much downloaders and crawlers as one might deem necessary.

In our model we assume that the crawlers have a unlimited supply of requests and that if there aren't any request left, it has to provide a new request to prevent the model from falling into a deadlock. Because this is a Uppaal model we can then proof that there is no possibility of a deadlock occurring within the execution of the model. Of course we can also proof other conditions. For example, we can proof that once the process is started and all downloaders and crawlers are available that there is either at least one response or request available. We can also proof that if the "Core" is in the "Output" state, there is actual new information, notice that only actual results have the boolean value of "useful" set to "true". Each of these conditions have been proven for one downloader and one crawler, as shown in figure 4.4.

Without any problems, we can use our method to generate a code framework for our simple web crawler. The generated code can be found in appendix A. We can see that

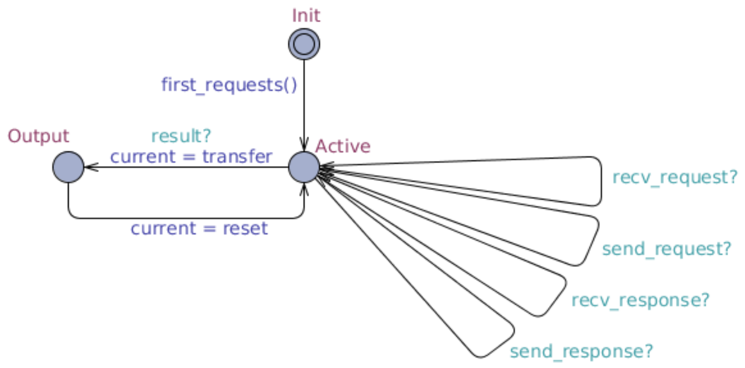


Figure 4.1: Example model: The “Core” template, data manager and semaphore.

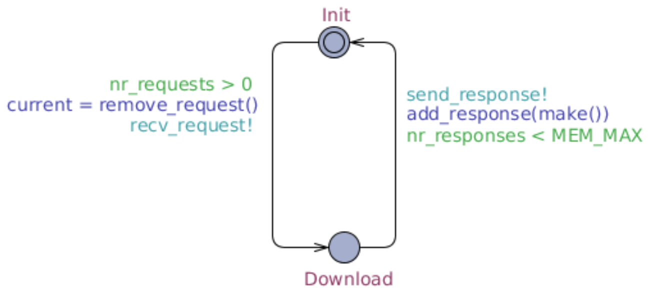


Figure 4.2: Example model: The “Downloader” template, request handler

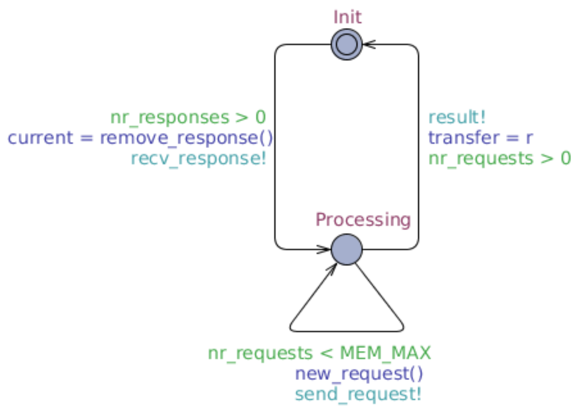
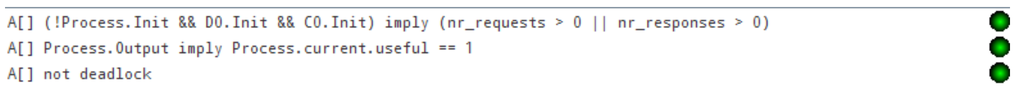


Figure 4.3: Example model: The “Crawler” template, data processor



```
A[] (!Process.Init && D0.Init && C0.Init) imply (nr_requests > 0 || nr_responses > 0)   ●
A[] Process.Output imply Process.current.useful == 1                                     ●
A[] not deadlock                                                                         ●
```

