Dataflow and State Machine Extraction from C/C++ Code

Lewe Andersen

Master's Thesis
December 2019

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth
M.Sc. Niklas Rentz

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

An important part of software development besides writing the actual source code is the creation and maintenance of documentation. A good documentation can help new developers to gain an overall better understanding about an existing project or help by maintaining older parts of a project. This task is very time consuming and often neglected, which results in not existing or untrustworthy documentation.

To overcome this problem this thesis introduces methods to automatically extract information out of C/C++ code to create an SCCharts model, which can be visualized by the layout tool chain of KIELER to create actual documentary diagram views.

Two different kinds of models are extracted. The first represents an algorithm with a dataflow graphic. It displays the in- and outputs of the algorithm and connects them with actors to indicate their changes during the calculation. The second model represents a state machine defined in the source code. It contains a representation for the defined states, input events, and transitions.

Acknowledgements

In the first place, I would like to thank my advisers Niklas Rentz and Steven Smyth as well as Professor Dr. Reinhard von Hanxleden not only for giving me the opportunity to work on this topic, but also for the time they invested in guiding and reviewing this thesis.

Further, I want to thank all members of the Real-Time and Embedded Systems Group at the Kiel University for their creative thoughts and suggestions on design questions coming up during my work on the thesis.

Contents

List of Acronyms **ix**

1 Introduction **1**
1.1 The Dataflow Approach . . . 1
1.2 State machines . . . 2
1.3 Problem Statement . . . 5
1.4 Outline . . . 5

2 Used Technologies **7**
2.1 KIELER . . . 7
2.1.1 SCCharts . . . 8
2.1.2 KiCo . . . 10
2.1.3 KLighD . . . 11
2.1.4 ELK . . . 12
2.2 CDT . . . 12

3 Related Work **15**
3.1 Extraction . . . 15
3.1.1 Control and Dataflow extraction from VHDL . . . 15
3.1.2 State machine extraction form HDL . . . 16
3.2 Code Comprehension . . . 16
3.2.1 ExplorViz . . . 16
3.2.2 Sourcetrail . . . 17
3.3 Visualization . . . 18
3.3.1 SCADE Suite . . . 18
3.3.2 LabView . . . 18
3.3.3 Ptolemy II . . . 19
3.4 Related Theses . . . 19
3.4.1 Model-based Compilation of Legacy C Programs . . . 20
3.4.2 From Lustre to Graphical Dataflow Programs . . . 20

4 Concept **23**
4.1 Dataflow Extraction . . . 23
4.1.1 SCChart Dataflow . . . 23
4.1.2 C Elements . . . 24
4.1.3 Static Single Assignment . . . 32
4.1.4 Not Supported Constructs . . . 35
4.1.5 Structflow Abstraction . . . 36
4.2 State Machine Extraction . . . 37
4.2.1 State Machines in C++ . . . 37
4.2.2 Extract the Centralized State Machine . . . 42
4.2.3 Extract the Class-Based State Machine . . . 46

4.3 Highlighting . . . 47
4.3.1 Highlight Annotations . . . 47
4.3.2 Highlighting Dataflow . . . 48
4.3.3 Highlighting State Machines . . . 48

5 Implementation **51**
5.1 C Parser . . . 51
5.2 Dataflow Extraction . . . 52
5.2.1 Function Extraction . . . 53
5.2.2 Compound Statement Extraction . . . 55
5.2.3 Control Statement Extraction . . . 55
5.2.4 Expression Extraction . . . 59
5.2.5 Valued Object Handling . . . 60
5.3 Statemachine Extraction . . . 62
5.3.1 Centralized Statemachine . . . 62
5.3.2 Class Based State Machine . . . 65

6 Evaluation **67**
6.1 Dataflow Extraction . . . 67
6.1.1 Basic Extraction Example . . . 67
6.1.2 Structflow Extraction Example . . . 68
6.1.3 Diagram Evaluation . . . 70
6.2 Statemachine Extraction . . . 70
6.2.1 Centralized State Machine Extraction Example . . . 71
6.2.2 Class-Based State Machine Extraction Example . . . 71
6.2.3 Diagram Evaluation . . . 72

7 Conclusion **75**
7.1 Future Work . . . 75
7.1.1 Missing Features for the Dataflow Extraction . . . 75
7.1.2 Filtering . . . 75
7.1.3 Additional State Machine Functionality . . . 76
7.1.4 Work on State Machine Patterns . . . 76
7.1.5 Secondary Notation . . . 76
7.1.6 Combining the Extractions . . . 77

Bibliography **79**

List of Acronyms

AST	Abstract Syntax Tree
CDT	C/C++ Development Tooling
ELK	Eclipse Layout Kernel
GCD	Greatest Common Divisor
HDL	Hardware Description Language
IDE	Integrated Development Environment
KiCo	KIELER Compiler
KLighD	KIELER Lightweight Diagrams
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
MoC	Model of Computation
SCADE	Safety Critical Application Development Environment
SCChart	Sequentially Constructive StateChart
SSA	Static-Single-Assignment
UI	User Interface
VHDL	Very High Speed Integrated Circuit Hardware Description Language

List of Listings

1.1 C code of an exemplary arithmetic function . . . 2
1.2 Definitions of states, events, and other variables . . . 4
1.3 Definition of core function with the behaviour switch statement . . . 4
2.1 Textual representation of the ABRO program . . . 9
2.2 Textual representation of the dataflow model . . . 9
2.3 Textual representation of the referencing SCChart . . . 10
2.4 Textual representation of the referenced SCChart . . . 10
2.5 A small function in C Code . . . 13
4.1 The source code of a function definition . . . 25
4.2 The source code of a small `CompoundStatement` . . . 26
4.3 The source code of a `CompoundStatement` including a function call . . . 28
4.4 The source code of a `while` statement . . . 29
4.5 The source code of a `for` statement . . . 30
4.6 The source code of an `if` statement . . . 31
4.7 The source code of a `switch` statement . . . 33
4.8 The source code for a couple of sequential assignments . . . 34
4.9 The source code for a couple of sequential assignments transformed into an SSA form . . . 35
4.10 The source code for an exemplary function of the structflow abstraction . . . 37
4.11 The `enum` to define the states . . . 39
4.12 The `enum` to define the trigger events . . . 39
4.13 The local variables and functions needed excluding the central `switch` statement . . . 39
4.14 The central behaviour `switch` statement . . . 39
4.15 The core state machine class . . . 41
4.16 The abstract class inherited by each state class . . . 41
4.17 The `enums` to list the state machines states and input events . . . 41
4.18 The evaluate function of one of the state classes . . . 41
4.19 The source code of the state `enum` with marking comment . . . 43
4.20 The source code of the central `switch` statement . . . 44
4.21 The evaluation function is marked by a comment in the definition of the abstract `Statebase` class . . . 47
5.1 The `process` function of the parser processor . . . 52
5.2 The global variables used for the dataflow extraction . . . 52
5.3 The creation of a function state with its name . . . 53
5.4 The extraction of the inputs for the function state . . . 54
5.5 The extraction of the output for the function state . . . 54
5.6 The function to translate a `while` statement into SCCharts dataflow . . . 56
5.7 The function to find all read variables of the given statement . . . 57
5.8 The translation of the `switch` statement's body without the translation of `case` and `default` statements . . . 58
5.9 The creation of the `case` and `default` states inside the `switch` statement . . . 58
5.10 The algorithm to translate a `FunctionCallExpression` . . . 60
5.11 The first part of the algorithm to retrieve a valued object by its variable's name . . . 61

5.12 The algorithm to find the AST node following the given comment node in the source code 63
5.13 The algorithm to extract the transitions out of the central behaviour `switch` statement . 64
5.14 The algorithm to extract the transitions out of the evaluation function of a state's class 66
6.1 The source code of the `euclid` function 67

List of Figures

1.1 A dataflow graphic and the corresponding source code . . . 2
a Dataflow graphic for the presented function . . . 2
1.2 State machine graph for a simple coffee vending machine . . . 3
1.3 A possible implementation of the coffee vending machine in C code . . . 4

2.1 Example view on the KIELER user interface . . . 7
2.2 SCCharts implementation of the basic synchronous program ABRO . . . 9
a Graphical representation of the ABRO program . . . 9
2.3 SCCharts implementation of a simple dataflow model . . . 9
a Graphical representation of the dataflow model . . . 9
2.4 A simple SCChart that uses a reference of another SCChart . . . 10
a Graphical representation of the referencing SCChart . . . 10
2.5 Transformation process of KLighD . . . 11
2.6 Overview about the ELK infrastructure . . . 12
2.7 An AST and its corresponding C code . . . 13
a The corresponding AST . . . 13

3.1 Example of an ExplorViz 3D city model . . . 16
3.2 Example of a Sourcetrail project . . . 17
3.3 An example SCADE dataflow graph . . . 18
3.4 An example Model in LabView . . . 19
3.5 Example models in Ptolemy II . . . 20
a Ptolemy dataflow model . . . 20
b Ptolemy state machine model . . . 20

4.1 The components of an SCCharts dataflow model . . . 24
4.2 A function declaration shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 25
a The AST of a function definition given by the CDT . . . 25
b The SCCharts representation of the function definition . . . 25
4.3 A small `CompoundStatement` shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 26
a The AST of a small `CompoundStatement` given by the CDT . . . 26
b The SCCharts representation of a small `CompoundStatement` . . . 26
4.4 A `CompoundStatement` with a function call shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 28
a The SCCharts representation of a `CompoundStatement` with a function call . . . 28
b The AST of a `CompoundStatement` including a function call given by the CDT . . . 28
4.5 A `while` statement shown in code with the corresponding AST, the final representation in SCCharts dataflow, and the possible design of a state with a condition port . . . 29
a The SCCharts representation of a `while` statement . . . 29
b The AST of a `WhileStatement` node given by the CDT . . . 29

c Concept of a `while` state with condition port . . . 29
4.6 A `for` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 30
a The SCCharts representation of a `for` statement . . . 30
b The AST of a `ForStatement` node given by the CDT . . . 30
4.7 An `if` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 31
a The AST of an `if` statement given by the CDT . . . 31
b The representation of an `if` statement after the translation in SCChart dataflow . 31
4.8 A `switch` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow . . . 33
a The SCCharts representation of a `switch` statement . . . 33
b The AST of a `SwitchStatement` node given by the CDT . . . 33
4.9 The result of the translation of sequential assignments following the so far established translation rules . . . 34
a The SCCharts representation following the so far established translation rules . . 34
4.10 The result of the translation of sequential assignments with the extension of SSA . . . 35
a The SCCharts representation after the translation with the use of SSA . . . 35
4.11 The representation of a function for the structflow abstraction . . . 37
a The SCCharts representation of the exemplary function . . . 37
4.12 All elements of the definition concept for the centralized state machine approach . . . 39
4.13 All elements of the definition concept for the class-based state machine approach . . . 41
4.14 The representation of states in the centralized state machine approach and its respective AST . . . 43
a The AST of the state `enum` . . . 43
4.15 The central `switch` statement with the corresponding AST representation . . . 44
a The AST for one of the `if` statements in the central `switch` statement . . . 44
b The AST for the central `switch` statement without the children of `if` statement nodes . . . 44
4.16 The extracted state machine graphic for the coffee vending machine example . . . 45
4.17 A screenshot of KIELER showing an example of the highlighting for a small dataflow model . . . 48
4.18 A screenshot of KIELER showing an example of the highlighting for the centralized state machine extraction . . . 49
4.19 A screenshot of KIELER showing an example of the highlighting for the class-based state machine extraction . . . 49

