JOHANNES KEPLER
UNIVERSITY LINZ

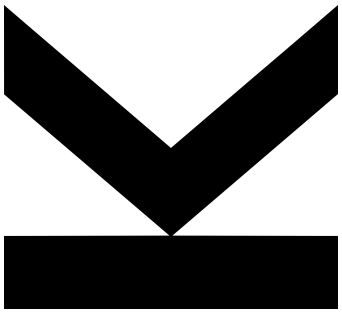
Author / Eingereicht von
Jonathan Kudlich,
Keanu Pöschko
k11818866,
k11824891

Submission / Angefertigt am
Institute for System Software

Thesis Supervisor / First Supervisor / BeurteilerIn / ErstbeurteilerIn / ErstbetreuerIn
a.Univ.-Prof. Dipl.-Ing. Dr. **Herbert Prähofer**

April 28, 2023

A Framework for Static Analysis of IEC 61131-3 Languages



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, April 28, 2023

Jonathan Kudlich,
Keanu Pöschko

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

Programmable Logic Controllers (PLCs) are essential for industrial automation systems that automate complex processes. However, the availability of tools for supporting static analysis of PLC programming languages is significantly limited. To address this issue, this thesis presents a language-independent framework focused on IEC-61131-3. It implements and describes data structures and algorithms that form a basis for analysing IEC-61131-3 based languages. It also implements a concrete application of these data structures in the form of constant folding. The developed tool chain is a significant step in bridging the gap in available tools for the static analysis of PLC programs.

Kurzfassung

Softwareprogrammierbare Steuerungen (SPS) sind essentiell for industrielle Automatisierungssysteme die komplexe Prozesse automatisieren. Die Anzahl von Werkzeugen, die die statische Analyse von SPS Programmiersprachen ermöglichen, ist jedoch stark limitiert. Um diesem Problem entgegenzuwirken, präsentiert diese Arbeit ein sprachunabhängiges Framework das sich auf IEC-61131-3 konzentriert. Sie implementiert und beschreibt Datenstrukturen und Algorithmen, die eine Basis für die Analyse von IEC-61131-3 basierten Sprachen bieten. Sie zeigt außerdem eine konkrete Anwendung dieser Datenstrukturen in der Form von Constant Folding. Die entstandene Toolchain ist ein signifikanter Schritt um die Lücke in Werkzeugen für die statische Analyse von SPS Programmen zu schließen.

April 28, 2023

Jonathan Kudlich,
Keanu Pöschko

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

JOHANNES KEPLER
UNIVERSITY LINZ

List of Figures

JOHANNES KEPLER
UNIVERSITY LINZ

List of Tables

JOHANNES KEPLER
UNIVERSITY LINZ

Listings

JOHANNES KEPLER
UNIVERSITY LINZ

1 Introduction and Motivation

IEC 61631–3 [6] is a standard for languages for Programmable Logic Controller (PLC) Programs. One language within this standard is Structure Text, which is a language similar to Pascal.

Static code analysis is a software engineering technique which allows acquiring information about the structure but also possibly about runtime properties of a program without executing it. In [11, Prähofer et al. 2015] a tool has been presented which supports advanced analysis techniques for PLC programs based on the IEC 61131–3 standard. However, as it relies on a specific dialect of the IEC 61131–3 standard, the tool is not generally available.

Therefore, in this work a new tool environment has been implemented. In contrast to the former, this tool chain supports the language elements of the Structured Text defined in the standard.

These theses have been produced in collaboration with the Software Competence Center Hagenberg, who have developed a framework that is capable of compiling source code into an ASTM-structure. This framework is used in our toolchain to perform the first step of the working process where the analysed code is parsed into ASTM format.

1.1 Context: PLC, IEC 61131

Programmable Logic Controllers (PLCs) are a widely spread tool for industrial automation systems. Many of the PLCs in use today conform by the IEC 61131 standard. Specifically, they also conform to the languages described in the third section of the standard. However, despite the software systems used in industrial programming becoming increasingly large and complex [15], the number of supporting tools for those languages is still severely lacking.

1.2 Approach

The goal of these theses is to present a tool chain for the static analysis of structure and data flow in programs. The presented tool chain starts by translating the programs into an intermediate representation, an Abstract Syntax Tree (AST) based on the AST Metamodel (ASTM) specified by the OMG. The ASTM is a model for representing code from different programming languages in a comprehensive, standardized way. After translating the program code into the ASTM the toolchain extracts useful meta-information from the ASTM and prepares the information for further analysis. The meta-information extracted from the ASTM consists of the Interprocedural Control Flow and Call Graphs, the Memory Model of the project as well as the Data Flow of the program, represented by its Single Static Assignment form and the computation of constant folding in the code. The complete process is detailed in Figure 1.1.

The main focus of these theses lies on the family of languages that are based on the IEC-61131–3 standard, however, one of our requirements is that the analysis should be independent of the used language. While this is not entirely possible, the requirement has been fulfilled to the greatest feasible extent. This means that the presented toolchain is able to perform basic analysis on most supported languages, while detailed analysis is available only for IEC-61131–3 based languages.

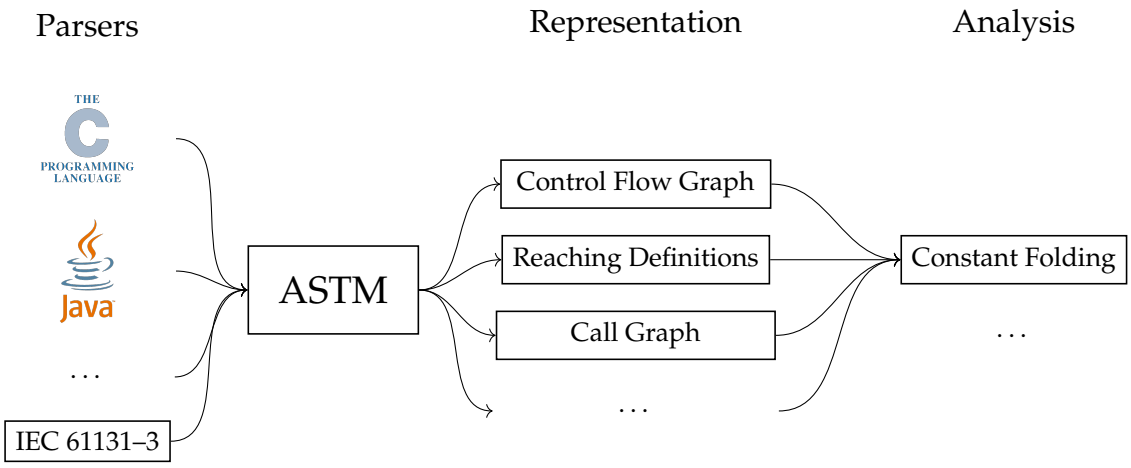


Figure 1.1: Analysis process

Due to this focus on IEC-61131–3 languages combined with limited time and manpower not all features of the supported languages were required to be supported.

1.3 Structure of the Theses

In the following we will outline the structure of the theses.

To start, Chapter 2 gives a short overview on the IEC 61131–3 standard. It then focuses in on the ST language defined in the standard, giving a short overview of the most important syntax elements. Finally, it introduces a short ST program that will be used to illustrate several concepts throughout this paper.

In Chapter 3 the abstract syntax tree metamodel is introduced as an abstract representation for programs of any language. The chapter then outlines the concepts of the metamodel which have the most significant impact on our work. It finishes by introducing the `GASTMSwitch`, a class that has been extensively used in our implementation.

Chapter 4 presents the control flow graph as an analysis tool for programs. It then shows our implementation of a control flow graph and how it is built from the ASTM representation that we use as our basis.

Chapter 5 then introduces the concept of a call graph and shows how our control flow graph is extended with a call graph.

In Chapter 6 a memory model, which represents all instances of a program, is introduced. For this purpose we introduce a tree like data structure called the instance tree.

Chapter 7 introduces an algorithm for analyzing the data-flow of a CFG . Specifically, it introduces the algorithm to compute the reaching definitions of a given program.

Chapter 8 presents the concepts of constant folding and constant propagation. The chapter concludes by showing how constant folding and constant propagation were implemented as part of our data-flow analysis.

Finally, Chapter 9 recapitulates what we have accomplished in these theses and gives a short outlook on potential future work.

1.4 Joint work of Bachelor applicants

This is a joint work by Jonathan Kudlich and Keanu Pöschko. Thus, the present report represents the Bachelor theses of the two Bachelor applicants where

- Chapters 4 and 5 represents the sole work of Jonathan Kudlich and
- Chapters 6, 7, 8 represents the sole work of Keanu Pöschko.

The introductory Chapters 1, 2, and 3 and the conclusion in Chapter 9, however, have been written in cooperation.

2 IEC 61131–3 Programs

The IEC 61131 standard [6] was released by the IEC in 1993 [7]. It is a widely accepted standard in the industry. in its third section, the standard defines five languages in which PLCs can be programmed. Of these, two are textual languages, and three visual languages:

- Instruction List (IL)

  Instruction List programs consist of assembly-style instructions that define the PLCs behavior.

- Structured Text (ST)

  Structured Text programs look similar to languages like Pascal and C and provide the ability to write code on a somewhat higher level than IL . The syntax is outlined in more detail in Section 2.1.

- Function Block Diagram (FBD)

  Function Block Diagrams are programs that focus of the flow and interactions of data within a program unit.

- Ladder Diagram (LD)

  Ladder diagrams are visual programs that model the behavior of the PLC as an electrical circuit.

- Sequential Function Chart (SFC)

  Sequential Function Charts are visual programs that focus on the order of the steps that a PLC goes through as it is executing its task.

2.1 Structured Text (ST)

Structured Text is the high level textual programming language defined in the IEC 61131–3 standard. Its syntax shows strong similarities to that of Pascal. These similarities as well as key differences between the languages explored in [13].

Like most high level programming languages, ST provides the usual control structures of `IF` and `ELSIF`, `WHILE`, and `FOR` for conditions and loops. Additionally, it provides a `REPEAT ... UNTIL` construct which is similar to Cs `do ... while` loops. The control structures and their respective meanings are listed in table 2.1.

The main structural unit in a PLC program are called program organisation units (POUs). These exist in several variations:

- **Function Blocks** (FBs) are structures that have internal fields and can be instantiated, similar to a class in Java. Unlike classes however, they do not define methods that can be called arbitrarily called. Instead, they define a single procedure body that is executed when the FB is called.
- **Functions** are procedures similar to those found in other high level programming languages. They can have both input, output, and inout parameters, as well as local variables.
- **Programs** behave like normal procedures but act as the main entry point into an ST program.

Listing 2.1: If-then-else statement.

```
1 IF <condition1> THEN
2     <statements1>
3 ELSIF <condition2> THEN
4     <statements2>
5 ELSE
6     <statements3>
7 END_IF;
```

`<statements1>` is executed if `<condition1>` is true, `<statements2>` is executed if the first condition is false and `<condition2>` is true. If neither condition is true, `<statements3>` is executed.

Listing 2.2: While loop.

```
1 WHILE <condition> DO
2     <statements>
3 END_WHILE;
```

If `<condition>` is true, `<statements>` is executed until `<condition>` is false.

Listing 2.3: For loop.

```
1 FOR <variable> := <start> TO <
      stop> BY <change> DO
2     <statements>
3 END_FOR;
```

`<variable>` starts counting at `<start>` and changes by `<change>` after each execution of `<statements>` until its value changes beyond `<stop>`

Listing 2.4: Repeat-until loop.

```
1 REPEAT
2     <statements>
3 UNTIL <condition>;
```

`<statements>` is executed and repeated as long as `<condition>` remains true.

Listing 2.5: Switch-case statement.

```
1 CASE <variable> OF
2     <v1>, <v2>: <statements1>
3     <v3>      : <statements2>
4 ELSE
5     <statements3>
6 END_CASE;
```

If `<variable>` equals either of `<v1>` or `<v2>`, `<statements1>` is executed. Otherwise, if `<variable>` equals `<v3>`, `<statements2>` is executed. If none of those options is taken, `<statements3>` is executed.

Table 2.1: Control flow structures of structured text.

Listing 2.7 in the next section gives an example for how an FB would be defined in ST . As can be seen there, variables in a POU are defined using the `VAR`, `VAR_INPUT`, `VAR_OUTPUT`, and `VAR_INOUT` keywords before the start of the procedure body.

ST also allows the definition of custom data structures using the `STRUCT` and `TYPE` keywords:

Listing 2.6: Example for a custom datatype.

```
1 TYPE <name> :
2     STRUCT
3         <name> : <type>;
4         <name> : <type>;
5         <name> : <type>;
6     END_STRUCT;
7 END_TYPE
```

2.2 Running Example

In this chapter we will present an example program, that will be used throughout the following chapters. Our running example consists of three files: `LevelControl.st`, `Pump.st` and `Sensor.st`. Together they model a control system for a storage tank or similar appliance. The code for the three files is shown in Listings 2.7 through 2.9.

Listing 2.7 shows the code of the `LevelControl` POU. This class is the top unit and models the control logic of the system. It defines two sensors that are responsible for detecting a high and low level in the tank, as well as a pump that can be used to actively refill the tank. Finally, it also defines a boolean variable that models whether the tank is currently being filled or emptied. During execution, it waits for the tank to be emptied and activates the pump once the low sensor triggers its signal. It then waits for the tank to be refilled by the pump, which is signalled by

the high sensor. Once the tank is refilled, the pump is deactivated and the cycle starts anew.

Listing 2.7: Content of LevelControl.st

```
1  FUNCTION_BLOCK LevelControl
2  VAR
3    pump       : Pump;
4    highSensor : Sensor;
5    lowSensor  : Sensor;
6    fill       : BOOL := FALSE;
7  END_VAR
8  IF fill THEN
9    pump(TRUE);
10   highSensor(TRUE, 100, 0);
11   lowSensor(FALSE, 10, 0);
12   IF highSensor.state THEN
13     fill := FALSE;
14   END_IF;
15 ELSE
16   pump(FALSE);
17   highSensor(FALSE, 100, 0);
18   lowSensor(TRUE, 10, 0);
19   IF NOT lowSensor.state THEN
20     fill := TRUE;
21   END_IF;
22 END_IF;
23 END_FUNCTION_BLOCK
```

Listing 2.8 shows the source code of the `Pump` POU. This POU is kept very simple, turning a single output on and off, depending on its input signal. The input signal is a simple boolean which indicates whether the pump should be turned on, while the output is a floating point number which is set to one if the input is true and zero otherwise.

Listing 2.8: Content of Pump.st

```
1  FUNCTION_BLOCK Pump
```

```
2  VAR_INPUT
3    on       : BOOL;
4  END_VAR
5
6  VAR_OUTPUT
7    out    : REAL;
8  END_VAR
9
10 IF on = TRUE THEN
11   out := 1.0;
12 ELSE
13   out := 0.0;
14 END_IF;
15 END_FUNCTION_BLOCK
```

Listing 2.9 shows the source code of the `Sensor` POU. The sensors can be configured with a threshold and a level or disabled by setting their respective active state to false. Each sensor outputs its state when queried, which is a boolean value that indicates if the level is greater than the threshold. If the sensor is disabled, it will simply output false.

Listing 2.9: Content of Sensor.st

```
1  FUNCTION_BLOCK Sensor
2  VAR_INPUT
3    active     : BOOL;
4    threshold  : REAL;
5    level      : REAL;
6  END_VAR
7
8  VAR_OUTPUT
9    state      : BOOL;
10 END_VAR
11
12 IF active THEN
13   IF level > threshold THEN
14     state := TRUE;
```

```
15    ELSE
16      state := FALSE;
17    END_IF;
18  ELSE
19    state := FALSE;
20  END_IF;
21  END_FUNCTION_BLOCK
```

3 ASTM Abstract Syntax Tree Representation

An abstract syntax tree (AST) is a common model for representing and manipulating code in memory [9]. Our project uses the Abstract Syntax Tree Metamodel (ASTM) defined by the OMG [4]. This defines the ASTM as consisting of two parts:

- The Generic Abstract Syntax Tree Metamodel (GASTM) that defines concepts common to all programming languages described by the ASTM.
- Several Language Specific Abstract Syntax Tree Metamodels (SASTMs) which define language specific extensions to the GASTM.

Our project uses a framework called eKNOWS, which is written and maintained by the SCCH[1]. It is based on Eclipse MoDisco, a framework by the Eclipse Foundation that includes a GASTM implementation [8]. The eKNOWS framework extends this GASTM with a SASTM for structured text.It also provides several frontends that can parse source files of supported languages and convert them into their ASTM representation.

Figures 3.1, 3.2, 3.3 and 3.4 show some relevant ASTM classes and their relations to each other. Figure 3.1 shows the top level structure of the ASTM. `GASTMObject` is the ancestor class of all other classes within the ASTM and fulfills a similar

[1] `https://www.scch.at/software-science/projekte/detail/eKNOWS`

role to the one `Object` has in Java. `GASTMSemanticObject` and `GASTMSourceObject` contain classes that model meta information and are expanded in figures 3.2 and 3.3, respectively. `GASTMSynatxObject` contains the classes that model the program structure described by the code. This class is the most relevant to our project and is expanded in figure 3.4.

