Extension of Ultimate for pthread programs

- Handling of outside-loops-pthread occurences -

Results of the bachelor project of
Rezart Qelibari

Supervisor
Matthias Heizmann

Albert Ludwigs University Freiburg
Department of Computer Science - Chair of Software Engineering

A huge thank you to my supervisor Matthias Heizman!

Chair of Software Engineering
Albert Ludwigs University Freiburg
http://swt.informatik.uni-freiburg.de

Title:
Extension of Ultimate for outside-loops-pthread programs

Project Period:
Summer Semester 2016

Author:
Rezart Qelibari

Supervisor:
Matthias Heizmann

Page Numbers: 17

Date of Completion:
November 28, 2016

Abstract:

This document describes the results of my attempt to extend Ultimate to understand pthread programs, where pthread is not inside a loop. In short: I had to extend the Boogie grammar with two new keywords: 'fork' and 'join', refering to *pthread_create* and *pthread_join* and with them their handling in the Abstract Syntax Tree and the Control Flow Graph.

Contents

Todo list

Chapter 1

Introduction

1.1 The Project

1.1.1 Goal

The goal of my bachelor project at the chair of Software Engineering is to extend Ultimate [1] - a framework for program analysis and verification written in Java - to be able to map pthread [2] programs to Ultimate's internal model, so Ultimate can perform verification tasks on it. At the end Ultimate should be able to solve some of the SV-Comp pthread tasks of the year 2016 (found on github).

Behoben: 1. it's -> it, 2. zu viel "it" man weiß nicht worauf es sich bezieht

1.1.2 Preconditions

Ultimate processes programs in four steps:

1. Translate program language to Boogie language, represented by an Abstract Syntax Tree (AST)
2. Use the AST to create a Control Flow Graph (CFG)
3. Use the CFG for the desired analyzation and verification

Behoben: Probleme bereits in letzter (ca. 2-3 Wochen alt) Feedback E-Mail beschrieben

The current version of Ultimate can already handle all four steps on a C-program, which does not use pthread. Further on, Ultimate's toolchain can handle nodes with multiple outgoing edges (graph fork) inside the CFG.

Behoben: branch == if-then-else?

[1] The Ultimate Project Site: https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/
[2] An overview of pthread: https://computing.llnl.gov/tutorials/pthreads/

1.1.3 Idea/Solution

The preconditions lead to the following idea to solve the goal:

1. Extend the AST to represent pthread calls properly e.g. with all important information neccessary to map pthread, hence extend Boogie.

   Behoben?: what is a point of concurrency?

2. Extend the CFG to represent the AST extension usefully, e.g. such that the verifier can handle it.

3. Find a suitable map of pthread function calls in C to the Boogie extension of point 1.

1.1.4 Terminology

- **Fork**
  This document uses fork multiple times with the following meanings:

  1. fork statement - a Boogie keyword which will be defined in this document
  2. fork branch - a branch in the graph of the program flow, which is induced by a fork statement. This means the branch begins at a location in the CFG corresponding to a fork statement.

- **CFG**
  A control flow graph (CFG).

Chapter 2

Extend the AST: Fork and Join

2.1 Idea

After reading a draft on CIVIL (another attempt to provide a translation of concurrency to Boogie) we decided to introduce two new keywords in Boogie, namely **fork** and **join** to represent concurrency.

2.2 Fork

2.2.1 Syntax

```
fork <expr> <id> <args>;
```

A fork statement consists of the *fork* keyword followed by the following points:

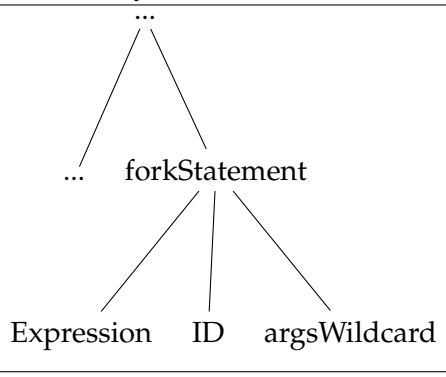
- <expr> - a Boogie expression whose type is int.
- <id> - an function or procedure identifier.
- <args> - arguments to pass to the function or procedure.

2.2.2 Description

A *fork statement* represents - as the name indicates - a branch in the program flow. Semantically the code of <*id*> will be run in parallel to the code following the *fork statement*.