6.1 The code for the `euclid` function with the extracted representation . . . 67
a The extracted diagram of the `euclid` function . . . 67
6.2 The extracted diagram of the structflow from one of the pitch tracking algorithms from Reuter . . . 68
6.3 A cut-out of the extracted graphic of one expanded function state . . . 69
6.4 The extracted state machine from the ardumower project . . . 72
6.5 The extracted state machine from the class-based industry example . . . 73

List of Tables

6.1 The results of the survey to evaluate the extracted dataflow diagram 70
6.2 The results of the survey to evaluate the extracted state machine diagram 74

Chapter 1

Introduction

The process of software development does not only consist of writing the actual source code that builds the software. Another important part of it is the creation and maintenance of documentation about the designed system. A good documentation does not only help the developer to keep a better overview about the project, but also helps others to get a better and faster understanding about the functionality and connections inside the project.

Regardless of these advantages of a well-documented project, the documentation for many projects is outdated, as stated by different surveys such as the one done by Lethbridge et al. [LSF03] or by Singer [Sin98]. Singer states that even if most developers appreciate good documentation, the time needed for its creation or maintenance does lead to inconsistency, which further leads to a lack of trust from the developers to the documentation. Therefore even if documentation exists, Forward et al. [FL02] show that the more specialized the documentation becomes the more rarely it is used.

A possible solution for a part of this problem is the introduction of code comprehension tools. These are tools that automatically create views or graphics for a given project to assist a developer with the overview about different aspects of the project. For example, the tool ExplorViz (described in more detail in Section 3.2.1) reads a whole software system and creates a visualization about all parts of the system and the connections between them.

The benefit of these tools is that the time needed for the manual creation and maintenance of such graphics is taken away from the user by automating this process, which can lead to a more detailed and more up to date documentation.

On the other hand the open source modelling tool Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) does not only provide the possibility for the textual development with the language Sequentially Constructive StateCharts (SCCharts), but also comes with an integrated tool chain to automate the creation of a visual representation of said language. Therefore, if a method could be found to extract certain information out of given source code and translate it into SCCharts, it would be possible to automatically create a visualization for this information, which can be used as documentation for the extracted information of the given code, without the need for the time consuming task to manually create these graphics.

1.1 The Dataflow Approach

The approach to define a function—in a programming language like C—is to sequentially connect different assignments, function calls or control statements.

In the dataflow approach a function is designed similar to an electrical circuit. It has defined in- and outputs which are connected through different actors to represent the behaviour of the described model. Listing 1.1 shows a simple arithmetic function defined in C code and Figure 1.1a shows a representation of this function in a dataflow graphic. The function receives three values x, y, and z as parameters and sequentially uses arithmetic assignments to return one single result. First, y is

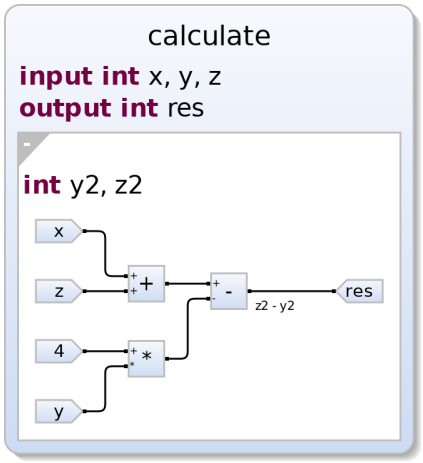
```c
int calculate(int x, int y, int z) {

  int y2 = 4 * y;
  int z2 = x + z;
  int res = z2 - y2;

  return res;

}
```

Listing (1.1) C code of an exemplary arithmetic function



(a) Dataflow graphic for the presented function

Figure 1.1. A dataflow graphic and the corresponding source code

multiplied by 4 and stored in a local variable $y2$, then x is added to z and stored in a local variable $z2$. Finally, $z2$ is subtracted by $y2$ which builds the result of the function.

In the dataflow graph this function is represented like a data circuit with 3 variable inputs, corresponding to the parameters of the C function, one fixed input 4 and one output, in this example called *res*. The inputs x and z are connected with the inputs of an add-actor and y and 4 are connected with the inputs of a multiply-actor. The outputs of these two actors are then connected to the inputs of a subtract-actor, the output of which is connected to the output of the function. As soon as all inputs of such a dataflow function are present the calculation is started, thus the first two actors for the addition and multiplication are executed concurrently. Sequentiality can only be presented by connecting the input of one actor with the output of another.

This kind of representation gives a good overview about where and how different values are used in a function, as already shown by Davis et al. [DK82]. On the first look a user can for example see where a specific input is used or which steps are needed to calculate the result. This helps a developer to get a fast understanding about where the different data a function is using is coming from. It can also help with bug fixing by quickly analysing where a variable is last used without the need to navigate through multiple lines of code.

1.2 State machines

A state machine is a system used to control the behaviour of a digital system. It communicates to its environment with pre-defined in- and outputs and internally consists of a finite set of states and a finite set of transitions which connect these states. Each transition has a certain trigger that—if it resolves to `true`—leads to a state change along the transition and a potential change in the outputs. The easiest way for a visual representation of such a state machine is a graph, where each state is represented by a node and each transition by a directed edge.

Figure 1.2 shows the visual representation of a simple coffee vending machine. The state machine models a coffee vending machine that gives coffee at a price of 1.50 Euro without giving change. It

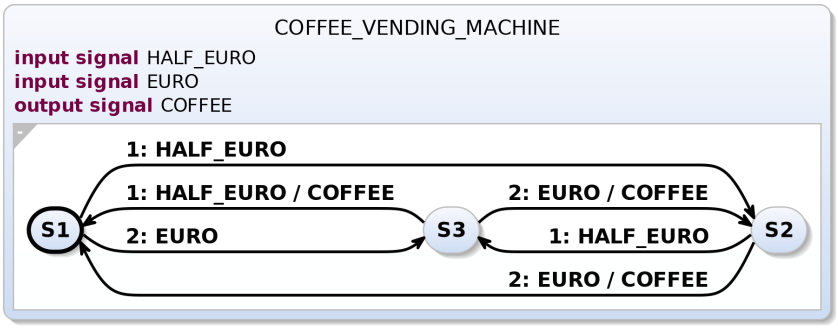


Figure 1.2. State machine graph for a simple coffee vending machine

only accepts 50 cent or 1 Euro coins. It consists of 3, states each with two outgoing and two incoming transitions. The bold encirclement around the state *S*1 marks the initial state that is the state at which the model begins its execution. The graphic shows that with the input of a 50 cent coin the active state of the model switches to *S*2 and the input of a 1 Euro coin switches to state *S*3. This shows that the states correspond to following conditions:

S1 → no payment received

S2 → 50 cent received

S3 → 1 Euro received

Each outgoing transition of a state corresponds with one of the two kinds of payment and adds up over the states. Each time an amount of 1.50 Euro is reached the output signal coffee is given.

A possible implementation of this state machine in C code is shown in Figure 1.3. Here the states are defined with an `enum` in line 2 to 6 of Listing 1.2 and the current state is saved inside the variable `currentState`. A transition from one state to another is initiated by a call of the `changeState` function. The active input is calculated by the `calculateEvent` function, and whether the coffee output signal is given is saved in the `COFFEE` variable. The `main` function starts the state machine by setting the initial state in line 3 of Listing 1.3. Thereafter starting an infinite loop that with each iteration first resets the in- and output of the previous iteration, followed by calculating the current input. After that one `switch` statement tests whether a transition has to be made by first checking which state is active and thereafter checking if the input matches the condition for a state change, which can be seen in lines 14 to 20 of Listing 1.3 for state `S1`.

This kind of state machine definition surely does implement the state machine shown in Figure 1.2. Nevertheless, the problem with a code definition of a state machine is that it needs a lot of code for a state machine even as simple as the shown example. What can be clearly seen is that with each additional state, not only an extra entry for the `enum` is needed, but also another `case` block needs to be added to the central `switch` statement, and the addition of an event leads to a similar increase of code size. This means that the larger the defined state machine is, the more complex and therefore confusing the source code becomes. The graphical representation on the other hand gives a clear overview about what states there are, which in- and output events are needed, and how the states are connected. This shows the big difference in comprehensibility between a state machine in source code and its graphical representation that becomes even bigger, the more complex the state machine is.

```c
// State Definitions
enum States{
  S1,
  S2,
  S3
};

// Event Definitions
enum Events{
  HALF_EURO,
  EURO
};

// Control Variables
int COFFEE;
int event;
int currentState;

// Transition Function
void changeState(int nextState) {
  currentState = nextState
}

// Function to determine the Inputs
void calculateEvent() {
  // Read Inputs
}
```

Listing (1.2) Definitions of states, events, and other variables

```c
int main() {
  // Setting the Inital State
  currentState = S1;
  // The main loop to iterate over the
       state machine
  while (1) {
    // Resetting the in and outputs
    COFFEE = 0;
    event = -1;
    // Read the inputs
    calculateEvent();
    // The switch control statement to
         define the SM behaviour
    switch (currentState) {
    // STATE S1
    case S1:
      if (event == HALF_EURO) {
        changeState(S2);
      } else if (event == EURO) {
        changeState(S3);
      }
    break;
    // STATE S2
    case S2:
      if (event == HALF_EURO) {
        changeState(S3);
      } else if (event == EURO) {
        changeState(S1);
        COFFEE = 1;
      }
    break;
    // STATE S3
    case S3:
      if (event == HALF_EURO) {
        changeState(S1);
        COFFEE = 1;
      } else if (event == EURO) {
        changeState(S2);
        COFFEE = 1;
      }
    break;
    }
  }
}
```

Listing (1.3) Definition of core function with the behaviour switch statement

Figure 1.3. A possible implementation of the coffee vending machine in C code

1.3 Problem Statement

The common way for graphical documentation of software is to use a graphical tool and to design the graphic manually with a palette or drag and drop like features. This gives the developer freedom with the choice of the final design of the graphic but takes a lot of time, especially for larger projects. At the latest when the project has changed and the graphic has to be updated, the originally well-designed layout does not longer serve its purpose. Then again a lot of time has to be invested to redesign the graphic such that it can clearly fulfil its purpose of software documentation.