Figure 4.4: Example model: Proven conditions

the structure for the web crawler is complete, however functionality should be added before the crawler will provide us with actual results. As our model contained only very simple types for requests, responses and results, we would most likely extend these. A request object should most likely be extended to contain a URI of the source that is requested. The response object should then contain the content retrieved, which is most likely hypertext. A result object could for example contain the title of the page. Because the types have changed, we would have to change the functions which use these types. In our case this would be "crawlerNew_request" and "downloaderMake", each of these functions need to fill the new information in the objects in a way that is thread-safe and in such a way that the information previously known stays the same. For example "crawlerNew_request" could read a new URI from a thread-safe storage mechanism and "downloaderMake" could add the contents of the URI, which can already be acquired in the "downloaderDownload" state. The code framework will then be functional when the download, processing and output functions, representing the states of the different templates, are functional. Each of these functions can be easily created using Google Go code. Notice that these functions still need to be thread-safe.

So by extending three types and creating three functions, the generated code framework becomes an operational web crawler. Even better, if we can assume that the method is correct, we know that certain conditions hold within the web crawler if we can prove that the additions to the model haven't changed the structure. Making this method at least an quick way to acquire a working, and assumingly correct, program from the Uppaal model.

Chapter 5

Validity

One of the most important aspects of our method is the validity of the resulting generated framework. The validity of these frameworks can be split into two parts, first the validity of the code conversion for the functions and other declarations and the validity of concurrent structure in Google Go representing the templates in Uppaal and their instantiations. To formally proof that both the generated code framework and the Uppaal model are semantically similar, we would need formal semantic models for both Uppaal and Google Go. As these models are not readily available as of yet, a formal proof can't be given within this paper. However, using informal lines of reasoning we can argue the validity of the model.

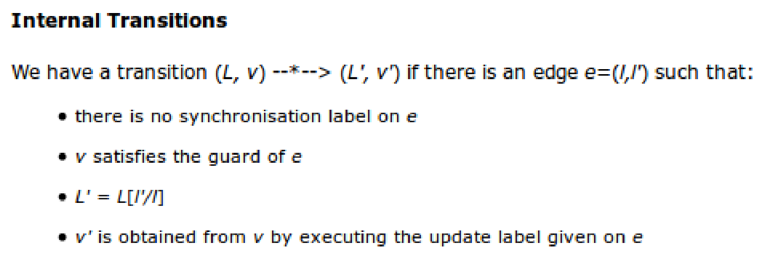
The main argument for the validity of the general conversion of the code from Uppaal to Google, is the similarity between the two. As both programming languages are strongly related to C, the syntax of Uppaal statements is usually similar to the Google Go syntax. For all statements that are so similar we can assume that the generated code is semantically the same as the code in Uppaal, as both Uppaal and Google Go use a straight forward memory model and statements in both languages are evaluated in a sequential order. There are however some differences between the two, other than in which order a declaration is stated, these consist of structures that are not available in Google Go. These structures can be split into two categories.

The first category contains the statements containing types special to Uppaal. Other then the types that were explicitly outside the scope of our algorithm, these are scalars and range types. Both are a special kind of integer that restricts the usage of the variable in such a way that there are less possible states for Uppaal to check. Scalars are integers with a restriction in regard to the possible operations that can be performed on the variable. Because operations that were prohibited in Uppaal, these operations will still not be performed in the generated Google Go code and because the other operations, that can be performed on scalars, are semantically similar to the same operation on a integer, the model is semantically equal when these variables are converted to integers. When using range types, the same operations can be used as with an integer. There is however a restriction in which values a these variables can take. However the value restrictions are not enforced by the operations on a variable, the user must add the restrictions himself. We can therefore assume that a range type is semantically similar to an integer.

The second category of statements are the statements that cannot be expressed in the same way in Google Go. In section 3.1 we find various statements in which this is the case. For example the "do-while" statement is not part of the Google Go syntax. Although we cannot express these statements in Google Go, we can express them in multiple statements in Uppaal which have the same result, which can be easily converted. However proving that these rewritten statements have the same effect as the original statement would again require a semantical proof. As there are only a few of these rewritten statements and because the rewrites usually only involve the splitting of a statement into multiple statements, we trust that the reader understands the similarity of the rewritten statement and the original statement.

