Verifying Properties on a Program Using Static Analysis and Model Checking

Aymeric Fages
(supervisor: Adrien Pommellet)

Technical Report n^o2020, June 2020
revision f7227a02

In this paper, we present different solutions to transform a program into a finite state system. Then we introduce our functional pipeline that allows us the say whether a program verifies a LTL formula. We describe all the aspect of this pipeline from LLVM tools and a C++ algorithm for model abstraction to the Spot library.
In a first part of this report, we explain the type of finite graph that we want to obtain in order to apply model checking. In a second part, we define important models such as pushdown systems or Büchi automata. In a third part, we explain how to find an under-approximation and over-approximation of a pushdown system as a finite system. Then finally, we show how those theoretical solutions were implemented into an actual functional pipeline.

Dans ce rapport, nous présentons les différentes solutions pour transformer un programme en un système d'états finis. Puis nous présentons notre programme fonctionnel qui nous permet de dire si un programme en C vérifie ou non une formule LTL. Nous décrivons tous les aspect de ce programme, que ce soit de l'utilisation de LLVM, d'un algorithme C++ d'abstraction de modèles ou encore de la bibliothèque Spot.
Dans une premiere partie, nous expliquons quel type de graphe fini nous voulons obtenir comme modèle du programme. Dans une deuxième partie, nous définissons des notions importantes de la verification de modèles, tels que les systèmes à pile ou les automates de Büchi. Dans une troisieme partie, nous expliquons comment trouver une sous-approximation et sur-approximation d'un système de poussée vers un systeme d'états finis. Pour finir, nous montrons comment ces solutions théoriques sont implementées en un vrai programme utilisable.

Keywords
Spot, Control Flow Graph, Model Checking, Static Analysis, LTL

Laboratoire de Recherche et Développement de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
afages@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

1.1 The Use of Model Checking

There are several reasons to use Model checking on programs. For instance, programming errors can have huge consequences when they happen on very critic programs such as rocket guidance programs or nuclear power plant cooler systems.

1.2 Static Over Dynamic Analysis

Dynamic analysis consists in choosing a set of inputs for a program, launch the program with the given inputs, then verifying that the output is conform to the expected behaviour of the program. When it comes to malware analysis, dynamic analysis usually has to be performed under a virtual environment to avoid infection. However, static analysis is not about emulating the program but it is about analysing the source code of it. Static analysis allow us to have an exhaustive analysis over the program without relying on the quality of test inputs.

1.3 How Model Checking Works

Model checking is about checking if a program verifies a property. In most cases, this simple problem is very hard to give an answer to. For example, let the property be "The program will stop". It is proven that no Turing Machine can tell if a program will eventually stop or not. The idea of model checking is to then transform the program into a model of the program, and express the property through a formula.

$$\begin{tikzcd}
\boxed{\text{Program}} \arrow[r, "\text{Verifies}"] \arrow[d, dotted] & \boxed{\text{Property}} \arrow[d, dotted] \\
\boxed{\text{Model}} \arrow[r, "\models"] & \boxed{\text{Formula}}
\end{tikzcd}$$

A very useful tool for model checking is the Spot library Duret-Lutz and Poitrenaud (2004) which allows us to tell if a finite graph or automaton of our program verifies a LTL formula. LTL formulas are one of the most classic formula used in model checking and introduced by Pnueli (1977). LTL express properties about the program for an infinite execution of it. Using a finite graph as model of the program is very classic as well in current state of the art model checking. That is why most of this report will be focusing on finding a way to transform a program into a finite graph which will be an approximation of the program.

1.4 Paper Outline

The purpose of Chapter 2 is to explain the kind of finite graph we want as a model for model checking. In Chapter 3, we will introduce important definitions on finite state system, Büchi automata, pushdown systems and LTL formulas. We introduce in Chapter 4 a way to approximate a pushdown system and its induced context-free language into a finite state system and its regular language. Chapter 5 explains how solutions presented in Chapter 2 and 4 are implemented as part as a program pipeline that directly checks if a program verifies a formula. Finally, Chapter 6 concludes this report by talking about possible evolutions for the project.

Chapter 2

Graph Representation of the Program

We previously said that the most common model of the program used to verify formulas is a finite graph. Finite graph represent a regular language whereas programs represent a possibly much more complex language which is a Turing-machine language. Thus, we need a way to approximate a program into a regular language. To do so, we shall have a precise idea of the graphs that we want to obtain, and of the ways to process it from a program source code.

Choosing the right graph representation firstly depends on what kind of fact about the program we want to be able to describe.

A first idea would be to describe the exact value of variables at any time in the program. Have such an information about the program would come very handy as in a lot of programs we want to avoid variable to have certain values. But getting a finite graph containing all possible variable values would mean storing all the possible configurations of the program's memory, which would be ridiculously large. So this is not possible to achieve.

Another idea is to describe ranges of variable values in specific domains of the program. Have such a piece of information would be useful as well, and since we only keep information on a range of value (e.g variable a is greater than 2) and not on all possible values, that would be something we could possibly achieve. This is something that has been done in other papers such as BABENYSHEV and RYBAKOV (2016).

