Universiteit Leiden
The Netherlands

Opleiding Informatica

Towards Learning Software Models:

making documentation easier

Daniël Fokkinga

Supervisors:
Dr. M.M. Bonsangue & Dr. H.C.M. Kleijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/08/2017

Abstract

In this thesis we apply the learning algorithm by Dana Angluin in order to build an automata model of C code. We propose abstractions to make the model tractable and present a prototype tool, I-Learner, for learning the call graph of an arbitrary piece of C code. The resulting automaton plays a similar role of UML interaction diagrams, but the current implementation does not allow for modelling recursive calls.

Contents

Chapter 1

Introduction

This thesis is written as part of the bachelor Computer Science at Leiden University and the topic is model learning from software, which means creating a model that shows the behaviour of software. Creating models representing software is a big part of the documentation process in Software Engineering [Vli08]. The goal of this project is to build such a model automatically for a program without looking at its code. The model that will be produced will be a deterministic finite automaton (DFA). In order to learn a DFA we will use the L* algorithm from Dana Angluin, discussed in Chapter 3. Because of the restricted amount of information contained in a DFA, a selection will be made on the information contained in the programs that we want to model. Moreover, we need a method to gain this information during the execution of the program and once this information has been collected it will have to serve the L* algorithm in order to build a DFA. These problems are discussed in Chapter 4 and the solutions to these problems described in Chapter 5. The software built according to these methods, *I-Learner*, will be evaluated in Chapter 6 and the produced models by *I-Learner* will be compared to UML diagrams, which are commonly used models in Software Engineering. The conclusions and suggestions for future work can be found in Chapter 7.

Chapter 2

Definitions

A deterministic finite automaton M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$. Q is the set of states. Σ is the set of input symbols, called the alphabet. $\delta : Q \times \Sigma \to Q$ is the transition function. q_0 is the initial state and $F \subseteq Q$ the set of accepting states. M is able to accept or reject strings over the alphabet Σ. A non-empty string $a = x_0x_1x_2...x_n$ over the alphabet Σ is accepted by M if and only if there exist a sequence of states $y_0, y_1, y_2, ..., y_n$ in Q where $y_0 = q_0$, $y_{i+1} = \delta(y_i, x_i)$ for all $0 \leq i \leq n$ and $y_{n+1} \in F$. The empty string is accepted by M if and only if $q_0 \in F$. [HMU06]

In this thesis we will also use the definition δ^*, where for a string x, $\delta^*(q_0, x)$ is the state where the automaton in is, after following all the transitions belonging to the symbols of x, starting in q_0.

Chapter 3

A learning algorithm

In 1987, Dana Angluin published a paper in which she presented her L* algorithm [Ang87]. This algorithm is able to learn a regular language resulting of the minimum deterministic finite automaton (DFA) recognizing it.

3.1 Minimally Adequate Teachers

In order for the algorithm to gain knowledge about the language U for which it is building a DFA, it uses a *minimally adequate teacher*. This teacher should be able to answer two types of questions about the language from the learner.

First of all, a *membership query*: the learner presents a string t and the teacher answers *yes* if t is a member of the language, otherwise the teacher answers *no*.

The second type of question is a *conjecture*: the L* algorithm will in this case present a DFA M. The teacher then answers *yes* if M recognizes the language U. The answer should be a counterexample, a string t, if M is not equivalent to U. t will then represent a difference between M and U. This string will be either accepted by M but is not a member of U, or is rejected by M and is a member U.

The L* algorithm only gains information from the teacher in the form of its answers to the learner's questions. Therefore L* works abstracting from how a teacher is implemented and can work with any kind of implementation as long as the answers to the questions are identical for a particular language U. Where the representation of U can differ depending on the teacher as well.

3.2 Data structures

Initially, the algorithm L*, is assumed to know a set A, the finite alphabet over which the unknown language U, is composed.

Information about U that the learner learns during an execution is organized into an *observation table*. This table consists of two sets of strings and a finite function. The first set of strings is a nonempty finite prefix-closed set S and the second set is a nonempty finite suffix-closed set E. The finite function is called T and maps $((S \cup S \cdot A) \cdot E)$ to $\{0, 1\}$. The value of T for a string u is that $T(u) = 1$ if and only if u is a member of U.