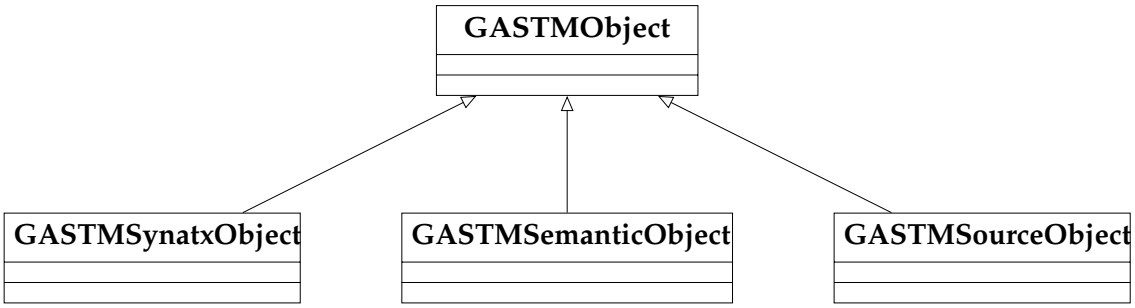


Figure 3.1: GASTMObject adopted from [4, p. 59]

Figure 3.2 shows the child classes of `GASTMSemanticObject`. These consist of the `Project` class and several variants of scopes. A `Project` is the largest organisational unit and what our framework expects as input. `Scope` and its subclasses can be used to find definitions of classes and variables by name. However, this functionality was not fully supported by the framework at the time of our project, so we were unable to rely on it.

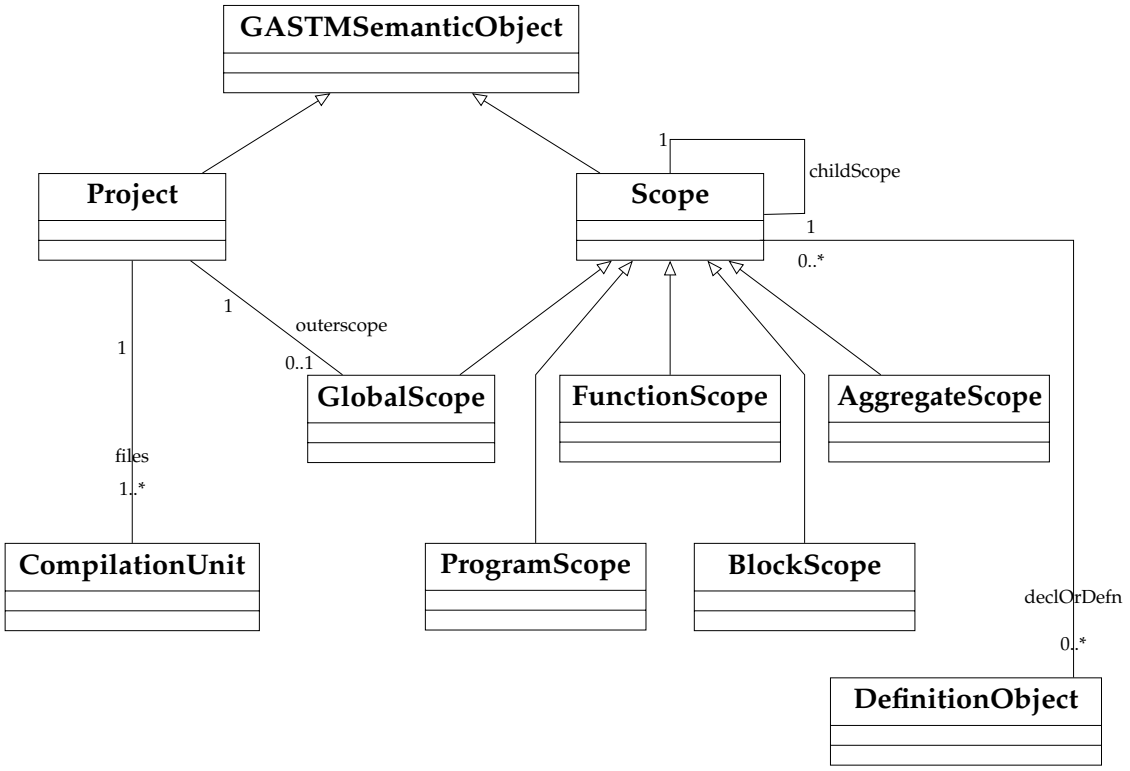


Figure 3.2: GASTMSemanticObject adopted from [4, p. 60]

Figure 3.3 shows the subclasses of `GASTMSourceObject`. These classes model the meta information of the source files. This information consists of source file names and languages, but also positional data such as code lines and positions. In ST, each file of a project is compiled into an individual `CompilationUnit`. These are then stored in the `Project` object that is finally passed to our framework.

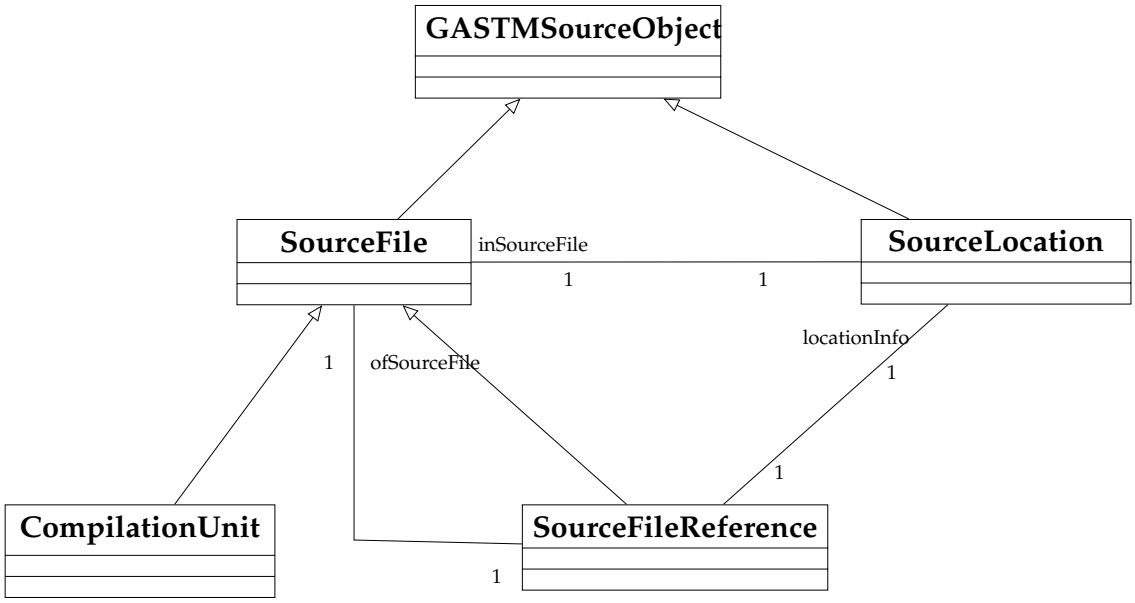


Figure 3.3: GASTMSourceObject adopted from [4, p. 61]

Figure 3.4 shows the children of `GASTMSyntaxObject`. All imperative statements in ST are expressed by subclasses of the `Statement` class. These subclasses include e.g. `IfStatements`, `WhileStatements` and `BlockStatements`. Within statements, `Expressions` are used to represent the exact syntax. The classes for type definitions, variable definitions and function definitions are implemented as subclasses of `DefinitionObject`.

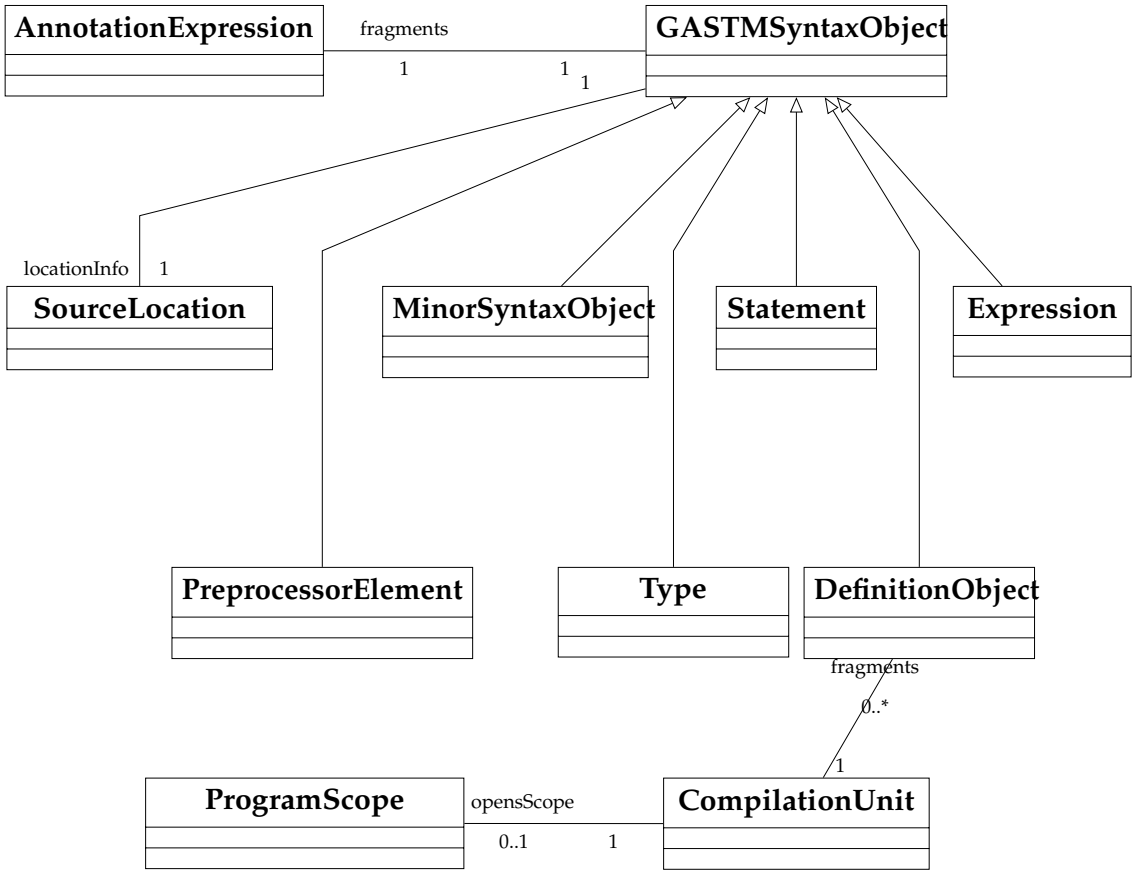


Figure 3.4: GASTMSyntaxObject adopted from [4, p. 62]

3.1 GASTMSwitch

One class that is extensively used in our code is the `GASTMSwitch`. The idea behind the `GASTMSwitch` is to be able to react to the exact class of a given `GASTMObject` without needing an excessive amount of type checks. This is implemented via a structure that is similar to a visitor pattern, but considers the inheritance structure of the GASTM: The `visit` method for each GASTM class has a default implementation that redirects to the `visit` method of its superclass if the class-specific method is not implemented or returns null.

In our framework, `GASTMSwitches` are used in combination with a `GASTMTraverser` or a custom traversal algorithm in order to find specific arrangements of GAST nodes. An example for this is the `CFGraphBuilder` class. This class implements a `GASTMSwitch` that looks for any object within the GAST that can initialize a function. Specifically, it looks for `FunctionDefinitions`, which are used to describe functions and `DefinitionObjects`, which are used to describe Function Blocks. This is necessary since Function Blocks do not have a `FunctionDefinition` for their code, but rather include their code directly. The `CFGraphBuilder` is then applied to a `Project` object via a `GASTMTraverser`. The traverser then looks through the entire structure of the GAST and calls the builder on any relevant node.

4 Control Flow Graph (CFG)

4.1 Basic Principle of CFG

A Control Flow Graph (CFG) is a directed graph that represents the control flow of a program [2]. Representing the control flow information in a graph makes it easier to automatically analyze the execution paths a program may take, i.e., it allows algorithms developed for graph analysis to be applied to code. CFGs are therefore important prerequisites for further analysis steps, such as calculating the SSA-form of a program (see Section 7) or performing symbolic execution [14].

The nodes within a CFG represent so-called basic blocks, that is, sections of code that will always be executed sequentially without the possibility of branching. The edges between the simple blocks represent the conditional paths that the code may follow. That means, the targets of the outgoing edges from a basic block are all blocks of code that can be executed after the end of the current block. For showing when which path is taken, edges can be annotated with the conditions that have to hold for the code to follow the edge. Most CFGs also include two special nodes that represent the start and end of the program, called the start and return node respectively. Graphically, basic blocks are usually depicted as rectangles, while the start and return node are depicted as triangles.

The following figures show some examples of how basic code structures are represented in CFGs. Figure 4.1 shows the representation of an if-then-else construct. The code that includes the conditional expression is in the top block. The two

optional branches are connected with edges labeled “true” for the then-block and “false” for the else-block. Figure 4.2 shows a while loop. The block representing the loop body is connected to the condition with a “true” edge and loops back to the condition block upon execution. The “false” branch of the condition leads on to the next code after the loop, showing how the loop is exited.

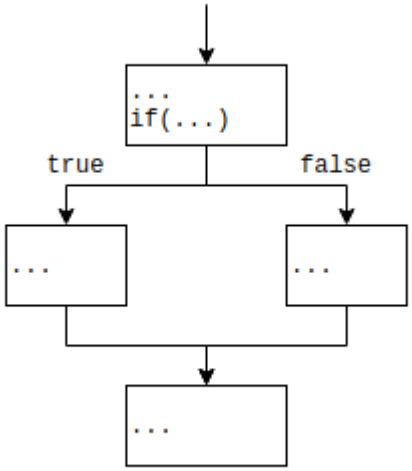


Figure 4.1: CFG for a if-then-else construct

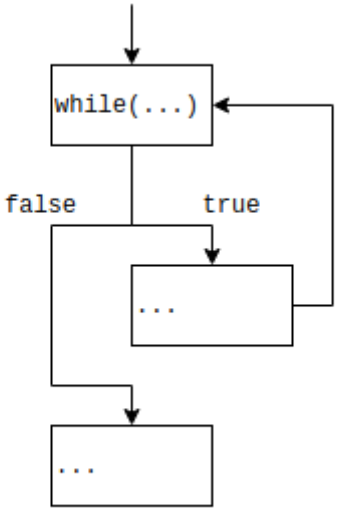


Figure 4.2: CFG for a while-loop[1]

Figure 4.3 shows the expected CFG for our running example from Section 2.2.

[1]Note that the while-statement stands in a separate basic block. This happens because only the condition is reevaluated when repeating a loop.

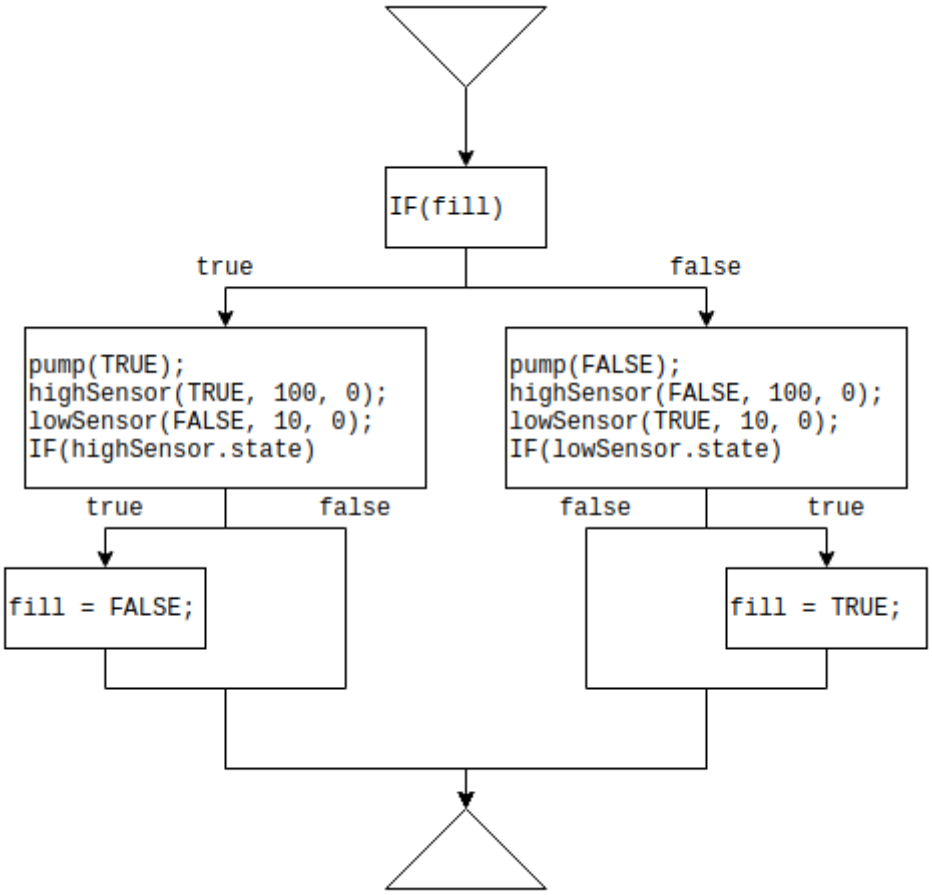


Figure 4.3: CFG for the running example

It can be seen that the `if`-statement that starts the procedure generates a first basic block and then splits the flow into two branches. Each of those branches then contains a list of statements that are executed followed by another `if`-statement. These nested `if`-statements split the program again, this time producing one branch with a single statement, and an empty `else`-branch as there is no `else`-statement.