One last possibility is to check properties on function calls. Model checking has been used a lot in order to check if a program is a malware for security purposes. Being able to check facts about API calls could greatly help malware detection. This is what we will most focus on as all our graph will contain properties about function calls.

2.1 Callgraphs

So we need a finite graph of our program that describes function calls. Intuitively, the first idea that comes to mind is to use callgraphs

Let a callgraph of a program $\mathscr{G} = (V, E)$ where V is a set of functions of a program and E a set of edges such that $\forall e \in E, e = (f_1, f_2) \in V^2$, f_1 calls f_2.

Callgraphs are quite popular graphs that are used to see which functions are called by a specific function. They are also very useful to detect functions that are never used, or to have a vague idea of a program's structure. Here is a simple example of the callgraph we get from LLVM Lattner and Adve (2004) for a C or C++ program.

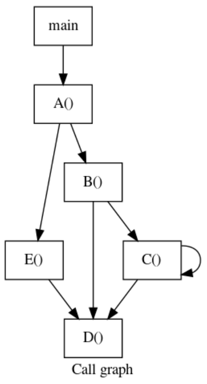

Figure 2.1: Simple callgraph on a C++ program

Figure 2.1 shows a simple callgraph where it is easy to read that function A calls function B and E, function C calls function D and itself, function D does not call any other function. Now let us have a look at another small C program.

```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    open("number42.txt", O_RDONLY);

    if (argc==42)
        close(3);

    return 0;
}
```

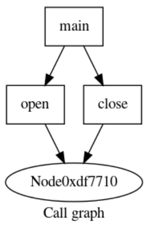

Figure 2.2: Callgraph for a simple C program

On figure 2.2, we can clearly see that main function calls open and close. So now let us imagine that we want to check that our programs always calls close() after calling open(). From the graph it is impossible to tell. Indeed, we cannot tell what function is called first, or if a function is either always called or very rarely called.**We miss too much information about the program's execution flow.**

Although callgraphs are easy to get, they miss too much information to check in most function calls properties.

2.2 Control Flow Graph (CFG)

We criticised callgraphs for missing information about execution flow. LLVM provides another type of graphs called control flow graphs (CFG). These graphs supply for each function an oriented graph where states are label by assembly code lines, and edges represent the decision of the program regarding a jump.

We call CFG of a function a graph $\mathscr{G} = (V, E, i, j)$ where V is a set of different block of assembly code of the function and E is a set of edges $e \in V^2$ such that $\forall e = (v_1, v_2)$, there is a conditional or unconditional jump from block v_1 to block v_2. $i \in V$ the entrance block of the function, $j \in V$ the exit block of the function.

We call simplified CFG of a function a graph $\mathscr{G} = (V, E, i, j)$ where V is a set of different function calls inside the function and E is a set of edges $e \in V^2$ such that $\forall e = (v_1, v_2)$, there is a conditional or unconditional jump between the function calls in v_1 and the function calls in v_2. $i \in V$ the entrance block of the function, $j \in V$ the exit block of the function. Note that functions calls of V can either be other functions of the same program or API calls.

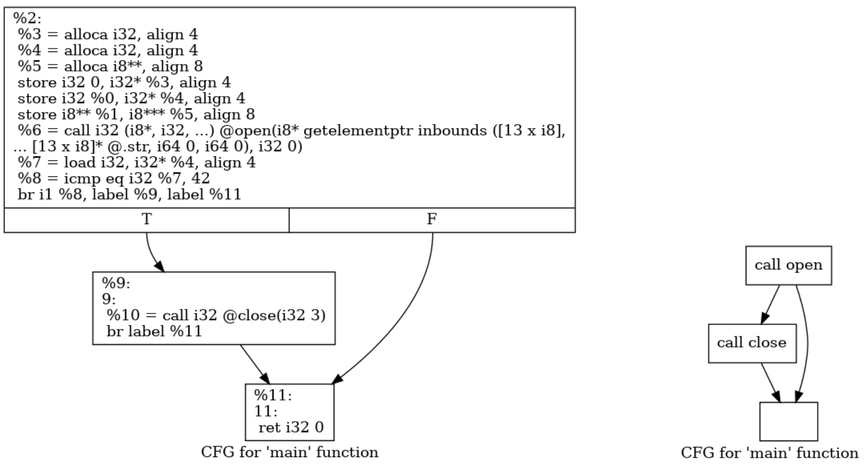
```
%2:
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
 %5 = alloca i8**, align 8
 store i32 0, i32* %3, align 4
 store i32 %0, i32* %4, align 4
 store i8** %1, i8*** %5, align 8
 %6 = call i32 (i8*, i32, ...) @open(i8* getelementptr inbounds ([13 x i8],
... [13 x i8]* @.str, i64 0, i64 0), i32 0)
 %7 = load i32, i32* %4, align 4
 %8 = icmp eq i32 %7, 42
 br i1 %8, label %9, label %11
```
```
%9:
9:
 %10 = call i32 @close(i32 3)
 br label %11
```
```
%11:
11:
 ret i32 0
```

Figure 2.3: CFG for main function from code on figure 2.2

Left image on figure 2.3 shows the LLVM CFG obtained from the program on 2.2. Since we are only interested in function calls, we can easily remove any useless information and assembly code lines and only keep function calls. We then get the graph on the right.

