Implementation of a data-flow analysis for Python scripts in RCE

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informationstechnik

der Dualen Hochschule Baden-Württemberg Mannheim

von

Lukas Rosenbach

Abgabe am September 7, 2021

Bearbeitungszeitraum:	14.06.2021 – 06.09.2021
Matrikelnummer, Kurs:	1922656, TINF18IT1
Abteilung:	Institute for Software Technology: Intelligent and Distributed Systems
Ausbildungsfirma:	German Aerospace Center (DLR)
Betreuer der Ausbildungsfirma:	Dr. rer. nat. Alexander Weinert
Gutachter der Dualen Hochschule:	Dipl.-Inf. Reiner Hüchting

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

THEMA

Implementation of a data-flow analysis for Python scripts in RCE

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

* falls beide Fassungen gefordert sind

Köln, den September 3, 2021

Abstract

This thesis describes the development and implementation of a data-flow analysis to determine certain functional properties of Python scripts in the integration software RCE. RCE is an open source software where engineers and scientists can integrate their own tools and combine them into workflows. Python scripts are used to convert between data types used by RCE and tool-specific formats. These scripts are error prone because the user can miss the conversion of some values in a script. In this thesis we describe a method for automatic analysis of the Python scripts and implement a prototype of this analysis in Java.

Zusammenfassung

Diese Bachelorarbeit beschreibt die Entwicklung und Implementierung einer Datenflussanalyse zur Ermittlung bestimmter funktionaler Eigenschaften von Python-Skripten in der Integrationssoftware RCE. RCE ist eine Open-Source-Software, in die Ingenieure und Wissenschaftler ihre eigenen Tools integrieren und zu Workflows kombinieren können. Zur Konvertierung zwischen den von RCE verwendeten Datentypen und toolspezifischen Formaten werden Python-Skripte verwendet. Da der Benutzer die Konvertierung einiger Werte in einem Skript übersehen kann, sind diese Skripte fehleranfällig. In dieser Bachelorarbeit definieren wir diese Analyse zur Auswertung der Python-Skripte und implementieren einen Prototyp der Analyse in Java.

Contents

List of Figures

List of Tables

Abbreviations

Listings

1. Introduction 1
- 1.1. Context . . . 1
- 1.2. Related Work . . . 2
- 1.3. Problem Definition . . . 2

2. Preliminaries 4
- 2.1. Tool Integration in Remote Component Environment (RCE) . . . 4
- 2.2. Data Flow Analysis . . . 5

3. Analysis Definition 7
- 3.1. Analysis Overview . . . 8
- 3.2. Control Flow Graph Generation . . . 9
- 3.3. Constants Transfer Functions . . . 15
- 3.4. Constant Propagation . . . 17
- 3.5. Control Flow Graph Reduction . . . 19
- 3.6. Output Transfer Functions . . . 22
- 3.7. Output Analysis . . . 23

4. Implementation 25

5. Evaluation 28

6. Conclusion 36

Bibliography 38

List of Figures

2.1. Tool execution in RCE 4
2.2. Control flow graph of Listing 2.1 6

3.1. Analysis flow 9
3.2. Control flow graph of Listing 3.3 12
3.3. Control flow graph of Listing 3.4 14
3.4. Control flow graph with if statement 20
3.5. Reduced control flow graph from Figure 3.4 20
3.6. Control flow graph with for-loop 21
3.7. Reduced control flow graph from Figure 3.6 21

5.1. Control flow graph iteration 0 29
5.2. Control flow graph iteration 1 30
5.3. Control flow graph iteration 2 32

List of Tables

3.1. Meta variables 10

Abbreviations

DLR German Aerospace Center /
Deutsches Zentrum für Luft- und Raumfahrt e. V.

RCE Remote Component Environment

ANTLR ANother Tool for Language Recognition

Listings

2.1. Example script 5

3.1. Example script 1 7
3.2. Example script 2 8
3.3. Example script with an if-statement 12
3.4. Example script with function calls 13
3.5. Example script where an if-statement can be reduced 19
3.6. Example script where a for-loop can be reduced 21

4.1. Pseudocode of data flow analysis algorithm 26

5.1. If-statement nested in for-loop 28
5.2. First reduction of listing 5.1 31
5.3. Second reduction of listing 5.1 33
5.4. While-loop 33
5.5. Simple function in a script 34
5.6. Complex function in a script 34

B.1. Expression Tests 44

1. Introduction

In this chapter we describe the context of this thesis in Section 1.1 followed by discussing related work in Section 1.2 and the problem definition in Section 1.3.

1.1. Context

The German Aerospace Center (DLR) is the German research centre for aeronautics and space and conducts research and development activities in the fields of aeronautics, space, energy, transport, security, and digitalization [8]. It consists of 54 research institutes and facilities at 30 different locations in Germany [8]. This thesis was written at the Institute for Software Technology. This institute does research and development in software engineering technologies, and the incorporation of these technologies into DLR software projects [12]. It consists of the departments Intelligent and Distributed Systems, Software for Space Systems, and Interactive Visualization and High-Performance Computing [12]. The research topics of the department Intelligent and Distributed Systems are applied software technologies for distributed and decentralized software systems, intelligent and knowledge-based software systems, artificial intelligence, software engineering and software analytics, and human factors in software engineering [13].

Remote Component Environment (RCE) [19] is an open source software developed mainly by the Intelligent and Distributed Systems department. According to Boden et al. [6], RCE is an integration environment for engineers and scientists from a wide range of disciplines to integrate software tools, create and execute distributed workflows, and evaluate the obtained results. Users integrate software for calculations,

simulations, and evaluations via a graphical wizard in RCE. These tools are then combined with other components provided by RCE into workflows. Finally, these workflows can be run locally on the users machine or distributed in a network of connected RCE instances.

1.2. Related Work

At the moment of writing this thesis there is no alternative software with the same concept and goal of RCE. However, there are some tools with similarities. A proprietary software similar to RCE is ModelCenter by Phoenix Integration [16]. OpenMDAO [11] is an open-source framework that had some similarities with RCE in the past, but now only focuses on optimization. Finally, Apache Nifi [4] and Knime [15] implement graphical editors like RCE. However, these tools only focus on data science applications.

CPAchecker [7] is a tool for configurable software verification that is also written in Java and utilizes a data flow analysis to find bugs in programs. However, CPAchecker can only analyse programs written in C or Java. BLAST [5] is another tool to verify C programs. An open source tool to analyse Python programs is Pylint [17]. In contrast to the work in this thesis Pylint is implemented in Python.

Noll [21] describes the general concept of data flow analyses in his lecture about static program analysis. The use of data flow analyses in compilers are described by Alfred V Aho et al. [1] in the book "Compilers: principles, techniques and tools". Erich Gamma et al. [10] presents different approaches of handling functions of a program in a data flow analysis.

1.3. Problem Definition

The integration of a tool in RCE is a complex process. Boden et al. [6] describe that first the user has to define inputs and outputs of the tool using the native datatypes of RCE. Then the user provides a console command to start the tool. In addition

to that there is also a script section, where the user can define a Python [18] pre- and post-script. These are executed before and after the tool execution typically to convert the datatypes used by RCE to tool-specific formats and vice versa.

A common source of error are the pre- and post-script. When writing these scripts, it is important to use all inputs and write result values to all outputs. Otherwise, the tool fails to run in a workflow. This is especially costly in terms of time because these errors only appear after running a concrete workflow. This results in a long feedback loop of adjusting the scripts and running a workflow again.

The goal of this thesis is to develop and implement a data-flow analysis for the pre- and post-script to determine if all inputs and outputs are used in the scripts. This could inform the user about errors directly while writing the script and not while executing the tool in a workflow. Of course, this analysis does not catch all sources of error, but by eliminating one main source, it would significantly shorten the feedback loop.

2. Preliminaries

In this chapter we describe fundamentals of this thesis. In Section 2.1 we describe the concept of integrating a tool in RCE. In Section 2.2 we describe the basics of a data flow analysis.

2.1. Tool Integration in RCE

Every software can be integrated in RCE to be used as a tool in a workflow. To integrate a new tool, the user has to define the necessary inputs and outputs. These are information in various formats that the tool needs to execute and returns after execution. Inputs and Outputs can be primitive data types like Boolean, integer or float values or complex data types like text, tables, files or directories. However, every software has other requirements on how to execute it and transfer the input and output values. Because of this reason the user has to define three scripts when integrating a tool. These are the Pre-Execution-Script, the Tool-Execution-Script and the Post-Execution-Script. As shown in Figure 2.1, when a tool is run, first the Pre-Execution-Script is executed to process the provided input values, to be used by the tool. Then the tool is executed with the prepared input values. Finally, the Post-Execution-Script processes the output of the tool, to be used in the workflow as the input values of the next tool.