To argue the validity of the concurrent structure of the generated framework in Google Go, we need to argue that in the framework the same processes are started as in the Uppaal model, that once these processes are started the same state changes are possible in the model and our generated code and that the program ends at the same time. The first step in the model is the initiation the various templates. As each of the variables in the system statement in the Uppaal model is evaluated to it's declaration in our Generated code, both the Uppaal model and the Generated code start the same (number of) instantiations of the templates using the same parameter. In our model the program would stop when deadlock is achieved, meaning that there is no possible transition which can be taken. Like the model, the generated program stops when there is no active process, meaning that all processes are waiting for a transition. This is achieved because the main thread in the generated code will lock after starting the instantiations of the templates and the program will automatically stop if none of the processes are active.

Next we need to argue that the generated code can reach all states which can be reached in the Uppaal model and no other states. In our generated code the active location of a template instantiation is represented by a variable called "state" in the main function of a template, the state variables of all goroutines form the location vector. The state of the program contains both all variables in the global space and in the started goroutines that were generated by the algorithm. In our algorithm we assure that the state can only be changed using a transition. Which means that if our transitions have the same semantics as the transitions in the Uppaal model, then the semantics of our generated framework is similar to the semantics of the Uppaal model. In the Uppaal semantics [8] there is a distinction between three different kind of transitions: internal transitions, binary synchronizations and broadcast synchronizations. As broadcast synchronizations aren't part of our algorithm yet, we will discuss only the first two kind of



Internal Transitions

We have a transition (*L*, *v*) --*--> (*L'*, *v'*) if there is an edge *e*=(*l*,*l'*) such that:

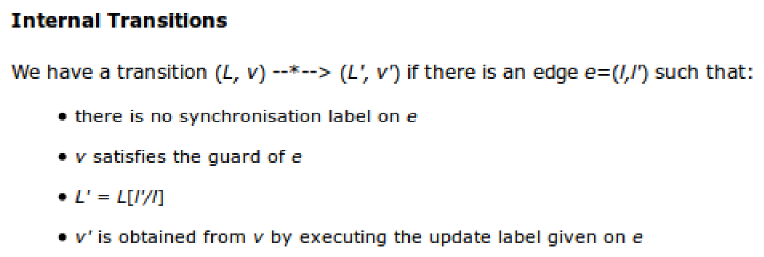
- there is no synchronisation label on *e*
- *v* satisfies the guard of *e*
- *L'* = *L*[*l'*/*l*]
- *v'* is obtained from *v* by executing the update label given on *e*

Figure 5.1: The requirements for an internal transition in Uppaal.

synchronizations.

According to the Uppaal semantics [8], a internal transition in Uppaal within the scope of our algorithm satisfies the requirements as given in figure 5.1. These conditions also hold in our generated model. The first requirement holds because when there is no synchronization statement we might still use a closed channel in our select statement, but as the channel is closed, the application can just proceed and thus doesn't have actual synchronization. The guard statement of the transition also holds within the current state, because if the guard statement doesn't hold within the current state, either the transitions is not triggered or the transition is canceled by the transition that changed the state. The third condition states that in the location vector the new location replaces the old location. This condition hold as once the new state is known, the new active location is returned to the main function of the process replacing the old active state. Like in the Uppaal model, the new state is the result of applying the update function to the old state. This is guaranteed because our framework only let's the update function change the values of the variables and the update function is only executed if a transition is triggered and not canceled. Note that the no other transition can perform a update statement at the same time due to the used semaphore and that once executed other transitions that were triggered will be canceled, thus making sure only the update statement of the executed transition will be evaluated.