To overcome this time consuming process this thesis will present methods to extract dataflow and state machines out of given C/C++ files and transform them into SCCharts. Due to the integration of automatic layout with the help of KIELER Lightweight Diagrams (KLighD) and Eclipse Layout Kernel (ELK) into KIELER, these SCCharts can then be used to create easy to read documentation graphics. This way the design of a documentation graphic takes significantly less time as if done manually and also the update of a graphic after a change in the source code does not end in hours of redesigning a graphic.

1.4 Outline

This section gives an overview about the structure of the following chapters of the thesis. Chapter 2 presents the technologies used for this thesis. An overview about KIELER and its used features are given and the Eclipse C/C++ Development Tooling (CDT) is presented. The following Chapter 3 introduces some related work. It mentions other work about the extraction of dataflow or state machines from source code and presents different tools that are either designed for code comprehension or have capabilities for the visualization of dataflow or state machines. Further some other theses of the research group with a related topic will be mentioned. Chapter 4 explains the concept of the thesis. It first addresses the dataflow extraction, how dataflow is handled in SCCharts, which C code constructs can be translated, and how they are translated. After that, the state machine extraction will be presented. It shows which different guidelines for state machine definitions in C code can be handled and how they are extracted. Chapter 5 gives a more detailed look on how the concept was implemented, first by giving a short look on the parsing of the source code and after that first explain the dataflow extraction implementation followed by the state machine extraction. In the next step, Chapter 6 evaluates the results of the thesis. It compares the extracted graphics with the given source code and tries to evaluate the efficiency of the presented tooling. Finally, Chapter 7 summarizes the results of this thesis and gives a conclusion before a short overview over potential future work is given.

Chapter 2

Used Technologies

For the extraction and presentation of dataflow and state machines from C/C++ code some pre-existing technologies were used. This chapter gives a more detailed view on the different tools and features that were used for the implementation. First, the tool KIELER and its used parts are described and after that a closer look on the CDT and its used features is presented.

2.1 KIELER

Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] is an open source research project focused on the model-based graphical design of complex systems. Based on the open source project *Eclipse* it combines a textual editor with a diagram view for the development with the textual and graphical language SCCharts. It further provides support for other languages like Esterel [Bd91] or Lustre [HCR+91], but for this thesis only the aspect of SCCharts is needed.

Figure 2.1 shows the user interface of KIELER marked with its four main components. Area 1 shows the textual editor used for model development. The visual representation of the designed model is

[1] https://rtsys.informatik.uni-kiel.de/kieler

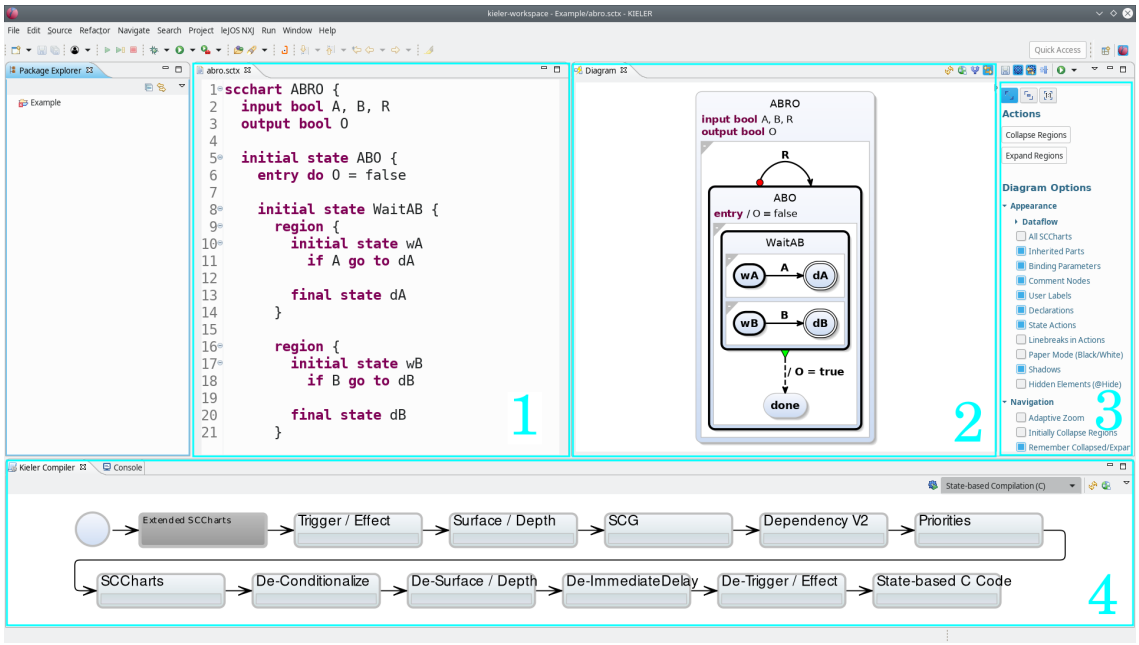


Figure 2.1. Example view on the KIELER user interface

built by an internal tool chain—described in more detail in Section 2.1.3 and Section 2.1.4—every time the model is updated and then shown in Area 2. Area 3 gives some user options for modifications on the created visual representation and Area 4 shows the selected compiler chain that can be used to show the different steps of compilation, which is described in more detail in Section 2.1.2.

2.1.1 SCCharts

The synchronous language Sequentially Constructive StateCharts (SCCharts) extends the classical synchronous Model of Computation (MoC) with the so called sequential constructiveness [HDM+13]. This basically extends the classical synchronous approach under certain preconditions with sequentiality during one calculation step. In its design SCCharts are based on StateCharts by Harel [Har87] and besides that also offer the possibility to define dataflow regions.

SCCharts use both a textual and visual representation to combine the benefits of both. The textual form accelerates the process of development through auto completion and copy-paste features [GKR+14] and, as stated by Sharafi et al. [SMS+13], the visual representation gives, as soon as the user gets used to the model, a better overview about the designed model even if it grows in size.

SCCharts are designed to model state machines by the use of states and transitions. Furthermore, each state can contain multiple regions to model additional behaviour inside a state. There are two different kinds of regions: *controlflow-* and *dataflow-regions*.

SCCharts Controlflow

A controlflow-region basically contains another SCChart; Figure 2.2 shows a basic example of such a hierarchical SCChart in both its textual and graphical representations. Similar to all SCCharts the *ABRO* model has one root state in its top level which contains the behaviour of the model. Like every state this root state can have input-, output- and other variable declarations. Further it has one controlflow-region which contains its behaviour. In the shown example the root state *ABRO* contains only one state named *ABO* which has a strong aborting self transition triggered by the input *R*, which means that every iteration with $R = true$ interrupts every behaviour inside of *ABO* and re-enters a reset version of the state. *ABO* then has a so called entry action which is triggered every time when the state is entered and in this cases sets the output *O* to *false*. Further it has its own controlflow-region which contains two other states. The initial state *WaitAB* is first entered when the state *ABO* is entered and it has a termination transition to the other state *done* which is taken when the behaviour inside of *WaitAB* is finished and sets the output *O* to *true*. The behaviour of *WaitAB* consists of two regions which are executed concurrently. The top region contains again two states. The initial state *wA* is left by a normal transition which is triggered by a tick with $A = true$ after the tick it is entered and then terminates in the state *dA*. The second region is equivalent to the other region with respect to the input *B*. As soon as both regions reach their final states the termination transition is triggered an the model stays in the state *done* until the next $R = true$ is reached.

SCCharts Dataflow

With the extension of SCCharts by dataflow-regions it became possible to model dataflow constructs as shown in Figure 2.3. Here an SCChart with the root state *Dataflow* is shown. The in- and outputs are defined in the same way as in the controlflow approach but the keyword `dataflow` opens up a dataflow-region. Instead of states and transitions this region consists of components that are connected by wires. The textual version of this model, as shown in Listing 2.2, simply defines two equations for the dataflow region in line 7 and 8. These equations are then for the graphical representation split up

```
scchart ABRO {
  input bool A, B, R
  output bool O

  initial state ABO {
    entry do O = false

    initial state WaitAB {
      region {
        initial state wA
          if A go to dA

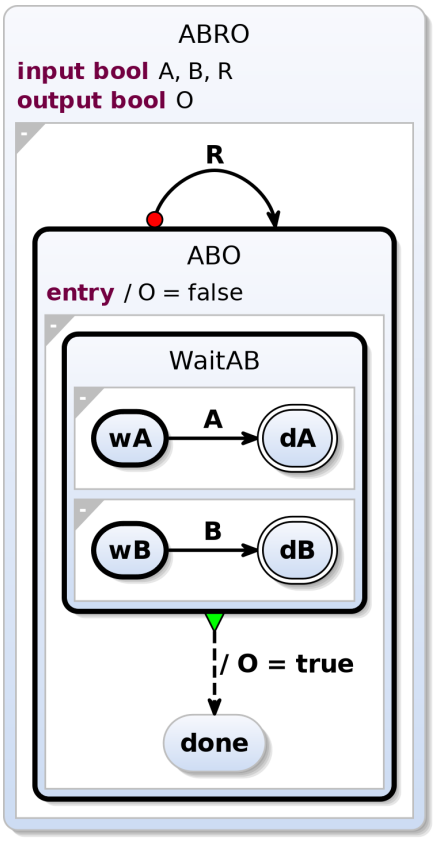
        final state dA
      }

      region {
        initial state wB
          if B go to dB

        final state dB
      }
    }
    do O = true join to done

    state done
  }
  if R abort to ABO
}
```

Listing (2.1) Textual representation of the ABRO program



(a) Graphical representation of the ABRO program

Figure 2.2. SCCharts implementation of the basic synchronous program ABRO

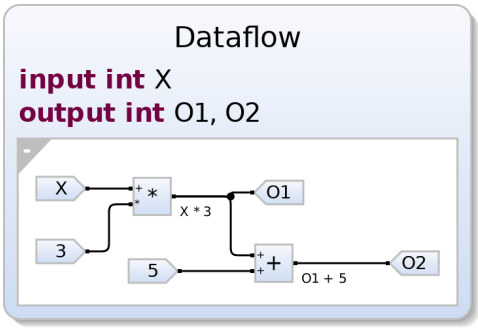
```
scchart Dataflow {

  input int X
  output int O1, O2

  dataflow {
    O1 = X * 3
    O2 = O1 + 5
  }

}
```