To initialize this observation table, L* starts with $S = E = \{\lambda\}$, where λ is the empty string.

The observation table can be seen as a two-dimensional array with rows labelled by elements of $(S \cup S \cdot A)$ and columns by elements of E. The entry of row s and column e will be equal to $T(s \cdot e)$. For all the rows s, with s an element of $(S \cup S \cdot A)$, $row(s)$ is the bitstring consisting of all values $T(s \cdot e)$ for all e in E.

In order to build a DFA, the algorithm uses the observation table. But for the observation table to be suitable to build a DFA it should satisfy two conditions. The first condition is called *closedness* and an observation table is *closed* if for each t in $S \cdot A$, there exists an s in S such that $row(t) = row(s)$. In other words for every row in $S \cdot A$ there is an equal row in S.

The second condition is called *consistency* and an observation table is *consistent* if for all s_1, s_2 in S such that $row(s_1) = row(s_2)$, for all a in A, $row(s_1 \cdot a) = row(s_2 \cdot a)$. In other words, for every pair in S that have the same row, for every symbol in A, if you add this symbol to both elements of the pair, each of their rows are still the same.

3.3 DFA propositions

Once the observation table (S, E, T) satisfies both conditions, a DFA $M(S, E, T)$ can be constructed. This DFA runs over the same alphabet A and the set of states Q consists of all the unique rows $row(s)$, with s in S. The initial state q_0 will be the row belonging to the empty word, $row(\lambda)$. The set of accepting states F will be all the rows $row(s)$, with s in S, for which $T(s) = 1$. Finally, the transition function δ will be denoted by $\delta(row(s), a) = (row(s \cdot a))$, with s in S and a in A. In other words for state in Q, $row(s)$ with s in S, there is a transition for each a in A to the state corresponding to $row(s \cdot a)$.

3.4 Operations of the algorithm

In order to make the observation table satisfy the two conditions, *closedness* and *consistency*, the algorithm has two operations; one in case the observation table is not *closed* and one if the observation table is not *consistent*.

If the observation table (S, E, T) is not *closed* then the learner finds a s_1 in S and an a in A where $row(s_1 \cdot a)$ does not equal $row(s)$ for all s in S. $s_1 \cdot a$ is then added to S, and T is extended to $((S \cup S \cdot A) \cdot E)$ using membership queries. In other words, the string belonging to the row in $S \cdot A$ that does not have an equal row in S, will be added to S. In Subsection 3.5, we refer to this operation as *MakeClosed*.

If (S, E, T) is not consistent, the learner finds a pair s_1, s_2 in S, an e in E and an a in A, where $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$. The string $a \cdot e$ is then added to E, and T extended to $(S \vee S \cdot A) \cdot (a \cdot e)$ using membership queries. Intuitively, there are two entries in the observation table that makes it so that there exists a pair in S that has the same row, but not the same row for every symbol in A added to both elements of the pair. These two entries both belong to the same column e in E and the last symbol of their row is an a in A, $a \cdot e$ will then be added to E. In Subsection 3.5, we refer to this operation as *MakeConsistent*.

3.5 Algorithm outline

1: Set S to λ
2: Set E to λ
3: Ask membership queries for λ and for each a in A
4: Construct (S, E, T)
5: **repeat**
6: **while** (S, E, T) not *closed* or not *consistent* **do**
7: **if** (S, E, T) is not *consistent* **then**
8: *MakeConsistent* (see Subsection 3.4)
9: **end if**
10: **if** (S, E, T) is not *closed* **then**
11: *MakeClosed* (see Subsection 3.4)
12: **end if**
13: **end while**
14: Propose conjecture $M = M(S, E, T)$
15: **if** Teacher replies with counterexample t **then**
16: Add t and all its prefixes to S
17: Extend T to $(S \cup S \cdot A) \cdot E$ using membership queries
18: **end if**
19: **until** Teacher replies *yes* to conjecture M
20: Halt and output M

Figure 3.1: Outline of the L* algorithm

Chapter 4

Problems

In order to apply the L* algorithm from Dana Angluin discussed in Chapter 3 to software that we want to model, some problems have to be solved. In this chapter we describe these problems.