Like the conditions for the internal transitions, the conditions for the binary synchronization, as shown in figure 5.2, hold. The first condition holds, if a binary synchronization is triggered the sender in our code will send a boolean channel to the receiver though a channel, this is a binary channel as in Google Go also only one sender and one receiver is triggered. Like the internal transition, the guard condition of both edges hold on the state. If this isn't the case, both transitions will not be triggered or they will both be canceled. If the transition is complete the transition functions of both processes will return the new active location replacing the current active location and satisfying the third condition. Like internal transitions only the update statement of the first triggered transition will be executed and it is the only statement which impacts the state. Because the transition is a synchronization, both the update statement on the


Internal Transitions

We have a transition (*L*, *v*) --*--> (*L*', *v*') if there is an edge *e*=(*l*,*l*') such that:

- there is no synchronisation label on *e*
- *v* satisfies the guard of *e*
- *L*' = *L*[*l*'/*l*]
- *v*' is obtained from *v* by executing the update label given on *e*

Figure 5.2: The requirements for an binary synchronization in Uppaal.

sending and the receiving edge will be executed. As stated in the last requirement we enforce the order in which the update statements are executed using the channel send upon first synchronization, executing the update statement of the sender first, then the update statement of the receiver. Hereby we satisfy the last condition.

As we argued both the validity of the general conversion and the validity of the concurrent structure as a whole, we conclude that, in absence of semantical proof, our method is, in all likeliness, valid.

Chapter 6

Related Work

In the rise of Model Driven Development, there is a lot of related work on both the generation of source code and test cases from models. Most of these methods focus on generation of source code or test cases from UML models, as these models are accepted to be a standard for designing new systems. For example UML state diagrams can be used to generated test cases [22]. A paper that hits closer to home is a paper discussing an UML tool to generate a simulation program [15]. This enables the user to use the generated simulation to estimate the performance of the system specified in UML. To generate such a simulation requires various of UML diagrams and even some textual specifications. The simulation itself however isn't an actual application and thus differs from our approach. Furthermore we notice that generation based on UML tends to discards possible approaches using concurrency.

When we look at research that includes the modeling of concurrency within (a piece of) software, we see that most research is not focused on Model Based Development, but rather the other way around. There is much research focused on the generation of a concurrency model from source code. For example research has been done to verify the systems of Lego Mindstorms^{TM}[21]. In this research a model in Uppaal has been generated for the various control programs and the scheduler of the Lego Mindstorms^{TM}.

Although there isn't yet another paper on generating source code from Uppaal, there are two papers that relate to the subject. The first paper presents a technique to focusing on the automatic generation of real-time conformance test cases from timed automata specifications in Uppaal [20]. This paper does not dive into the generation of actual source code, but shows an insight into the abstract layers of Uppaal. Very similar to our project is the TIMES tool[14], like our method it's able to eventually generate source code, but unlike our method it is not based on Uppaal models, although the model is quite similar, and is focused on scheduling problems instead of a general approach in generating source code from the model.

This paper is among one of the first scientific works that includes the Google Go programming languages. Notable for future work is a thesis by Sören Jeserich [24], which is a first attempt to provide formal semantics to Google Go. But although little scientific work has been published on concurrency in Google Go, Google itself however provides us with a lot of explanation on best practices concerning concurrency. For example blog posts were made concerning advanced concurrency structures, like pipelines and cancellation [13] and contexts [12], in these blog posts a Google employee explains how to handle various structures one might find when using concurrency. However there are also talks on concurrency, like the talk explaining the important difference between concurrency and parallelism [18]. These bests practices and the language specification of the Google Go programming language [10] are the foundation of the concurrency principles used in the generated code.

Chapter 7

Conclusions and Future Work

In this thesis, we have presented a new technique for generating a Google Go code frameworks for a subset of Uppaal models. A framework generated by this method contains the concurrent base of a piece of software and behaves in accordance to the Uppaal model. Using an example we have demonstrated the use of the proposed method and argued it's use. Trough an informal argument we discussed the validity of the generated code by the method. However, not all Uppaal models are covered by this method and there is no definite proof of the validity of the method.

Therefore future work on this approach would most likely be focused on these aspects. Using semantic models of Uppaal and Google Go, one could formally prove the semantic equality between the generated code and the Uppaal models. One might also consider performing a case study on a larger scale and demonstrate the use of this approach. Another possibility is to extend the method to cover a broader scope of Uppaal models. One might cover time constructs (clocks and time restrictions) using the "time" package in Google Go. At the moment there is no construct in Google Go to capture the functionality of broadcast channels, however one might consider a similar structure as used in the cancel governor. Finally one might extend the method to take priorities, of both channels and processes, into account. However, several other related subjects might also be interesting for research. One might for example consider the translation from Google Go source code to a Uppaal model or the translation from and to other model checkers.