Figure 2.1.: Tool execution in RCE

2.2. Data Flow Analysis

Noll [21] states that a data flow analysis is a traditional form of program analysis. The idea is to describe how analysis information flows through a program. A data flow analysis can be defined as a system:

$$System = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi) \tag{2.1}$$

The first step of performing a data flow analysis is to label individual relevant locations of a program. These are the labels *Lab*. They label statements that can change the analysed information that flows through the program. Analysed information can for instance be the values of the variables, to determine if they can be replaced with constants, or the availability of expressions. E are the extremal labels that determine where the analysis starts. Next, a control flow graph is created that determines in which order the labels of the program are reached. One label can lead to and can be reached by multiple other labels representing branching in the program. The connections of the nodes are defined by the flow relation F. Listing 2.1 and Figure 2.2 show an example script with its corresponding control flow graph. Here, all variable assignments and if clauses are labelled.

```
1 a = 1
2 b = 2
3 if a > b:
4   c = 3
5 else:
6   c = 4
7 d = 5
```

Listing 2.1: Example script

The complete lattice $(D, \sqsubseteq)$ defines the analysed information. It comprises functions that map the outputs to their state and ensures that the analysis always terminates. ι are the extremal values which define the state at the start of the analysis. φ is the monotonic transfer function, which defines how the labelled parts of a script affect the variables which are being analysed.

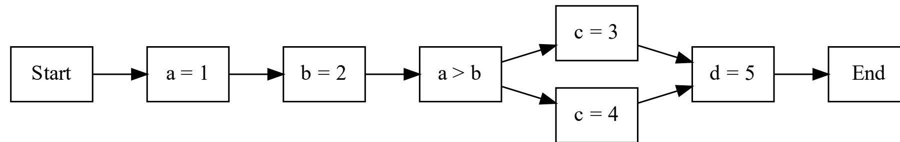

Figure 2.2.: Control flow graph of Listing 2.1

With the definition of the data flow system *System*, the actual analysis is executed. For each label an equation is created that describes how the labelled part of the program affects the analysed information. These equations are then combined into an equation system. Then the equation system can be solved by fix point iteration. The result is a concrete information state at each label of the program.

When defining a data-flow analysis one has to define a grammatical representation of the program's language, which parts of a program are labelled, which information is analysed, the original condition, and whether the analysis is performed forwards or backwards through the program. We define such data-flow analyses to analyse the pre- and post-script in the next chapter.

3. Analysis Definition

This chapter gives a formal definition of an analysis solving the problem described in Chapter 1. The straightforward approach would be to simply search the script for commands that read an input value or write to an output value and check whether all inputs and outputs are used. Unfortunately, this is unsuitable because the analysis of the scripts has to consider the concrete values of each variable and the control flow structure of the script.

For instance, Listing 3.1 shows a post script that contains commands to write to the outputs `output_a` and `output_b`. Assuming `output_a` and `output_b` are the only outputs, a simple search would conclude that every output is used by the script. However, when the script is executed, not all outputs are used if `x` is not in the interval between 0 and 3.

```
1 if x > 0:
2   RCE.write_output("output_a", 42)
3 if x < 3:
4   RCE.write_output("output_b", 42)
```

Listing 3.1: Example script 1

Furthermore, the example in Listing 3.2 shows that the values of the variables in the script have to be evaluated to determine which outputs are used. In this example a simple search would conclude that not all outputs are used by the script, while in reality values are written to all outputs.

```
x = "output_a"
y = "output_b"
RCE.write_output(x, 42)
RCE.write_output(y, 42)
```

Listing 3.2: Example script 2

In this chapter we define a data-flow analysis that solves these problems. Because the analysis of the pre- and post-script are very similar, the remaining part of this thesis will only focus on post-scripts for the sake of readability.

3.1. Analysis Overview

To simplify the process of analysing the post-script, we divide the analysis into six parts as shown in Figure 3.1. The first step is to generate the control flow graph from the script (see Section 3.2). After that, we analyse all functions in the script to determine their transfer function which will be used in the next step (see Section 3.3). These transfer functions describe how the values of the variables in the script are changed by the functions defined in the script. The third step is a constant propagation analysis to find constants for variables in the script (see Section 3.4). Next, we reduce the control flow graph if possible where constant values restrict the flow of the script (see Section 3.5). In this step we remove dead branches of if-statements and unroll for-loops to improve the analysis results. The three steps constants transfer functions, constant propagation and control flow graph reduction are repeated until the control flow graph cannot be reduced any more. This process always terminates, because with each iteration the number of if-statements and for-loops in the script is reduced until eventually all if-statements and for-loops are removed or the reduction of the remaining if-statements and for-loops is not possible. Finally, in the last two steps we analyse which outputs are written by the script. First the transfer functions for the output analysis in the last step are determined (see Section 3.6). These transfer functions describe which outputs are written by the functions based on their parameters. Then the output analysis is executed to determine the written outputs of the script (see Section 3.7).

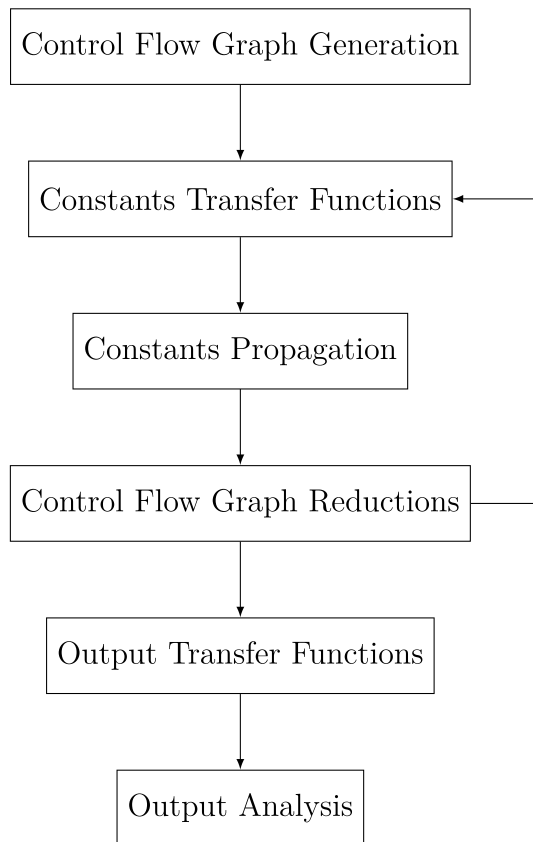

Figure 3.1.: Analysis flow

3.2. Control Flow Graph Generation

The first step of generating the control flow graph from a Python script is to parse it. Because of the scope of this thesis, the analysis can only handle a subset of python syntax. We define the syntax with the following grammar. In the grammar we label statements that are important for the analysis with $[...]^l$:

$$A ::= r \mid x \mid B \mid L\texttt{[}A\texttt{]} \mid \texttt{(}A\texttt{)} \mid A_1\texttt{+}A_2 \mid A_1\texttt{-}A_2 \mid A_1\texttt{*}A_2 \mid A_1\texttt{/}A_2 \in AExp \tag{3.1}$$

$$B ::= b \mid x \mid L\texttt{[}A\texttt{]} \mid \texttt{(}B\texttt{)} \mid A_1\texttt{==}A_2 \mid A_1\texttt{!=}A_2 \mid A_1\texttt{>}A_2 \mid A_1\texttt{<}A_2 \mid A_1\texttt{>=}A_2 \mid A_1\texttt{<=}A_2 \mid \texttt{not } B \mid B_1 \texttt{ and } B_2 \mid B_1 \texttt{ or } B_2 \in BExp \tag{3.2}$$

$$C ::= [x\texttt{=}A]^l \mid [x\texttt{=}S]^l \mid [x\texttt{=}B]^l \mid [x\texttt{=}L]^l \mid [x\texttt{= functioncall(...)}]^l \mid C_1\, C_2 \mid [\texttt{RCE.write_output(}S\texttt{,}S\texttt{)}]^l \mid [\texttt{def function(...)}]^l \mid \texttt{if } [B]^l : C_1 \texttt{ else } C_2 \mid \texttt{while}([B]^l)\texttt{:}C \mid \texttt{for } [x \texttt{ in } L]^l\texttt{:}C \in Cmd \tag{3.3}$$

$$S ::= s \mid x \mid A \mid B \mid L\texttt{[}A\texttt{]} \mid \texttt{(}S\texttt{)} \mid s\texttt{+}s \in SExp \tag{3.4}$$

$$L ::= l \mid x \mid \texttt{[}M\texttt{]} \mid L\texttt{+}L \in LExp \tag{3.5}$$

$$M ::= A\texttt{,}M \mid B\texttt{,}M \mid S\texttt{,}M \mid A \mid B \mid S \tag{3.6}$$

Furthermore, we use the following meta variables in the definition of the analysis:

Category	Domain	Meta variable
Numbers	$\mathbb{R}$	r
Truth values	$\mathbb{B} = \{true, false\}$	b
Strings	$\mathbb{S}$ = Text	s
Lists	$\{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}\}$	l
Variables	$Var = \{x, y, ...\}$	x
Outputs	$Out = \{x, y, ...\}$	o
Arithmetic expressions	$AExp$ (see Formula 3.1)	A
Boolean expressions	$BExp$ (see Formula 3.2)	B
String expressions	$SExp$ (see Formula 3.4)	S
List expressions	$LExp$ (see Formula 3.5)	L
Commands (statements)	Cmd (see Formula 3.3)	C

Table 3.1.: Meta variables

The selected subset of Python includes basic arithmetic operations with the five data types Boolean, integer, float, string and list. Values can be assigned to variables and outputs can be written. Also, the result of a function call can be assigned to a variable. The analysis includes three control structures: while-loops, for-loop

and if-else-statements. To simplify the analysis each if-statement is followed by an else-statement and there is no elif-statement. More complex if-structures can be trivially transformed into a series of if-else-statements.