Listing (2.2) Textual representation of the dataflow model



(a) Graphical representation of the dataflow model

Figure 2.3. SCCharts implementation of a simple dataflow model

```
import "CF"

scchart DataflowReference {

  input int X
  output int O

  ref CF T

  dataflow {
    T.X = X
    O = T.O
  }

}
```

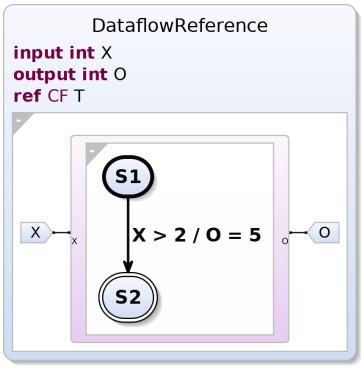
Listing (2.3) Textual representation of the referencing SCChart

```
scchart CF {
  input int X
  output int O

  initial state S1
  if X > 2 do O = 5 go to S2

  final state S2
}
```

Listing (2.4) Textual representation of the referenced SCChart



(a) Graphical representation of the referencing SCChart

Figure 2.4. A simple SCChart that uses a reference of another SCChart

into its basic components. The first equation $O1 = X \cdot 3$ consists of 4 components; one input flag for the defined input X and one for the literal 3, and on the outgoing side is an output flag for the defined output $O1$. The core is represented by an actor for the multiplication. At the left side are two wires that connect the inputs with the actor and on the right side is one wire that connects the output with it. The second equation is represented on the same way but this time the output of the first equation is an input of the second one. Therefore the output wire of the first actor is not only connected with the output flag for $O1$ but also with the input side of the second actor.

Referencing

Another useful feature of SCCharts that is used for this thesis is the possibility of referencing other SCCharts or states. As shown in Listing 2.3 the keyword `ref` creates the object T of type CF, where CF is a reference for the SCChart defined in Listing 2.4. After that the in- and outputs of object T can be used for the equations in the dataflow-region. The referenced SCChart is then shown as an expandable actor in the graphical representation as shown in Figure 2.4a. Further it is of no difference whether the referenced SCChart uses dataflow or controlflow elements or both and it is even possible to reference single states. In controlflow-regions a referenced SCChart is used as a state but requires a mapping of the outer in- and outputs to its own in- and outputs. The same method for referencing works in controlflow-regions where another SCChart can be referenced and used as a state.

2.1.2 KiCo

One feature of KIELER for the development of complex systems is its special compiler implementation, which is named KIELER Compiler (KiCo) [SSH18]. KIELER provides multiple different compile systems. Area 4 in Figure 2.1 shows a view on the selected *process system*. In KiCo there exists one process system for each transformation that is supported, e. g., a transformation from SCCharts to C Code as selected in the figure. The system itself consists of multiple model-to-model transformations, which are called

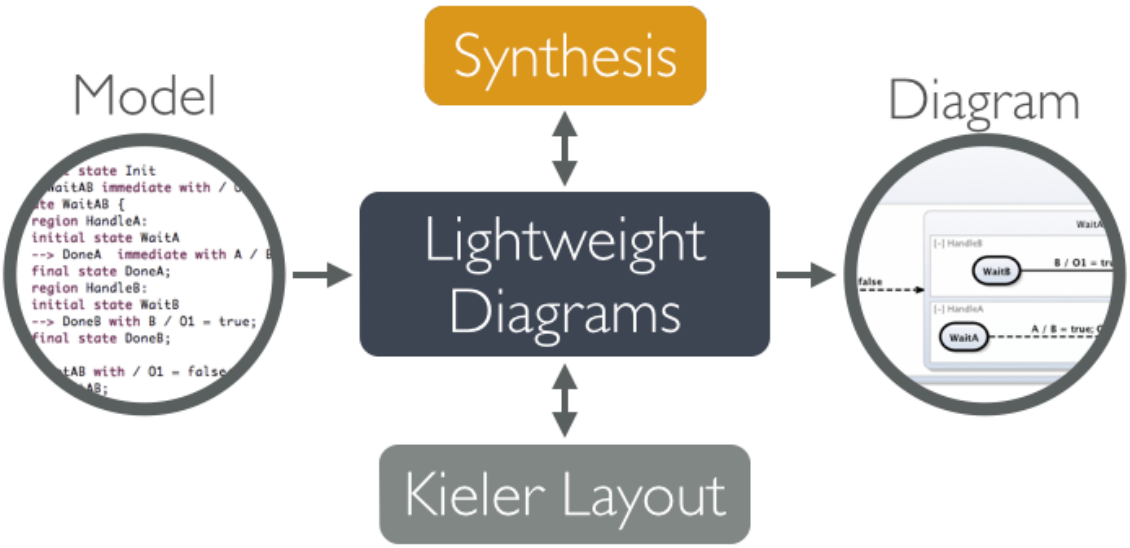


Figure 2.5. Transformation process of KLighD[2]

processors. Each processor takes the result of the previous one and hands its own result to the following one. On this way a processor chain is created that builds the process system. As shown in the Figure the processor chain is visualized with a name for each processor. In KIELER it is further possible to inspect the result of each processor to get further insight in each step of the transformation and to give further possibilities for bug fixing. For the transformation of SCCharts to C Code for example only a couple of core features of SCCharts have actual translation rules to C Code. The first processors of the process system for this transformation actually transform the so called extended features of SCCharts into the core features before the actual transformation to C Code is done. Each step of this transformation can then be reviewed by the selection of the corresponding processor in the processor chain.

2.1.3 KLighD

The KIELER subproject KIELER Lightweight Diagrams (KLighD) [SSv13] is one part of the diagram visualization of KIELER. KLighD is the tool that takes the textual defined SCChart and produces the corresponding diagram. This process takes place in different steps as shown by Figure 2.5.

First a synthesis is needed to transform the given textual model into the underlying KGraph, which is the basic model to represent graphs in KIELER. After that layout algorithms from ELK are used to attach the needed layout information onto the different KGraph elements before KLighD finally transforms the KGraph in the model needed by the diagram view of KIELER where it is rendered and displayed.

[2]Source: https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=10751615

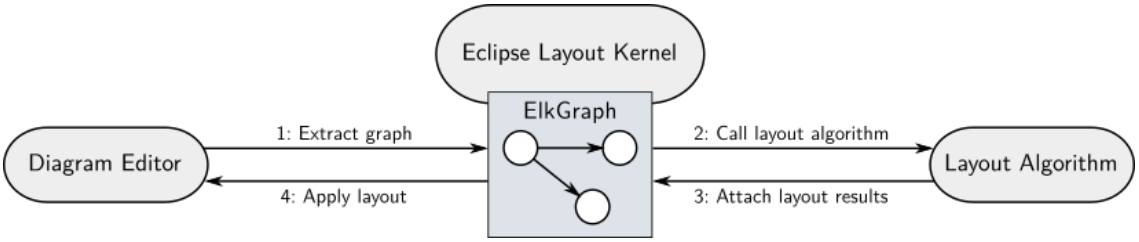


Figure 2.6. Overview about the ELK infrastructure[4]

2.1.4 ELK

The Eclipse Layout Kernel (ELK)[3] combines a collection of layout algorithms with an infrastructure to connect them with diagram views as shown in Figure 2.6. It gets a graph and uses a preselected layout algorithm on it to compute coordinates for each element of the graph and finally hand it back to the diagram view to render the graph. On this way every time the model is updated, ELK can be called to compute a new layout. Therefore it completely takes away the time consuming process for a user to find a place for a newly defined node or a way for a new edge. Further it is possible to define user options for a certain algorithm which ELK then provides through its infrastructure.

2.2 CDT

The plug-in C/C++ Development Tooling (CDT)[5] for Eclipse provides a fully functional Integrated Development Environment (IDE) for development in C/C++ on the Eclipse platform. It supports an editor combined with syntax highlighting paired with project management and creation tools. Furthermore, it also provides tools for a more detailed work on the source code.

The CDT is already integrated in KIELER to provide an editor for C/C++ code and for this thesis it is not only needed for the editor for the source files to work on but it will also be used to generate an Abstract Syntax Tree (AST) out of the given source code, which is needed to extract the needed information out of the given code.

AST

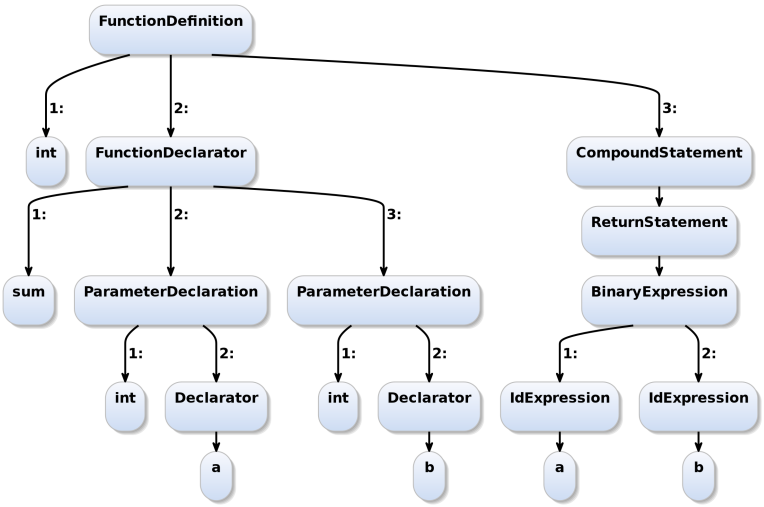
An AST denotes a tree structure that represents the abstract syntactic structure of source code. The CDT can create an AST representation for any C/C++ Code. Each node of the AST corresponds to one construct of the original source code and is connected with other nodes to represent the connection between them.

Figure 2.7 shows a simple function defined in C Code with the corresponding AST generated by the CDT. It can be seen that the root node of the tree corresponds to the function definition shown in the code. The node corresponds to the whole definition from line 1 to 5 and the further statements of the function are represented by the children of this root node. The first child is a node that says simply *int* and corresponds with the declaration of the return type of the function given in line 1. The function declarator node corresponds with the actual declaration of the function in line 1 and it has 3 children; one for the function name and further one for each parameter. The last child of the function definition

[3] https://www.eclipse.org/elk/
[4] Source: https://www.eclipse.org/elk/documentation.html
[5] https://www.eclipse.org/cdt/

```
1 int sum(int a, int b) {
2
3   return a + b;
4
5 }
```

Listing (2.5) A small function in C Code



(a) The corresponding AST

Figure 2.7. An AST and its corresponding C code

is a node named compound statement and corresponds with the whole body of the function. In this case it has only one child, namely the return statement of line 3. The `return` statement has a return expression as its child which is in this case a binary expression, namely an add operation with both parameters as its operands. What can be seen is that different to a real syntax tree, where every token of the source code has a representation, in the abstract syntax tree only the structural information of the code is included.