4.1 The alphabet

4.1.1 Alphabet choice

Because the L* algorithm requires to know the alphabet over which the language is composed that is to be learned, we need to decide how to determine the alphabet, a finite set of symbols, for any program for which we want to build a DFA. From this set of symbols we should be able to construct strings that represent the behaviour of this program. Besides that, it should also be decidable, for any program that we want to model, how the corresponding alphabet would look like.

4.1.2 Gaining information from a program execution concerning the alphabet

After it is decided how the alphabet of a program is composed, the next problem to solve is how to gain information concerning this alphabet from a program execution. In other words, it should be possible to build strings over the chosen alphabet that represent an execution from the program. Such a string should be derived without looking at the code, thus during runtime of the program. The reason we want to gain this information without looking at the source code is because in practice, access to this code might not always be available. For example, when working with libraries or if you only have access to an executable that you want to know more about.

4.2 The teacher

The L* algorithm assumes that there is a *minimally adequate teacher* present to answer questions, namely the *membership queries* and *conjectures*. To learn a model representing behavioural aspects of a program, a teacher with the ability to answer these questions should be designed.

4.2.1 How to answer membership queries

When the L* algorithm presents a string t over the alphabet, discussed in Subsection 4.1.1, the teacher should be able to test this string t. This means checking whether or not the t represents possible behaviour of the program. If this is the case, the teacher can answer *yes* or *no* if not.

4.2.2 How to answer conjectures

When the L* algorithm asks the second type of question, a *conjecture*, it proposes a DFA M, see Subsection 3.3. The teacher should then be able to check if M is a correct model of the given program.

This means, first of all, checking if every string, over the given alphabet, that represents possible behaviour of the program, is accepted by M. If not, a string t should be given to the L* algorithm. t should be a string that belongs to the program but is not accepted by M and acts as a counterexample.

Moreover, it should also be checked that every string accepted by M is also a string that represent possible behaviour of the program, if not, a string t should be given. t should be a string that is accepted by M but is not a string that belongs to the program and will be seen as counterexample.

If there is not such a string t for both checks, the teacher should answer *yes*.

Chapter 5

Method

In order to build a DFA model of a program we have developed the software *I-Learner*. *I-Learner* is a combination of an implementation of the L* algorithm by Dana Angluin and a teacher able to answer the questions from L* based on an input program. To create this teacher we had to solve the problems mentioned in Chapter 4. In this chapter, we discuss the design of *I-Learner*, meaning how we solved these problems.

I-Learner is implemented in C++ and works on C programs.

5.1 The alphabet

As discussed in Subsection 4.1.1, every input program needs an alphabet to represent its behaviour. With *I-Learner*, the decision has been made to take the calls and returns of functions from a program into account. Thus the alphabet would be a set of two types of symbols. If we choose to model a program A, the alphabet would consist of, for every function x of A, the symbols $Call(x)$ and $Return(x)$.

A string of symbols over this alphabet can be used to represent a possible execution path of the program. The DFA of a program P would then determine the set of strings representing some possible execution paths of the program. This set is considered to be the language of P, $L(P)$.

Because we are working with C programs and every program would therefore have to include a function call and return to the `main` function during its execution, *I-Learner* excludes this function.

5.2 Tracing information during execution

To determine $L(P)$ for a program P, we have to execute P a number of times, this number is defined by the user of *I-Learner*, and trace the function calls and returns during all these executions. All the strings representing these traces of functions calls and returns together, will then form the set $L(P)$.

To trace this information during the execution of a program, *I-Learner* expects the input program to be compiled with the `-finstrument-functions` flag from GCC [GCC]. If a C program is compiled with the `-finstrument-functions` flag from GCC two extra functions will be added to each function that is compiled, namely one extra function when it is called and one when it returns. Using these two extra functions, *I-Learner* can save the name of the function once it has been called and when it is returned, thus building a string of calls and returns.