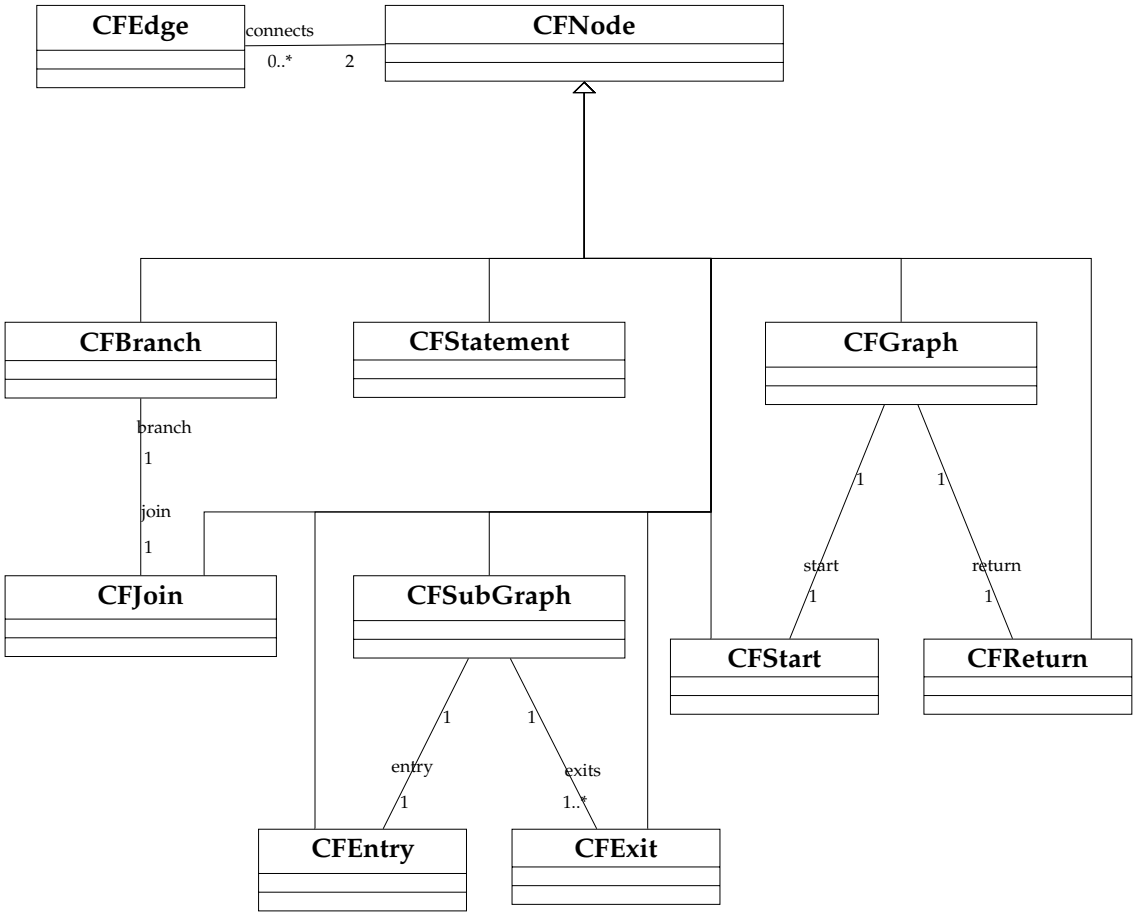


Figure 4.4: Class system for the CFG

4.2 Class System for CFG

A set of classes have been implemented to represent CFGs. These classes are located in the package `at.jku.ssw.analysis.cfg` and are described in this section.

The general class structure can be seen in Figure 4.4. The base class for all CFG nodes is the `CFNode` class. It provides features for modeling the graph structure, namely methods for storing and querying preceding and following nodes as well

as fields for data generated by the SSA and constant folding (see Section 7 and Section 8).

Our CFG inserts for each connection between nodes a `CFEdge` object. This stores whether the edge is a "negated" edge. This is later used to distinguish the branches of an `if-then-else` construct, i.e., the `then`-branch has its `negated` flag set to `false`, while the `else`-branch has it set to `true`.

Basic statements (that do not alter the flow of the program) are stored in the graph as `CFStatement` nodes. Unlike other CFG implementations, like e.g. the representation shown in the previous section, which collect sequential statements into block nodes, our implementation stores each statement in a separate node. This is necessary for integrating the call graph into the CFG later (see Section 5).

Statements that represent branching points in the graph (i.e. conditionals and loops) are represented by `CFBranch` nodes. Each `CFBranch` has an associated `CFJoin` that represents the point where the different branches join again.

The full CFG of a procedure is represented by a `CFGraph` object that stores the entry and exit point for the routine it represents. Each `CFGraph` has only one such entry point (represented by a `CFStart` node) and only one exit point (represented by a `CFReturn`).

The package also includes the classes `CFSubGraph`, `CFEntry` and `CFExit` which are only used while building the CFG and do not occur in the final graph. `CFSubGraph` is used to represent nested `CFGraphs` and `CFEntry` and `CFExit` are used to represent the entries and possible exits of these nested graphs, respectively (for more information on how these classes are used, see Section 4.3).

Figure 4.5 shows the CFG for our running example from Section 2.2. It shows the created objects and their most important relationships, as well as the information they contain, i.e., the `CFBranches` show the condition that they depend on, the `CFStatements` show the statements they represent and the `CFEdges` show whether

they are negated or not. Note that these fields are not represented as strings in the program, but are instead references to the relevant `GASTMObjects` of the GASTM.

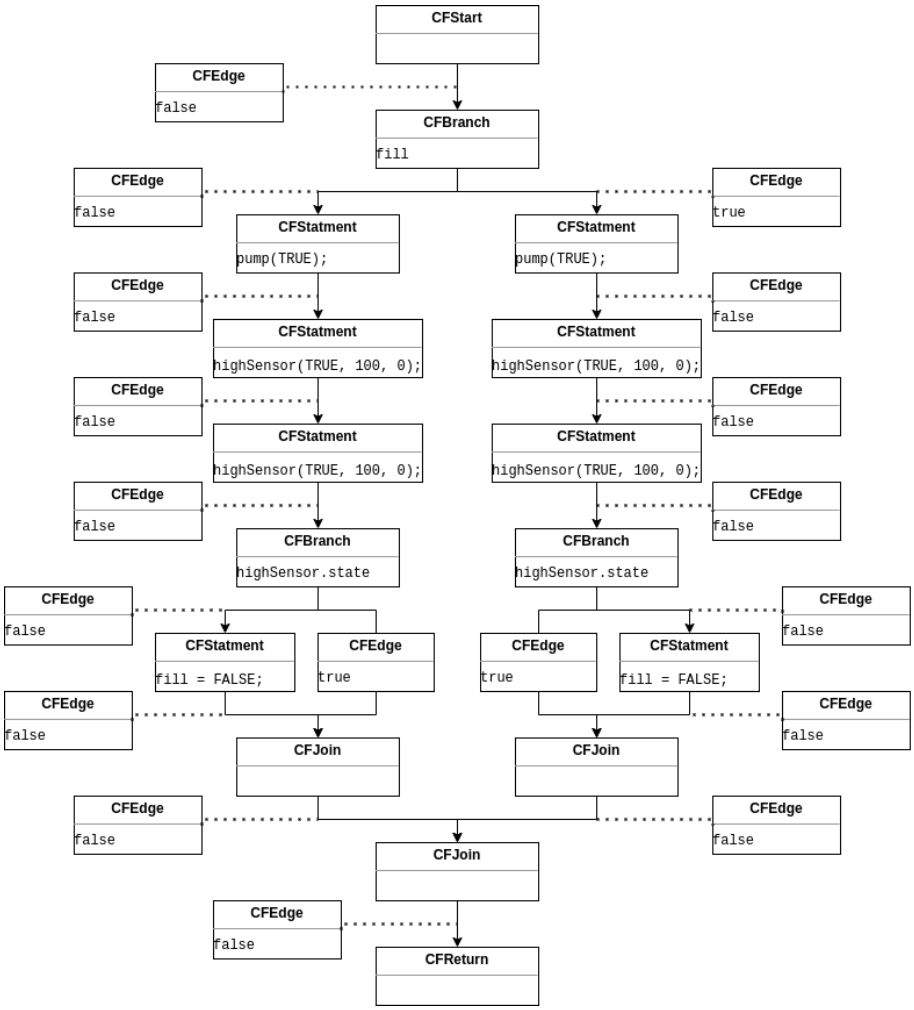


Figure 4.5: CFG for the running example

Figure 4.6 shows the CFG for the simple program shown in Listing 4.1 that computes the sum of the natural numbers up to a given `n` using a while loop. The loop condition is encapsulated in a `CFBranch`, which has outgoing edges to the loop body as well as the end of the loop. The body of the loop is split in two blocks as

it contains two separate statements. Note that the `CFBranch` of the loop still has an associated `CFJoin`-node.

Listing 4.1: Example procedure for summing up the numbers from 1 to n.

```
1  FUNCTION_BLOCK Sum
2  VAR_INPUT
3    n : INT;
4  END_VAR
5
6  VAR
7    i : INT;
8  END_VAR
9
10 VAR_OUTPUT
11   sum : INT;
12 END_VAR
13
14 i := 1;
15 sum := 0;
16 WHILE i <= n
17   sum = sum + i;
18   i := i + 1;
19 END_WHILE
```

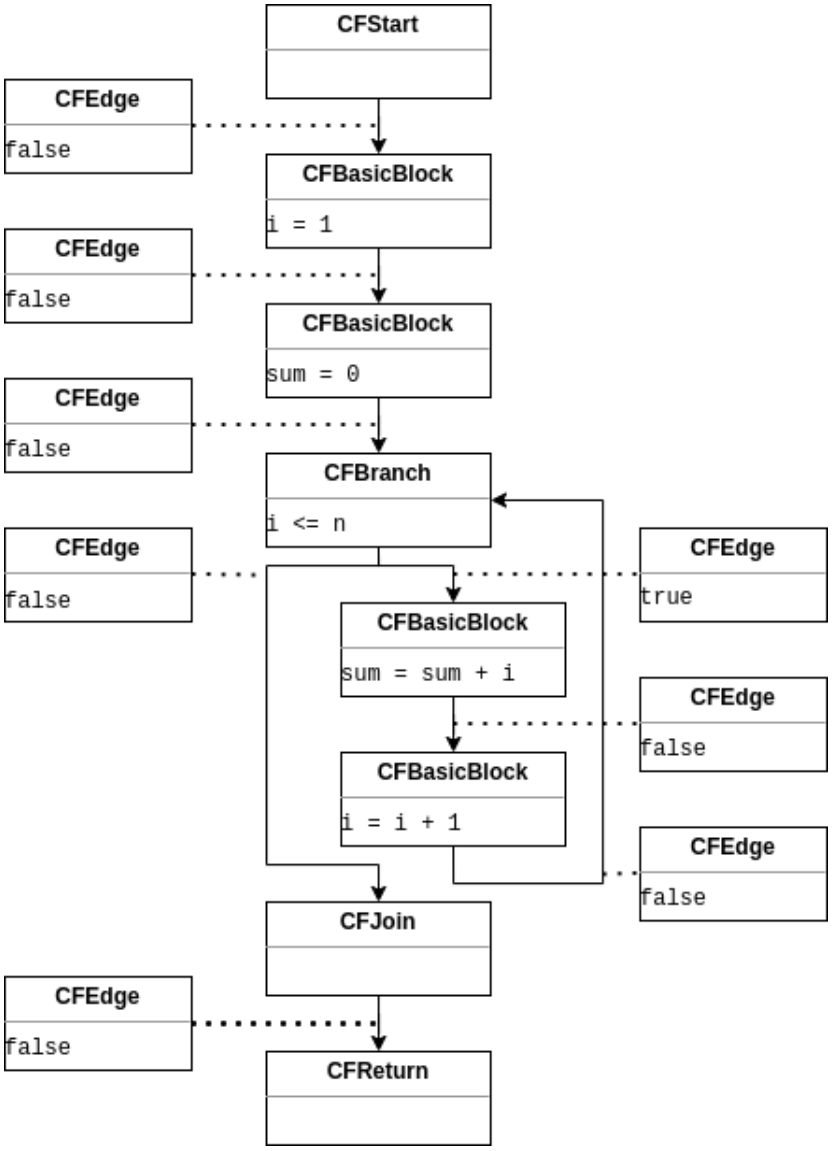


Figure 4.6: CFG for the example program using a loop from Listing 4.1

4.3 Builder for CFG

4.3.1 Principal approach

The process of building the CFG is split into two steps. In the first step, nested subgraphs are built from the ASTM. The second step connects these subgraphs to create a final control flow graph.

The subgraphs created by the first step are analogous to the recursive calls a parser would make when parsing the original code [3]: A new subgraph is created at any place where an arbitrary block of statements could be placed. More specifically, the bodies of loops and if-then-else statements are represented by subgraphs.

Figure 4.7 shows how the subgraph abstraction works for the example from Listing 4.2. The part of the code that is hidden in a subgraph is marked in red. The advantage of this approach is that the algorithm for the first step needs not bother with the complexity of nested statements. In this example, even if `foo()` would be replaced by arbitrarily complex logic, we would not have to consider it, as all complexity would be hidden behind the subgraph abstraction.

Listing 4.2: Example procedure for showing the principle of subgraphs.

```
1  FUNCTION_BLOCK Subgraphs
2  VAR_INPUT
3    bar : INT;
4  END_VAR
5
6  IF bar <= 5
7    foo();
8  END_WHILE
```

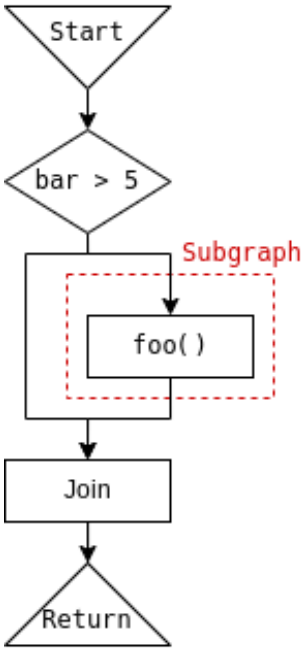


Figure 4.7: Subgraph abstraction for the example from Listing 4.2.

4.3.2 CFSubGraphBuilder

The first stage is the subgraph stage (performed in `CFSubGraphBuilder`). In this stage, the algorithm considers only one level of nesting at a time. Within this level the subgraph builder creates nodes for the control flow in this level, substituting lower levels with `CFSubGraph` nodes which are built recursively. This is done to handle `break`, `return` and `continue` statements properly, as each of those can be challenging in a single-stage algorithm.

The builder is implemented as a `GastmSwitch` with cases for the various `GASTMObjects` it should handle and the procedure `buildSubGraph` to start the building process. The `buildSubGraph` method uses a custom traversal method to iterate through the nodes of the AST. The principal logic behind this iteration is shown in Figure 4.8. As can be seen, the algorithm generates nested `CFSubGraphs` depending on the kind of `GASTMObjects` it handles during iteration:

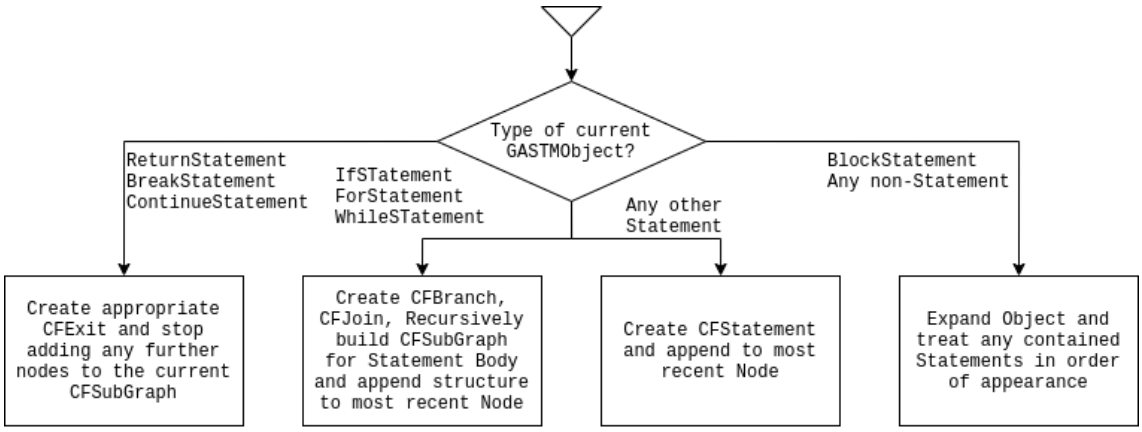


Figure 4.8: Principle for the iteration of the `CFSubGraphBuilder`

The following paragraphs explain how each type of object is handled:

- `IFStatements`, `WhileStatements`, and `ForStatements` are modeled as `CFBranch` nodes. The respective code-blocks (for `for` and `while` this means the loop body, for `if` this means the then- and else-bodies) are generated as new `CFSubGraphs` which are built via a call to a new `CFSubGraphBuilder`. Finally, the `CFJoin` block is set as the current block. (This means that future statements are appended to this node instead of the others. From outside, it looks like branching statements are treated as a singular, linear node.)

  The difference between `IfStatements` and `WhileStatements/ForStatements` is the way the `CFSubGraphs` are attached to the `CFBranch` and `CFJoin`. `IFStatements` have both of their branches leading to the `CFJoin` node, while for the subgraphs of the various loops lead back to the `CFBranch` node, with the negated branch connected to the `CFJoin`.

- `BreakStatements`, `ContinueStatements` and `ReturnStatements` represent the end of the current `SubGraph`. For each such statement the algorithm generates a `CFExit` node with the appropriate `ExitType` which is then added to its `CFSubGraph`. There are a total of four exit types (`RETURN`, `BREAK`, `CONTINUE`

and SEQ (sequential)) that are used by the second step of the algorithm to find the appropriate connection points for the exits.

Unlike other node types, the children of exits are not traversed by the algorithm. This has the side effect that dead code is not added to the CFG, which makes it easier to detect and report dead code occurrences.

- For all other Statements (except for statements that do not contain instructions, mainly BlockStatements) CFStatements are created and appended to the current subgraph.