Unlike callgraphs, we can clearly see on 2.3 that there is a path where we call open but not close. As CFGs give us information about the program's control flow, we can conclude on much more properties about the program's function calls. However, CFGs as we get them from LLVM are not interprocedural: we get one graph for each function. We then need a way to get one interprocedural graph from those multiple graphs of each function.

The most intuitive way to make the graph interprocedural is to insert functions into the main graph: for each node, for each function call inside the node, we insert the graph of the function between the call node and the node above it as shown on 2.4.

We will see later that this technique will not work in case of procedure recursive calls, and we will need to find another way to insert it.

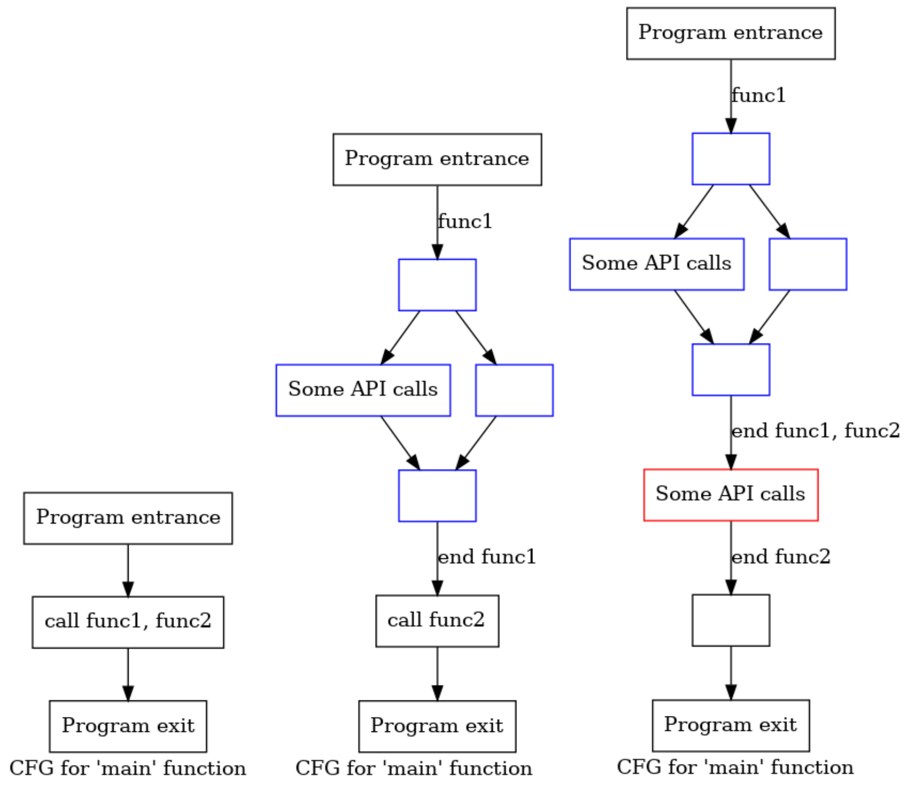

Figure 2.4: Example of function's graph insertion

Chapter 3

Model Checking on Automata

3.1 Finite State System and Büchi Automaton

3.1.1 Finite State System

Definition 3.1.1.1. Let R be a set of all possible execution runs of a program, r one of the run of R such that $r \in R$. t_r is a sequence of all the atomic proposition met during the run r. t_r is the trace associated to the run r.
Moreover, T_R is a set containing all possible traces for all the possible runs $r_i \in R$ of a program such that $t_r \in T_R$.

Definition 3.1.1.2. A finite state system $\mathscr{A}$ is a tuple (Q, Δ, Σ) where Q is a finite set of states, Σ a finite label alphabet, $\Delta = \{(x, l, y) \in Q \times \Sigma \times Q\}$ a finite state of transitions or edges labeled by an element of Σ^*. Intuitively, a finite state system can be seen as a simple edge-labeled oriented finite graph.

Definition 3.1.1.3. Let $Runs(\mathscr{A})$ be the set of all infinite runs of $\mathscr{P}$ starting from its initial state $q_0 \in Q$. Let $Traces(P)$ be the set of all infinite traces of $\mathscr{P}$ starting from its initial state $q_0 \in Q$.

3.1.2 Büchi automaton

Definition 3.1.2.1. A Büchi automaton $\mathscr{B}$ is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite input alphabet, $\delta \in Q \times \Sigma \cup \{\varepsilon\} \times Q$ a set of transitions, $F \subseteq Q$ a set of accepting states, and $q_0 \in Q$ the initial state.