In the defined Python grammar statements are marked with a label $[...]^l$. For each statement with a label a corresponding node is created in the control flow graph. The nodes are connected according to this flow function:

$$
\begin{aligned}
flow([x = A]^l) &:= \emptyset && (3.7)\\
flow([x = \text{functioncall}(...)]^l) &:= \emptyset && (3.8)\\
flow([\text{RCE.write_output}(S, S)]^l) &:= \emptyset && (3.9)\\
flow(C_1\, C_2) &:= flow(C_1) \cup flow(C_2) \\
&\quad \cup (final(C_1) \times \{init(C_2)\}) && (3.10)\\
flow(\text{if } [B]^l : C_1 \text{ else } C_2) &:= flow(C_1) \cup flow(C_2) \\
&\quad \cup (l, init(C_1)) \cup (l, init(C_2)) && (3.11)\\
flow(\text{while}([B]^l) : C) &:= flow(C) \cup (l, init(C)) \\
&\quad \cup (final(C) \times \{l\}) && (3.12)\\
flow(\text{for } [x \text{ in } L]^l : C) &:= flow(C) \cup (l, init(C)) \\
&\quad \cup (final(C) \times \{l\}) && (3.13)
\end{aligned}
$$

We define *init* and *final* as the entry and exit of a flow in a straight forward way to maintain readability. The parse trees of expressions that are part of the statements are saved with the corresponding nodes. With the parse trees the expressions can then be easily evaluated in the analyses. A complete example of a control flow graph with expression trees is shown in Figure 3.2. It was generated from Listing 3.3. The nodes of the control flow graph are represented by rectangles. The nodes of the expression trees are represented by circles.

```
1 a = 1
2 if x > y:
3   b = 2
4   c = 3
5 else:
6   b = 2
7   c = 4
8 d = 5
```

Listing 3.3: Example script with an if-statement

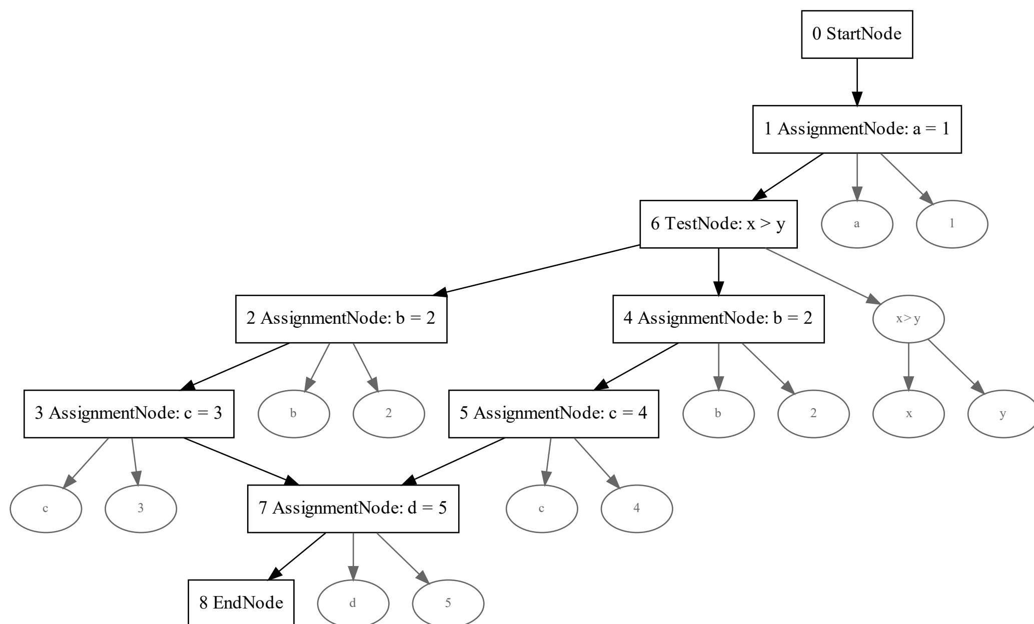

Figure 3.2.: Control flow graph of Listing 3.3

The analysis also takes into account functions defined in the script with the statement "`def function_name(...):`". For these, separate control flow graphs are created. Then the function call nodes are connected with special edges to the graphs of the functions. Here a function call node is split into two nodes that are connected through the function graph. The first node has a special edge to the start node of

the function. the second node has a special edge to the end node of the function. An example control flow graph with function calls is shown in Figure 3.3. It was generated from Listing 3.4. Special edges are represented with dotted arrows. Lines without arrows show which function call node parts belong together.

```
a = 1

def testFunction(x, y):
    f = x + y
    RCE.write_output(f, 1)
    return 2

b = 1
out1 = testFunction(a + 1, b)
out2 = testFunction(3, 4)
c = 1
```

Listing 3.4: Example script with function calls

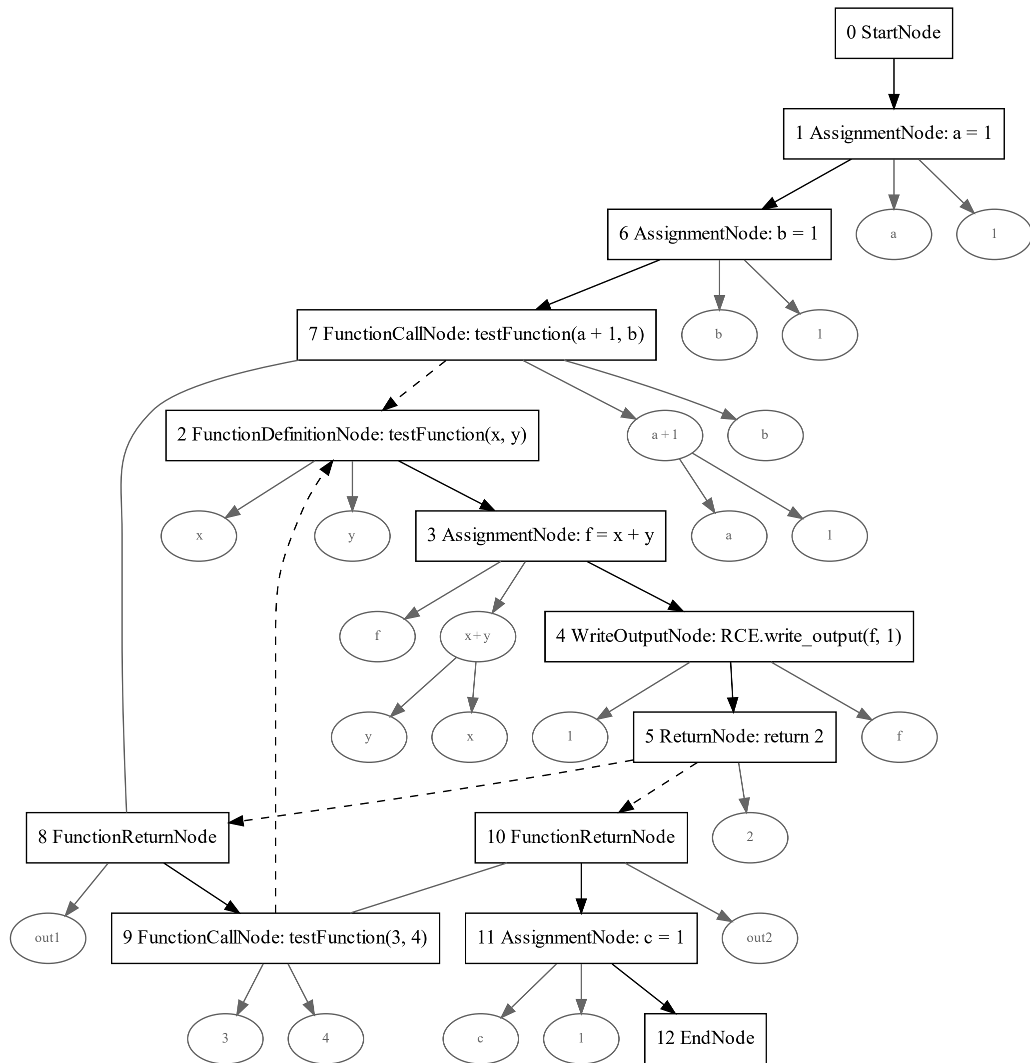

Figure 3.3.: Control flow graph of Listing 3.4

In the next section we will define the second part of the analysis where the functions in the script are analysed.