To build $L(P)$ for a program P, *I-Learner* simply executes P a finite amount of times and saves each string representing the current execution path. When P needs an input parameter, *I-Learner* simply generates a random input for each time it needs to be executed. After the finite amount of executions of P have been performed, *I-Learner* will consider $L(P)$ to be complete.

5.3 Assumptions and limitations

Because *I-Learner* needs to generate input for every program itself, we assume that *I-Learner* will only receive programs P with one integer number as input. This input can then be randomly generated by *I-Learner*. However, because, in general there are still an infinite amount of possible inputs, e.g. in the case of an integer, it is not possible to execute P for every possible input. *I-Learner* currently can only execute P n times, each time with a randomly generated input, with n a finite number. Besides, $L(P)$ might also be growing in time, instead of being bound by n, because the possibility of a certain execution path through a program can depend on another factor besides the input. Therefore, it is possible, that *I-Learner* will generate a DFA M accepting an incomplete $L(P)$, thus M will have to be seen as an *approximation* of a behaviour model for P.

Moreover, because we have chosen to trace information during the execution using `-finstrument-functions` flag from GCC, it is required to have access to the code of the program that we want to model. In practice, this may not always be the case, for example when dealing with libraries or executables, see Section 4.1.2. In these cases, it is unfortunately not possible to use this method.

5.4 Answering a membership query

When the L* algorithm answers a membership query for a string t, *I-Learner* can simply go over each string x in $L(P)$ and see if x matches t. If such a matching string for t is found, *I-Learner* answers *yes*, otherwise *no*. This is possible because $L(P)$ is finite.

5.5 Answering a conjecture

When the L* algorithm proposes a DFA $M = (Q, A, \delta, q_0, F)$, see Subsection 3.3, *I-Learner* first checks if every string x in $L(P)$ will be accepted by M. This is done by checking if state q is part of F, for $q = \delta^*(q_0, x)$. If q is

part of F, q is an accepting state and accepts M the string x. In case of a string x that is not accepted by M this string x will be returned to L* as counterexample.

When every string x in $L(P)$ is accepted by M, *I-Learner* performs an additional check. Because every string accepted by M, also has to be in $L(P)$. To check this, *I-Learner* does the following:

A set X is constructed, X will contain pairs of strings of symbols from A and states from Q. X will initially contain (λ, q_0). For each pair (x, q) in X, if x is not a prefix of any of the strings in $L(P)$, then *I-Learner* checks if there is a reachable accepting state from q, if this is the case, x will be returned to L* as a counterexample, if not, q will be marked and the following action will be taken. For every symbol a in A, the pair $(a, \delta(q, a))$ will be added to X as long as $\delta(q, a)$ is not marked. This will be repeated until all states are marked or a counterexample is found. When no counterexample is found, *I-Learner* will answer that the conjecture is correct.

Chapter 6

Evaluation

In this chapter, the methods discussed in Chapter 5 will be evaluated. To do this, we look at some of the resulting models produced by the software, *I-Learner*, for a set of several input programs.

6.1 Results

6.1.1 Simple branching program

Listing 6.1 shows a simple C program consisting of 4 functions. Each function can either return itself without calling another function, or call one other function. With the exception of the function d, that cannot call another function.

The alphabet of this program for *I-Learner* is $\{Call(a), Return(a), Call(b), Return(b), Call(c), Return(c), Call(d), Return(d)\}$.

Depending on the input of this program, an execution path can be determined with 4 possibilities, thus 4 possible strings in the language.

The result that *I-Learner* has produced is visible in figure 6.1. In this DFA the four different branches are clearly visible and after each function call, it can be seen which options the program has left. This DFA has been produced, restricting *I-Learner* with the fact that it can only generate random integers between 1 and 4 for the input.