The language $\mathscr{L}(\mathscr{B})$ accepted by $\mathscr{B}$ the set of all infinite sequences w in Σ^ω such that there is an infinite run r of $\mathscr{B}$ with trace w starting in state q_0 that visits accepting states from F infinitely often.

3.2 Pushdown Systems

Pushdown models are a natural model for sequential program with possible recursive procedure calls.

3.2.1 Definition of a Pushdown System

Definition 3.2.1.1. A pushdown system (PDS) is a tuple $\mathscr{P} = (P, \Sigma, \Gamma, \Delta)$ where P is a finite set of control states, Σ a finite input alphabet, Γ a finite stack alphabet, and $\Delta \subseteq P \times \Gamma \times \Sigma \times P \times \Gamma^*$ a finite set of transition rules.

A configuration of $\mathscr{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ is a control state and $w \in \Gamma^*$ a stack content. Let $Conf_{\mathscr{P}} = P \times \Gamma^*$ be the set of all configurations of $\mathscr{P}$.

For each $a \in \Sigma$, we define the transition relation $\xrightarrow{a}_{\mathscr{P}}$ on configurations as follows: if $(p, \gamma)(p', w) \in \Delta$, for each $w' \in \Gamma^*$, $\langle p, \gamma w' \rangle \xrightarrow{a}_{\mathscr{P}} \langle p', ww' \rangle$. From these relations, we can then infer the immediate successor relation $\rightarrow_{\mathscr{P}} = \cup_{a \in \Sigma} \xrightarrow{a_i}_{\mathscr{P}}$.

To some PDSs we match an initial configuration $c_0 \in Conf_{\mathscr{P}}$ of the form $c_0 = \langle p0, \rangle$, where $\perp \in \Gamma$ is a special bottom stack symbol and $p_0 \in P$ a control state. We then introduce these PDSs as quintuplets of the form $\mathscr{P} = (P, \Sigma, \Gamma, \Delta, c_0)$.

A run r starting from a configuration c is a sequence of configurations $r = (c_i)_{i \geq 0}$ such that $c_0 = c$ and $\forall i \geq 0, c_i \xrightarrow{a_i} c_{i+1}$. The word $(a_i)_{i \geq 0}$ is then said to be the trace of r. Traces and runs may be finite or infinite.

Let $Runs(\mathscr{P})$ be the set of all infinite runs of $\mathscr{P}$ starting from its initial configuration c_0. Let $Traces(\mathscr{P})$ be the set of all infinite traces of $\mathscr{P}$ starting from its initial configuration c_0.

3.2.2 From CFG to Pushdown System

Let f a function of a program, $\mathscr{G}_f = (V_f, E_f, i_f \in V_f, j_f \in V_f)$ the CFG of f.
Let $\mathscr{P} = (P, \Sigma, \Gamma, \Delta)$ the PDS from all $\mathscr{G}_{f_i}$ such that:

- $P = \{p\}$
- Σ is the set of all API calls in the program.
- $\Gamma = \cup V_f$ for all function f of the program
- Δ is defined as follow: $\forall f$ function of the program, $f_i, f_j \in V_f$
  - if $f_i \xrightarrow{a} f_j \in E_f$, where $a \in \{\varepsilon\} \cup \Sigma$, then $\langle p, f_i \rangle \xrightarrow{\varepsilon} \langle p, f_j \rangle$ $(switch)$.
  - if $j_i \in V_f$ is the exit node of f, then $\langle p, f_i \rangle \xrightarrow{\varepsilon} \langle p, \varepsilon \rangle$ $(return)$.
  - if $f_i \xrightarrow{call\ g} f_j$ where g an intern function, then $\langle p, f_i \rangle \xrightarrow{\varepsilon} \langle p, i_g f_j \rangle$ $(call)$.

A pushdown system $\mathscr{P}$ can potentially have a infinite number of state due to its infinite stack. The language $L(\mathscr{P})$ of all the traces of all the possible runs of $\mathscr{P}$ is context-free on the Chomsky hierachy. As said in 1.3, we need a finite graph representation of the program, i.e. a model M which all possible traces language $L(M)$ would be regular.

3.3 Linear Temporal Logic (LTL)

The most widely used variant of temporal logics is the linear temporal logic LTL introduced by Pnueli in Pnueli (1977). We will then use LTL formulas as defined in Pnueli (1977).

Let AP be a finite set of atomic propositions used to express events or properties about a program.

Definition 3.3.1. The set of LTL formulas is given by the following grammar:

$$\varphi, \psi ::= \perp \mid p \in AP \mid \neg\varphi \mid \varphi \vee \psi \mid X\varphi \mid \varphi U \psi$$

$\perp$ stands for the predicate 'always true'. X and U are called the next and until operators: the former means that a formula should happen at the next step, the latter, that a formula should hold at least until another formula becomes true.

LTL Semantic 3.3.2. Intuitively, we interpret the folowing operators G, F, U, X:
Gφ (Globally) means that φ is always true at any step of the program.
Fφ (Finally) means that φ is eventually true.
φUψ (Until) means that φ has to be true at least until ψ is true.
Xφ (neXt) means that φ has to be true at the next state.