3.3. Constants Transfer Functions

The second step of the analysis determines the constants transfer functions. These are necessary for the next step of the analysis, because the constant propagation analysis has to take functions into account. Sharir and Pnueli [20] show that the functions of a script can be analyses separately from the script to reduce complexity. In this step we analyse the functions of the script, how they change the variables in the script based on their parameters. Then in the next step, we can use the transfer functions to analyse the whole script.

This transfer function analysis is defined as a data-flow system as described in Section 2.2:

$$System = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi) \tag{3.14}$$

The generation of the control flow graph is described in Section 3.2. The program labels Lab and the flow relation F are defined in Section 3.2. E are the extremal labels. For the constants transfer function analysis E is defined as

$$E := \{init(C)\} \tag{3.15}$$

This means that the analysis start at the beginning of the script.

$(D, \sqsubseteq)$ defines a complete lattice required for the analysis.

$$D := \{\delta \mid \delta : Var_c \to c \cup \{\bot, \top\}\} \tag{3.16}$$

$$\begin{aligned} f_1 \sqsubseteq f_2 \Leftrightarrow \forall x \in Var_c((f_1(x) = c \Rightarrow (f_2(x) = c) \vee (f_2(x) = \top)) \\ \wedge (f_1(x) = \top \Rightarrow f_2(x) = \top)) \end{aligned} \tag{3.17}$$

for

$$c \in A \cup B \cup S \cup L \tag{3.18}$$

δ is defined as:

$$\delta(x) = a \in A \quad : \text{x is defined by an arithmetic expression} \tag{3.19}$$

$$\delta(x) = b \in B \quad : \text{x is defined by a boolean expression} \tag{3.20}$$

$$\delta(x) = s \in S \quad : \text{x is defined by a string expression} \tag{3.21}$$

$$\delta(x) = l \in L \quad : \text{x is defined by a list expression} \tag{3.22}$$

$$\delta(x) = \bot \quad : \text{x is undefined} \tag{3.23}$$

$$\delta(x) = \top \quad : \text{x is overdefined} \tag{3.24}$$

ι are the extremal values. Here, ι defines the start value of each variable at the beginning of the script. Every variable x initially has an unknown default value:

$$\iota := \delta \in D \text{ where } \delta(x) := \bot \text{ for every } x \in Var_c \tag{3.25}$$

φ is the monotonic transfer function, which defines how the labelled parts of a script affect the variables which are being analysed.

$$\varphi_l(\delta) := \begin{cases} \delta[x \mapsto x, y \mapsto y, ...] & \text{if } B^l := \text{def function_name(x, y, ...):} \\ \delta[x \mapsto replace_\delta(B)] & \text{if } B^l := [x = B] \\ \delta[x \mapsto replace_\delta(A)] & \text{if } B^l := [x = A] \\ \delta[x \mapsto replace_\delta(S)] & \text{if } B^l := [x = S] \\ \delta[x \mapsto replace_\delta(L)] & \text{if } B^l := [x = L] \\ \delta & \text{else} \end{cases} \tag{3.26}$$

The $replace_\delta$ function replaces all variables in an expression with the corresponding variables expression of the current state. For example $replace_\delta(x+y)$ returns $3 \cdot a + 42$, if the current variable states are $x = 3 \cdot a$ and $y = 42$.

3.4. Constant Propagation

The constant propagation analysis is the third step of the analysis and analyses the values of the variables in the script. It uses the results from the constants transfer functions analysis to analyse function calls in the script. The analysis is also defined as a data-flow system as described in Section 2.2:

$$System = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi) \tag{3.27}$$

The generation of the control flow graph is described in Section 3.2. The program labels Lab and the flow relation F are defined in Section 3.2. E are the extremal labels. For the constant propagation E is defined as

$$E := \{init(C)\} \tag{3.28}$$

This means that the analysis start at the beginning of the script.

$(D, \sqsubseteq)$ defines a complete lattice which necessary for the analysis to work.

$$D := \{\delta \mid \delta : Var_c \rightarrow e \cup \{\bot, \top\}\} \tag{3.29}$$

$$\begin{aligned} f_1 \sqsubseteq f_2 \Leftrightarrow \forall x \in Var_c((f_1(x) = c \Rightarrow (f_2(x) = c) \vee (f_2(x) = \top)) \\ \wedge(f_1(x) = \top \Rightarrow f_2(x) = \top)) \end{aligned} \tag{3.30}$$

for

$$e \in \mathbb{B} \cup \mathbb{R} \cup \$ \cup \{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \$\} \tag{3.31}$$

δ is defined as:

$$\delta(x) = b \in \mathbb{B} \qquad \text{:x has a constant truth value} \tag{3.32}$$
$$\delta(x) = r \in \mathbb{R} \qquad \text{:x has a constant numeric value} \tag{3.33}$$
$$\delta(x) = s \in \mathbb{S} \qquad \text{:x has a constant string value} \tag{3.34}$$
$$\delta(x) = l \in 2^{\mathbb{B} \cup \mathbb{R} \cup \mathbb{S}} \qquad \text{:x has a constant list value} \tag{3.35}$$
$$\delta(x) = \bot \qquad \text{:x is undefined} \tag{3.36}$$
$$\delta(x) = \top \qquad \text{:x is overdefined} \tag{3.37}$$

ι are the extremal values. Here ι defines the start value of each variable at the beginning of the script. Every variable x has an unknown default value:

$$\iota := \delta \in D \text{ where } \delta(x) := \bot \text{ for every } x \in Var_c \tag{3.38}$$

φ is the monotonic transfer function, which defines how the labelled parts of a script affect the variables, which are being analysed.

$$\varphi_l(\delta'\delta w) := \begin{cases} \delta'[x \mapsto val_{\delta'}(B)]\delta w & \text{if } B^l := [x = B] \\ \delta'[x \mapsto val_{\delta'}(A)]\delta w & \text{if } B^l := [x = A] \\ \delta'[x \mapsto val_{\delta'}(S)]\delta w & \text{if } B^l := [x = S] \\ \delta'[x \mapsto val_{\delta'}(L)]\delta w & \text{if } B^l := [x = L] \\ \delta'[x \mapsto for_{\delta'}(val_{\delta}(L))]\delta w & \text{if } B^l := [x \text{ in } L] \\ \delta'[p_1 \mapsto a, p_2 \mapsto b, ...]\delta'\delta w & \text{if } B^l := [x = \text{functioncall(a, b, ...)}] \\ \delta[x \mapsto func_{\delta'}(\text{functioncall(...)})]w & \text{if } B^l := [x = \text{functionreturn}] \\ \delta'\delta w & \text{else} \end{cases} \tag{3.39}$$

$p_1, p_2, ...$ are the parameters of the function. The functions val and for are defined in Appendix A.1. The function $func$ calculates the result of the function call by

utilizing the results of the constants transfer functions analysis.

In this analysis a stack of information flows through the script. Every time a function is entered a new element is put on the stack. When a function returns, the top element of the stack is removed.