As an example, we will now show how the CFSubGraphBuilder builds the first step of the CFG for our running example. For this, we will assume that the method buildSubGraph is called with the root GASTMObject of the LevelControl function block as parameter. This is a POU object produced by parsing the file where LevelControl is described.

The algorithm starts by creating a new CFSubGraph. After that, it starts going through the contents of the POU object it received as parameter. Since the variable declarations at the beginning of the class are not considered by the builder, the first IfStatement in the block is the first item that the builder handles, i.e., the caseIfStatement() method is called for this item. This method first generates a CFBranch node for the statement, which automatically extracts the condition of the statement and generates a linked CFJoin node. It then creates a CFSubGraph for the then-branch by calling a new GFSubGraphBuilder on the respective Statement.

The first three statements of the then-block are all procedure calls, which are encoded as three CFStatements in the CFG, the state of the graph at this point is shown in Figure 4.9. Then, another IfStatement follows which is built and appended to the last CFStatement. Building the inner if-statement works in the same way as for the outer if-statement, with the only difference that, since there is no else branch, a negated CFEdge is created directly between the CFBranch and its CFJoin. Since no other statement follows that inner if, a CFExit with the SEQ-type

is appended to the last node and the second builder terminates. The result of the algorithm at this point is shown in Figure 4.10

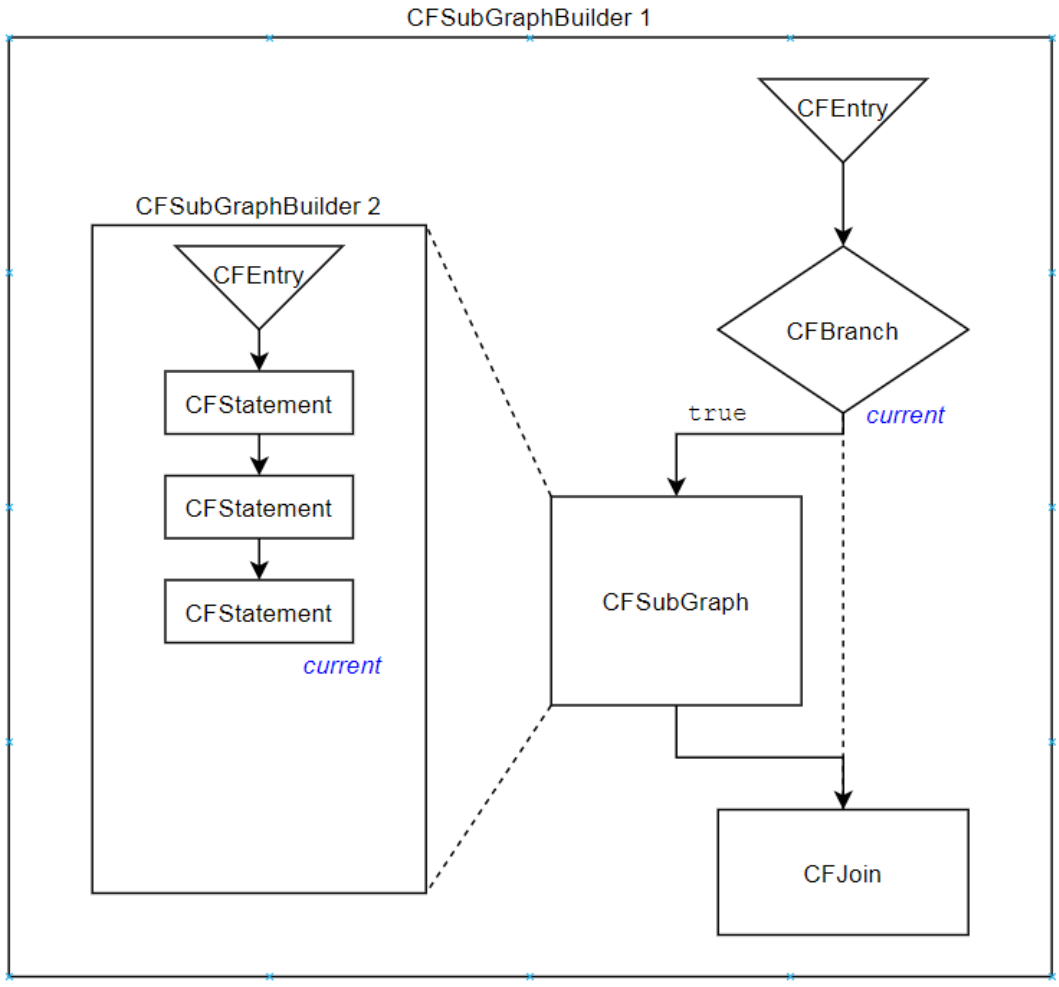


Figure 4.9: Process of building the CFSubGraphs

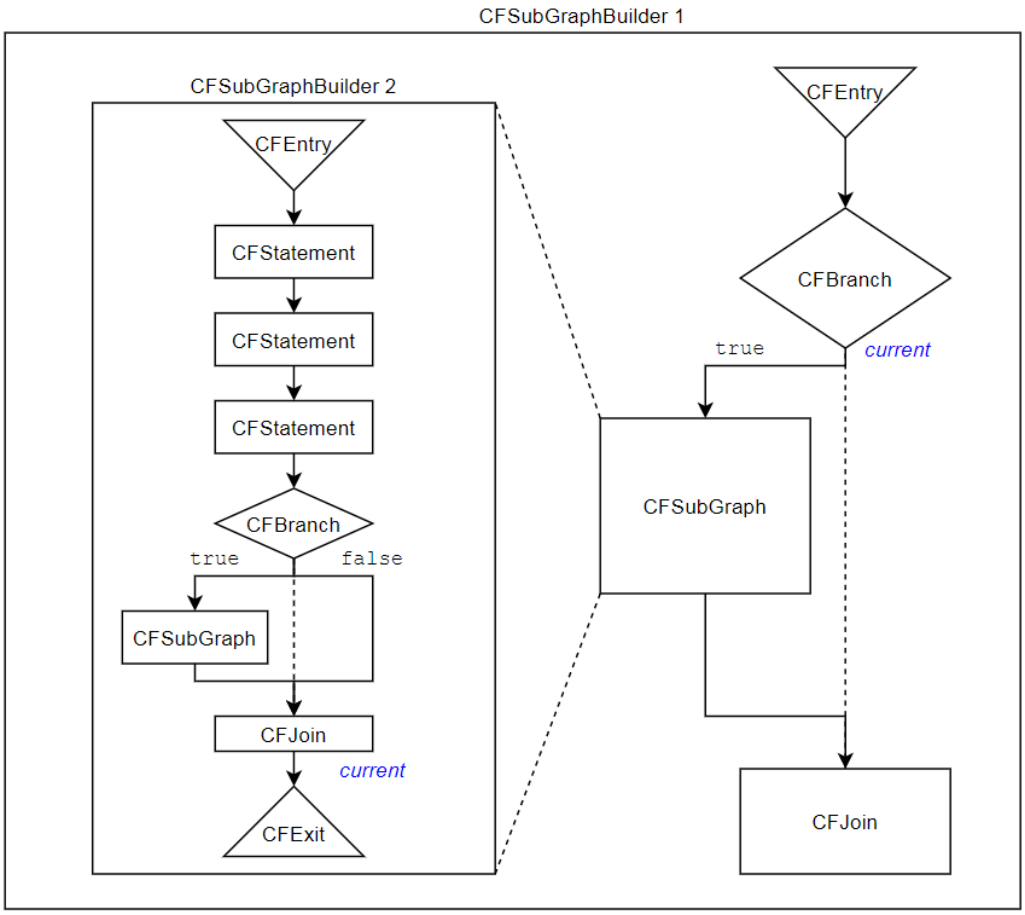


Figure 4.10: Process of building the CFSubGraphs

With this the `CFSubGraph` that represents the then-branch of the outer if-statement is complete. The else-branch is then built in the same way. Both the then-branch and the else-branch are finally connected to the `CFJoin` of the outer if. After having completed both branches of the outer if, the `CFSubGraphBuilder` has finished processing the `IfStatement` and continues to look for the next statement on its path. As it finds none, it appends a `CFExit` with the `SEQ` type and returns the finished `CFSubGraph`, the finalized graph can be seen in Figure 4.11.

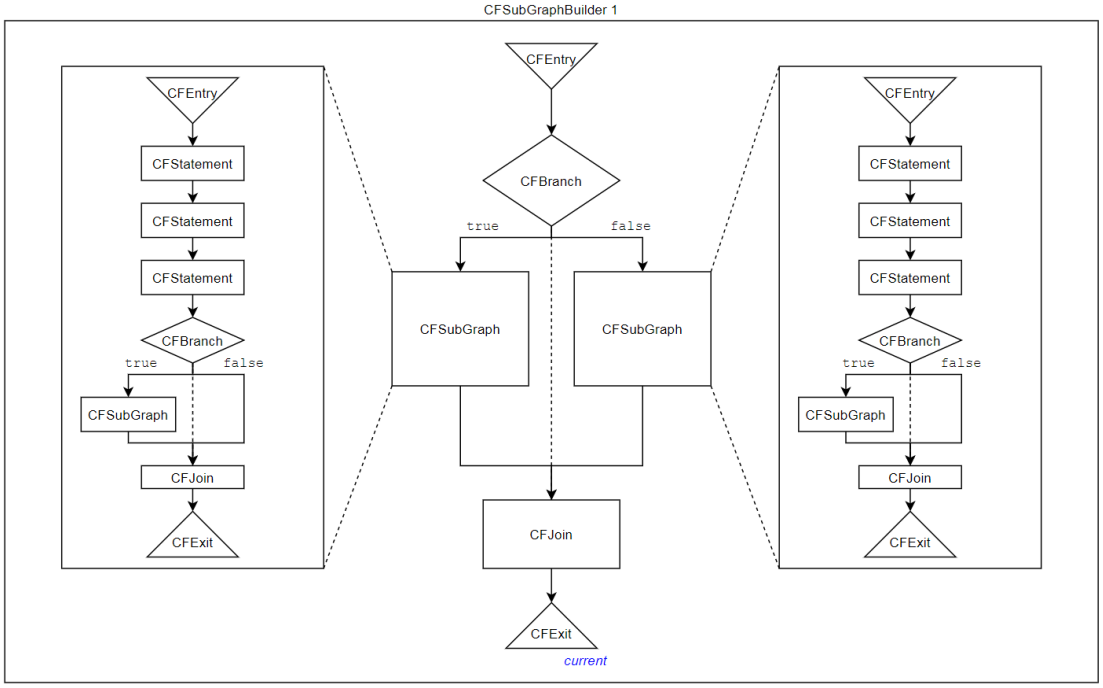


Figure 4.11: Process of building the CFSubGraphs

The full code of the `CFSubGraphBuilder` can be found in `CFSubGraphBuilder.java`.

4.3.3 flattenSubGraph()

The second stage of the algorithm is concerned with "unwrapping" the nested `CFSubGraphs` into a single `CFGraph`. This is accomplished by the method `flattenSubGraph()` in the `CFGraphBuilder` class. This method is called with a subgraph which it then eliminates from the CFG. It does this by replacing it with its "inner" graph and returning any `CFExits` that remain unprocessed to the caller. Similar to the building of the subgraphs, `flattenSubGraph` calls itself recursively to process nested structures before processing the current level. It iterates through the `CFNodes` in the graph, collecting the `CFExits` from all subgraphs it comes across as well as any `CFExits` it finds.

It then tries to resolve as many of the collected exits as possible by connecting the parents of the exits to their intended targets. What the proper target for each exit is, depends on the `ExitType` of the exit:

- *Sequential* exits connect to whatever the next node of the subgraph is. Due to way the `CFSubGraphBuilder` connects the subgraphs it creates, this automatically connects loops to their branches and `if`s to their joins.
- *Return* exits are not connected to anything. They are collected from all subgraphs and connected to the CFGs `CFReturn` node after the first `flattenSubGraph` call returns.
- For *Break* and *Continue* exits, it is important whether the `CFBranch` that created the subgraph belongs to an `IfStatement` or a loop statement. If it belongs to an `IfStatement`, neither break exits nor continue exits are resolved, as neither `break` nor `continue` interact with `if` or `else`. Otherwise, continue exits are connected to the responsible `CFBranch` and break exits to its associated `CFJoin`.

Once all exits are processed, the subgraph is "inlined", that is, its `CFEntry` is replaced by the predecessor of the subgraph and the `CFSubGraph` node itself is deleted.

Again, we will go through our running example to illustrate how this process works. Figure 4.12 depicts the state of the CFG at the beginning of the procedure, figure 4.13 shows the flattening process.

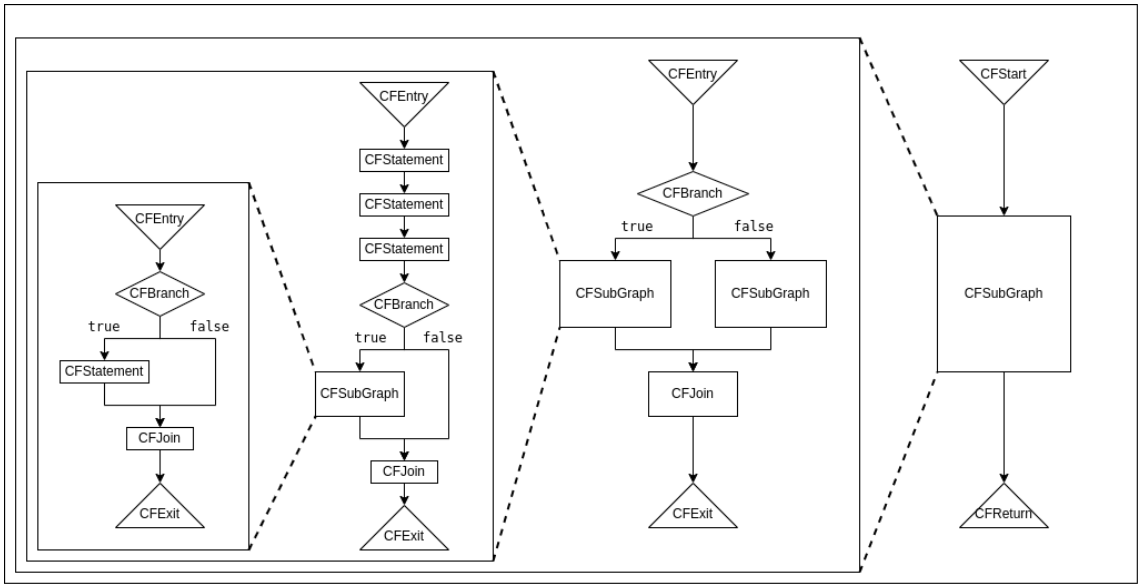


Figure 4.12: State before the flattening.

The procedure first enters the subgraph in the left branch of the outermost if-statement by recursively calling `flattenSubGraph()` on it, iterating through its statements until it finds the left branch of the inner if-statement. It then does another recursive call on the `CFSubGraph` in the then-branch. This subgraph only contains a single statement, which is ignored, followed by a sequential CFExit. This sequential Exit is removed, and the Node before it is connected to the node that follows the containing `CFSubGraph`. Since there are no further nodes in this subgraph, its start is removed and its predecessor connected to its first statement.

Next, the second call of `flattenSubGraph()` finds the sequential exit of its subgraph, removes it and connects the `CFJoin` of the inner if-statement to the `CFJoin` of the outer if-statement. Since there are no more nodes to traverse at this point, it connects the node before its subgraph (the `CFBranch` of the outer if) to the node after its subgraphs `CFEntry` and then deletes the subgraph by removing all connected edges. The procedure is illustrated in Figure 4.13.

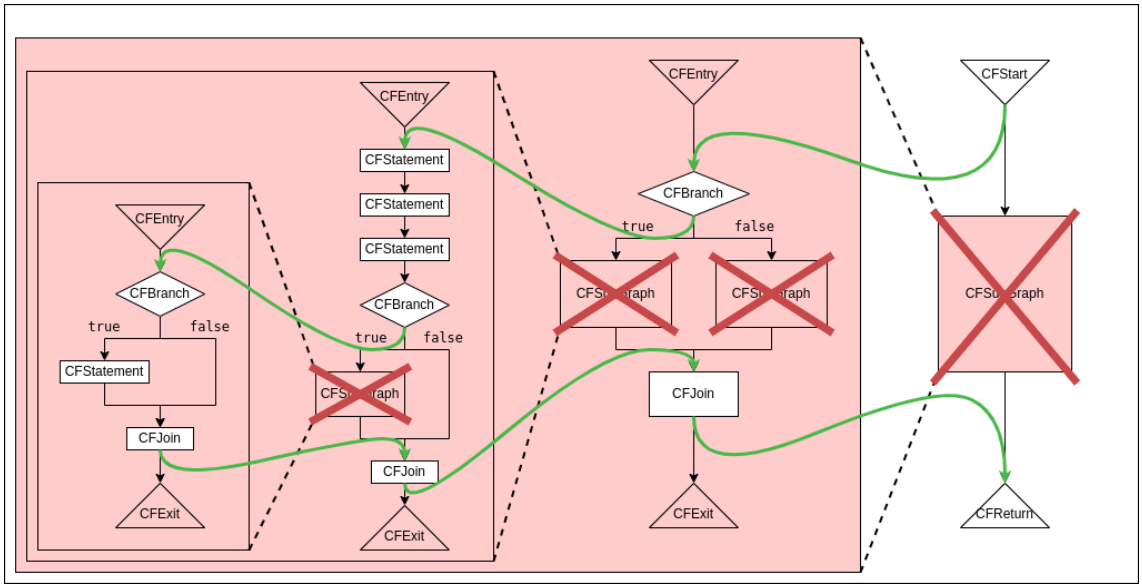


Figure 4.13: Process of flattening the CFSubGraphs

The same happens to the subgraph in the else-branch of the outer if. Once both branches of the outermost if-statement are inlined, the subgraph containing the if-statement itself is inlined by replacing its `CFExit` with the `CFReturn` of the `CFGraph` and its `CFEntry` by the `CFStart` of the graph. The following figures show the final CFG generated by this procedure (Figure 4.14), as well as the CFGs of the procedures called by our example (Figures 4.15 and 4.16).