Theorem 3.3.3. Given a LTL formula φ, there is a Büchi automaton B on the alphabet $\Sigma = 2^{AP}$ such that $\mathscr{L}(B) = \mathscr{L}(\varphi)$ Y.Kesten et al. (1993)

The model checking problem 3.3.4. Given a LTL formula φ, a PDS $\mathscr{P}$, the model checking problem consist on checking whether $\forall t \in \mathit{Traces}(p),\ t \models \varphi$.

Chapter 4

Approximation of a Pushdown System

A pushdown system $\mathscr{P}$ of our program is easy to get using LLVM Lattner and Adve (2004). But applying model checking to check if a PDS verifies a formula is more complex than applying it to a finite state automaton. Furthermore, we would like to use the Spot library Duret-Lutz et al. (2016) in order to check if the model verifies a formula. Yet, Spot does not support LTL formula verification on PDS. One solution would be to use another model checking library or implement our own PDS model checking techniques, but it would probably be too tedious. Instead, we will transform our model into a finite state graph which is an approximation of the PDS. Doing so means finding an approximation for direct or indirect procedure recursion calls by transforming the language of the trace of $\mathscr{P}$ into a regular language.

4.1 Under-approximations technique

4.1.1 The Use of an Under-Approximation

Let φ be an LTL formula and F $= \mathscr{L}(\varphi)$.
Let $\mathscr{P}$ be a PDS and E$= \mathit{Traces}(\mathscr{P})$.
Let $\mathscr{P}$ be a finite state system which is an under-approximation of $\mathscr{P}$ and U $= \mathit{Traces}(\mathscr{U})$.

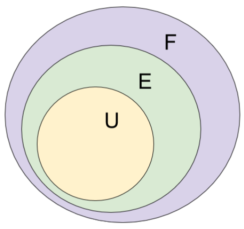

Figure 4.1: Here, p verifies γ

As shown on 4.1:
$\mathscr{U}$ is an under-approximation of $\mathscr{P}$ so U $\subseteq$ E.
If $\mathscr{P}$ verifies γ then E $\subseteq$ F.
This means that
$\exists t_u \in \mathrm{U} \mid t_u \notin \mathrm{F} \Rightarrow \mathrm{E} \nsubseteq \mathrm{F}$.
Thus, if $\mathscr{U}$ does not verify φ, then $\mathscr{P}$ does not verify φ. Although an under approximation does not allow us to conclude on whether a PDS verifies a formula, it can be used to refute it.

4.1.2 Under-Approximations Technique

If we were to find a finite state graph for a sequential program with no procedure recursion, the easy way would be to simply get the main function of the program, and insert other functions graph to the main graph for each function call until the is no function call left. Now, if we use the same technique on a program containing recursive calls, we will insert functions an infinite number of time until eventually getting memory overflows as shown on figure 4.2. And from this problem came a simple approximation intuition: we just limit the insertion depth, meaning the number of possible nested insertion in the context.

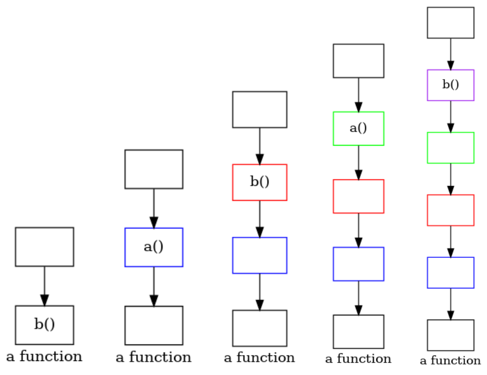

Figure 4.2: Applying simple insertion algorithm to a program's model with recursion calls

Let $\mathscr{P} = (P, \Sigma, \Gamma, \Delta_{\mathscr{P}})$ a PDS, $tr(\mathscr{P})$ the set of possible traces of $\mathscr{P}$.
Let $\mathscr{A} = (Q, \Delta_{\mathscr{A}}, \Sigma)$ a finite state system such that $\mathscr{P}$ and $\mathscr{A}$ are on the same dictionary Σ.
Let $n_{max} \in \mathbb{N}$ a depth bound.
$Q = \{\langle p, w\rangle \ \in Conf(\mathscr{P}) \mid \langle p, w\rangle \ \in P \times \Gamma^{n_{max}}\}$
$\Delta_{\mathscr{P}} = (p, w, a, p', w') \in P \times \Gamma \times \Sigma \times P \times \Gamma$
$\Delta_{\mathscr{A}}$ is defined such that: $\forall c_1, c_2 \in P \times \Gamma^{n_{max}}$, if $c_1 \xrightarrow{a}_{\mathscr{P}} c_2$ then $c_1 \xrightarrow{a} c_2 \in \Delta_{\mathscr{A}}$.
Q is finite and $\mathscr{A}$ is an under-approximation of $\mathscr{P}$.