3.5. Control Flow Graph Reduction

After the constant propagation analysis, we try to reduce the control flow graph to refine the analysis results. There are two cases where a control flow graph can be reduced. The first case are if statements where their conditions are constant values. They can be reduced by removing the invalid branch. Figure 3.4 and 3.5 show an example reduction of an if condition in Listing 3.5.

```
1 x = 1
2 if x > 0:
3   RCE.write_output("y", 1)
4 else:
5   x = 42
```

Listing 3.5: Example script where an if-statement can be reduced

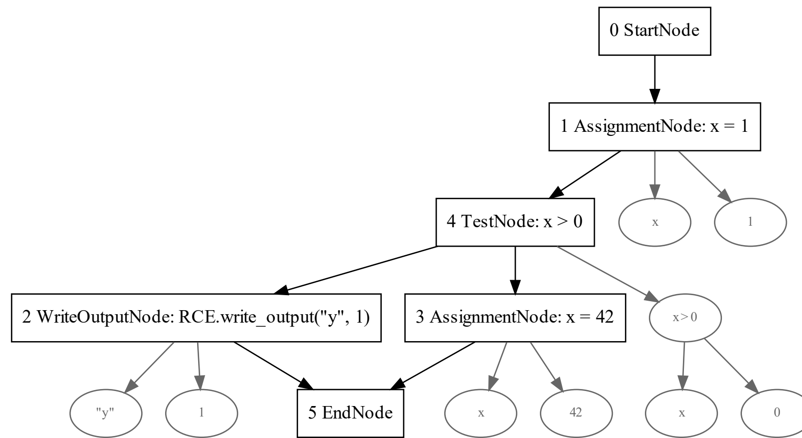

Figure 3.4.: Control flow graph with if statement

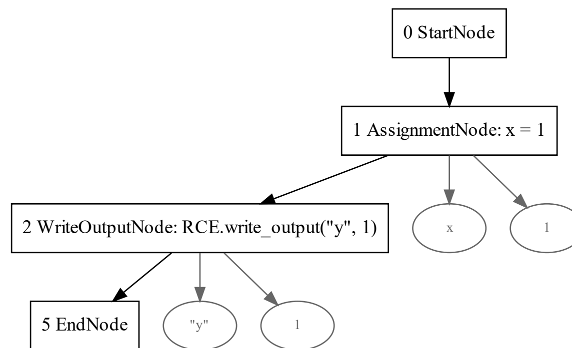

Figure 3.5.: Reduced control flow graph from Figure 3.4

The second case are for-loops where the list to iterate over is constant. Here, the for-loop can be unrolled. Figure 3.6 and 3.7 show an example reduction of a for-loop in Listing 3.6.

```
1 list = ["a", "b", "c"]
2 for out in list:
3   RCE.write_output(out, 1)
```

Listing 3.6: Example script where a for-loop can be reduced

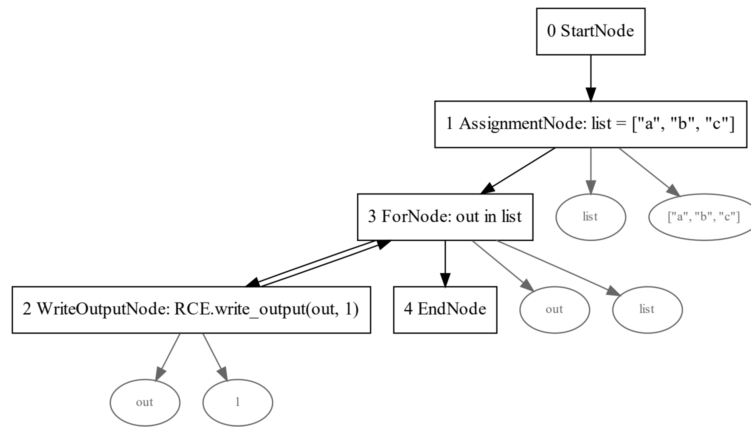

Figure 3.6.: Control flow graph with for-loop

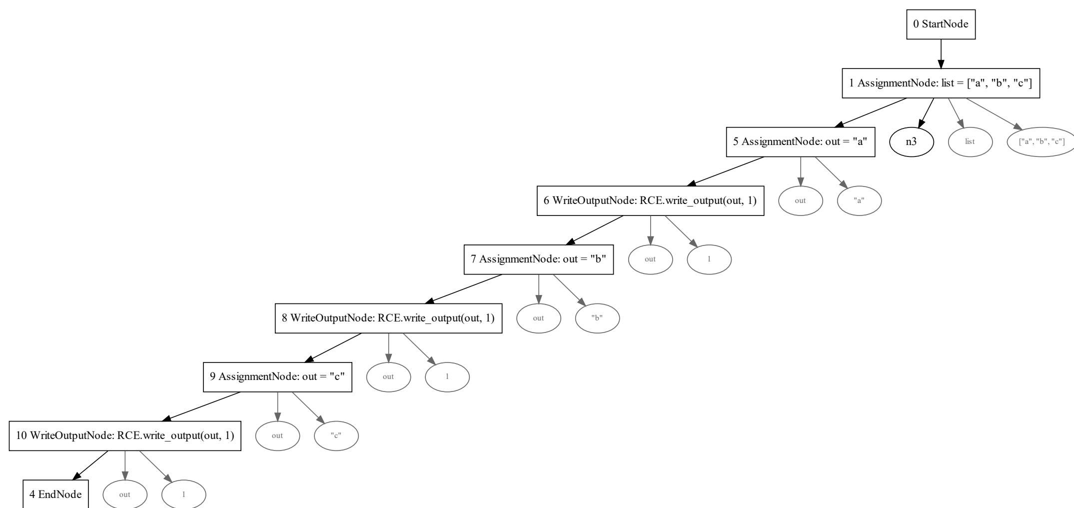

Figure 3.7.: Reduced control flow graph from Figure 3.6

Also, nested if-statements and for-loops can be reduced. If the control flow graph is reduced, the analysis of the constants can be executed again to get more accurate results.

3.6. Output Transfer Functions

To execute the final step of the analysis, the output analysis, first the functions defined in the script have to be analysed for the written outputs. The analysis of the function returns transfer functions describing which outputs the function writes based on its parameters.

This transfer function analysis is defined as a data-flow system as described in Section 2.2:

$$System = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi) \tag{3.40}$$

The generation of the control flow graph is described in Section 3.2. This analysis uses the graph that was already reduced in Section 3.5. The program labels Lab and the flow relation F are defined in Section 3.2.

E are the extremal labels. For the constants transfer function analysis E is defined as

$$E := \{init(C)\} \tag{3.41}$$

This means that the analysis start at the beginning of the script.

The complete lattice $(D, \sqsubseteq)$ is defined as sets of string expressions, that represent the written outputs:

$$(D, \sqsubseteq) := (2^{SExp}, \subseteq) \tag{3.42}$$

At the beginning no output is written:

$$\iota := \emptyset \tag{3.43}$$

The monotonic transfer function adds an string expression to the set of written outputs, when the method `RCE.write_output` is called. When a new function begins the list of written outputs is empty:

$$\varphi_l(\delta) := \begin{cases} \delta \cup s_1 & \text{if } B^l := [\text{RCE.write_output}(s_1, s_2)] \\ \emptyset & \text{if } B^l := [\text{def function_name(x, y, ...):}] \\ \delta & \text{else} \end{cases} \tag{3.44}$$

(3.45)

3.7. Output Analysis

In the final step of the analysis we analyse which outputs are written by the script. The output analysis is based on the result of the constant propagation analysis. Every variable, where the constant propagation analysis found a constant value, is replaced by that value. Like the constant propagation analysis the output analysis is also defined as a data-flow system as described in Section 2.2:

$$System = (Lab, E, F, (D, \sqsubseteq), \iota, \varphi) \tag{3.46}$$

The program labels Lab, extremal labels E, and flow relation F is the same as in the data flow system of the constant propagation analysis. (see Section 3.4)

The complete lattice $(D, \sqsubseteq)$ comprises functions that map the outputs to their state:

$$D := \{\delta \mid \delta : Out_c \rightarrow e \cup \{\bot, \top\}\} \tag{3.47}$$

$$f_1 \sqsubseteq f_2 \Leftrightarrow \forall x \in Out_c((f_1(x) = c \Rightarrow (f_2(x) = c) \vee (f_2(x) = \top)) \\ \wedge (f_1(x) = \top \Rightarrow f_2(x) = \top)) \tag{3.48}$$

for

$$c \in \{\text{written}, \text{not_written}\} \tag{3.49}$$

δ is defined as:

$$\delta(x) = \text{written} \qquad \text{:output x is written} \tag{3.50}$$
$$\delta(x) = \text{not_written} \qquad \text{:ouput x is not written} \tag{3.51}$$
$$\delta(x) = \bot \qquad \text{:the state of output x is undefined} \tag{3.52}$$
$$\delta(x) = \top \qquad \text{:the state of output x is overdefined} \tag{3.53}$$

At the beginning of the analysis no output is written:

$$\iota := \delta \in D \text{ where } \delta(x) := \bot \text{ for every } x \in Out_c \tag{3.54}$$

The monotonic transfer function sets an output to written, when the method `RCE.write_output` is called. At the beginning of the script all outputs are set to not_written:

$$\varphi_l(\delta) := \begin{cases} \delta[s_1 \mapsto \text{written}] & \text{if } B^l := [\text{RCE.write_output}(s_1, s_2)] \\ \delta[\forall x \in Out_c : x \mapsto \text{not_written}] & \text{if } B^l := [\text{start node}] \\ \delta & \text{else} \end{cases} \tag{3.55}$$

4. Implementation

In this chapter we describe the implementation of the analysis defined in Chapter 3. We implemented the analysis in Java [14] with the integrated development environment Eclipse [9].