2.2.3 Schematic Translation

Table 2.1: Schematic corelation of a *fork statement* and it's AST representation

C	Boogie	Abstract Syntax Tree
pthread (&thread_id, NULL, someProcedure, someArg)	fork 5 someProcedure someVarAsArg;	

Behoben? I cannot see that this is a translation. From "what" to "what" are you translating? This section is only about an extension of Boogie, I have no idea what we could translate without mentioning C or a CFG

2.3 Join

2.3.1 Syntax

```
1 join <expr>;
```

A join statement consists of the *join* keyword followed by the following point:

"Behoben: followed by the following", points but I can only see one point

- <expr> - a Boogie expression whose type is int. The int value has to match the <expr> value of a *fork* statement in the program.

  Shall join have an option to pass an <id> to save an optional return value?

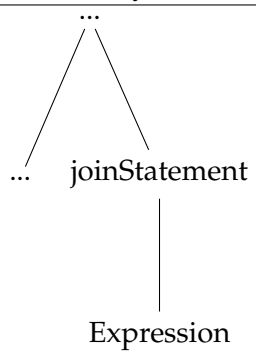
2.3.2 Description

A *join statement* waits for the *fork branch* definied by *<expr>* to finish and as such synchronizes two parallel program flows.

I cannot typecheck this. According to your definition a "fork branch" is a certain structure in the control flow graph - we cannot wait for a structure in a CFG. Maybe we need more terminology and introduce e.g., the notion of a "thread" or a "process"

2.3.3 Schematic Translation

Table 2.2: Schematic corelation of a *join statement* and it's AST representation

Boogie	Abstract Syntax Tree
join 5;	

Chapter 3

Extend the CFG: Fork and Join

Du versuchst hier gleichzeitig eine Erweiterung des CFG und eine Übersetzung von Boogie nach CFG vorzustellen. Das ist sehr schwierig. Vielleicht wäre es besser beides nacheinander vorzustellen.

3.1 Idea

One of the preconditions is, that the verifier recognizes branches in the CFG as parallel program flows. As such we can map the *fork statement* as the begin of a branch in the CFG and a *join statement* as a connection of two branches.

Was sind "preconditions"? Eigentlich verstehe ich den ganzen Abschnitt 3.1 nicht.

3.2 Fork

3.2.1 Description

A *fork statement* is represented in the CFG as a branch of the graph. This is done by two edges starting at the same location L_{start} in the CFG. One edge is called *callee edge* e_{callee} and points to the first location of <*id*> procedure/function L_{proc} (respectively to an helper location, as we will see) and thereby opening a new branch in the CFG which we will call a *fork branch*. The other edge is called *caller edge* e_{caller} and points to the location following the *fork statement*.

Ich denke ich verstehe was du meinst, aber finde die Beschreibung nicht gut.

3.2.2 Fork branch, helper locations, helper variables

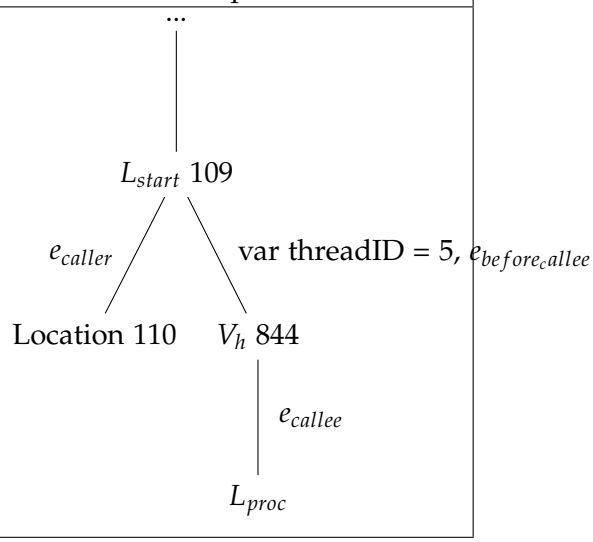
To map concurrency right, the Verifier needs to know the "thread id" (as defined by the <*expr*> part of the *fork statement*) of a *fork branch* in order to use the right *join statement* later (if any). For this reason we need a *helper location* V_h and a *helper*

variable threadID (which can be local in the scope of the fork branch). We add the *helper location* V_h between L_{start} and L_{proc}. The *callee edge* is then $e_{callee} = (V_h, L_{proc})$ and a new edge $e_{before_callee} = (L_{start}, V_h)$ is inserted. The new edge is annotated with "*threadId* = <*expr*>", where "<*expr*>" is the value of the evaluated expression provided in the *fork statement*.