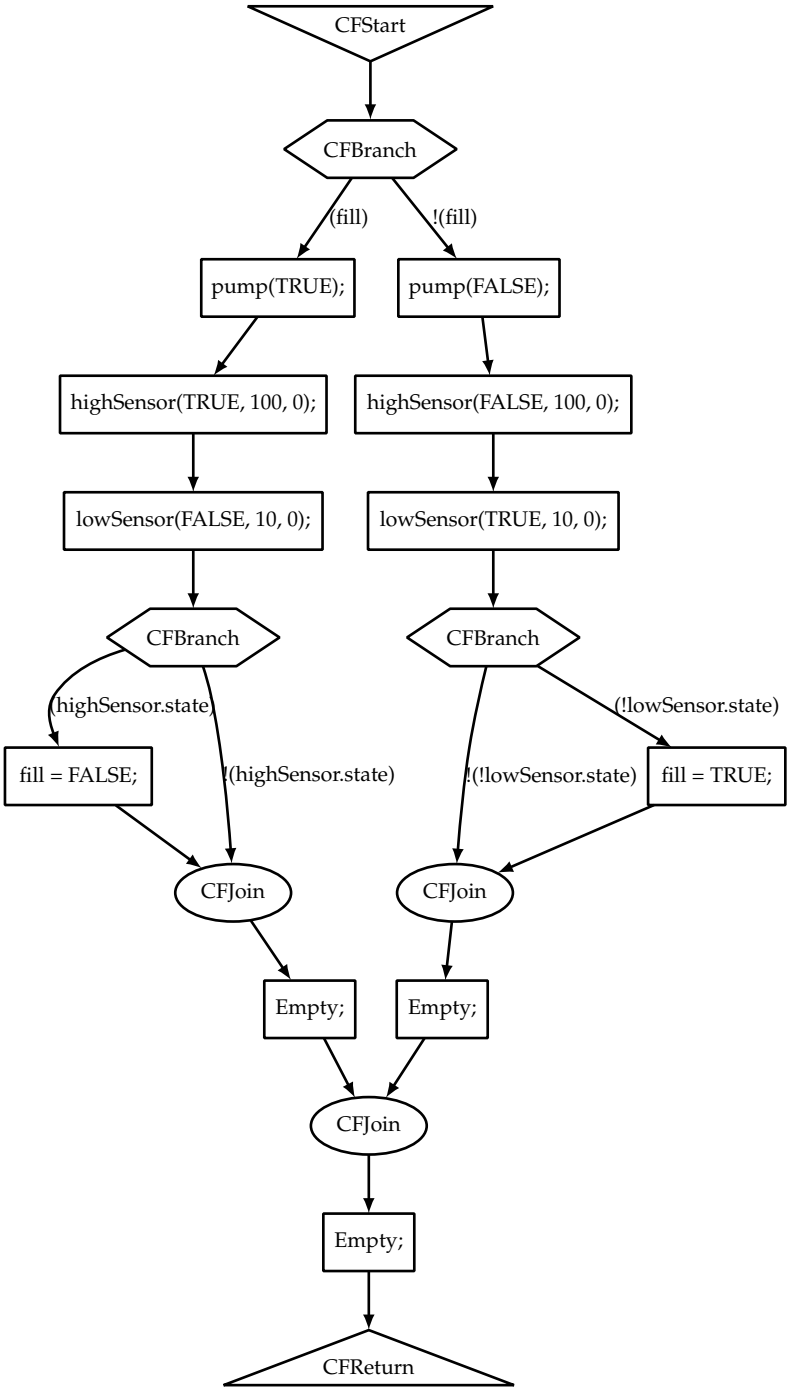


Figure 4.14: CFG generated by LevelControl.

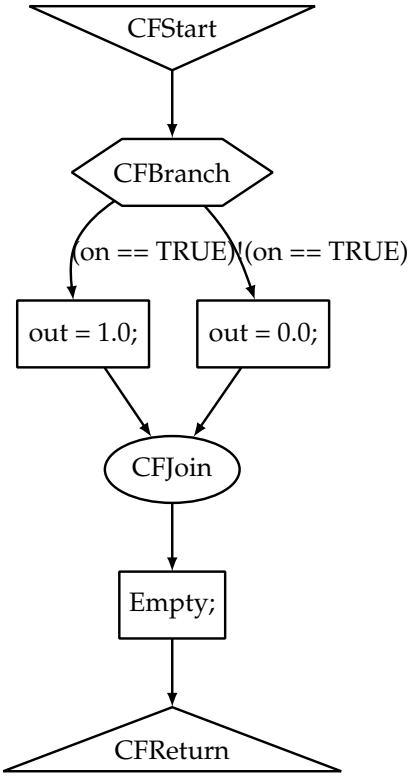


Figure 4.15: CFG generated by Pump.

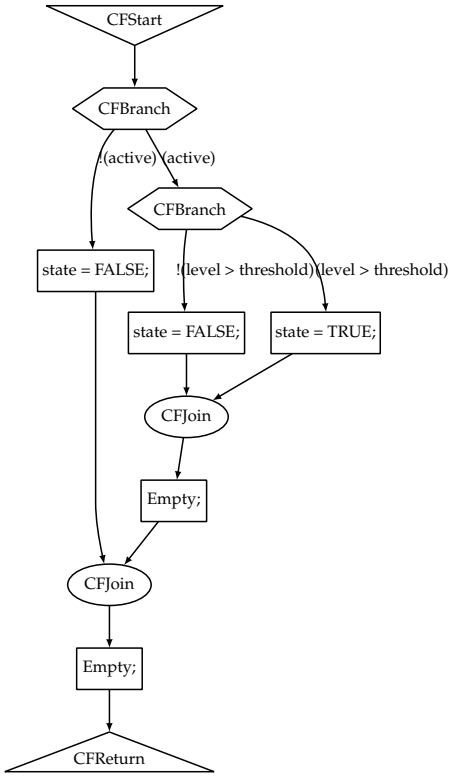


Figure 4.16: CFG generated by Sensor.

5 Call Graph

5.1 Introduction to Call Graphs

A Call Graph (CG) is a graph representing caller-callee relationships between procedures. It is a variant of the previously presented Control Flow Graph [9]. In it, each procedure is depicted by one or several nodes that depict the procedure or the procedure with its current context, respectively. The relationships are depicted as edges. Usually all calls from one procedure node to another are collapsed into a single edge from the caller to the callee. Depending on the purpose and specific implementation, the amount of context information stored in a CG varies. This ranges from completely context-aware CGs that store the context of every call to every procedure separately to context-insensitive graphs that store no context information at all.

Our implementation does not fully adhere to this definition of CGs, however. Instead, it increases the complexity of the model to reduce the complexity in time and memory needed to traverse and store the graph. The most significant difference is that the relationship edges our implementation creates do not originate from the caller themselves, but rather from within the CFG of the calling procedure. This avoids the need for storing context information in the procedures, which in turn avoids the need for making multiple copies of each procedure. Context information is added later through the SSA algorithm, which annotates each procedure

call with the variable values at that code position (See Section 7). This gives us a small memory footprint, while not loosing any information.

This is possible because the IEC-61131–3 languages do not allow dynamic allocation of memory, implicitly ruling out all recursive calls, as well as most of the dynamically bound procedures that are present in many other languages.

To make the structure of the call graph more understandable, we will now go through our running example again, showing first what a classic call graph representation would look like (Figure 5.1), and then how our implementation realizes the structure (Figure 5.2).

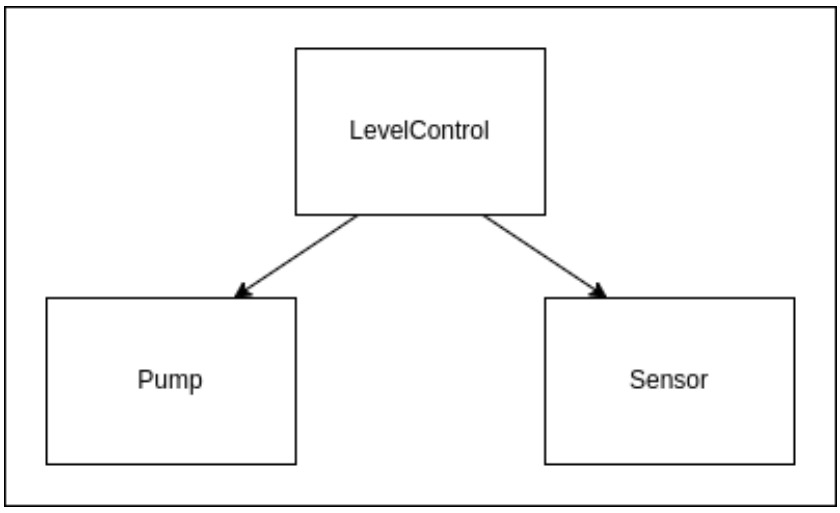


Figure 5.1: Call graph of LevelControl using a conventional, context-free call graph implementation

Figure 5.1 shows how a context-free call graph represents the two calls from lines 9 and 16 of LevelControl as a single edge between LevelControl and Pump. Similarly, it also represents the calls to highSensor (lines 10 and 17) and lowSensor (lines 11 and 18) as a single edge from LevelControl to Sensor. This not only looses the parameters of the individual calls, but also the number of calls happening and which instances of the function blocks are called.

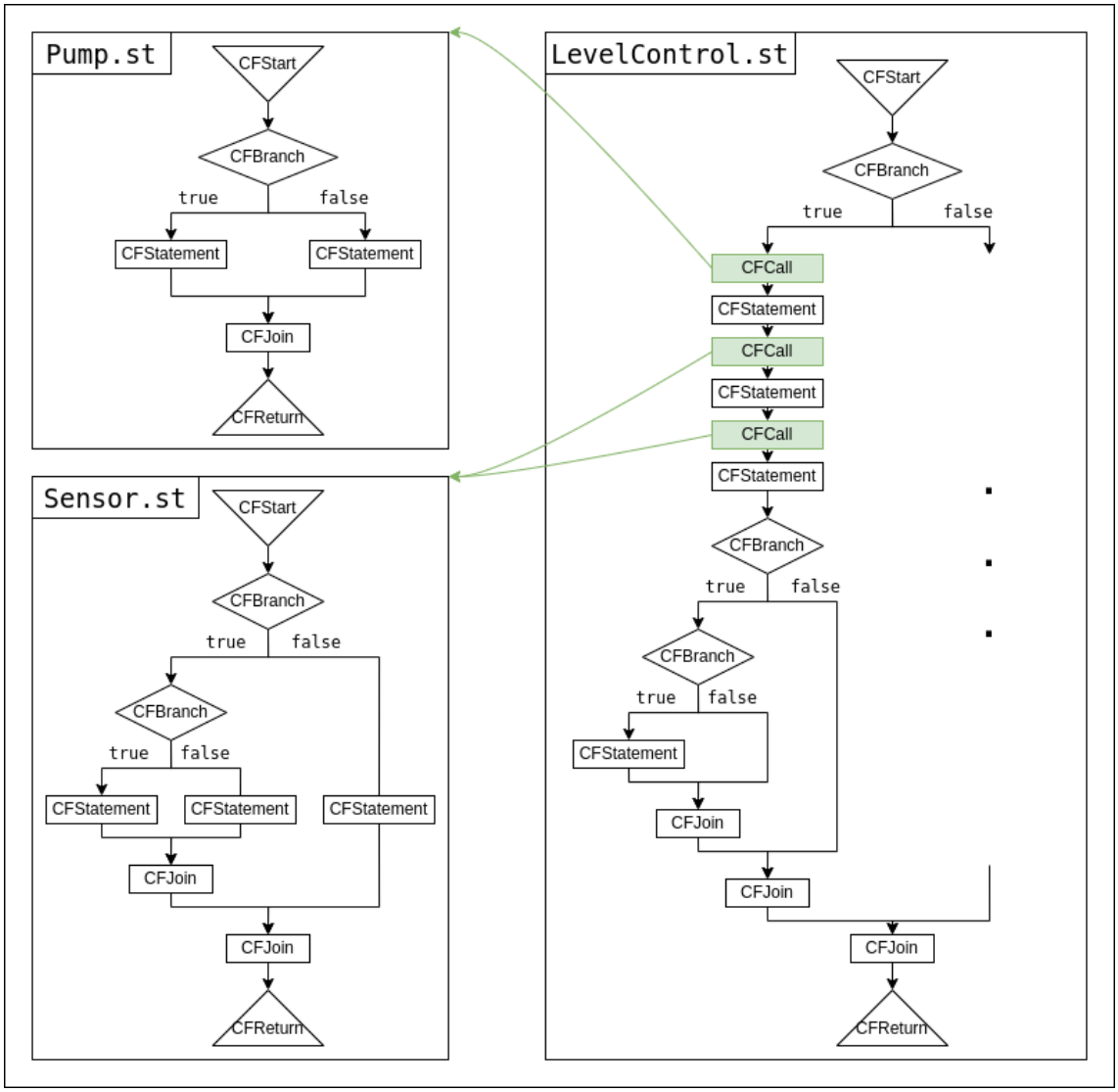


Figure 5.2: Call graph of LevelControl in our implementation. The objects added to realize the graph are marked in green.

Figure 5.2 shows how the call graph is embedded into the CFG in our implementation. For each call a new `CFCall` node that points to the CFG of the callee is inserted into the graph of the caller. In combination with SSA, which calculates values for all variables at each point in time, we can then determine the full context of each procedure call (see Section 7).

The above figures illustrate the advantages and disadvantages of our system. On one hand, our system gains additional accuracy, as we know where exactly each procedure is called, and how many calls to each procedure happen. We can also add context information without the need for additional nodes by annotating our CFCalls with the necessary information. For an implementation of this, refer to Section 7. On the other hand, our implementation is more complex to compute and query than a direct implementation of a CG would be. For example, finding all procedures called by LevelControl would be a very easy operation in a simple CG, whilst it would require a full traversal of the CFG in our implementation.

5.2 Class System for Call Graph

The implementation of the CG reuses and extends the classes from chapter 4.2. The class `CFCall` is a subclass of `CFNode` that allows us to embed the CG within the existing CFG structure. Each `CFCall` represents a call to a procedure, with the called procedure being stored in the `CFCall` node as a pointer to the callee's `CFGraph`. The `CFCall` node also stores context information that is specific to each individual call, like parameter values or assumptions about the state of the program at this point. The extended class diagram is shown in Figure 5.3.

Figure 5.2 shows the object structure of our running example once the CG is integrated into the CFG structure. As the right branch of the outermost if contains the same calls as the left one, it has been omitted to keep the figure small.

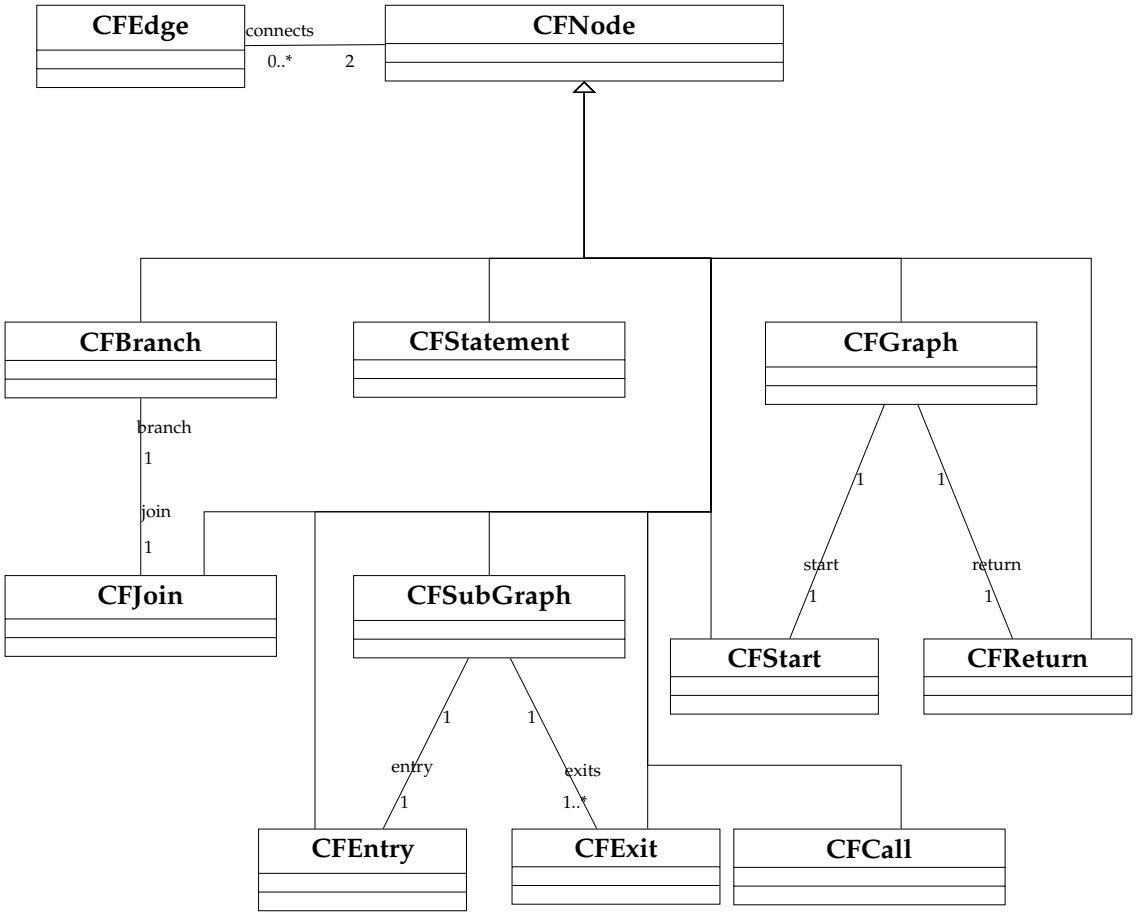


Figure 5.3: Class system for the CFG and CG.