The first part of the analysis, the generation of the control flow graph, begins with parsing the script. To parse the script we use the framework ANother Tool for Language Recognition (ANTLR) [2]. ANTLR is a general parser generator. Given the Python3 grammar developed by Bart Kiers [3] ANTLR generates a lexer and a parser class in Java, which are used to parse the Python scripts. As described by Alfred V Aho et al. [1], the lexer reads the stream of characters of the script and turns them into tokens. The parser then analyses the tokens and generates a parse-tree. ANTLR also generates a base-visitor class to operate on the parse-tree. This class implements the visitor design pattern. According to Erich Gamma et al. [10], the purpose of the visitor design pattern is to separate the algorithm from the object structure on which it operates. The base-visitor class provides functions for each class in the object structure, that we can specialize for our algorithm. In this case we derive two classes from this visitor that generate the control flow graph and the expression trees of the nodes. We do not need to change the classes that form the parse-tree.

All data flow analyses are implemented in the same way. The general structure of the analysis is shown in pseudocode in Listing 4.1. We implement a class that represents the analysed information state of one node in the graph and a class that combines all these states. These states are initialized as defined in Chapter 3. Then the program iterates over all nodes and updates their states. Here, for each node, we combine

the state of all nodes that lead to this node and apply changes based on the current node.

For example, we run the constant propagation analysis and the current node is an assignment node: `x = 3`. In the current state of the node every variable is undefined. Two nodes lead to the assignment node. In the state of the first node `x` is set to 2 and `y` is set to 1. In the state of the second node `x` is set to 2 and `y` is set to 2. When we combine these two states, we get a new state where `x` is set to 2 and `y` is set to overdefined. Finally, the state is changed by the assignment node and `x` is set to 3. The new state of the assignment node is `x = 3` and `y = overdefined`.

Iteration over all nodes ends when no more state is changed. Then we have reached a fixed-point. The complete lattice, defined in the data flow analysis, ensures that the analysis always terminates. For example, in the constant propagation analysis the state of each node is undefined in the beginning. Over multiple iterations the states are set to a value or overdefined. However, a state of a node that is set to a value can only be changed to overdefined and an overdefined state always stays overdefined. In the extreme case the analysis stops because all states are overdefined and no state can be changed any more.

```
1  def analyse(all_nodes):
2    current_state = new State(all_nodes)
3    state_changed = true
4    while state_changed:
5      state_changed = false
6      for node in all_nodes:
7        current_state.update_state_of_node(node)
8        if state_of_node_changed(node):
9          state_changed = true;
10   return current_state
```

Listing 4.1: Pseudocode of data flow analysis algorithm

For the control flow graph reduction we also iterate over all nodes. If there is an if-node with a constant value we remove the nodes from the unreachable branch. If there is a for-loop-node the node and the nodes of the loop are replaced with an unrolled version of the loop.

In the end we receive an analysis state from the output analysis that stores the state of each output at each node of the control flow graph. Now we can get all outputs that are written at the last node of the script and know whether an output is missing.

5. Evaluation

In this chapter we evaluate the analysis defined in Chapter 3 by demonstrating which scripts the analysis can handle correctly. The analysis can evaluate many expressions. Appendix B.1 is a test script for all the supported expressions. The analysis can also analyse control structures. Listing 5.1 shows a script where an if-statement is nested in a for-loop.

```
1 list = ["a", "b", "c"]
2 i = 0
3 for out in list:
4   if i != 1:
5     RCE.write_output(out, i)
6     i = i + 1
7   else:
8     i = i + 1
```

Listing 5.1: If-statement nested in for-loop

The analysis starts with the control flow graph shown in Figure 5.1. In the first iteration the analysis finds that the for-loop iterates over the constant list `["a", "b", "c"]` and expands the for-loop, as shown in Figure 5.2. A corresponding reduced script is shown in Listing 5.2. Finally, the analysis reduces the if-statements by finding the constant values of the condition, as shown in Figure 5.3. The representation of the reduced graph as a script is shown in Listing 5.3. Now the script is a sequence of statements without branching. First `i` is set to 0 and `out` is set to "a". Then the output `a` is written with the value 0. After that `i` is incremented by one and `out` is set to "b", but output `b` is not written. Finally, `i` is incremented by one again, `out` is set to "c" and output `c` is written with the value 2. In the end `i` is incremented

by one, which is irrelevant for the analysis. The analysis has correctly simplified the control flow graph and determines that the outputs `a` and `c` are written by the script, while the output `b` is not written.

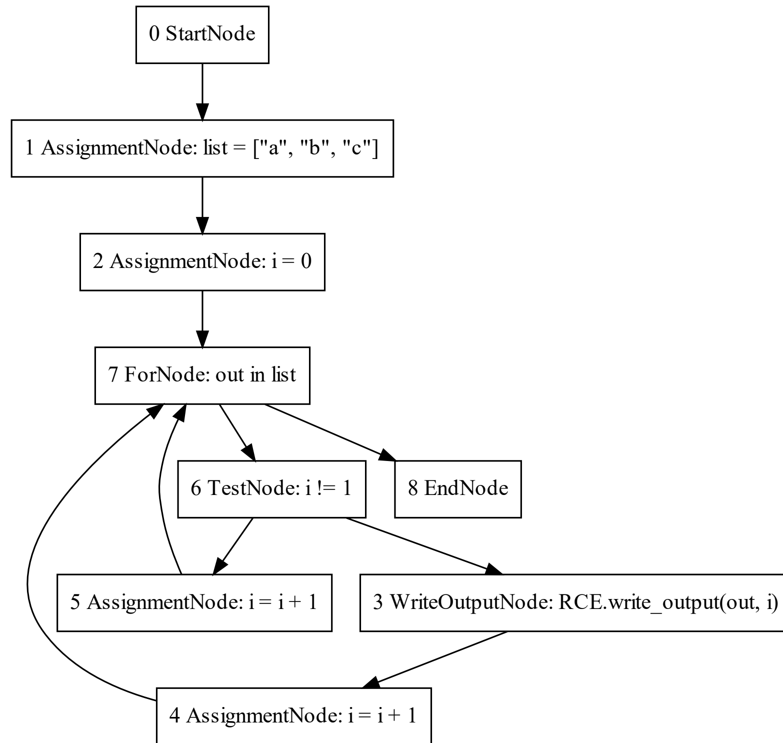

Figure 5.1.: Control flow graph iteration 0

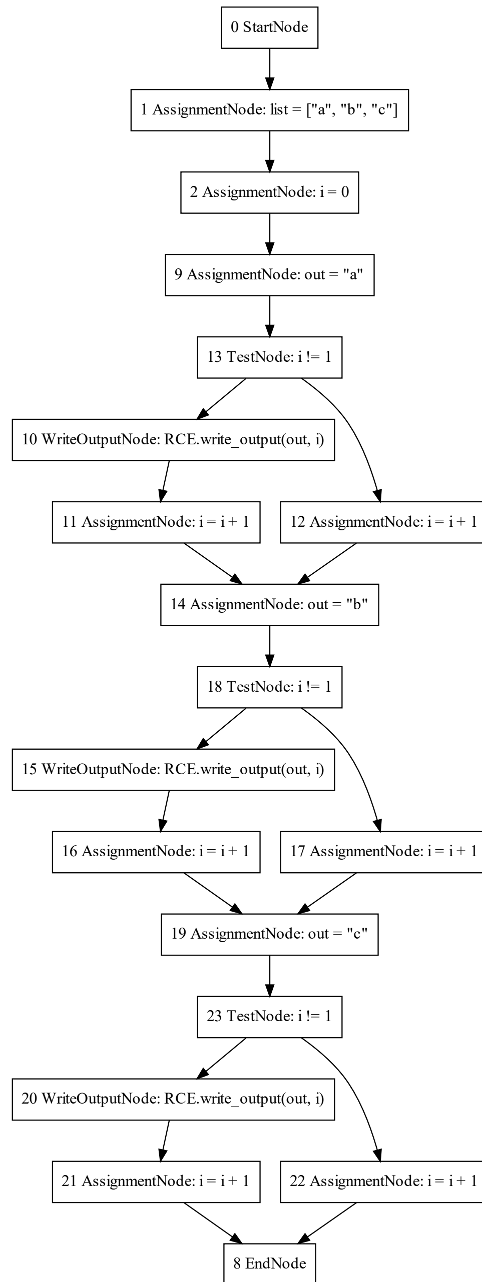

Figure 5.2.: Control flow graph iteration 1

```
1  list = ["a", "b", "c"]
2  i = 0
3  out = "a"
4  if i != 1:
5    RCE.write_output(out, i)
6    i = i + 1
7  else:
8    i = i + 1
9  out = "b"
10 if i != 1:
11   RCE.write_output(out, i)
12   i = i + 1
13 else:
14   i = i + 1
15 out = "c"
16 if i != 1:
17   RCE.write_output(out, i)
18   i = i + 1
19 else:
20   i = i + 1
```