1. Warum eine helper Location? Können wir die Zuweisung auch an e_{caller} schreiben? 2. Warum nicht die besprochene `isThreadActive` Variable? 3. Was ist "scope of the Fork branch"?

3.2.3 Schematic Translation

Table 3.1: Schematic Translation of a *fork statement* into the CFG

Boogie	Control Flow Graph
fork 5 someProcedure someVarAsArg;	

3.3 Join

3.3.1 Description

Du solltest beschreiben was ein Join in einem CFG ist!

A *join statement* introduces a location in the CFG, which can have

can have?

multiple incoming edges. Those edges come from all existing *fork branches*.

Klingt als wolltest du vom Fork eine Kante zum Join ziehen

3.3.2 Fork branch, helper variables, annotations

Incoming edges of *fork branches* are annotated with an "*assume threadID* == <*expr*>"

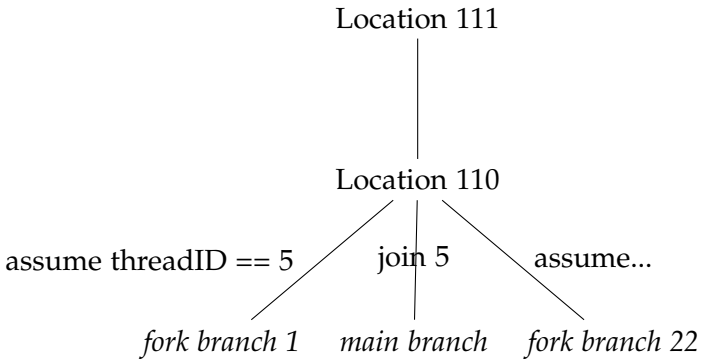
all edges?

where <*expr*> is the value of the evaluated expression of the *join statement*, so that only the right *fork branches* are joined. Another thing the Verifier needs, is a way to know, which join was already used and which not. To accomplish this we need to add a global variable for each *join statement*, initialize with the boolean value *false* and set it to true the moment a *fork branch* can be joined.

Den Zweck dieser globalen Variable verstehe ich nicht. Kannst du das irgendwie ausführlicher beschreiben?

3.3.3 Schematic Translation

Table 3.2: Schematic Translation of a *join statement* into the CFG

Boogie	Control Flow Graph (180 deg. rotated)
join 5;	

Warum gibt es drei eingehende Kanten? Durch was wird dies verursacht? Was ist "fork branch"?

Chapter 4

Example

4.1 Simple Pthread Example

This shows a little example which simulates a run of Ultimate https://preview.overleaf.com/public/wjbnm my changes. Let's take the following C program and see the different steps:

```
extern void __VERIFIER_error() __attribute__ ((__noreturn__));

#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int __VERIFIER_assert(int expression) { if (!expression) { ERROR:
    __VERIFIER_error(); }; return 0; }

int i = 0;

void thread1(void * arg)
{
    i = 5;
}

void thread2(void *arg)
{
if (i == 5) i = 6;
}


int main()
{
    pthread_t t1, t2;