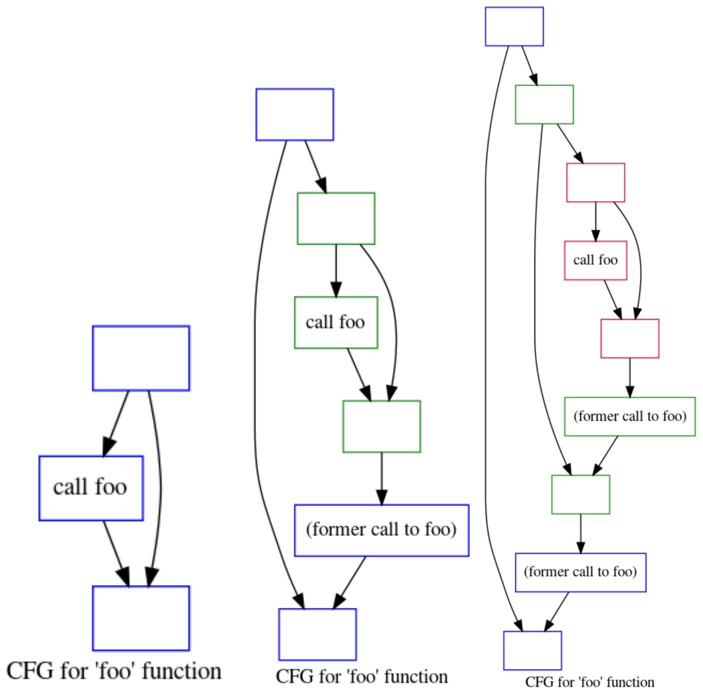

Figure 4.3: Example of the under-approximation method application with depth 2

4.2 Over-approximations technique

4.2.1 The Use of an Over-Approximation

Let φ be an LTL formula and $\mathrm{F} = \mathscr{L}(\varphi)$. Let $\mathscr{P}$ be a PDS and $\mathrm{E} = \mathit{Traces}(\mathscr{P})$.
Let $\mathscr{O}$ be a finite state automaton which is an over-approximation of $\mathscr{P}$ and $\mathrm{O} = \mathit{Traces}(\mathscr{O})$.

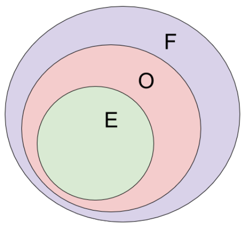

Figure 4.4: Here, p verifies γ

As shown on 4.4:
$\mathscr{O}$ is an over-approximation of $\mathscr{P}$ so $\mathrm{E} \subseteq \mathrm{O}$.
If $\mathscr{P}$ verifies φ then $\mathrm{E} \subseteq \mathrm{F}$.
This means that
$\forall t_o \in \mathrm{O}, t_o \in \mathrm{F} \Rightarrow \mathrm{E} \subseteq \mathrm{F}$.

Thus, if $\mathscr{O}$ verifies φ, then $\mathscr{P}$ verifies φ. Although over-approximation does not allow us to conclude on whether a PDS does not verify a formula, it can be used to prove that a PDS verifies a formula.

4.2.2 Over-Approximations Technique

Whereas the under-approximation technique came out intuitively from the natural use of a PDS, the over-approximation technique may not be so inherent. To find it, we used a simple example of a recursive function call.

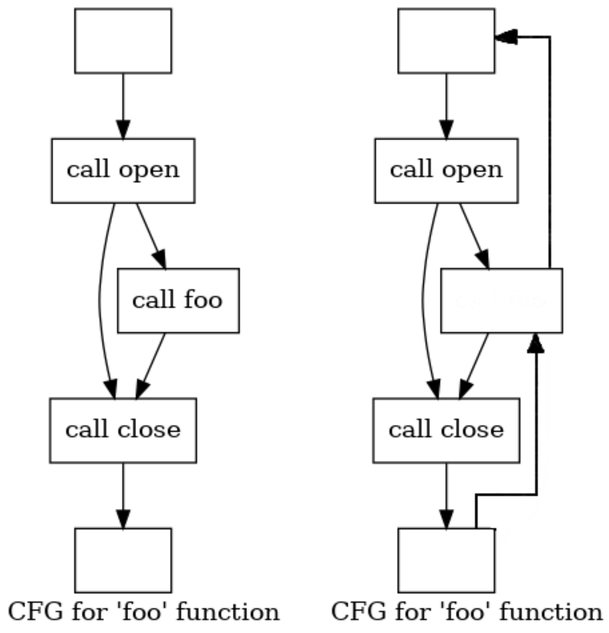

Figure 4.5: Example of the over-approximation method application

Left figure on 4.5 shows a PDS p where $AP_p = \{\text{call open, call foo}\}$ being the possible atomic propositions used by p and where the foo function recursively calls itself. The language associated to the possible traces of p for all its possible run $L(P)$ is $(CallOpen)^n \cdot (CallClose)^n, n \in \mathbb{N}^*$ as we call open the same number of time as close.