Bibliography

[1] The go programming language. http://golang.org/.

[2] The go programming language - frequently asked questions (faq). http://golang.org/doc/faq.

[3] Go talks. http://talks.golang.org/.

[4] Package sync. http://golang.org/pkg/sync/.

[5] A tour of go. http://tour.golang.org/.

[6] Uppaal. http://www.uppaal.org/.

[7] The go memory model. http://golang.org/ref/mem, 2012.

[8] Uppaal language referance. http://www.uppaal.com /index.php?sida=217&rubrik=101#Declarations, 2012.

[9] Go 1.2 release notes. http://golang.org/doc/go1.2#stack_size, 2013.

[10] The go programming language specification. http://golang.org/ref/spec, 2013.

[11] What is the most frequent concurrency issue you've encountered in java? http://stackoverflow.com/questions/461896 /what-is-the-most-frequent-concurrency-issue-youve-encountered-in-java, 2013.

[12] Sameer Ajmani. Go concurrency patterns: Context. http://blog.golang.org/context, 2014.

[13] Sameer Ajmani. Go concurrency patterns: Pipelines and cancellation. http://blog.golang.org/pipelines, 2014.

[14] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 60–72. Springer, 2004.

[15] LB Arief and Neil A Speirs. A uml tool for an automatic generation of simulation programs. In *Proceedings of the 2nd international workshop on Software and performance*, pages 71–76. ACM, 2000.

[16] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0. vol, 2006.

[17] Russ Cox. Guarded selective waiting. `https://groups.google.com/d/msg/golang-nuts/ChPxr_h8kUM/mntIttBSZDUJ`, 2012.

[18] Andrew Gerrand. Concurrency is not parallelism. `http://blog.golang.org/concurrency-is-not-parallelism`, 2013.

[19] Vyukov D. Gerrand, A. Introducing the go race detector. `http://blog.golang.org/race-detector`, 2013.

[20] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using uppaal. In *Formal Approaches to Software Testing*, pages 114–130. Springer, 2004.

[21] Torsten K Iversen, Kåre J Kristoffersen, Kim Guldstrand Larsen, Morten Laursen, Rune G Madsen, Steffen K Mortensen, Paul Pettersson, and Chris B Thomasen. Model-checking real-time control programs: verifying lego mindstorms tm systems using uppaal. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 147–155. IEEE, 2000.

[22] Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, and Sung-Deok Cha. Test cases generation from uml state diagrams. *IEE Proceedings-Software*, 146(4):187–192, 1999.

[23] Paul E McKenney. Is parallel programming hard, and, if so, what can you do about it? *Linux Technology Center, IBM Beaverton*, 2011.

[24] Sören Jeserich. Modellierung von Google Go im π-Kalkül. Master's thesis, University of Oldenburg, 2012.

[25] Frits Vaandrager. A first introduction to uppaal. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, 18, 2011.

Appendix A

Generated Crawler

A.1 basic_structures.go

```
package main

import "sync"

type Cancellor struct {
        channels []chan bool
        mutex    sync.Mutex
}

func (c *Cancellor) GetChannel() <-chan bool {
        b := make(chan bool, 1)
        c.mutex.Lock()
        c.channels = append(c.channels, b)
        c.mutex.Unlock()
        return b
}

func (c *Cancellor) cancellation() {
        for _, b := range c.channels {
                close(b)
        }
}

func (c *Cancellor) CallUpdate() {
        c.mutex.Lock()
        c.cancellation()

        c.channels = make([]chan bool, 0)
        c.mutex.Unlock()
}

func when(b bool, c chan chan bool) chan chan bool {
        if !b {
                return nil
        }
        return c
}
```