Listing 5.2: First reduction of listing 5.1

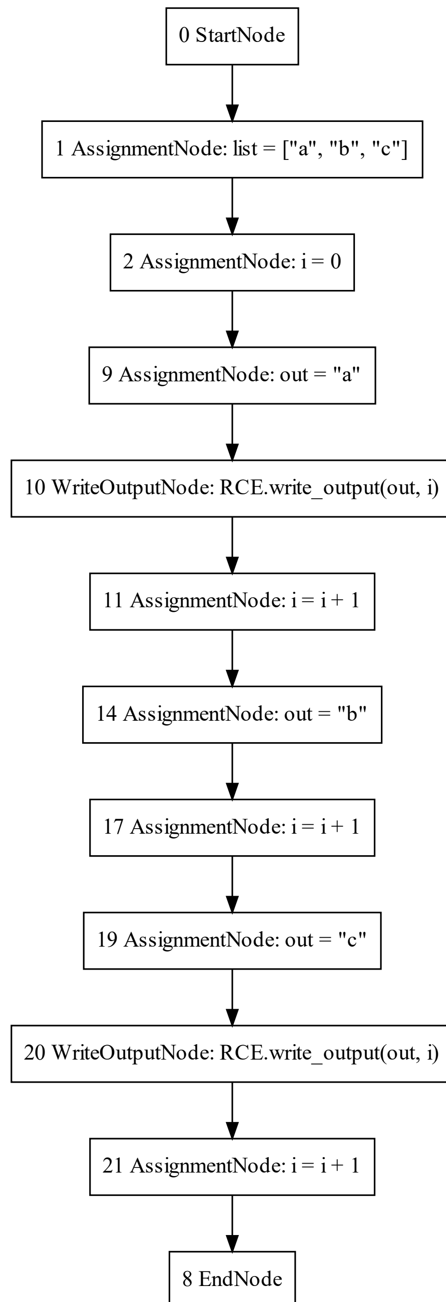

Figure 5.3.: Control flow graph iteration 2

```
list = ["a", "b", "c"]
i = 0
out = "a"
RCE.write_output(out, i)
i = i + 1
out = "b"
i = i + 1
out = "c"
RCE.write_output(out, i)
i = i + 1
```

Listing 5.3: Second reduction of listing 5.1

The analysis can also analyse while-loops. However, with multiple iterations of the loop the iteration variable can change and the analysis results can become less precise. Listing 5.4 shows an example script where the analysis cannot determine the written outputs. Here, the constant propagation analysis cannot find a constant value for `x`. Therefore, the analysis of the written outputs cannot evaluate which output is written in line 3.

```
x = 1
while x < 5:
  RCE.write_output(x, 42)
  x = x + 1
```

Listing 5.4: While-loop

The analysis can also evaluate scripts with simple functions. Listing 5.5 shows an example script. The analysis correctly determines that the function writes an output when called and that the name of the output consists of the prefix string and "out". The analysis gives the correct result, that the outputs `aout`, `bout` and `cout` are written.

```
def write_output(prefix, value):
  out = prefix + "out"
  RCE.write_output(out, value)
  return True

x = write_output("a", 1)
x = write_output("b", 1)
x = write_output("c", 1)
```

Listing 5.5: Simple function in a script

However, more complex functions cannot be fully represented in the analysis, which in turn results in a less accurate result that includes fewer written inputs. Listing 5.6 shows an example script that concerns this limitation. The analysis cannot determine when the function writes and when not. Therefore, the analysis results that no output is written by the script.

```
def write_output(outputname, condition):
  if condition:
    RCE.write_output(outputname, 42)
    return True
  else:
    return False

x = write_output("a", True)
x = write_output("b", False)
x = write_output("c", False)
```

Listing 5.6: Complex function in a script

When the analysis determines a written output, it is guaranteed that this output will be written by the script. This concludes that the analysis is sound. However, if the analysis does not find an output, it is not certain that there is an error in the script. Either the user really made a mistake with the script, or the analysis is not accurate enough. The analysis is sound but incomplete.

We tested the implemented analysis with 26 test-scripts to determine the correctness and performance. These are short test-scripts with an average of seven lines per

script and three or more outputs to analyse. The average runtime of analysing these scripts is $42ms$ with a minimum of $11ms$ and a maximum of $92ms$. Considering that post-scripts in RCE are usually quite short and simple, the analysis developed in this thesis could provide feedback to the user in real time about possible errors in the script. All test-scripts are designed to be analysed correctly. Therefore, they do not test how powerful the analysis is.

6. Conclusion

In this thesis we developed, implemented, and evaluated a prototype to analyse post-scripts in RCE to determine if all outputs are written in the scripts, as described in Chapters 3, 4, and 5. We defined an analysis with six steps. First we parse the script and generate a control flow graph. Then we perform two data flow analyses to find constant values for variables in the script. The first data flow analysis is only applied to the functions defined in the script. The second analysis then uses this information to analyse the whole script. After the constant values are found we reduce the control flow graph by removing dead branches of if-statements and unrolling for-loops. We repeat the steps of finding constant values and reducing the control flow graph until the graph can no longer be reduced. Finally, we perform two more data flow analyses to determine which outputs are written by the script. As with the constant analysis, the first data flow analysis is only applied to the functions while the second analysis is applied to the whole script. At the end of the analysis we obtain a list of outputs that are always written by the script.

We implemented a prototype of the analysis in Java and evaluated it. The prototype shows that the analysis developed in this thesis can analyse a large variety of scripts that include if-else-statements, for-loops, while-loops, functions and various expressions. However, the analysis has its limitations with scripts that include more complex functions and while-loops where variables change with the iterations. Every output the analysis finds is guaranteed to be written, but the analysis does not always find all written outputs. Therefore, the analysis is sound but incomplete. When an output is not found by the analysis, the user should be informed about a possible error in the script. Due to a runtime of the analysis in the range of double-digit milliseconds the analysis can give the user feedback in real time.

To conclude, we successfully developed and implemented a prototype to analyse post-scripts in RCE. The analysis is fast and powerful enough to analyse typical post-scripts written by users in RCE and can give valuable feedback in real time about possible missing outputs.

In the future the prototype developed in this thesis can be expanded and integrated into RCE. The analysis could also be applied to more Python constructs like elif-statements and the remaining missing arithmetic expression, like modulo or integer division. Furthermore, classes defined in the script could be taken into account. In addition to the output analysis, the analysis of the prescript, analysing the used inputs, can also be implemented with the same approach. Finally, the user feedback could be improved by pointing out unsupported syntax or constructs that make the analysis imprecise like while-loops. Also `write_output` statements where the analysis can not find a constant value and in turn can not evaluate which output is written could be marked. In all these feedback cases the user could then adapt the script so that the analysis can determine the correctness of the entire script regarding the written outputs.

Bibliography

[1] Alfred V Aho et al. *Compilers: principles, techniques and tools*. 2020, pp. 4–8.

[2] *ANTLR (ANother Tool for Language Recognition)*. URL: https://www.antlr.org/ (visited on 08/18/2021).

[3] *ANTLR4 grammar for Python 3*. URL: https://github.com/bkiers/python3-parser (visited on 08/18/2021).

[4] *Apache Software Foundation, Cloudera, Hortonworks: Apache Nifi*. URL: https://nifi.apache.org/ (visited on 07/28/2021).

[5] *BLAST: Berkeley Lazy Abstraction Software Verification Tool*. URL: http://mtc.epfl.ch/software-tools/blast/index-epfl.php (visited on 08/02/2021).

[6] Brigitte Boden et al. *RCE: An Integration Environment for Engineering and Science*. 2020. arXiv: 1908.03461 [cs.SE].

[7] *CPAchecker: The Configurable Software-Verification Platform*. URL: https://cpachecker.sosy-lab.org/index.php (visited on 08/02/2021).

[8] *DLR at a glance*. URL: https://www.dlr.de/EN/organisation-dlr/dlr/dlr.html (visited on 01/11/2021).

[9] *Eclipse*. URL: https://www.eclipse.org/ (visited on 08/30/2021).