However, because the execution path is depending on the input, there is still a chance that after, for instance 10 executions of the program, not every possible execution path has been explored by *I-Learner*. If for example, the execution path where c calls d before returning itself is missing, the transition $(q_3, Return(c)) = q_4$ will not be present in the resulting DFA.

```c
#include <stdlib.h>

void d(){
  int x = 1;
}

void c(int x){
  if(x > 3)
    d();
}

void b(int x){
  if(x > 2)
    c(x);
}

void a(int x){
  if (x > 1)
    b(x);
}

int main(int argc, char *argv[]){
  a(atoi(argv[1]));
  return 0;
}
```

Listing 6.1: Simple C program, where atoi() is a function from stdlib.h that converts the string argument to an integer

6.1.2 Detecting (infinite) loops

Because *I-Learner* will make sure that the resulting DFA will *only* accept the language $L(P)$ of the input program P and because of the restriction on $L(P)$, see Section 5.3, *I-Learner* is not able to detect infinite loops in a program. For example, a program with one function that is able to call itself recursively will result in the DFA in figure 6.2. To create this DFA, we had to restrict the function in a way that it would only call itself maximally 5 times. By increasing this limit by 1, the DFA in figure 6.3 would be the result, where the extra recursive call results in two additional states, q_{11} and q_{12}. Removing this limit would result in a possibly infinite DFA where each extra recursive call would need two additional states.

These results can be expected when it is considered that the language of a program that we wish to model in this case is context-free and would require a pushdown automaton to be modelled correctly. These types of programs are therefore not suitable to be modelled by *I-Learner*.

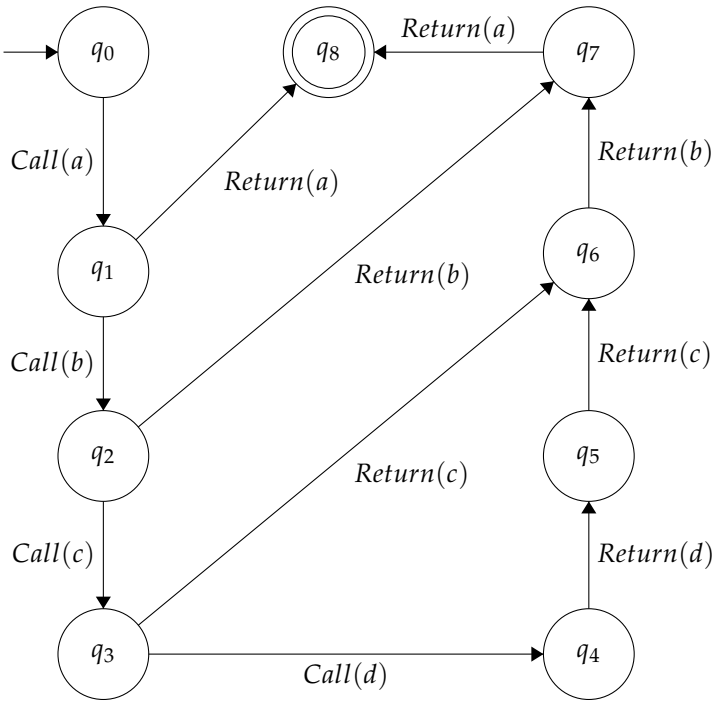

Figure 6.1: Result of the simple C program

6.1.3 Relation to UML

Interaction diagrams, part of the Unified Modelling Language (UML), are often used in Software Engineering to model collaboration of a group of objects [BRJ99]. If we relate this to the collaboration between functions in the simple C program, described in Section 6.1.1, it is possible to see which functions depend on each other or communicate in the program. Typically, an interaction diagram models a single use-case or sequence. For example, in the case of our simple C program, see figure 6.4. Whereas the DFA produced by *I-Learner*, figure 6.1, shows a collection of possible sequences.

Moreover, state transition diagrams are also used to describe behaviour of objects in the software engineering process, the results of *I-Learner* can be used to help to create these state transition diagrams in order to describe processes.