The closest regular language containing $L(P)$ we could think of would then be $(CallOpen)^+ \cdot (CallClose)^+$. So now, we want to find a way to get a finite state graph for which the language of its possible traces for all its possible runs would be the former. Obtaining a finite state graph from this PDS would first mean removing function calls (those that are not atomic propositions), that is removing call foo. And by linking the state which calls the function foo to the top of the function, and linking the bottom (return statement) of the function to the calling state, we get an automaton which language is $(CallOpen)^+ \cdot (CallClose)^+$.

So here we have an over-approximation technique which consist in linking the call state to the beginning of the function and linking the end of the function to the call state. It works for direct recursive calls, e.g calls to the function that is on top of the function stack.

Now, we need a similar solution for indirect calls: we keep a context of insertion (a list of all the previous insertions which is equivalent to the list of function in the stack of p). Whenever we encounter a call to a function, we check if we have inserted that function before in this context, e.g if inserting the function would create a nested insertion. If we have not, we insert it in a classic way. If we have, we link like described above the call states to the previously inserted function.

Note that in the worst case, i.e. when every function of the program calls every other function, we multiply the size of our graph by itself.

Let $\mathscr{P} = (P, \Sigma, \Gamma, \Delta_{\mathscr{P}})$ a PDS.
The infinite set C of configuration of $\mathscr{P}$ has to be reduced to a finite approximation. To do so, we define a congruence relation $\sim$ on C so we get a finite set of equivalence class $C/\sim$.

The congruence $\sim$ is defined as follow:
Let Γ^* be the set of all possible stacks of $\mathscr{P}$.
$\forall P_1 \in \Gamma^*, P_1 = h_1 \cdot P_{1_r}$ where $h_1 \in \Gamma$ the top of stack P_1 and $P_{1_r} = P_1/\{h_1\}$ the rest of the stack.

$\forall P_2 \subseteq \Gamma^*,$ if h_1 appears in P_{1_r} and $\exists P_3 \subseteq \Gamma*$, $P_2 = P_1 \cdot P_3$, then $P_1 \sim P_2$: P_1 and P_2 are part of the same equivalence class. Two stack have the same equivalence class if the smallest one contains the same symbol of Γ twice, and the biggest one is equal to the smallest stack to which we added elements of Γ.

$\forall P \in \Gamma^*, |P| > |\Sigma| \Rightarrow \exists P' \in \Gamma^*$, $|P'| < |P|$ and $P \sim P'$. Thus, there is a finite number of equivalence class.

The finite system $\mathscr{O} = (C/\sim, \Sigma, \Delta_{\mathscr{O}})$ is then introduced as an over-approximation of $\mathscr{P}$, such that:
$\Delta_{\mathscr{P}} = (p, w, a, p', w') \in P \times \Gamma \times \Sigma \times P \times \Gamma$
$\Delta_{\mathscr{O}}$ is defines as follow: if $C_1, C_2 \in \mathit{Conf}(\mathscr{P})$ and $\exists c_1 \in C_1, c_2 \in C_2$ such that $c_1 \xrightarrow{a}_{\mathscr{P}} c_2$ then $c_1 \xrightarrow{a} c_2 \in \Delta_{\mathscr{O}}$

The latter finite state system $\mathscr{O}$ describes the over approximation obtained by following the insertion algorithm describe in 4.5.

Chapter 5

Implementation

5.1 From Program to Multiple Graphs

For now, the only way we have of obtaining a CFG from a program is from C source code. LLVM tool allow us to extract a CFG for each function of the program as *.dot* text files that represent a graph.

In order to only keep function calls, we use *sed* (stream editor) and regular expressions to remove useless information and only keep lines involving a call to a function. Unfortunately, since assembly code function calls uses bracket enclosing, sed cannot extract nested function calls (e.g $call\ close(call\ open()))$. So using we extract function calls from the rest of the expression using a simple C++ program.

Now, as graphs as .dot files are hard to analyze and modify, we need a way to parse *.dot* files and extract the graph. The Boost library provides a tool to parse a .dot file and get the graph as a C++ adjacency list implementation. We can then store all the graphs as part of the same list of all the graphs of all functions of the program. From there, it is going to be much easier to manipulate our graphs.

5.2 Linking the Graphs Together

Now that we have all our graphs as mutable C++ graphs, we can start inserting graphs into the main one. The principal graph is the graph of the function *'main'* and we will insert other graphs into this one.

Under-approximation and over-approximation techniques are performed as part of the linkage process.

5.2.1 Under-Approximation Linkage

Under-approximation is done like following:

- 1. We set a count to 0.
- 2. For all new nodes that we never went on, we insert functions graphs according to the called function in the node.
- 4. We increase the count.

- 5. If the count is less than the maximum insertion depth, we repeat from step 2. Else we exit.

5.2.2 Over-approximation linkage

Over-approximation is done like following:

- 1. We initialise an empty list.
- 2. We apply 3. to all nodes
- 3. For one node, we extract function calls.
  If the function is not inside the list, we insert functions graphs according to the called function in the node. Then we push the name of the function and its top and bottom nodes inside the list, and we apply 3. to the newly inserted nodes. Finally, we pop the function name and nodes out of the list.
  If the function is already in the list, it means it was already inserted, and we link the calling node to the top of the function that was inserted before, and the bottom of the function to the calling node.