[10] Erich Gamma et al. *Elements of reusable object-oriented software*. Vol. 99. Addison-Wesley Reading, Massachusetts, 1995, pp. 331–344.

[11] Justin S. Gray et al. “OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization”. In: *Structural and Multidisciplinary Optimization* 59.4 (04/2019), pp. 1075–1104. DOI: 10.1007/s00158-019-02211-z. URL: https://doi.org/10.1007/s00158-019-02211-z.

[12] *Institute for Software Technology*. URL: https://www.dlr.de/sc/en/desktopdefault.aspx/ (visited on 01/11/2021).

[13] *Intelligent and Distributed Systems*. URL: https://www.dlr.de/sc/en/desktopdefault.aspx/tabid-1199/1657_read-3066/ (visited on 01/12/2021).

[14] *Java*. URL: https://www.java.com/en/ (visited on 08/30/2021).

[15] *KNIME AG: KNIME*. URL: https://www.knime.com/ (visited on 07/28/2021).

[16] *Phoenix Integration: Model Center*. URL: https://www.phoenix-int.com/ (visited on 07/28/2021).

[17] *Pylint*. URL: https://pylint.org/ (visited on 08/02/2021).

[18] *Python*. URL: https://www.python.org/ (visited on 08/30/2021).

[19] *RCEnvironment*. URL: https://rcenvironment.de/ (visited on 01/15/2021).

[20] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences . . ., 1978, pp. 189–211.

[21] *Static Program Analysis, Lecture 2: Dataflow Analysis I*. URL: https://moves.rwth-aachen.de/wp-content/uploads/WS1617/spa/l2-handout.pdf (visited on 01/15/2021).

Appendix A.

Constant Propagation

A.1. val and for function

$$val_\delta(x) := \delta(x) \tag{A.1}$$
$$val_\delta(b) := b \tag{A.2}$$
$$val_\delta(r) := r \tag{A.3}$$
$$val_\delta(s) := s \tag{A.4}$$
$$val_\delta(l) := l \tag{A.5}$$

$$val_\delta(B_1 \text{ and } B_2) := \begin{cases} b_1 \wedge b_2 & \text{if } b_1, b_2 \in \mathbb{B} \\ false & \text{if } b_1 = false \text{ or } b_2 = false \\ \bot & \text{if } b_1 = \bot \text{ or } b_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.6}$$

$$val_\delta(B_1 \text{ or } B_2) := \begin{cases} b_1 \vee b_2 & \text{if } b_1, b_2 \in \mathbb{B} \\ true & \text{if } b_1 = true \text{ or } b_2 = true \\ \bot & \text{if } b_1 = \bot \text{ or } b_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.7}$$

$$val_\delta(\text{ not } B_1) := \begin{cases} \neg b_1 & \text{if } b_1 \in \mathbb{B} \\ \bot & \text{if } b_1 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.8}$$

$$val_\delta(A_1 == A_2) := \begin{cases} r_1 = r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.9}$$

$$val_\delta(A_1 != A_2) := \begin{cases} r_1 \neq r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.10}$$

$$val_\delta(A_1 > A_2) := \begin{cases} r_1 > r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.11}$$

$$val_\delta(A_1 < A_2) := \begin{cases} r_1 < r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.12}$$

$$val_\delta(A_1 >= A_2) := \begin{cases} r_1 \geq r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.13}$$

$$val_\delta(A_1 <= A_2) := \begin{cases} r_1 \leq r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.14}$$

$$val_\delta(A_1 + A_2) := \begin{cases} r_1 + r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.15}$$

$$val_\delta(A_1 - A_2) := \begin{cases} r_1 - r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.16}$$

$$val_\delta(A_1 * A_2) := \begin{cases} r_1 \cdot r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ 0 & \text{if } r_1 = 0 \text{ or } r_2 = 0 \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.17}$$

$$val_\delta(A_1 / A_2) := \begin{cases} r_1 / r_2 & \text{if } r_1, r_2 \in \mathbb{R} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.18}$$

$$val_\delta(S_1 + S_2) := \begin{cases} \text{concatinate } s_1 \text{ and } s_2 & \text{if } s_1, s_2 \in \mathbb{S} \\ \bot & \text{if } r_1 = \bot \text{ or } r_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.19}$$

$$val_\delta(L_1 + L_2) := \begin{cases} \text{concatinate } l_1 \text{ and } l_2 & \text{if } l_1, l_2 \in \{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}\} \\ \bot & \text{if } l_1 = \bot \text{ or } l_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.20}$$

$$val_\delta(L_1[A_1]) := \begin{cases} l_1[a_1] & \text{if } a_1 \in \mathbb{R} \text{ and } l_1 \in \{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}\} \\ \bot & \text{if } a_1 = \bot \text{ or } l_1 = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.21}$$

$$for_\delta(l) := \begin{cases} l[0] & \text{if } l \in \{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}\} \text{ and } length(l) = 1 \\ \bot & \text{if } l \in \{[a_1, ..., a_n] \mid n \in \mathbb{N}, a_i \in \mathbb{B} \cup \mathbb{R} \cup \mathbb{S}\} \text{ and } length(l) = 0 \\ \bot & \text{if } l = \bot \\ \top & \text{otherwise} \end{cases} \tag{A.22}$$

for

$$b_1 := val_\delta(B_1) \tag{A.23}$$
$$b_2 := val_\delta(B_2) \tag{A.24}$$
$$r_1 := val_\delta(A_1) \tag{A.25}$$
$$r_2 := val_\delta(A_2) \tag{A.26}$$
$$s_1 := val_\delta(S_1) \tag{A.27}$$
$$s_2 := val_\delta(S_2) \tag{A.28}$$
$$l_1 := val_\delta(L_1) \tag{A.29}$$
$$l_2 := val_\delta(L_2) \tag{A.30}$$

Appendix B.

Evaluation

B.1. Expression Tests

```
1  e01 = 1 + 2                  # 3
2  e02 = "ou" + "t"             # "out"
3  e03 = 2.2 + 4.6              # 6.8
4  e04 = 1.1 + 42               # 43.1
5  e05 = 8.2 + "test"           # "8.2test"
6  e06 = [1, 2.2, "x"] + ["y", 42] # [1, 2.2, "x", "y", 42]
7
8  e07 = 1 - 2                  # -1
9  e08 = 4.4 - 2.2              # 2.2
10 e09 = 42 - 1.1               # 40.9
11
12 e10 = 2 * 3                  # 6
13 e11 = 1.2 * 2.4              # 2.88
14 e12 = 42 * 1.2               # 50.4
15 e13 = 3 * "out"              # "outoutout"
16
17 e14 = 16 / 4                 # 4
18 e15 = 8.8 / 4.4              # 2.0
19 e16 = 10.4 / 2               # 5.2
20
21 e17 = True and True          # True
22 e18 = True and False         # False
23 e19 = False and True         # False
24 e20 = False and False        # False
25
26 e21 = True or True           # True
27 e22 = True or False          # True
28 e23 = False or True          # True
29 e24 = False or False         # False
30
31 e25 = not False              # True
32 e26 = not True               # False
33
34 e27 = 3 == 3                 # True
```

```
35  e28 = 4 == 5                   # False
36  e29 = 3.3 == 3.3               # True
37  e30 = 3.7 == 3.8               # False
38  e31 = "test" == "test"         # True
39  e32 = "test1" == "test2"       # False
40  e33 = True == True             # True
41  e34 = False == False           # True
42  e35 = True == False            # False
43  e36 = [1, 2, 3] == [1, 2, 3] # True
44  e37 = [1, 2] == [1, 2, 3]      # False
45
46  e38 = 3 != 3                   # False
47  e39 = 4 != 5                   # True
48  e40 = 3.3 != 3.3               # False
49  e41 = 3.7 != 3.8               # True
50  e42 = "test" != "test"         # False
51  e43 = "test1" != "test2"       # True
52  e44 = True != True             # False
53  e45 = False != False           # False
54  e46 = True != False            # True
55  e47 = [1, 2, 3] != [1, 2, 3] # False
56  e48 = [1, 2] != [1, 2, 3]      # True
57
58  e49 = 1 > 2                    # False
59  e50 = 2 > 1                    # True
60  e51 = 42 > 1.5                 # True
61
62  e52 = 2 >= 1                   # True
63  e53 = 1 >= 2                   # False
64  e54 = 42 >= 1.5                # True
65  e55 = 3 >= 3                   # True
66
67  e56 = 2 < 1                    # False
68  e57 = 1 < 2                    # True
69  e58 = 42 < 1.5                 # False
70
71  e59 = 2 <= 1                   # False
72  e60 = 1 <= 2                   # True
73  e61 = 42 <= 1.5                # False
74  e62 = 3 <= 3                   # True
```

Listing B.1: Expression Tests