In this way an AST provides all the information about the give source code which is needed to extract any information out of it. The CDT not only provides a mechanism to create an AST out of C/C++ Code, but also contains a detailed class system to represent each node of the AST and to provide all kinds of functions to navigate through them.

Chapter 3

Related Work

This chapter mentions different work that is related in one or another way with the topic of this thesis.

Section 3.1 describes two scientific papers that work on the extraction of either dataflow or state machines out of Hardware Description Languages (HDLs) and a short description about how this extraction is achieved is given.

Section 3.2 introduces other tools for code comprehension. It describes what kind of support these tools try to give to the developers, how the results look, and what kind of interaction the results offer.

After that, Section 3.3 mentions some tools that also offer some kind of visualization for either dataflow or state machines. It explains what the tools are used for and the visualization itself is described.

Finally, Section 3.4 gives an overview about other theses of the research group that are related to this and gives insight about the topics and methods of them.

3.1 Extraction

This section addresses the extraction of dataflow or state machines out of source code and therefore describes two different approaches for the extraction of one of the models out of source code. The first one describes the extraction of dataflow out of a Very High Speed Integrated Circuit Hardware Description Language (VHDL) program and the second one will describe a more flexible approach to extract a state machine out of different HDLs.

3.1.1 Control and Dataflow extraction from VHDL

Namballa et al. [NRE04] developed a tool to automate the process of creating control- and dataflow graphs out of given VHDL programs—a task that so far was only done manually. The automation of this process, which is a needed step in the VLSI design flow, not only saves a lot of time during the act of development but also takes out a potential error source.

First they translate the given VHDL source code into an AST like structure with the help of "(...) the standard compiler construction tools, Lex and YACC (...)" [NRE04, p. 3] where the operators appear as nodes and the operands appear as children of the corresponding nodes. Afterwards their tool traverses the created tree and transforms it into the wanted control dataflow graph by following distinct translation rules for each VHDL statement.

The main difference to the dataflow extraction part of the tool introduced by this thesis is that the resulting graph of the tool of Namballa et al. is a semantically equivalent representation of the given VHDL source code, whereas the resulting dataflow diagram of the tool introduced by this thesis is an abstract view meant to help the developer with code comprehension and therefore does not create a semantically equivalent SCChart.

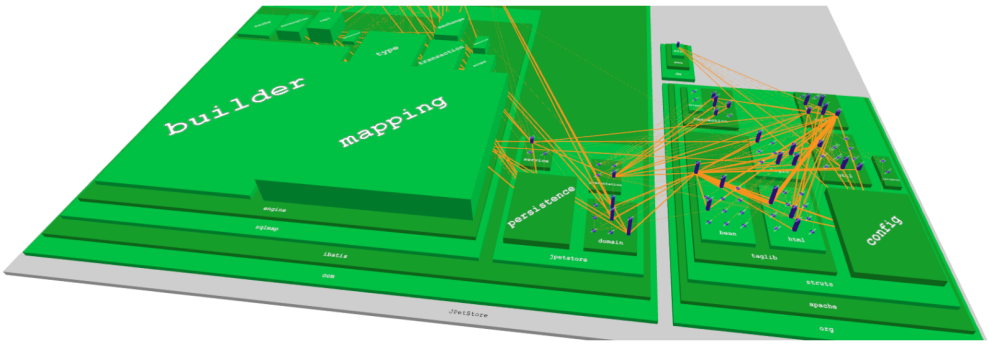


Figure 3.1. Example of an ExplorViz 3D city model[2]

3.1.2 State machine extraction form HDL

A method for the extraction of finite state machines out of programs defined with a Hardware Description Language (HDL) was developed by Jean-Charles Giomi [Gio95].

The first step of this method is to use the existing HDL parser to translate the given HDL program into a controlflow graph. This graph consists of one node for every statement of the previous textual description and further one unique entry node and one unique exit sink node.

After that the state machine is extracted out of this controlflow graph. The states are created from the wait statements of the program and for each path between two wait statements in the controlflow graph a transition between the two corresponding states is created. Furthermore, a data analysis is done to extract the transition triggers.

In this way a semantically equivalent state machine is extracted that maps every information of the given HDL program to a certain element of the resulting state machine.

3.2 Code Comprehension

There are various tools to help developers with code comprehension. This section will mention two of them. Both take a different approach to this topic than the tooling introduced by this thesis. They are both focused on the visualization of whole software systems and the inner connections of different parts of these systems. For example they give visualizations about all places of the system where a selected class is used.

3.2.1 ExplorViz

The research group Software Engineering at the Kiel University has developed the tool ExplorViz[1] [FKH17]. It takes a whole software system as input and creates a so-called 3D city model to display the communications between the different parts of the system. An example of this is shown in Figure 3.1. Here the different packages of the given system are displayed as green boxes. The model itself is interactive so the user can open the boxes to show its contents. Inside these boxes are either further packages, again displayed as boxes, or classes, which are displayed as blue cylinders.

[1] https://www.explorviz.net/

[2] Source: https://www.explorviz.net/img/screenshots/jpetstore_screen_v1.png

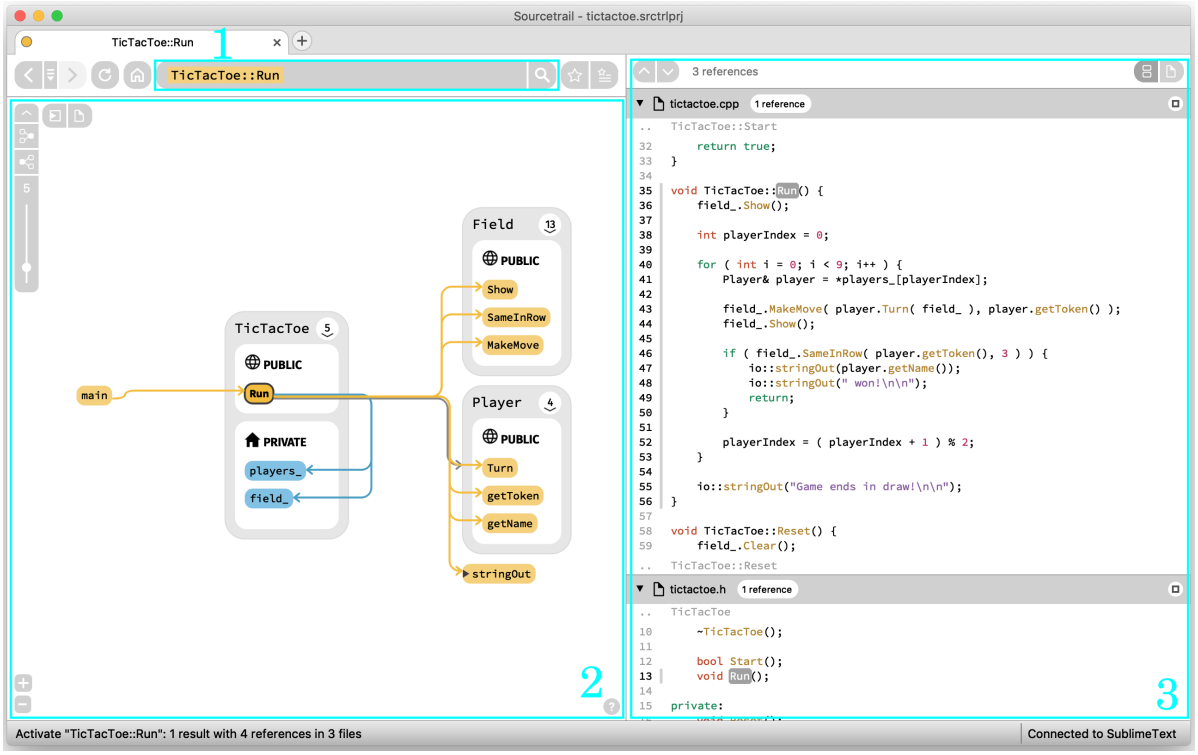


Figure 3.2. Example of a Sourcetrail project[4]

In addition to the display of the system structure, the tool also displays the communication between different parts of the system. For each time a class calls a function of another class, a yellow line is drawn between the corresponding blue cylinders of the two classes.

3.2.2 Sourcetrail

Another tool for code comprehension to be named here is called Sourcetrail[3], which is developed by the Coati Software KG. Sourcetrail is an interactive tool that again takes an existing software system as input and in the first steps indexes all its content to create an environment to easily navigate through the system contents.

After Sourcetrail has processed the system it provides a simple User Interface (UI) that is separated into three different connected parts, as shown in Figure 3.2.

1. A search field with an integrated auto-completion popup to search for every indexed object of the system like classes or functions etc.
2. A graph visualization that displays the selected object as an active node connected with every other object it is connected with. For example if the selected node is a function, like in the example shown in Figure 3.2, then the graph would contain a node for the function inside a hierarchical node for the containing class. Further, the function node would be connected with all indexed objects that either call this function or that this function is calling.

[3] https://www.sourcetrail.com/

[4] Source: https://www.sourcetrail.com/documentation/#UIIntro

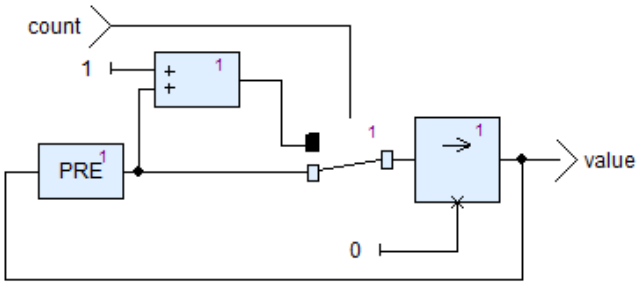


Figure 3.3. An example SCADE dataflow graph Soure: [Gri19]

3. A code view that displays all occurrences of the selected object in all indexed source files. Hereby the selected object is highlighted and a syntax highlighting is used.

3.3 Visualization

This section presents some other tools besides KIELER that can be used for the visualization of dataflow or state machines. Different to KIELER all these tools are mainly focused on a manually composed graphic.

3.3.1 SCADE Suite

The Safety Critical Application Development Environment (SCADE) Suite [CPP17] is a tool developed by Esterel Technologies. It uses textual and graphical engineering tools for the development of safety-critical software systems. It uses an integrated compiler which satisfies several safety standards for the development in avionic or automotive environments. It is based on the dataflow language Lustre and has its own graphical representation for it.