5.3 Applying Model Checking on the Model

Once we have a single interprocedural graph, a first step is to put all function calls to the edge of the graph, as Büchi automata as we defined them are transition-based.

We also make the last node of the graph loop on itself to simulate an infinite run of the program when the program finishes, as LTL formula only describe properties for infinite runs of the program.

We can now transform our graph into a Spot implementation Büchi automaton by describing the nodes and edges of our program.

Spot supplies useful operations on automata, such as an automaton negation, an automaton emptyness check, and automata multiplication that can be interpreted as the intersection of two automata. We can then check if a automaton set of possible traces is a subset of another using:

$$E \subseteq F \Rightarrow E \cap \bar{F} = \emptyset$$

Spot also supports transforming a LTL formula into a Büchi automaton. Using spot operations on automata, we can easily say if the set of traces of our Büchi automaton is a subset of the traces verified by the Büchi automaton obtained from the formula.

- If the over-approximation of the program is a subset of the formula, then the program verifies the formula.
- If the under-approximation of the program is not a subset of the formula, then the program does not verify the formula.
- If the over-approximation of the program is not a subset of the formula, but the under approximation is a subset of it, then we cannot conclude due to approximation imprecision.

5.4 Optimisation Issues

- Comparing two automata implies multiplying one to the other when processing intersection.
- Processing under-approximation, in one of the worst case, multiplies the number of state by the depth.
- Processing over-approximation, in the worst case, multiplies the number of state by itself.

We risk having really big graphs on big programs, provoking possible memory overflows and long computation times.
That is why we should remove useless nodes created by the function insertions by using accessibility check and by removing epsilon transitions.

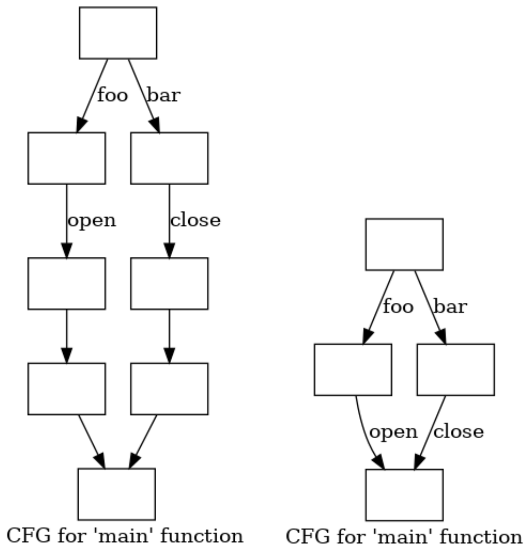

Figure 5.1: Example of ε-transition removal

Also, model checking algorithms work by going through our graph. But they might not go through the entire graph. We should only insert functions dynamically when Spot needs them. Those two possible improvements are not features of the current program but they could greatly increase our performances.

Chapter 6

Conclusion

In this report, we showed a way to show if a C program verifies or does not verify a LTL formulas using the Spot library. To do so, we mainly focused on how to get our program as a finite state system by firstly transforming it into a pushdown system using the Clang and LLVM analysis tools, and by finding approximations of the pushdown system into a finite system. We also provided a full functional pipeline that automatically gives a result from a *.c* source file and a LTL formula. However, keep in mind that our pipeline is not always able to conclude on whether the program verifies the property due to models approximations imprecisions.

Here is some improvement we could bring to this project:

- Proceed lazy insertions while doing spot verification in order to avoid useless insertion and remove useless ε-transitions in order to improve performance.
- Find a way to get interesting graphs from compiled code or other languages.
- Find a graph representation to express properties about the ranges of values of variables in domains of the program.

Chapter 7

Bibliography

BABENYSHEV, S. and RYBAKOV, V. (2016). Linear temporal logic ltl: Basis for admissible rules. (page 7)

Bermudez, M. E. and Schimpf, K. M. (1990). Practical arbi-trary lookahead lr parsing.

Besson, F., Jensen, T., Metayer, D. L., and Thorn, T. (2005). Model checking security properties of control flow graphs.

Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., and Xu, L. (2016). Spot 2.0 — a framework for ltl and omega-automata manipulation. (page 14)

Duret-Lutz, A. and Poitrenaud, D. (2004). Spot: an extensible model checking library using transition-based generalized buchi automata. (page 6)

Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. (2000). Efficent algorithms for model checking pushdown systems.

Heizmann, M., Hoenicke, J., and Podelski, A. (2013). Software model checking for people who love automata.

Kobayashi, N., Sato, R., and Unno, H. (2011). Predicate abstraction and cegar for higher-order model checking.

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis and transformation. (pages 8 and 14)

Pnueli, A. (1977). The temporal logic of programs. (pages 6 and 12)

Y.Kesten, Z.Manna, H.McGuire, and A.Pnueli (1993). A decision algorithm for full propositional temporal logic. (page 13)