A.2 main.go

```
package main

import "sync"

type State int

type Request struct {
        id int
}

type Response struct {
        id      int
        origin Request
}

type Result struct {
        useful bool
}

var nr_requests int = 0
var nr_responses int = 0

const MEM_MAX int = 5

var send_request chan chan bool
var recv_request chan chan bool
var send_response chan chan bool
var recv_response chan chan bool
var result chan chan bool

var transfer Result

var requests [MEM_MAX]Request
var responses [MEM_MAX]Response

func add_request(a Request) {
        requests[nr_requests] = a
        nr_requests++
}

func remove_request() Request {
        var r Request = requests[nr_requests-1]
        nr_requests--
        return r
}

func add_response(a Response) {
        responses[nr_responses] = a
        nr_responses++
}

func remove_response() Response {
        var r Response = responses[nr_responses-1]
        nr_responses--
        return r
}

var cancellor Cancellor = Cancellor{channels: make([]chan bool, 0)}
var lock sync.Mutex

func main() {
        send_request = make(chan chan bool)
        recv_request = make(chan chan bool)
        send_response = make(chan chan bool)
        recv_response = make(chan chan bool)
        result = make(chan chan bool)

        go Core()
        go Crawler()
        go Crawler()
        go Crawler()
        go Downloader()
        go Downloader()

        select {}
}
```

A.3 core.go

```
package main

const (
        CoreNone State = iota
        CoreInit
        CoreActive
        CoreOutput
)

func coreFirst_requests() {
        var i int
        for i = 1; i <= 3; i++ {
                var a Request = Request{i}
                add_request(a)
        }
}

func Core() {
        var current Result
        var reset Result = Result{false}
        state := CoreInit
        for {
                switch state {
                case CoreInit:
                        coreInit()
                        state = coreInitTransitions()
                case CoreActive:
                        coreActive()
                        state = coreActiveTransitions(&current)
                case CoreOutput:
                        coreOutput()
                        coreOutputTransitions(&current, &reset)
                }
        }
}

func coreInit() {

}

func coreInitTransitions() State {
        var state State = CoreNone
        pass := make(chan bool)
        close(pass)
        for state == CoreNone {
                switch {
                <- pass:
                        lock.Lock()
                        coreFirst_requests()
                        state = CoreActive
                        cancellor.CallUpdate()
                        lock.Unlock()
                }
        }
        return state
}

func coreActive() {

}

func coreActiveTransitions(current *Result) State {
        var state State = CoreNone
        for state == CoreNone {
                cancel := cancellor.GetChannel()
                select {
                case x := <-recv_request:
                        execute := <-x
                        if execute {
                                state = CoreActive
                                x <- true
                        } else {
                                state = CoreNone
                        }
                case x := <-send_request:
                        execute := <-x
                        if execute {
                                state = CoreActive
                                x <- true
                        } else {
                                state = CoreNone
                        }
                case x := <-recv_response:
                        execute := <-x
                        if execute {
                                state = CoreActive
                                x <- true
                        } else {
                                state = CoreNone
                        }
                case x := <-send_response:
                        execute := <-x
                        if execute {
                                state = CoreActive
                                x <- true
                        } else {
                                state = CoreNone
                        }
                case x := <-result:
                        execute := <-x
                        if execute {
                                *current = transfer
                                state = CoreActive
                                x <- true
                        } else {
                                state = CoreNone
                        }
                case <-cancel:
                        state = CoreNone
                }
        }
        return state
}

func coreOutput() {

}

func coreOutputTransitions(current *Result, reset *Result) State {
        var state State = CoreNone
        pass := make(chan bool)
        close(pass)
        for state == CoreNone {
                select {
                <- pass:
                        lock.Lock()
                        *current = *reset
                        state = CoreActive
                        cancellor.CallUpdate()
                        lock.Unlock()
                }
        }
        return state
}
```