As shown in Figure 3.3 the graphical view of a SCADE project consists of different operators. Each operator represents one Lustre equation and the input ports of an operator correspond to the inputs of the equation and the output ports of the operators correspond to the outputs of the equation. The connections between the operators then connect the outputs of an equation with the input of the next.

3.3.2 LabView

Another tool for graphical work with dataflow is called LabView[5]. Similar to SCADE, LabView uses an actor-centred approach for the visualization of its dataflow models. Each actor represents either an operation or a variable and the lines between them connect the in- and outputs of the different operations with each other, as shown in Figure 3.4.

Contrary to SCADE, LabView is not based on a certain dataflow language like Lustre, but implements its own language named G and its main application is collecting and processing data in the field automatization processes.

[5] http://www.ni.com/getting-started/labview-basics/

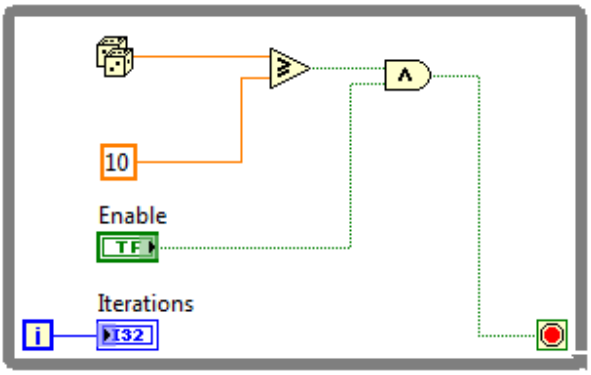


Figure 3.4. An example model in LabView[7]

3.3.3 Ptolemy II

The last tool to be named here is Ptolemy II[6]. Ptolemy II is a tool developed at the UC Berkeley and meant for the experimental design of all kinds of models. The underlying semantic meaning of a model is thereby not restricted by an underlying fixed concept but is configurable by the choice of different so-called directors.

Due to this broad capabilities of Ptolemy II it is possible to not only model dataflow systems but also state machines and it is further even possible to connect them with each other, as shown in Figure 3.5.

Figure 3.5a shows how different actors are connected to build a dataflow system in a similar way to SCADE or LabView. Each Actor represents one Operation or Variable and the in- and outputs are connected to the different operations. A special feature of Ptolemy is the director operator, in the example placed in the top left corner. This actor is with no wire connected to any other actor but it is used to determine the model of computation used for the described system.

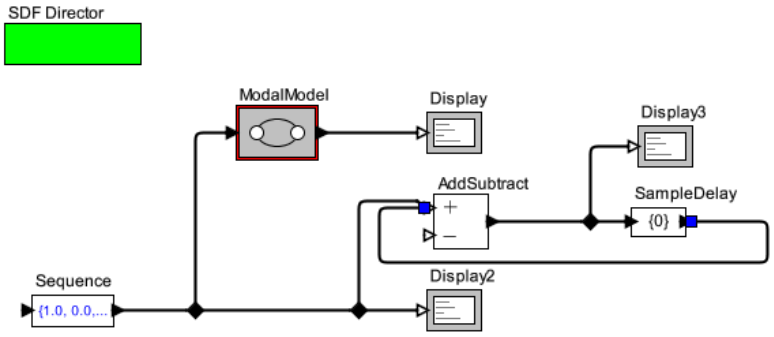
Figure 3.5b on the other hand shows a state machine modelled with Ptolemy II. It can be seen that the state machine is visualized as a connected directed graph. Each node of the graph represents a state with a label for its name. Each edge of the graph represents a transition—a possible state change—of the state machine, where the guard label gives information about the conditions that have to be fulfilled for the state change to occur and the output label gives information about the further effects this state change has. The arrows on the left or right side of the graphic state which in- or outputs this model has, which is not only needed to know which variables can be used in the guard or output expressions but is also needed when the modelled state machine is used as an actor in another model.

3.4 Related Theses

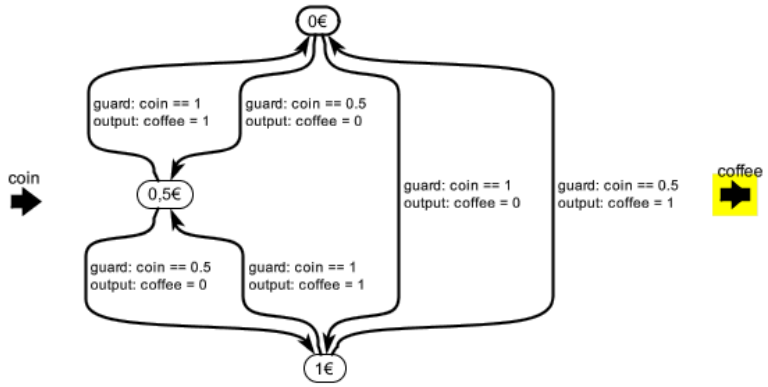
The last section of this chapter mentions other theses from the Real-Time and Embedded Systems Group from the Department of Computer Science of the Kiel University, which are related to the topics of this thesis.

[6] http://ptolemy.berkeley.edu/ptolemyII/

[7] Source: http://www.ni.com/getting-started/labview-basics/d/execution-structures



(a) Ptolemy dataflow model



(b) Ptolemy state machine model

Figure 3.5. Example models in Ptolemy II

3.4.1 Model-based Compilation of Legacy C Programs

In his bachelor thesis Stephan Lenga [Len16] introduces a new link to the tooling chain of KIELER to translate non-concurrent and non-dynamic legacy C code into a semantically equivalent SCChart. To achieve this he uses the eclipse CDT to build an AST out of the given legacy C code and then developed transformations for each possible node of the AST to reach a semantically equivalent SCChart representation.

Different to the tooling presented by this thesis Lenga did not use the dataflow capabilities of SCCharts and was focused on the semantically equivalent translation and not on the creation of a tool for code comprehension.

3.4.2 From Lustre to Graphical Dataflow Programs

A more dataflow focused work was done by Lena Grimm in her master thesis [Gri19]. She developed a method to replace the manually composed visualization of the dataflow language Lustre done in SCADE by a automatically composed visualization with the help of KIELER and SCCharts.

She developed a new plugin for KIELER to translate a Lustre program into an equivalent SCChart with as much concentration on the SCChart dataflow as possible. On this way KIELER automatically creates a visualization for the given Lustre program that is, by the use of SCChart dataflow, familiar to the design of a SCADE visualization but takes less effort of the developer through the benefits of textual development of the Lustre program.

Chapter 4

Concept

The following chapter gives a detailed look at the concept of this thesis' work. Section 4.1 lays out the process of dataflow extraction. It first describes what parts of SCCharts are used and gives a deeper insight on them. Thereafter, an overview about the different aspects of the C language that are considered by this thesis is given, before the translation rules for each of them are explained. Following that, it explains how the lack of sequentiality in SCCharts dataflow is overcome. Finally, an abstraction of this translation will be explained to visualize a special sort of algorithm that works on the manipulation of one central struct.

Section 4.2 gives an insight about the concept of state machine extraction. In the beginning, an overview about the needed elements of a state machine are described before laying out two different kinds of guidelines for implementing a state machine in C/C++ code, such that they can be extracted with the methods of this thesis.

As soon as these guidelines are defined it will be described how the actual extraction is done.

4.1 Dataflow Extraction

The basic concept for the extraction of a dataflow model out of a C code function is simple. The source code has to be analysed and for each element of the function a graphical representation has to be found. Therefore the first step is to build the AST of the function to analyse which elements it consists of. This AST then has to be traversed node by node to translate each node into a representation in the resulting SCChart. So before the different translation rules can be described, an overview about what elements there are in SCCharts, that can be used for this, has to be given.

4.1.1 SCChart Dataflow

For the creation of an SCCharts dataflow model it is important to lay out which elements SCCharts offer, to make it possible to decide which C code element can be represented by which SCCharts element.

As stated in Section 2.1.1 SCCharts uses so called regions to contain behaviour inside one state and for the creation of a dataflow model a dataflow region needs to be used, because the control-flow region can not contain dataflow. Thus the basic form of an SCChart dataflow model is a root state containing a dataflow region. Further a state can have multiple declarations. The same way as in C code a declaration is needed for the definition of a variable. Figure 4.1 shows a small SCCharts dataflow model. Area 1 illustrates a declaration—in this case a declaration for an input integer variable. Such a declaration declares one or more so-called valued objects, which themselves represent a kind of variable, as shown in Area 2. The difference between a variable and a valued object is that a valued object does not have to be linked to a variable but can also be linked to a reference declaration as shown in Area 3 and 4 of Figure 4.1. In this case the valued object does not represent some value as a variable would, but it represents another SCChart that can be used as an actor inside the dataflow region.

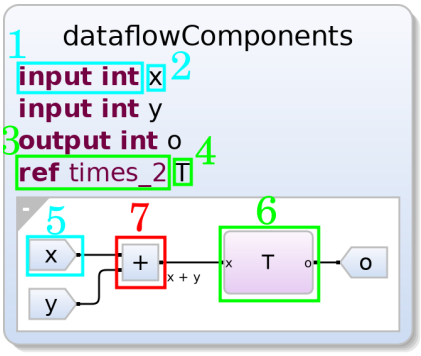


Figure 4.1. The components of an SCCharts dataflow model

The dataflow region itself consists of a collection of assignments. An assignment assigns an expression to a valued object. E. g., the assignment $o = x + y$ consists of the valued object o and the expression $x + y$ that is assigned to o. The representation of an expression in SCCharts is called a *KExpression*, but it will further be referred to as expression.

In the graphic this assignment is then represented by one element for each part of it. The valued object is placed on the right side of the graphic. The valued objects of the expression are placed on the left side and all of them are wired to the operator of the expression, shown by Area 7 of Figure 4.1.

As mentioned before, a valued object of a reference declaration can also be used as an actor, as shown by Area 6. To achieve this, one assignment for each in- and output of the referenced SCChart has to be placed in the dataflow region. Thus the dataflow region shown in Figure 4.1 contains two assignments: $T.x = x + y$ and $o = T.o$. Thereby $T.x$ corresponds with the input valued object x of the referenced SCChart and $T.o$ with the output respectively.

4.1.2 C Elements

For the translation of C code into a dataflow model the AST of the given source code is analysed and a translation rule is given for each different possible node.

In the first step the AST is searched for function definitions. The resulting model will consist of one root state with one controlflow region containing one state for each function defined in the given source code.