5.3 Builder for Call Graph

5.3.1 Main Principle

Our call graph building algorithm works on top of an already existing set of CFGs. It works by iterating through all nodes of each CFG, stopping at `CFStatements` and examining their contents. If it finds a procedure call within a statement, it resolves the procedure name to its `CFGraph` and then inserts a corresponding `CFCall` into the callers structure.

5.3.2 Finding the Procedure Calls

When started on a `CFGraph` object, the `CallGraphBuilder` traverses all nodes in that graph. For each node it first checks whether that node is a `CFStatement` or not. If it is, the statement contained in the node is examined and searched for instances of `FunctionCallExpressions`. These expressions are then collected into a list and the names of the procedure calls are extracted from them.

5.3.3 Resolving of Procedure Names

The algorithm for building the CG needs to be able to match the name of a procedure to that procedures `CFGraph` object. This process is called "resolving" and should be handled by the framework that builds the `GASTM`. Since the ASTM framework we used did not provide a method for procedure name resolving, we had to implement a simple name resolving method on our own.

To resolve procedure names, the `CallGraphBuilder` is passed the list of CFGs for all procedures in the project during its creation. Since all `CFGraphs` store their own

name, finding a procedure then becomes matter of looking up a procedures name in the list of CFGs.

In the running example from Section 2.2, we first build the CFG for each of the three function blocks. From this we obtain a list of CFGs, each of which stores the name of the function block it represents (see Figure 5.4). This list is then passed to the call graph builder.

Figure 5.4 outlines how the `CallGraphBuilder` uses this list to resolve procedure calls. First, the `CallGraphBuilder` finds an expression that requires it to create a `CFCall`. In our case, this is the call to the function block `Pump` in line 9 of LevelControl. It then instantiates the call object and extracts the name of the called procedure from the expression and looks for an appearance of that name in the list of CFGs. Here, this is simply the name "pump" which it quickly locates in the list of CFGs. Once it finds the name, it links the `CFCall` to the `CFGraph` and continues iterating through the CFG to find the next procedure call.

The builder first iterates through all CFGs in the list and creates a map from procedure names to their CFGs, called the function map. It then iterates through each CFG and inserts call nodes whenever a procedure is called (see Figure 5.2). Each time a `CFCall` is created, the builder extracts the name of the called procedure from the ASTM procedure call. It then looks up the name of the procedure in the function map. If it finds a match, it sets that CFG as the target of the current procedure call. If the name is not in the map, the target is set to `null`, meaning that the procedure could not be resolved.

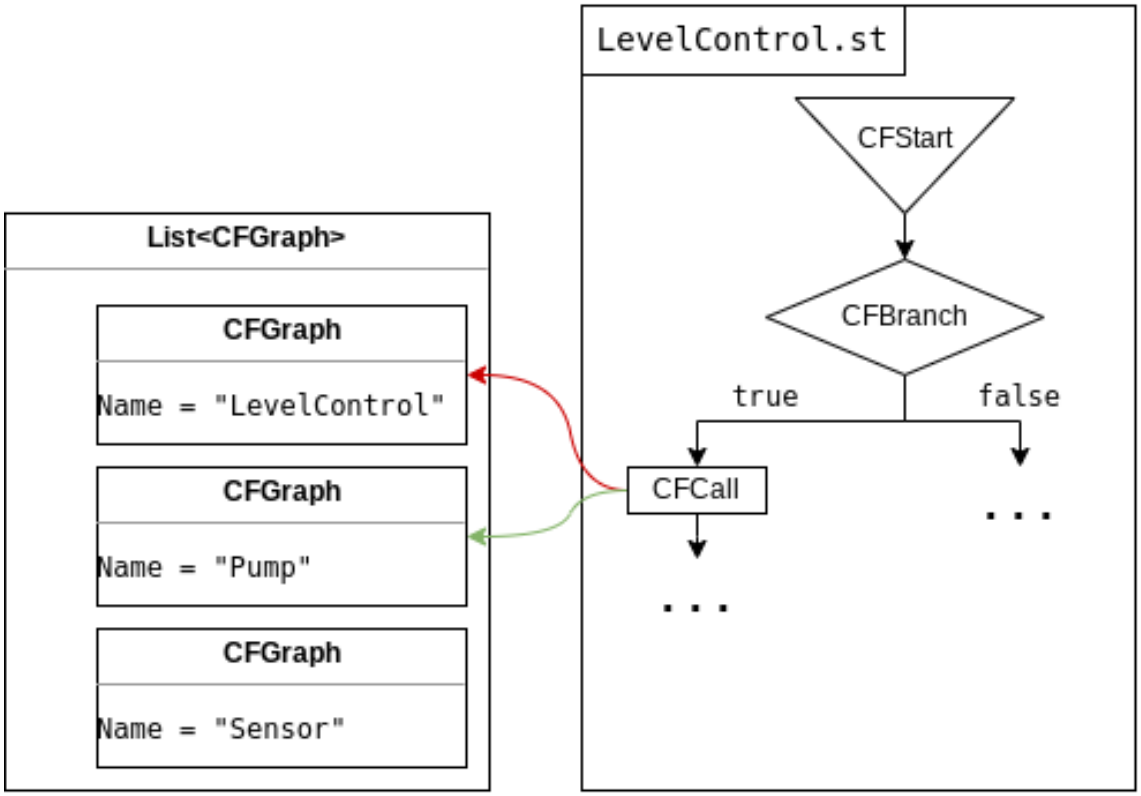


Figure 5.4: Operation principle of procedure name resolving.

6 Memory Model

6.1 Instance Tree (IT)

A representation of the program's data structures, the instance tree is a model which gives a view of all static memory allocations. Its primary task is to provide information about all objects, their compositions, and their dependencies. Each instance is assigned a node in the tree and the fields in a structure dictate the neighbour nodes of the instance. By assigning instances to nodes the instance tree makes it possible to quickly resolve complicated expressions such as `QualifiedIdentifierReferences` (e.g. "a.b.c"). This makes it possible to track which instance of a variable is modified, by simply following the path inside of the instance tree.

For our model we chose to represent three types of nodes:

1. `atomic` nodes hold atomic forms of data, which may not be broken down further, typically primitives. These nodes are represented as leaf nodes in the instance tree. They may also have a specific value associated with them.

2. `ref` nodes are nodes which hold references to other data in memory. Therefore, this node has a single edge to some other node in the tree. Only simple references are modeled by this node complex pointer arithmetic is not supported.

3. `structured` nodes represent composite data types e.g. classes. This type of node is composed of other nodes such as `atomic`, `ref`, or `structured`.

The memory model was used for program analysis and could also be applied to various optimization tasks. We used the instance tree for the following two applications: constant folding and reaching definitions.

Concretely, during constant folding values are assigned to `atomic` and `ref` nodes in the tree. By tracking assignments to these nodes and propagating the results of all operations, one can figure out which nodes always receive a constant value. Obviously if an instance remains constant, it can later safely be replaced by a constant value. This can be helpful during static analysis since it might indicate a code smell. Correspondingly, it was also used in the computation of reaching definitions in order to track variables associated with an assignment. The instance tree helps make this process a lot easier as resolving which variable is exactly assigned becomes a lot easier.

There are also some limitations that result from performing a static analysis since this type of instance tree tracks only static memory allocations (meaning that the memory requirements can be fully determined at compile time). Unlike static memory allocations, dynamic memory allocations could not be tracked in the same manner since they would actively grow and shrink this tree. Two types of problems result from restricting the evaluation to static analysis:

The instance tree is not generally applicable to all languages and is restricted to languages such as ST and other languages in which all allocations are entirely static. However, at least a subset of most languages can be supported as most languages allow for some allocations to be static.

Pointer arithmetic or complicated pointer assignments could also not be supported since they would have introduced additional problems which could likely not be solved statically. One must only imagine a pointer to an element contained

within an array to see how this analysis would indeed become very difficult, if not undecidable.

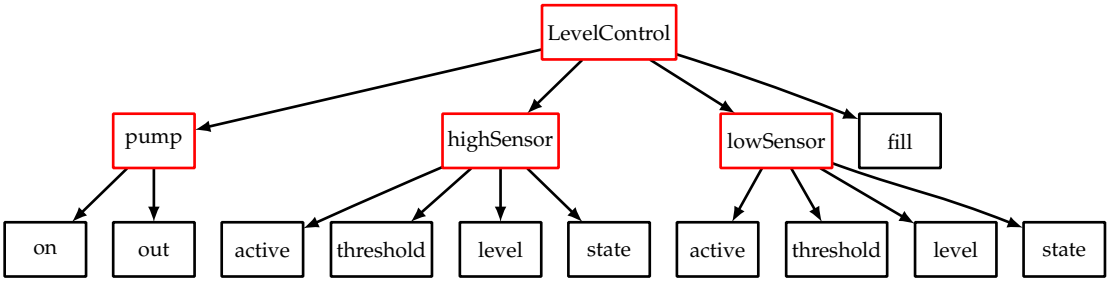


Figure 6.1: An example of an instance tree.

Figure 6.1 depicts a visual representation of an instance tree generated from the introductory ST example in section 2.2. In this representation all `structured` nodes are marked in red, while all `atomic` nodes are marked in black. The instance tree was generated by choosing the `LevelControl` function block as the starting point for the algorithm. This function block contains four member fields, namely `pump`, `highSensor`, `lowSensor` and `fill`. The atomic node `fill` is the only `atomic` member in the `LevelControl` function block and is therefore a leaf node. The other vertices below `LevelControl` are one of two structures, namely `Pump` or `Sensor`. Each of which form their own subtrees below the `LevelControl` vertex. `Pump` is made up of the boolean `on` and floating point number `out`. While `Sensor` is composed of the two booleans `active` and `state` along with the floating point numbers `threshold` and `level`.

Looking at a specific example of an expression containing a `QualifiedIdentifier-Reference` such as `highSensor.state`, one could now simply match it to the state node in the memory model. This allows an algorithm such as constant folding to associate a value with this node. Furthermore, if a specific assignment such as `highSensor.state = TRUE` is given, the value `TRUE` could be associated with the corresponding state node in the tree.

6.2 Instance Tree Builder

The task of the instance tree builder is to construct the instance tree. In the following section the construction of the instance tree is demonstrated based on the example of the previous section 6.1.

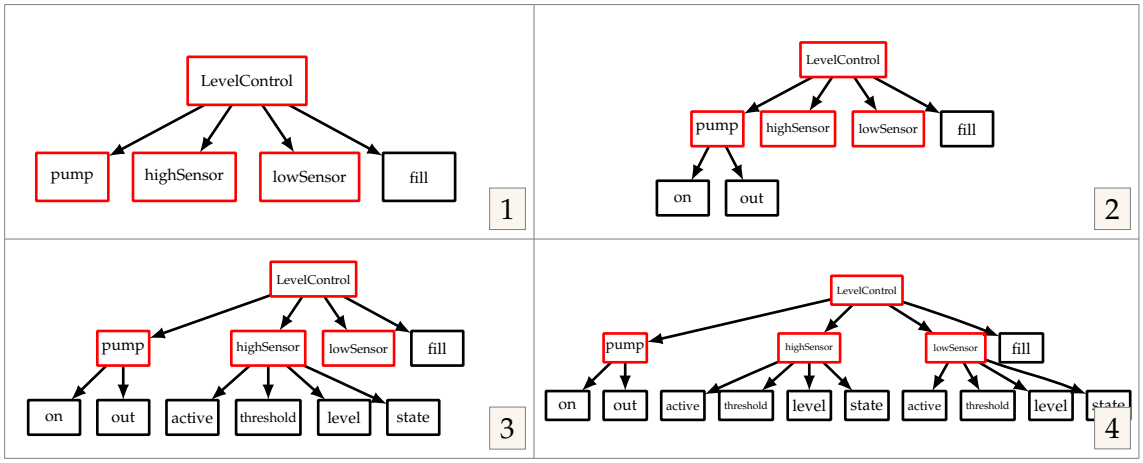


Figure 6.2: Four part instance tree construction

The above Figure 6.2 shows the recursive construction of the instance tree starting at an instance of the `LevelControl` Function Block. In step one all member fields of the `LevelControl` function block are obtained from the GASTM representation. Here `fill` an atomic boolean is added to the root of the tree. Then `pump`, `highSensor`, and `lowSensor` are recursively extended in step 2, 3, and 4 respectively. Once `pump` is reached, only atomic values may be added to the subtree and so the algorithm must backtrack and processes the next structure. The next structure would then be `highSensor` for which the atomics `active`, `threshold`, `level`, and `state` are added. In the last step the algorithm backtracks again and it extends the `lowSensor` node.

The idea for the builder is essentially similar to DFS (Depth First Search) algorithm, where nodes in the graph are exhaustively extended until an end is reached.

6.2.1 Instance Tree Builder Algorithm

In the following section the algorithm for constructing the instance tree will be described in more detail:

Algorithm 1 Instance Tree Builder

```
1:  function build(G, S)
2:      V ← {S}
3:      E ← {}
4:      for all m ∈ S do
5:          V ← V ∪ {m}
6:          if m is GLOBAL then
7:              E ← E ∪ {(G, m)}
8:          else if m is REF then
9:              E ← E ∪ {(S, m), (m, getTarget(m))}
10:         else if m is POU then
11:             (V1, E1) ← build(G, m)
12:             V ← V ∪ V1
13:             E ← E ∪ E1
14:         else
15:             E ← E ∪ {(S, m)}
16:     return (V, E)
```

The `build` function takes two parameters: The root of the instance tree `G` where all global variables are declared and the current structure which is being expanded `S`. Initially `S` will be equal to `G` since the algorithm is executed on the "root node of the tree" (`build(G, G)`). The algorithm returns a tuple of vertices and edges. Where the vertices of every subtree are either member fields of the root of that subtree or members of one of the children of the root.

The rough idea behind this algorithm is that each structure is recursively extended (similar to DFS). At each expansion step all `atomic` and `ref` nodes are resolved to their appropriate location and added to the current structure.

In detail this means: The first step of the algorithm creates a set of vertices and a set of edges, which will later be returned. The current structure is then added to

the set of vertices. Then each member field of the structure (and its corresponding instance) are visited. In any case the field is added to the set of vertices.

After adding the member to the set of vertices, one of four cases will be executed:

1. If the vertex is a global variable, an edge is added from the global node `G` to this instance.
2. If the vertex is a reference, an edge is added from the structure to the member and from the member to the references target.
3. If the vertex is a structure / POU, then similarly a vertex is added from the current structure and the vertex is recursively extended.
4. If the vertex is an atomic value, an edge is added from the parent structure to this atomic value.

At each stage the algorithm will return a subtree containing all instances and child instances of the given structure. Finally, the value returned can be assumed to contain all reachable instances and therefore forms an instance tree.

6.3 Class System for IT

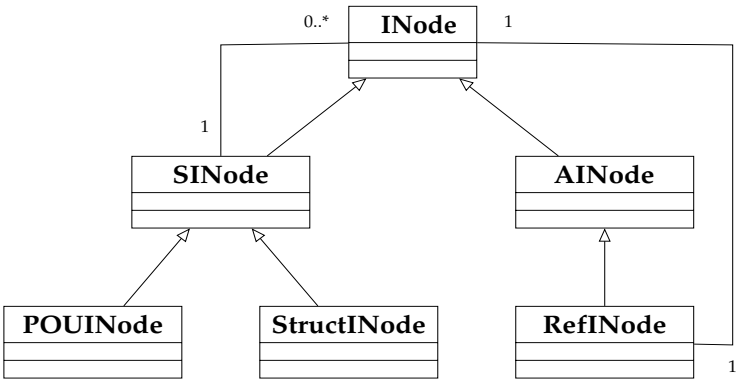


Figure 6.3: Class System for IT

Figure 6.3 presents a visual representation of the class system for the instance tree. The class system is comprised of the classes outlined below:

- The `INode` is the base class of each node of the instance tree. It stores GASTMObject which is associated with it, the name of the declaration, and its data type. The `INode` also maintains a reference to its parent `structured` node.
- `SINodes` (short for Structured Instance Node) represent the base class for both POUINodes and StructINodes and are hence responsible for dealing with instances of composite types. These `structured` nodes need to store both the specific declaration of an instance of a variable and the name of its field.
- `POUNodes` can be thought of as objects which have some kind of instance, code which may act upon this instance, and variables which may be modified by the object. Unlike regular objects POUs typically only have one function which acts upon its data. The data of a POU may also be modified from the outside e.g. by another Function or POU.
- `StructINodes` and `POUNodes` are very similar and the difference is merely a technical one since POUs may be any of the following: Program, FunctionBlock, Function. While `StructINodes` may only contain DataDefinitions (e.g. struct). These occur in the GASTM when languages other than ST are parsed.
- `AINode` AINodes (short for Atomic Instance Node) can only store atomic values like: Integer, Boolean, and Real. Atomic Nodes may also store slightly more complicated aggregate types like arrays composed of primitives. These nodes are used quite heavily during constant folding.
- `RefINode` are nodes which reference other nodes similar to pointers.