```

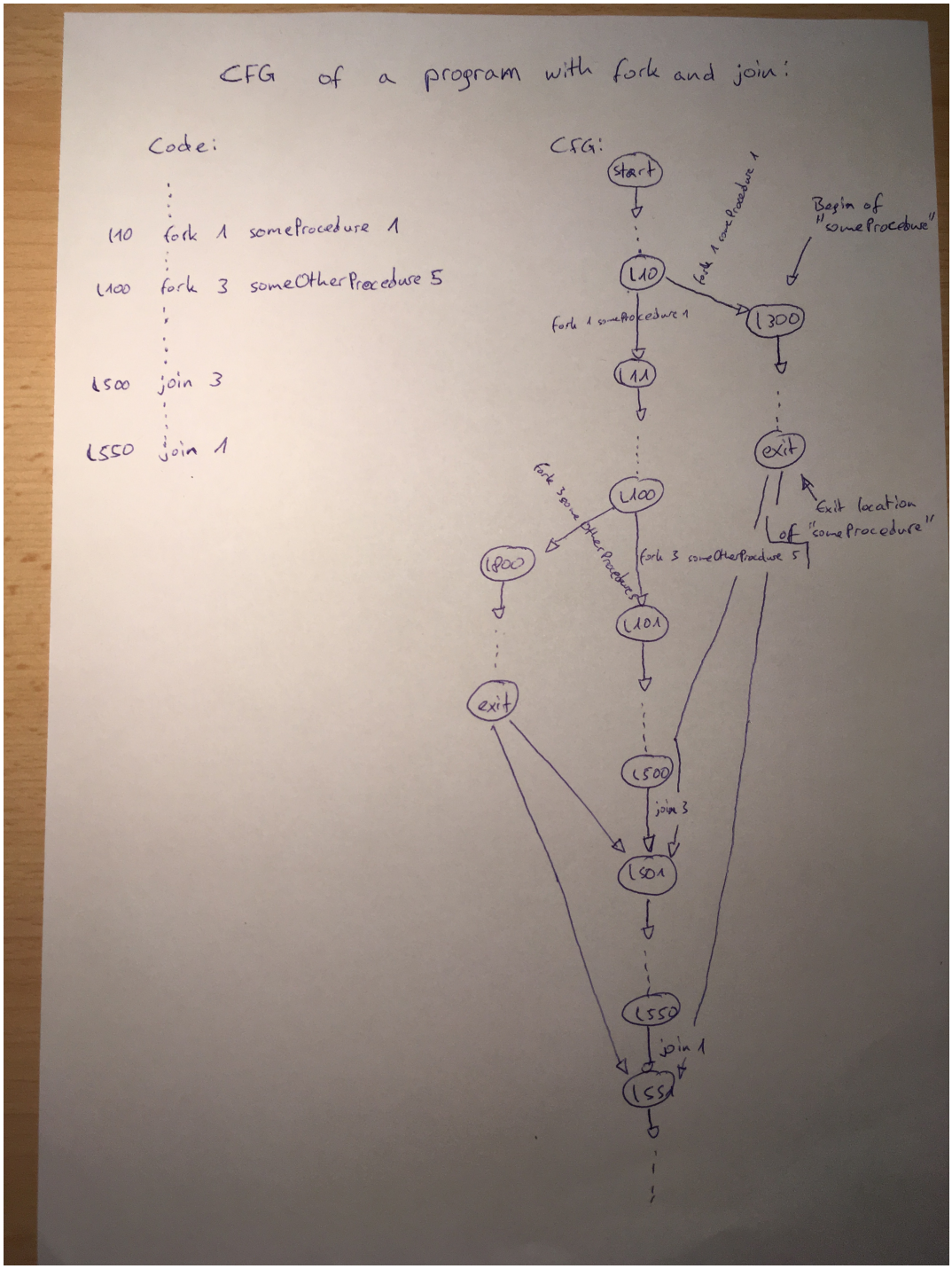
```
26    pthread_create(&t1, 0, thread1, 0);
27    pthread_create(&t2, 0, thread2, 0);
28
29    pthread_join(t2, 0);
30    pthread_join(t1, 0);
31
32    printf("i␣is␣%d", i);
33
34    __VERIFIER_assert(i == 0 || i == 6);
35    return 0;
36 }
```