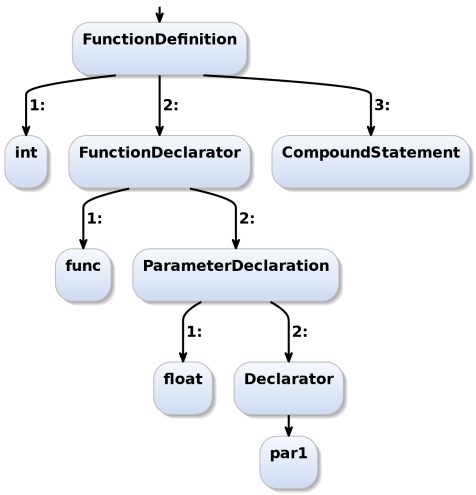
Function

A C code function definition such as in Listing 4.1 is represented in the CDT AST as shown in Figure 4.2a. The function definition consists of multiple different parts in the AST. In the first stage it is separated into three children. The first child represents the functions return type, in the shown example declared as *int*. The second child represents the function declaration, which on the one hand contains the function name and on the other hand contains all declared parameters, which are again separated into their types and a declaration of their names. The third and last child of the function definition is a so called `CompoundStatement` node, which represents the actual body of the function.

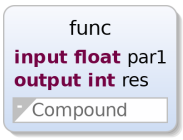
Figure 4.2b shows the representation of the function definition after the translation to SCCharts. As stated before, a function is represented by a state in the SCCharts model so that it can be referenced to model function calls, where the name of the function is also the name of the state. The parameters are represented by one variable declaration for each parameter with one valued object for each. All basic

```
1 int func(float par1) {
2   //Compound Statement
3 }
```

Listing (4.1) The source code of a function definition



(a) The AST of a function definition given by the CDT



(b) The SCCharts representation of the function definition

Figure 4.2. A function declaration shown in code with the corresponding AST and the final representation in SCCharts dataflow

types supported by SCCharts are also supported by this translation. Further the state has one declared output of the declared return type of the function. The name of this output valued object is set to be *res*, which leads to errors if this name is already used in the source code. This could be overcome, by internally renaming a use of that name, but is not considered by this thesis.

Finally the function state has one dataflow region to represent the function body which is contained in the AST node `CompoundStatement`.

Compound Statement

The translation of the `CompoundStatement` is the core of the whole process. As noted before it represents the body of a function, but also it is used to represent the body of each control statement, e. g., the `if` statement, which means that the translation of the `CompoundStatement` returns the actual dataflow model. Thus all the statements inside of it are translated into declarations or assignments which are added to the dataflow region of the containing state.

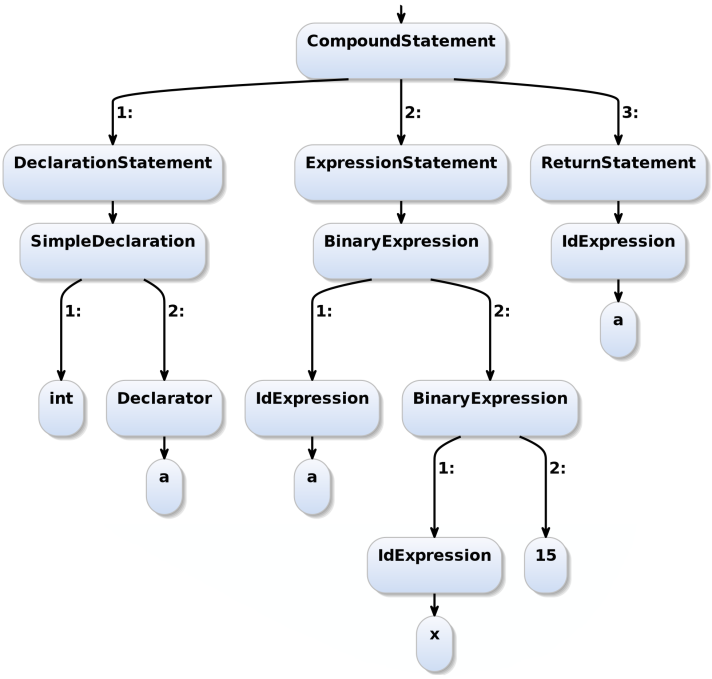
Listing 4.2 shows the source code for a simple function body. It consists of only one variable declaration, one assignment and then ends with a `return` statement. Each of these three statements is represented in the AST shown in Figure 4.3a by one respective node and for each of these one translation rule is given.

The translation of the variable definition is similar to the translation of parameters of a function definition. First an SCCharts variable declaration is created attached with a valued object representing the actual variable. This declaration is then added to the corresponding dataflow region.

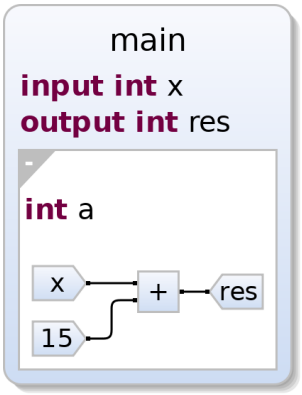
The translation of the assignment is divided into several steps. In the beginning, an SCCharts expression is created representing the right side of the equation and the valued object representing

```
1 int main(int x) {
2
3   int a;
4   a = x + 15;
5   return a;
6
7 }
```

Listing (4.2) The source code of a small `CompoundStatement`



(a) The AST of a small `CompoundStatement` given by the CDT



(b) The SCCharts representation of a small `CompoundStatement`

Figure 4.3. A small `CompoundStatement` shown in code with the corresponding AST and the final representation in SCCharts dataflow

the variable of the left side of the equation is retrieved. Thereafter an SCCharts assignment is created, assigning the expression to the valued object. This assignment is then added to the list of assignments of the dataflow region. This lead to the creation of the + operator in the dataflow region shown in Figure 4.3b with x and 15 as its inputs.

The `return` statement is then translated by creating an assignment to assign the given expression to the created output *res*. Since the declaration of the valued object for the variable *a* was assigned to the dataflow region, no token is drawn for it in the resulting graphic.

A slightly more complex function body is shown in Listing 4.3. It can be seen in line 4 that the expression, which gets assigned to the local variable *a*, is a function call. What can be seen in the AST shown in Figure 4.4b is that this function call has its own node as child of the binary expression node containing information about the name of the called function and an expression that is used as the parameter of the function. With this information a reference declaration can be created which references the state for the called function, if there is any, otherwise an empty state has to be created. A valued object for this declaration is created and the output of the state is linked to the corresponding valued object the source code assignment links to. Further, the parameter expression is transformed into a `KExpression` and with another assignment linked to the input of the referenced state. This creates the result shown in Figure 4.4a. If the called function was also defined in the given source code—and not for example a library function—the referenced state is expandable to show the behaviour of the called function. In this way, the resulting diagram can be navigated without showing every detail from the beginning so that the user can choose by himself, which details he wants to see and which not.

The last possible statements in the compound statement are the control statements such as loops. Each of these have no direct representation in dataflow. For the representation in this thesis an approach is choosen to represent each controlflow statement as a special actor built by a state and referenced into the dataflow region like it was done for a function call. In the following the translation rules for each controlflow statement are given.

While and Do-While Statements

The concept for translating a loop statement into an SCChart dataflow element is similar to the translation of a function definition. It will be explained in detail for the `while` statement.

In source code the keyword `while` starts the loop. It is followed by a conditional expression that states how long the loop is repeated and thereafter the loop body represents the behaviour of it.

The same elements can be found in the AST representation as shown in Figure 4.5b. The node `WhileStatement` states the begin of a while loop and has only two children: On the one hand there is an expression to represent the condition and on the other hand is the `CompoundStatment` to represent the loop's body.

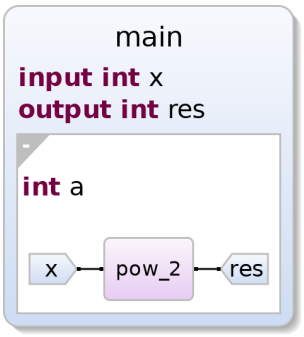
For the translation into SCCharts the statement is like a function definition represented as a state, as shown in Figure 4.5a. The name of this state will be *while_* extended by a number. During the translation the number of while statements met are counted to distinguish between the different created while states.

For the translation of the body another dataflow region is created wherein the `CompoundStatement` is translated, but different to the function definition this dataflow region has a label to represent the condition expression in plain text, as shown in Figure 4.5a.

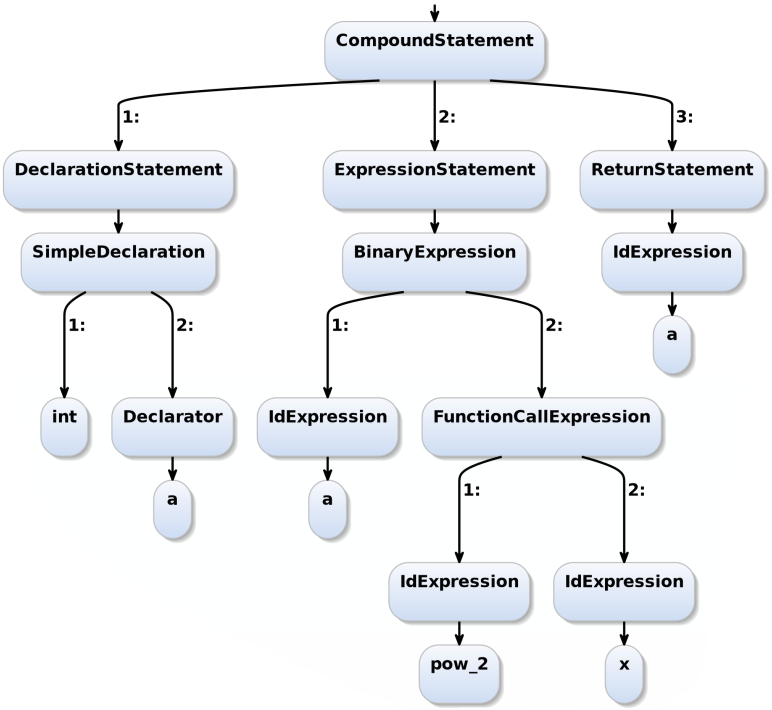
A possibility to visualize the condition expression with dataflow elements would have been to introduce a special port for the while state to receive a wire from the illustrated condition expression as shown in Figure 4.5c. The figure shows the visualized condition expression above the actual `while` state. This shows that this approach leads to a lot of additional elements in the resulting graphic,

```
1 int main(int x) {
2
3   int a;
4   a = pow_2(x);
5   return a;
6
7 }
```