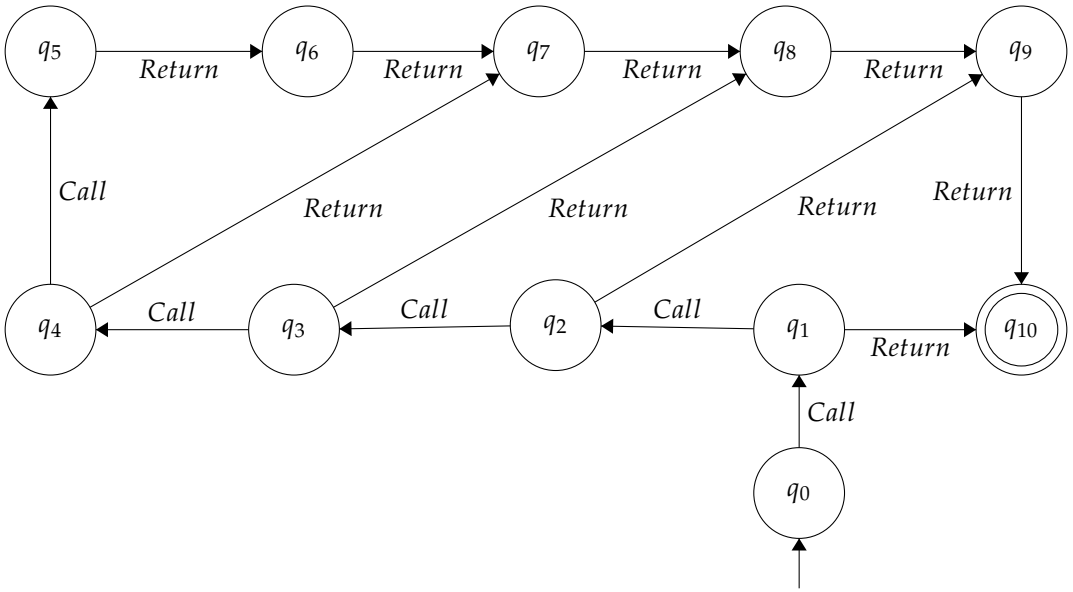

Figure 6.2: Result of C program with 1 recursive function

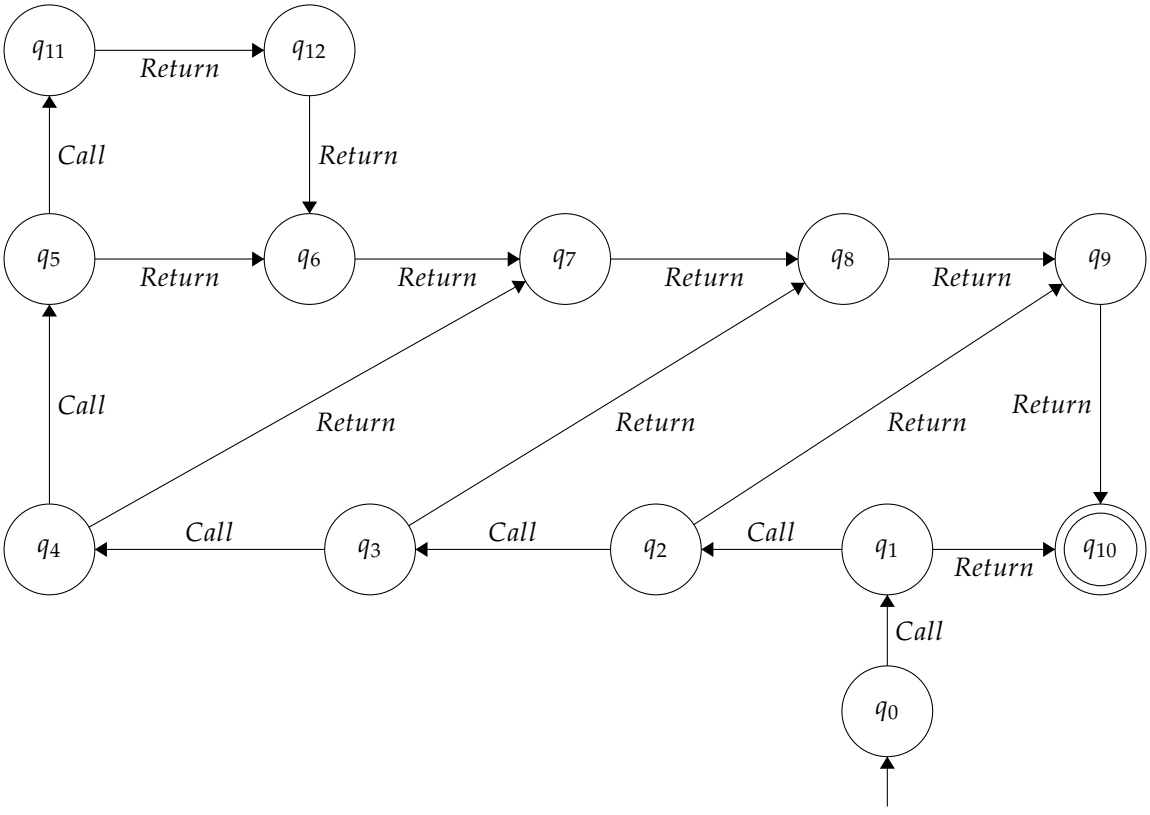

Figure 6.3: Result of C program with 1 recursive function (increased limit)

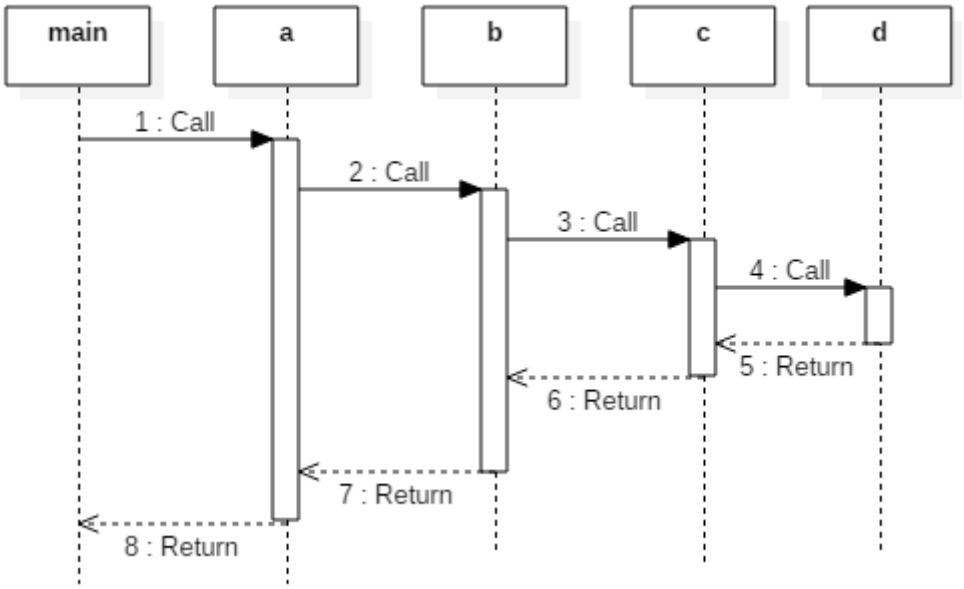

Figure 6.4: Interaction diagram example

Chapter 7

Conclusions

Our software, *I-Learner*, can trace and save information about a program during its execution. Using this information it is able to run the L* algoritm from Dana Angluin in order to build a DFA representing a model of the program in question. This model shows a collection of possible execution paths of function calls and returns. However, the possibility exists that a possible execution path will be missed by *I-Learner* and thus the produced model will have to be seen as an approximation. Moreover, *I-Learner* is not suitable for recursive programs or programs with loops in general, because *I-Learner* builds a DFA which accepts a regular language. The models produced by *I-Learner* can serve as substitution for, or addition to, certain UML diagrams, like the interaction diagram or the state transition diagrams.

As for future work, a possible next step would be to expand the selection of information from a program that *I-Learner* takes into account when creating a model. For example, when a function is called, the values of he parameters could be traced as well. This would lead to an even more informative model. Because *I-Learner* requires a program to be compiled with a specific compile flag, access to the program code is still required in most cases. It would be interesting to use a method that does not require this and we would be able to build models in a true “black box” spirit.

Bibliography

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[GCC] Code gen options - using the gnu compiler collection (gcc). `https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Code-Gen-Options.html`. Accessed: 10-8-2017.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Vli08] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd edition, 2008.