7 Data-Flow Analysis

In [9] data-flow analysis is defined as: "The purpose of data-flow analysis is to provide global information about how a procedure (or a larger segment of a program) manipulates its data." In other words, data-flow analysis seeks to determine how data flows between different parts of a program and how this influences other elements of the program. This is typically achieved by analyzing the CFG of the program and identifying data dependencies within it.

There are many kinds of data-flow analysis techniques. These techniques apply to a wide range of problems. Depending on the problem, a decision is made as to which algorithm is used. Examples of data-flow analysis techniques include:

- Constant propagation of constants through the statements of the program.
- Reaching definitions determining the set of assignments valid for each statement.
- Live variable analysis seeks to determine which values will be needed in the future through computation of so called "live sets".

Many other data-flow analysis techniques exist [9]. However, in the following sections we will focus on reaching definitions and constant folding.

Data-flow analysis has become an integral part of compilers and other software analysis tools and is used for a host of optimizations. There is an extremely wide range of use cases for data-flow analysis techniques, some of which have been mentioned here:

- Dead code elimination attempting to find unused code so-called "dead code" which may then potentially be removed inorder to simplify the program.
- Register allocation describes the process of mapping register to each variable can use live sets and graph coloring algorithms
- Optimization various other optimization techniques also benefit greatly from data-flow analysis e.g. CSE.

7.1 Reaching Definitions

One way to perform data-flow analysis is by performing a reaching definitions analysis. Definitions are statements in which some variable is assigned some value. The “reaching” part of reaching definitions refers to estimating the range of the validity of a definition. Combined, the reaching definitions algorithm seeks to determine the set of assignments valid in each basic block of a given CFG. These definitions can then be used to ascertain exactly which assignments are relevant to any part of the CFG.

Figuring out the exact smallest set of definitions that affect a given statement is a challenging task. This is because the problem of finding the minimal set of assignments is often too difficult to solve exactly. Instead, we compute an over-approximation to estimate the set of definitions that may affect a statement. This technique provides a conservative estimate of the set of assignments that hold for a given statement, allowing us to reason about the behavior of the program with a degree of confidence.

It is important to ensure that the set of assignments produced by reaching definitions is not entirely misrepresentative of the function, even though the precise set of assignments cannot be accurately determined. Reaching definitions analysis aims to produce a conservative estimate of which assignments hold for a given

statement. The resulting set of definitions is therefore a superset of the minimal set of definitions that affect a statement. This conservative estimate of all valid assignments can later be narrowed down further by excluding dead code or applying other data-flow analysis techniques, such as constant folding.

7.2 The Basics of Reaching Definitions

This section will introduce the basics of reaching definitions through the following examples.

Listing 7.1: A simple code example

```
1 i := 0
2 i := 1
3 j := i + 1
```

Listing 7.1 contains a simple code example composed of a few variable assignments. The example simply assigns two variables and `i` and `j`. It can easily be seen that the first assignment `i := 0` is not required. Computing the reaching definitions form for this example helps visualize this.

Listing 7.2: A simple code example with Reaching Definition annotation

```
1 { }          i := 0
2 { i := 0 } i := 1
3 { i := 1 } j := i + 1
```

The above listing 7.2 shows on the left the reaching definitions annotation and on the right the code shown in the previous listing. Using this reaching definition representation one can now easily infer that `i := 0` is no longer valid once the third statement is reached because the second statement "kills" the assignment

of the first statement. Hence, applying reaching definition analysis to this simple code example reveals exactly which assignments affect what statement.

Applying reaching definitions analysis to the pump function block shown in section 2.1 produces the Figure 7.1 below:

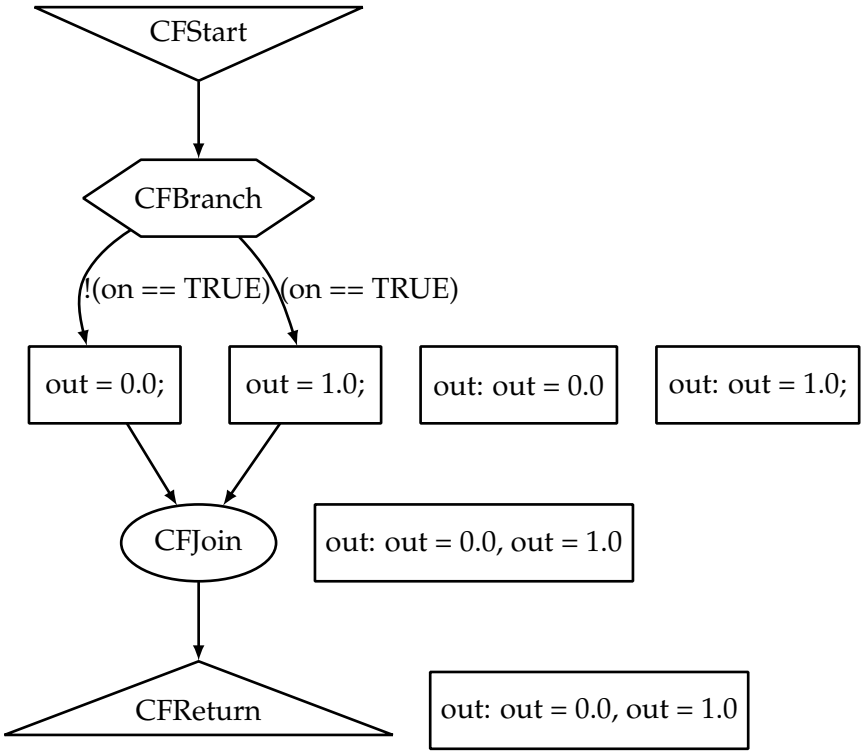


Figure 7.1: An example of reaching definitions

The figure shows a branch with a condition. On the left, `out` is assigned the value `0.0` and on the right `out` is assigned `1.0` respectively. Once the `CFJoin` node is reached both assignments `out = 0.0` and `out = 1.0` are valid because at this point either assignment could have been executed.

Therefore, the goal of reaching definitions is to compute which assignments hold in what node.

7.3 Formalizing Reaching Definitions

This section will introduce reaching definitions in formal manner. The problem of computing the reaching definitions can be formulated as solving as iteratively solving a set of data-flow equations until a fixpoint is reached. These data-flow equations define which assignments are valid for any given node in the CFG. We will start by introducing the individual data-flow equations and then combine them to show how the reaching definitions could be computed using them.

Our goal will be to compute two functions for every node in the CFG: $reach_{in}(i)$ and $reach_{out}(i)$. Where $reach_{in}$ denotes the set of assignments that reach the given node and $reach_{out}$ defines the set of assignments that remain valid for subsequent nodes. It is then obvious that $reach_{in}(\text{CFStart}) = \emptyset$, because no variables are assigned before the starting node. So, then it may be useful to define $reach_{in}$ as:

$$reach_{in}(i) = \bigcup_{j \in pred(i)} reach_{out}(j)$$

The definitions which reach this node as the definitions which other blocks produce. Adding to that $reach_{out}$ can be defined as:

$$reach_{out}(i) = (reach_{in}(i) \cap prsv(i)) \cup gen(i)$$

The set of definitions that reach the block without the set of definitions which the block invalidates plus the set of definitions which the block itself contains. Let $gen(i)$ denote the set of all assignments in the block i. Furthermore, let $prsv(i)$ denote the definitions which remain valid in block i.

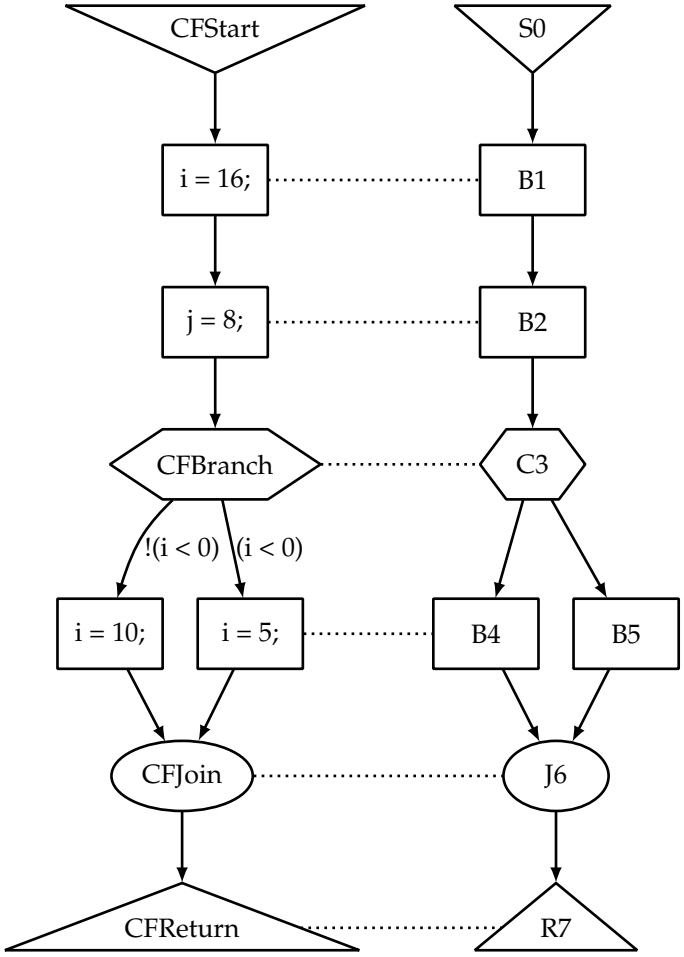


Figure 7.2: A simple cfg with labeled nodes

Figure 7.2 shows an example of a CFG with labels assigned to each node. This example will be used in the following section to demonstrate the computation of reaching definitions.

Table 7.1 holds the solutions to each of the data-flow equations for the CFG given in figure 7.2. Empty entries indicate that the set would be empty.

i	$gen(i)$	$prsv(i)$	$reach_{in}(i)$	$reach_{out}(i)$
S0		i={5,10,16}j={8}		
B1	i={16}	i={16},j={8}		i={16}
B2	j={8}	i={5,10,16},j={8}	i={16}	i={16}j={8}
C3		i={5,10,16},j={8}	i={16}j={8}	i={16}j={8}
B4	i={10}	i={10},j={8}	i={16}j={8}	i={10}j={8}
B5	i={5}	i={5},j={8}	i={10}j={8}	i={5}j={8}
J6		i={5,10,16},j={8}	i={5}j={8}	i={5,10}j={8}
R7		i={5,10,16},j={8}	i={5,10}j={8}	i={5,10}j={8}

Table 7.1: The computed Reaching Definitions for figure 7.2

The first of the data-flow equations $gen(i)$ can easily be explained using this example. The first non-empty entry $gen(B1)$ corresponds with the assignment $i = 16$ for example and so do all others respectively.

The second of the data-flow equation $prsv(i)$ denotes the set of assignments which are preserved in this node. Alternatively, one could think $prsv(i)$ as the inverse of the set $kill(i)$ which would denote the set of assignments which have been overridden. In the basic block B5 the assignment $i = 5$ is made. Hence, $prsv(i) = \{i = 5, j = 8\}$ which tells us that in B5 only those two assignments will remain. Put differently: The assignments $i = 10$ and $i = 16$ are "killed" in this block.

Lastly, $reach_{in}(i)$ and $reach_{out}(i)$ are computed. $reach_{in}(i)$ is simply the union of the parent $reach_{out}$ assignments for all cases. While $reach_{out}(i)$ is simply computed using the data-flow equation which was previously given.

7.4 Interprocedural Reaching Definitions

Up until this point only individual procedures have been discussed, now interprocedural reaching definitions will be introduced. Performing an interprocedural reaching definitions analysis allows for more information to be discovered about the assignments in a given program.

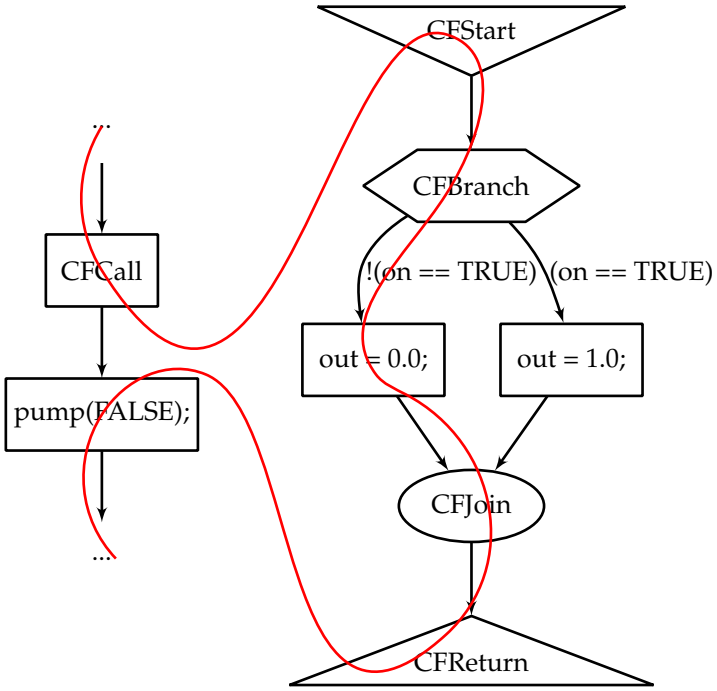


Figure 7.3: Interprocedural reaching definitions example

Figure 7.3 shows the path which would be taken by the algorithm inorder to evaluate these procedures.

The foundation of interprocedural analysis is the call graph which was introduced in chapter 5. The `CFCall` nodes generated are then used to find out where a call resolves to.

Before the reaching definitions form can be computed for the given procedure the following must happen: Firstly, the parameters for the function call must be extracted i.e. the exact assignments that relate to all parameters of the function must be collected. Secondly, a check should be performed to see if the function was already evaluated using these assignments. If the function was previously called with those exact assignments then the prior result should be returned.

Next, $reach_{in}(CFStart)$ is initialized using the relevant set of assignments. This takes the form of mapping nodes in the instance tree, namely the function call parameters, to a set of assignments relevant to them. The correct value for $reach_{out}(CFStart)$ can be assumed because $kill(CFStart) = \emptyset$ and $gen(CFStart) = \emptyset$. Finally, the Reaching Definitions form can be computed as usual by solving the data-flow equations recursively.

We acknowledge that the there are some major downside of this approach. Namely, the exponential runtime with respect to call stack depth might present a problem. While some algorithms do exist [12] to resolve this, they were simply out of the scope of this project.

7.5 Algorithm for Constructing Reaching Definitions

Prior to understanding the implementation of our reaching definitions algorithm, we must briefly familiarize ourselves with how the GASTM represents definitions. In the GASTM definitions are represented as `BinaryExpressions` with their operator set to `Assign`. These `BinaryExpressions` have a left and a right operand. We can extract these definitions from a statement using a `GASTMSwitch` and `GASTTraverser` which was discussed at length in section 3.1. This is necessary since certain programming languages like Java allow multiple assignments within one statement. The `GASTTraverser` will iterate over the tree and the `GASTMSwitch`

will be applied to every node within the tree. If a `BinaryExpression` containing an assignment is found, we can add it to the list of definitions.

Algorithm 2 Reaching Definitions Fixpoint Algorithm

```
1:  function reachingFixpoint(cfg)
2:      work ← {cfg.getStart()}
3:      while work ≠ ∅ do
4:          node ∈ work
5:          work ← work \ {node}
6:          old ← node.assigns
7:          node.assigns ← {}
8:          for all (pred, node) ∈ cfg do
9:              node.assigns ← join(node.assigns, pred.assigns)
10:         switch node.getType() do
11:             case CFStatement
12:                 for all expr ∈ node.getStatement() do
13:                     if expr is Assignment then
14:                         node.assigns ← node.assigns \ {(expr.getAssigned(), x)|
                                (expr.getAssigned(), x) ∈ node.assigns}
15:                         node.assigns ← node.assigns ∪ {(expr.getAssigned(), expr)}
16:             case CFCall
17:                 otherCFG ← node.getCFG()
18:                 otherCFG.getStart().assigns ← node.assigns
19:                 reachingFixpoint(otherCFG)
20:                 node.assigns ← otherCFG.getEnd().assigns
21:         if old = ∅ ∨ old ≠ node.assigns then
22:             for all (node, succ) ∈ cfg do
23:                 work ← work ∪ {succ}
```

The above algorithm 2 is a sketch of our Reaching Definitions implementation. It is a fixpoint algorithm which eventually converges to the set of assignments for every node.

The idea behind our implementation is essentially a graph traversal algorithm similar to DFS or BFS. To start, the work set is initialized with the starting node of the graph. The current node is processed and then inspected for the next nodes which should be expanded. These nodes are then added to the workset. Unlike DFS / BFS our algorithm does not terminate if all nodes in the graph have been

visited. The algorithm terminates once a fixpoint in the assignments has been reached, so the algorithm continues to run until the set of assignments becomes "stable".

Whenever a new node is visited, the assignments of all parent nodes are joined into one set. As described previously in the computation of $reach_{in}$ in section 7.3. Then a decision is made as to what node we are dealing with i.e. `CFStatement` or `CFCall`.

If a `CFStatement` is encountered, then it is searched for assignments as described earlier. Every assignment in the statement is handled as follows: First, all other assignments that are overwritten by this assignment are located, essentially the computation of the $kill(i)$ set follows. Then the set of assignments that are no longer valid $kill(i)$ is subtracted from the total set of assignments. Second, the set of assignments generated in this node $gen(i)$ is computed and added to the set of assignments.

If a `CFCall` is encountered, then the target CFGraph of this function call is located. To begin, $reach_{in}(CFStart)$ for the target CFGraph is set to the set of assignments which hold in the current node. Next, the algorithm is applied recursively to the target CFGraph. Finally, the results of the recursive application are propagated.