A.4 crawler.go

```
package main

const (
        CrawlerNone State = iota
        CrawlerInit
        CrawlerProcessing
)

func crawlerNew_request() {
        var stop bool = false
        var i int = 1
        for i <= MEM_MAX && !stop {
                var j int = 0
                for j < nr_requests && requests[j] != (Request{i}) {
                        j++
                }
                if j == nr_requests {
                        stop = true
                } else {
                        i++
                }
        }
        add_request(Request{i})
}

func Crawler() {
        var current Response
        var r Result = Result{true}

        state := CrawlerInit
        for {
                switch state {
                case CrawlerInit:
                        crawlerInit()
                        state = crawlerInitTransitions(&current)
                case CrawlerProcessing:
                        crawlerProcessing()
                        state = crawlerProcessingTransitions(&r)
                }
        }
}
```

```
func crawlerInit() {

}

func crawlerInitTransitions(current *Response) State {
        var state State = CrawlerNone
        for state == CrawlerNone {
                cancel := cancellor.GetChannel()
                c := make(chan bool)
                select {
                case when(nr_responses > 0, recv_response) <- c:
                        lock.Lock()
                        select {
                        case <-cancel:
                                c <- false
                                state = CrawlerNone
                        default:
                                *current = remove_response()
                                state = CrawlerProcessing
                                c <- true
                                <-c
                                cancellor.CallUpdate()
                        }
                        lock.Unlock()
                case <-cancel:
                        state = CrawlerNone
                }
        }
        return state
}

func crawlerProcessing() {

}

func crawlerProcessingTransitions(r *Result) State {
        var state State = CrawlerNone
        for state == CrawlerNone {
                cancel := cancellor.GetChannel()
                c := make(chan bool)
                select {
                case when(nr_requests < MEM_MAX, send_request) <- c:
                        lock.Lock()
                        select {
                        case <-cancel:
                                c <- false
                                state = CrawlerNone
                        default:
                                crawlerNew_request()
                                state = CrawlerInit
                                c <- true
                                <-c
                                cancellor.CallUpdate()
                        }
                        lock.Unlock()
                case when(nr_requests > 0, result) <- c:
                        lock.Lock()
                        select {
                        case <-cancel:
                                c <- false
                                state = CrawlerNone
                        default:
                                transfer = *r
                                state = CrawlerProcessing
                                c <- true
                                <-c
                                cancellor.CallUpdate()
                        }
                        lock.Unlock()
                case <-cancel:
                        state = CrawlerNone
                }
        }
        return state
}
```

A.5 downloader.go

```
package main

const (
        DownloaderNone State = iota
        DownloaderInit
        DownloaderDownload
)

func downloaderMake(current *Request) Response {
        var stop bool = false
        var i int = 1
        var r Response = Response{i, *current}
        for i <= MEM_MAX && !stop {
                var j int = 0
                for j < nr_responses && responses[j].id != i {
                        j++
                }
                if j == nr_responses {
                        stop = true
                } else {
                        i++
                }
        }
        r.id = i
        return r
}

func Downloader() {
	var current Request

	state := DownloaderInit
	for {
		switch state {
		case DownloaderInit:
			downloaderInit()
			state = downloaderInitTransitions(&current)
		case DownloaderDownload:
			downloaderDownload()
			state = downloaderDownloadTransitions(&current)
		}
	}
}

func downloaderInit() {

}

func downloaderInitTransitions(current *Request) State {
	var state State = DownloaderNone
	for state == DownloaderNone {
		cancel := cancellor.GetChannel()
		c := make(chan bool)
		select {
		case when(nr_requests > 0, recv_request) <- c:
			lock.Lock()
			select {
			case <-cancel:
				c <- false
				state = DownloaderNone
			default:
				*current = remove_request()
				state = DownloaderDownload
				c <- true
				<-c
				cancellor.CallUpdate()
			}
			lock.Unlock()
		case <-cancel:
			state = DownloaderNone
		}
	}
	return state
}

func downloaderDownload() {

}

func downloaderDownloadTransitions(current *Request) State {
	var state State = DownloaderNone
	for state == DownloaderNone {
```

```
                cancel := cancellor.GetChannel()
                c := make(chan bool)
                select {
                case when(nr_requests > 0, recv_request) <- c:
                        lock.Lock()
                        select {
                        case <-cancel:
                                c <- false
                                state = DownloaderNone
                        default:
                                add_response(downloaderMake(current))
                                state = DownloaderInit
                                c <- true
                                <-c
                                cancellor.CallUpdate()
                        }
                        lock.Unlock()
                case <-cancel:
                        state = DownloaderNone
                }
        }
        return state
}
```