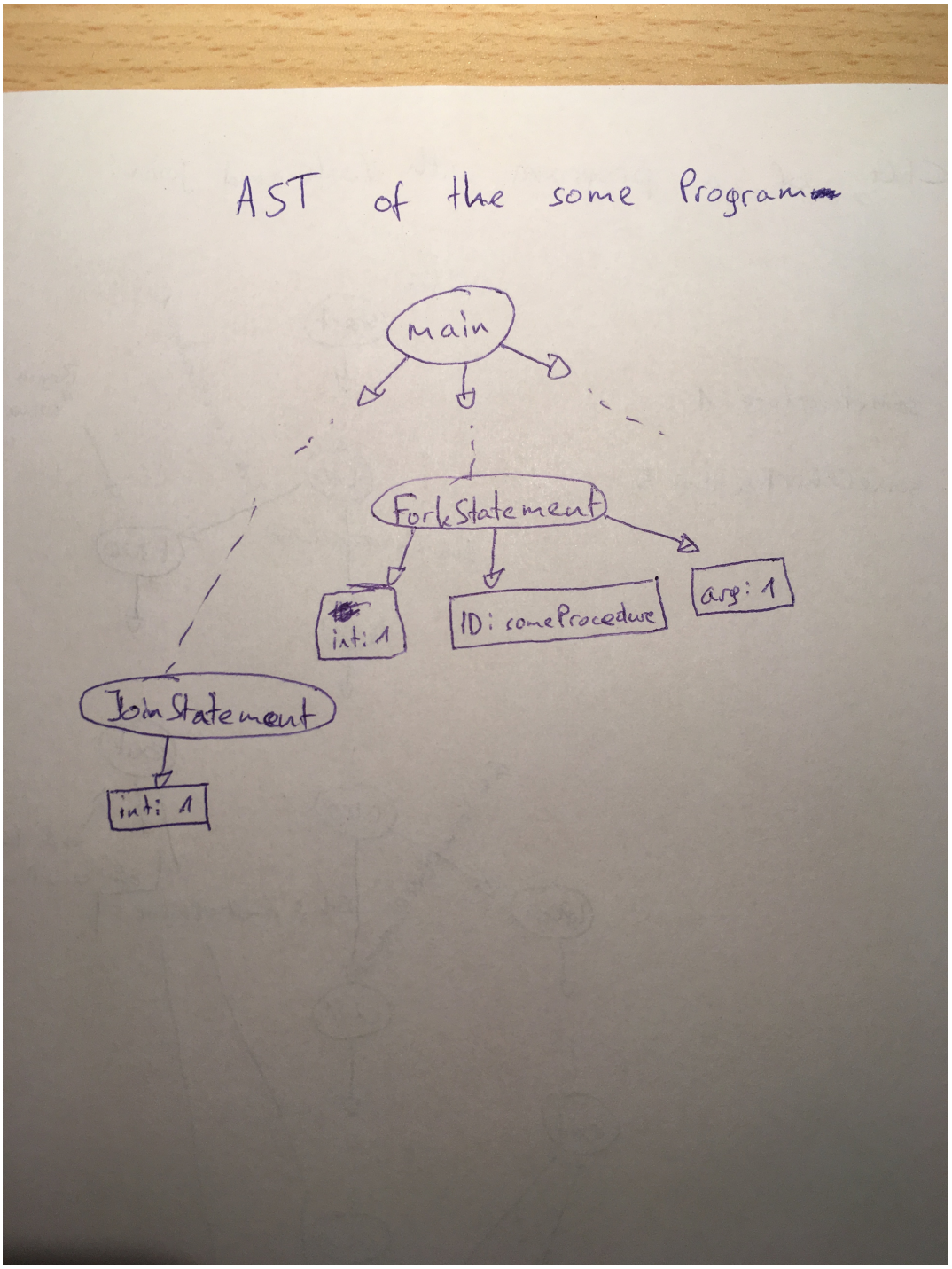
Listing 4.1: Example 1

The pthread to Boogie Translation makes the following of this:

```
1  implementation thread1(#in~x : int) returns ()
2  {
3      var ~x : int;
4
5    $Ultimate##0:
6      ~i~1 := 5;
7      return;
8  }
9
10 implementation thread2(#in~x : int) returns ()
11 {
12     var ~x : int;
13
14   $Ultimate##0:
15           if (i=5) i := 6;
16     return;
17 }
18
19 implementation main() returns (#res : int)
20 {
21   $Ultimate##0:
22     var t1:int, t2:int;
23     t1 := 0;
24     t2 := 1;
25     ~i~1 := 0;
26     fork t1 thread1 0;
27     fork t2 thread2 0;
28     join t1;
```

```
29      join t2;
30      return;
31  }
32
33  var ~i~1 : int;
34
35  procedure thread1(x : int) returns ();
36      modifies ~i~1;
37
38  procedure thread2(x : int) returns ();
39      modifies ~i~1;
40
41  procedure main() returns (#res : int);
42      modifies ~i~1;
43
```


CFG of a program with fork and join:
Code:
L10 fork 1 someProcedure 1
L100 fork 3 someOtherProcedure 5
L500 join 3
L550 join 1
CFG:
start
L10
L300
L11
exit
L100
L800
L101
exit
L500
L501
L550
L551
Begin of "someProcedure"
Exit location of "someProcedure"
fork 1 someProcedure 1
fork 3 someOtherProcedure 5
join 3
join 1


AST of the some Program
main
ForkStatement
int: 1
ID: someProcedure
arg: 1
JoinStatement
int: 1