Another noteworthy implementation detail about the sets of assignments is the use of the functional collection library Vavr [1]. This library makes it possible to create quick immutable copies of the sets of assignments. Being able to create these immutable copies is incredibly helpful because modified copies of these sets are attached to every node. So, one would typically either be stuck using bit sets or have to resort to copying around large amounts of data. What Vavr allowed us to do was to pick a middle path between using bit sets and having the convenience associated with regular sets.

8 Constant Folding

8.1 Basic Principle of Constant Folding

In [9] constant-folding is defined as: "Constant-expression evaluation, or constant folding, refers to the evaluation at compile time of expressions whose operands are known to be constant."

Put differently, constant folding allows us to take simple expressions such as $(3 + 5) * 7$ and evaluate them to their respective constants (56 for this example). If hypothetically the goal was to optimize a piece of code, then one could replace this constant expression by its constant value.

Constant folding also creates some challenging problems when language independence is a requirement. For example: Primitives such as booleans should work exactly the same across multiple languages, however, other data-types such as floating point numbers may present interesting edge cases. Therefore, it becomes obvious that some assumptions have to be made about the underlying data-type, which is encountered. For the purposes of our work, we assumed that floating point numbers conform to the IEEE 754 [5] standard.

In part because of how easy constant folding is to implement, it has found a wide range of uses cases and applications including - but not limited to - compiler optimizations, detection of code smells, and dead code elimination.

Constant propagation takes the value of the computed constant expressions and propagates them to subsequent statements. The idea here is that both steps become interleaved: A CF step is followed by a CP step until a fixpoint is reached. This technique allows for far more constant expressions to be evaluated, which results in a significantly more optimized result.

Constant propagation and reaching definitions go hand in hand, by virtue of trying to achieve similar goals: The reaching definitions algorithm propagates sets of assignments. The constant propagation algorithm on the other hand propagates sets of constant values. This similarity can be exploited inorder to execute both algorithms at once. Taking advantage of this can be quite beneficial as it helps to avoid duplicate code and reduces the runtime ever so slightly. In addition, adding support for interprocedural constant propagation was aided by our implementation of interprocedural reaching definitions, as described in section 7.4. Thus, our implementation of reaching definitions invokes constant folding whenever a statement, function call, or branch is encountered.

However, rather than simply propagating all constants, we opted to compute all possible "outcomes" of constant propagation and collect the results into a set. Essentially, if a constant assignment is encountered, the set of possible values is updated to reflect the new value of the variable. Whenever a `CFJoin` node is reached, the sets are merged into one set containing all possible values. Arithmetic operations are applied to the set e.g. $x * 2$ with $x \in \{1, 2\}$ results in the set $\{2, 4\}$.

Finally, if a loop is encountered, the set of all values is simply invalidated. Meaning that all values of all variables used or affected by the loop become unknown.

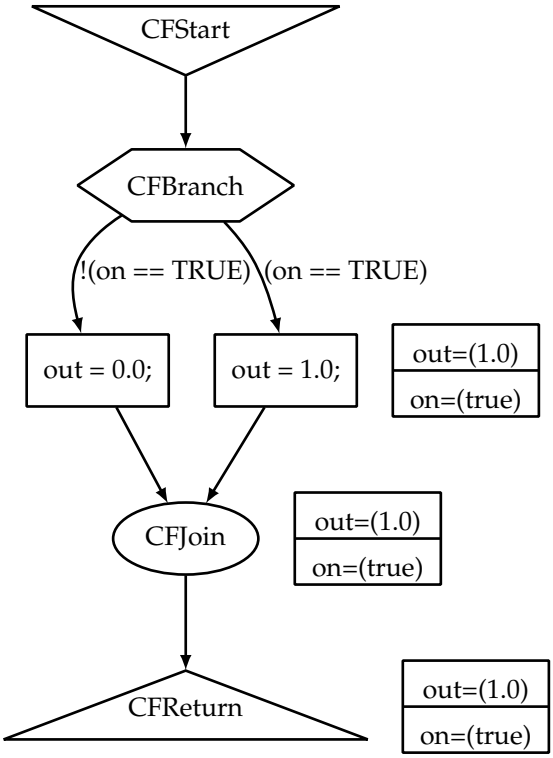


Figure 8.1: A simple example of constant propagation

The above Figure 8.1 illustrates constant propagation on the `pump` function block section 2.2 with `on` set to `TRUE`. Starting at the `CFStart` node `on` is simply propagated. On the right side of the CFG we can observe that the variable `out` is set to `1.0`. The left side is never executed, since the condition in the `CFBranch` evaluates to `FALSE`. Once the `CFJoin` is reached, both sets from each branch are merged. The constant value `1.0` is then returned in the `CFReturn` node. Applying constant propagation to this Function Block therefore revealed that this call to the Function Block can be replaced by the constant expression `1.0`.

8.2 Constant Expression Evaluation

Expressions are combined of literals, operations, identifier, and function calls. If for a given expression a definite value can be computed, than this expression is known as a constant expression. Evaluating constant expressions is referred to as constant folding.

Literals have a clearly defined value associated with them, which the evaluation function will attempt to propagate. Examples of literals include: `IntegerLiteral`, `RealLiteral`, and `BooleanLiteral`.

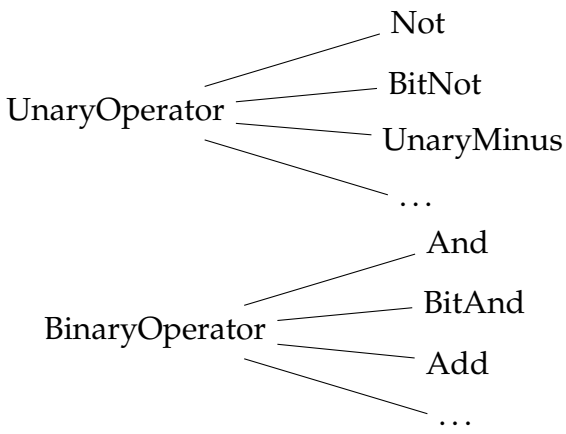


Figure 8.2: Example unary and binary operator types

Figure 8.2 depicts some different kinds of unary and binary operators which an expression in the GASTM may be comprised of. Unary operators as their name suggests have only one parameter while binary operators manipulate two values.

Operations are assigned rules of precedence which dictate how expressions such as $5 + 2 * 3$ are evaluated i.e. $(5 + 2) * 3$ vs. $5 + (2 * 3)$. The constant folding algorithm operates on a subset of the AST which contains expressions and the components which they are made of. These expressions are represented in the GASTM as nodes in a tree like structure. From this representation the rules of precedence become clear.

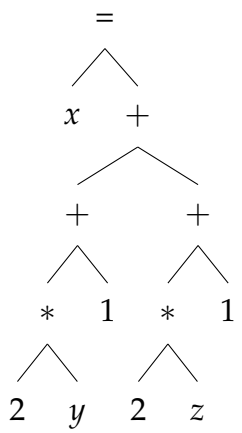


Figure 8.3: Example of an AST which eval would process

The above Figure 8.2 shows a simplified example of an expression in the GASTM representing $x = (2 * y + 1) + (2 * z + 1)$. This AST would then be evaluated bottom up recursively until only atomic values remain. If a variable whose value is undefined is encountered then the result of the evaluations is typically undefined.

8.2.1 Identifiers

Identifiers are the last missing piece of the puzzle. These come in two forms: `IdentifierReferences` and `QualifiedIdentifierReferences`.

`IdentifierReferences` are identifiers (essentially some string), which refers to a `DefinitionObject`. `DefinitionObjects`, according to the OMG-ASTM specification [4] are defined as "Constructs that define entities". To paraphrase they are defined as any of the following: variable definitions, type definitions, function definitions, etc. Figuring out which `DefinitionObject` is referenced by an identifier, is the job of the resolver. Our resolver implementation attempts to match some portion of the IT defined in 6.1 with the given identifier. We can therefore, associate a value with an identifier.

8.2.2 QualifiedIdentifierReferences

`QualifiedIdentifierReferences` (QIRs) are typically used to reference some kind of data or field inside an aggregate type (e.g. `person.name`). QIRs act upon an expression on the left and access some member on right. Inorder to match the QIR to some portion of the IT, two parts need to be resolved: The expression and the member which acts upon the expression. In the case of the expression only the expression type needs to be determined. The member is an `IdentifierReference` which using the expression type can be matched to a specific part of the instance tree.

These QIRs may become quite complex and for some languages even indeterminate. An example of more complicated QIR is: `("a"+"b").toString()`. Where `("a"+"b")` is the expression being acted upon and `toString` is the member being referenced.

The Figure below 8.4 depicts the AST which would be generated for the expression `a.b.c.d`, along with the relevant part of the Instance Tree of the program associated with this expression.

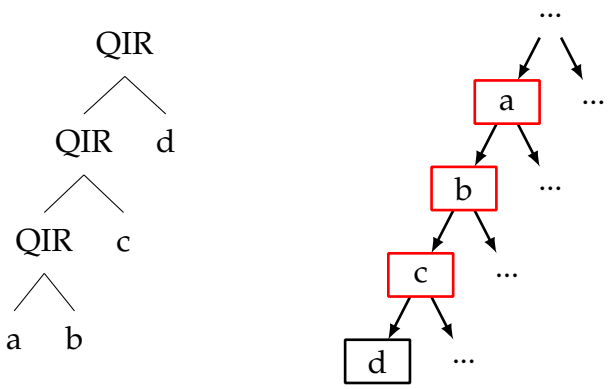


Figure 8.4: An illustration of an AST and its IT

The expression is evaluated bottom up, starting at `a.b`. First the qualifier and then the member must be resolved. For this the IT is used, by simply following the path shown in the IT one can obtain both the value and type information for such an expressions. Once the correct node has been located it can be assigned a value.

Algorithm 3 Simplified Resolving Algorithm

```
1: function resolve(qir)
2:     if qir.qualifier is QIR then
3:         n ← resolve(qir.qualifier)
4:     else
5:         n ← it.getMember(qir.qualifier.name)
6:     return n.getMember(qir.member.name)
```

Algorithm 3 shows exactly how the above figure 8.4 and other problems like it can be resolved.

8.2.3 Putting it all together

Algorithm 4 Constant Folding Algorithm

```
1:  function eval(expr)
2:      r ← {}
3:      if expr is Literal then
4:          r ← {expr.value}
5:      else if expr is BinaryExpression then
6:          switch expr.op do
7:              case Assign
8:                  left ← resolve(expr.left)
9:                  left.value ← eval(expr.right)
10:                 r ← left.value
11:             case Add
12:                 r ← execute(expr, +)
13:             case Sub
14:                 r ← execute(expr, −)
15:             case Times
16:                 r ← execute(expr, ∗)
17:             ...
18:     else if expr is UnaryExpression then
19:         switch expr.op do
20:             case Not
21:                 r ← execute(expr, !)
22:             case UnaryMinus
23:                 r ← execute(expr, −)
24:             ...
25:     else if (expr is IR) ∨ (expr is QIR) then
26:         r ← {resolve(expr).value}
27:     return r
```

The above algorithm 4 combines the steps discussed in the previous sections. First a decision is made if the given expression is any of the following: literal, binary expression, unary expression, or identifier. If the given expression is a literal then we can simply return the value associated with the literal.

If the expression is a binary expression, then we must distinguish between two cases. The first case is an assignment: Here we have to resolve the assignee and evaluate the assigned.

Algorithm 5 Execute Binary Expression

```
function execute(binexpr, op)
    r ← {}
    left ← eval(binexpr.left)
    right ← eval(binexpr.right)
    if left = right then
        for all value ∈ pou do
            r ← r ∪ {value op value}
    else if left ≠ {} ∧ right ≠ {} then
        for all value_1 ∈ left do
            for all value_2 ∈ right do
                r ← r ∪ {value_1 op value_2}
    return r
```

If a binary expression is encountered algorithm 5 is invoked with the appropriate operator. This algorithm evaluates both the left operand and right operand and then applies to the cartesian product of the two sets.

Algorithm 6 Execute Unary Expression

```
function execute(unaryexpr, op)
    r ← {}
    for all value ∈ eval(unaryexpr.operand) do
        r ← r ∪ {op value}
    return r
```

Both IRs and QIRs are resolved using the algorithm 3, which was presented earlier. Finally, algorithm 4 returns the empty set if the overall result is undefined, otherwise the set of values represents the possible values that the given expression might hold. If the size of the result is one, then the expression may be optimized to simply yield the value in the set.

8.2.4 Dealing with references

Up until now we left references entirely. However, now that the basics have laid out, references will be relatively easy. The instance tree already tracks all allocated instances ergo, if a reference is assigned some value, we can simply update the IT model to reflect this change.

```
struct { struct S {
    int value;
    struct S *ptr;
} a, b; } s;

void a() {
    s.a.ptr = &s.a;
    s.a.ptr = &s.b;
}
```

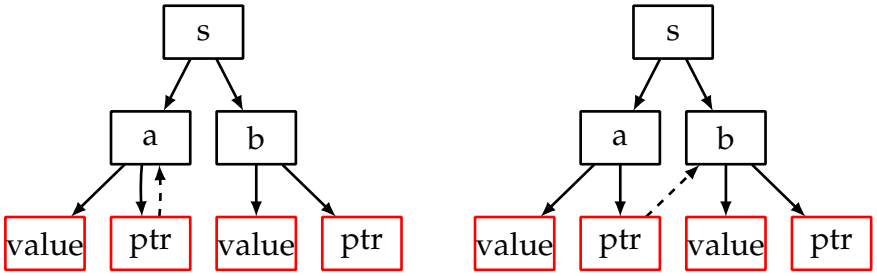


Figure 8.5: A example showing references in the IT

Figure 8.5 depicts how the example code would be reflected inside the IT. At first `s.a.ptr` points to `s.a` then the pointer is adjusted to point to `s.b`;

9 Conclusion and Outlook

9.1 Summary

In these theses we presented a framework for static code analysis of languages based on the IEC 61131–3 standard. Based on a GASTM representation of the source code, intermediate representations are generated, which can then be checked to generate knowledge about information in the code. We then presented the intermediate representations that were used.

First we introduced the CFG which is used to analyse the flow of each function by transforming the AST into a directed graph. The graph then represents all possible execution paths that can be taken on the graph. We showed how this graph can be constructed using a recursive algorithm to deal with complex code structures. We then introduced the CG as an extension of the CFG which adds the potential for contextual awareness to each function. Next, we showed how the data structure of a project is represented in an Instance Tree (IT). The IT shows the POUs and variables allocated by a program. We then gave a summary of the concept of reaching definitions before showing a practical application for the intermediate representations by applying a constant folding algorithm to our data structures.

The algorithms and classes presented in this paper were implemented as a Java framework and tested on several example programs in various languages. The implementation was then used in [10], where a tool for symbolic code execution based on our intermediate representation is presented.

9.2 Open Issues

Although the implementation covers a wide range of analysis methods, some open issues remain:

- **Nested Function Calls**: One issue that our implementation has is an inability to deal with nested function calls, as well as multiple calls within a single statement.

  The current implementation of the Call Graph builder is unable to accurately determine the order of multiple and potentially nested function calls within a single statement. Due to limitations of our code, the `CFCalls` for multiple calls are inserted as well as possible, but not necessarily correctly. These issues also extend to the constant folding algorithm, which has no concept of intermediate values, such as return values of functions passed on to other functions. Because of this, only code with one function call per statement can be guaranteed to be analysed correctly.

- **Further Analysis Methods**: With the constant folding algorithm, our toolchain provides a tool for code analysis which is based on our intermediate representations. However, there are plenty further options for refining and expanding our toolchain. For example, it could be possible to implement a framework for rules, similar to the one presented in [11]. Another option would be to use the intermediate representation to then perform symbolic execution on it, expanding on the work done in [10].

- **Language Constructs**: Due to time limitations not all language constructs defined by IEC 61131–3 were implemented in our algorithm. One notable example for this is the switch statement, which is currently not supported in the CFG .

JOHANNES KEPLER
UNIVERSITY LINZ

Bibliography

[1] URL: https://www.vavr.io/.

[2] Frances E. Allen. "Control Flow Analysis". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: https://doi.org/10.1145/390013.808479.

[3] JT Davie and Ronald Morrison. *Recursive descent compiling*. John Wiley & Sons, Inc., 1982.

[4] Object Management Group. "Abstract Syntax Tree Metamodel". In: (2011). URL: https://www.omg.org/spec/ASTM/1.0/PDF.

[5] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[6] International Electrotechnical Commission. *IEC 61131-3 Ed. 3.0 en:2013: Programmable controllers — Part 3: Programming languages*. 2013. URL: https://webstore.iec.ch/publication/4552.

[7] International Electrotechnical Commission. *IEC 61131-3:1993: Programmable controllers — Part 3: Programming languages*. 1993. URL: https://webstore.iec.ch/publication/19080.

[8] *MoDisco User Guide — help.eclipse.org*. https://help.eclipse.org/latest/nav/58_0. [Accessed 03-Dec-2022].

[9] Steven S Muchnick. *Advanced Compiler Design and Implementation*. Oxford, England: Morgan Kaufmann, Sept. 1997.

[10] Peter Pfeiffer. "A Tool for verifying PLC Safety Components using Symbolic Execution". In: (2022).

[11] Herbert Prähofer et al. "Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application". In: *IEEE Transactions on Industrial Informatics* 13.1 (2016), pp. 37–47.

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61. ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

[13] Nieke Roos. "Programming plcs using structured text". In: *International Multiconference on Computer Science and Information Technology*. Citeseer. 2008, pp. 20–22.

[14] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. "Global value numbers and redundant computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 12–27.

[15] Valeriy Vyatkin. "Guest editorial: Special section on software engineering in industrial automation". In: *IEEE Transactions on Industrial Informatics* 9.4 (2013), pp. 2337–2339.