Listing (4.3) The source code of a `CompoundStatement` including a function call



(a) The SCCharts representation of a `CompoundStatement` with a function call

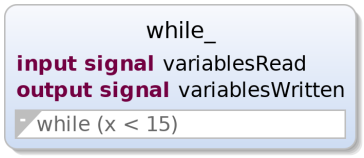


(b) The AST of a `CompoundStatement` including a function call given by the CDT

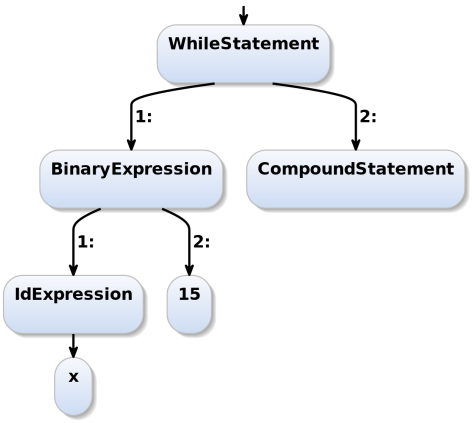
Figure 4.4. A `CompoundStatement` with a function call shown in code with the corresponding AST and the final representation in SCCharts dataflow

```
1 while (x < 15) {
2    //Compound
3 }
```

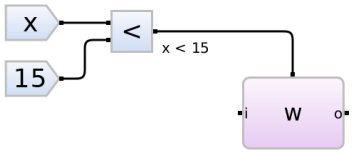
Listing (4.4) The source code of a `while` statement



(a) The SCCharts representation of a `while` statement



(b) The AST of a `WhileStatement` node given by the CDT



(c) Concept of a `while` state with condition port

Figure 4.5. A `while` statement shown in code with the corresponding AST, the final representation in SCCharts dataflow, and the possible design of a state with a condition port

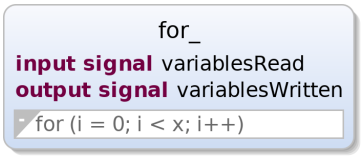
especially if the condition expression becomes more complex or if multiple loops are used. For the reason that the usual condition expression is short and does not influence the values of any variables it is chosen to only represent it as text.

The last step needed for the translation of the `while` statement is the definition of its in- and outputs. During the loop each variable used so far can be manipulated. Therefore the child nodes of the `CompoundStatement` node have to be checked for every possible assignment on a so far known variable. For each such assignment found, an output is added to the `while` state. On the same way each read variable is found and added as input to the state.

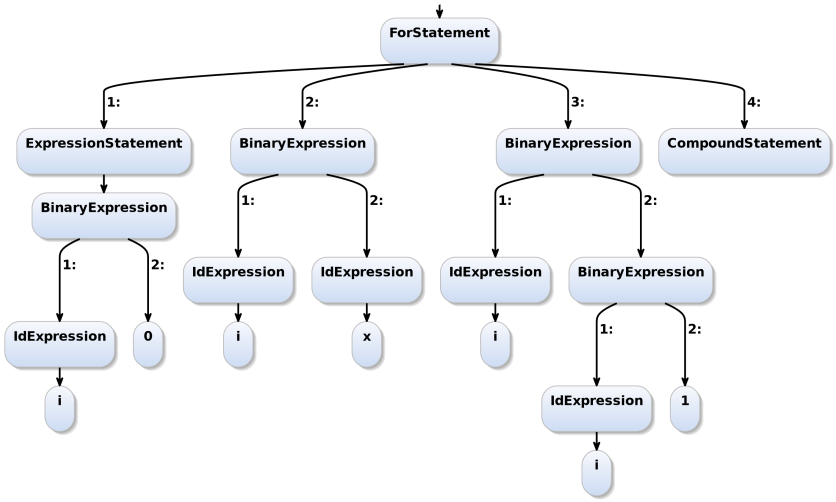
The created `while` state can now be referenced during the translation of the `CompoundStatement` containing the `while` statement and therefore be used as an actor in the resulting dataflow region connecting each read valued object with the inputs and each written with the outputs of the state. The translation of the `do-while` statement follows the same principle and will not be described in detail.

```
1 int i;
2 for (i = 0; i < x; i++) {
3   //Compound
4 }
```

Listing (4.5) The source code of a `for` statement



(a) The SCCharts representation of a `for` statement



(b) The AST of a `ForStatement` node given by the CDT

Figure 4.6. A `for` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow

For Loop

The translation of the `for` statement is very similar to the translation of the other loop statements.

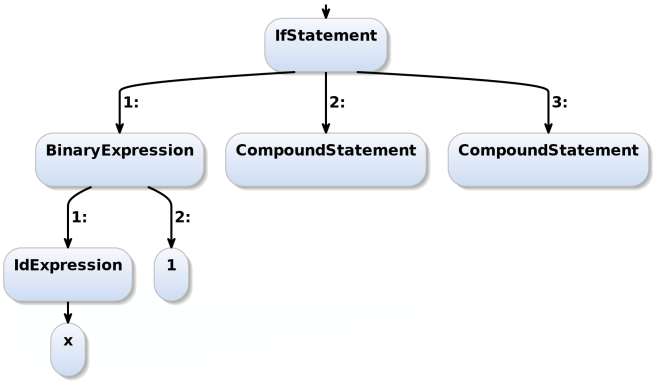
As shown by Figure 4.6a the statement is also translated into a state with the same type of naming concept. The state again contains a dataflow region to represent the loop body and the read and written variables are set as in- and outputs.

The difference of a `for` statement to the other two kinds of loops is that it does not only contain one condition expression. As shown by the AST pictured in Figure 4.6b the `for` statement has two more child nodes that also contain expressions. This is because a `for` statement does make use of three expressions. First an initialization to set the variable used for the loop on a starting value. Second the loop condition to determine how often a loop will be executed. Third an update expression that is executed after each loop iteration to update the loop variable.

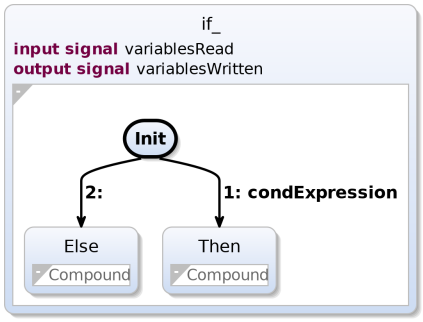
For the same reasons it was decided not to illustrate the condition expression of the other loop statements, it was decided not to illustrate the expressions of the `for` statement. Most of the time, the only variable affected by these expressions is the loop variable that does not interact much further with the rest of the program. Therefore the final graphic does not loose much expressiveness by representing these expressions as text labels of the dataflow region.

```
1 if (x == 1) {
2   //compound Then
3 } else {
4   //compound Else
5 }
```

Listing (4.6) The source code of an `if` statement



(a) The AST of an `if` statement given by the CDT



(b) The representation of an `if` statement after the translation in SCChart dataflow

Figure 4.7. An `if` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow

If Statement

The next control sequence is the `if` statement. The keyword `if` signals the begin of the statement, followed by the condition expression that is meant to be checked. If the condition holds, the *then branch* will be executed. If the condition fails, the *else branch* will be executed.

The AST representation of the `IfStatement` node, as shown in Figure 4.7a, has three children. One for the condition expression and one `CompoundStatement` for each branch.

For the translation into SCCharts the statement will again be represented as a state named following the same concept as before. The state will also hold the read variables as inputs and the written ones as outputs. The region however is a controlflow region, as shown in Figure 4.7b. It contains three states: An initial state with no extra behaviour and further one state for the then branch and one for the else branch. Each of these two states can be reached by an outgoing transition from the initial state and each of them has a dataflow region to contain the corresponding `CompoundStatement` translation.

Further, it can be seen that the condition expression is given as the trigger for the transition to the then state while the transition to the else state has no trigger. The priorities shown on each indicate that if the trigger fires, the then transition should be taken, since a lower priority is equivalent to a higher priority in SCCharts.

Switch Statement

The final control statement to be described is the `switch` statement. The behaviour of this statement is similar to multiple chained up `if` statements. It focuses on one variable and it is possible to determine different cases on which value the variable may hold. For each case a behaviour can be defined and also a default case can be given, as shown in Listing 4.7. What can be seen in the source code is that in C code each `case` indicates which code segment should be executed when a certain case is matched by the variable stated in the beginning. Further, a `break` indicates when the execution should be stopped and the switch statement should be left. In the shown example, the statement area for the case 1 is not ended by a `break` statement. This means that if the `switch` statement is entered with $x = 1$ then not only the statements after `case 1` are executed but also the statements after `case 2` until the `break` in line 9 is reached. Shown by the AST in Figure 4.8b is that no grouping can be found for the different statements. All statements inside the body of the `SwitchStatement` node are children of the corresponding `CompoundStatement`. Thus the only way to differentiate them is their ordering. It can be seen that the first child is a `case` statement with the literal 0 attached to it and the second child is the following `break` statement and so on. This sorting is enough to distinguish the cases and build up a translation into SCCharts.

The translation shown in Figure 4.8a is built up the same way as the translations described before. The statement itself is again represented as a state named following the same concept as before and again the read and written variables are attached as in- and outputs.

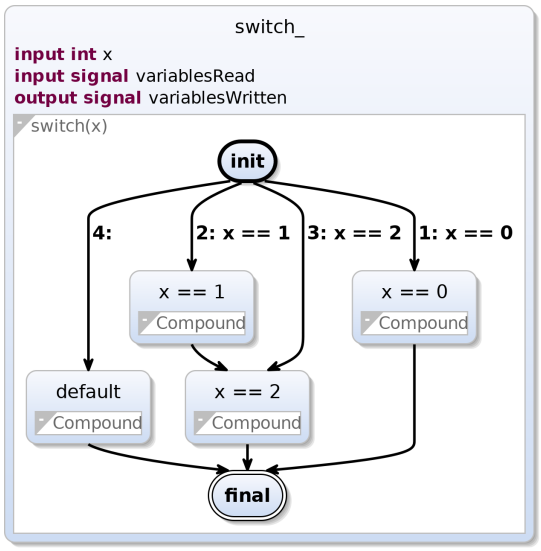
Just as with the `if` statement, the `switch` state contains a controlflow region to represent the behaviour. This region has an initial state *init*, a final state *final* and besides these one state for each `case` statement and one for a potential `default` statement. Except for the initial and final state each of these states again has its own dataflow region to illustrate the behaviour defined in the source code after the `case` statement. Besides that, transitions are used to represent the way through the `switch` statement. Therefore the initial state has one outgoing transition for every `case` statement with a trigger representing the defined case and another outgoing transition for the `default` statement without a trigger but with the lowest priority. These transitions make sure that the correct case state is entered. Furthermore, each case state has exactly one outgoing transition to show where the controlflow is leading after the calculations inside the case state are finished. If a case state has a closing `break` statement at its end, then the outgoing transition leads to the final state. If the `break` statement is missing, then the outgoing transition leads to the next `case` statements state following the ordering in the source code. Finally the default statement has a fixed outgoing transition to the final state, because it is the last control segment and therefore the switch statement is always left after the default part is finished.

4.1.3 Static Single Assignment

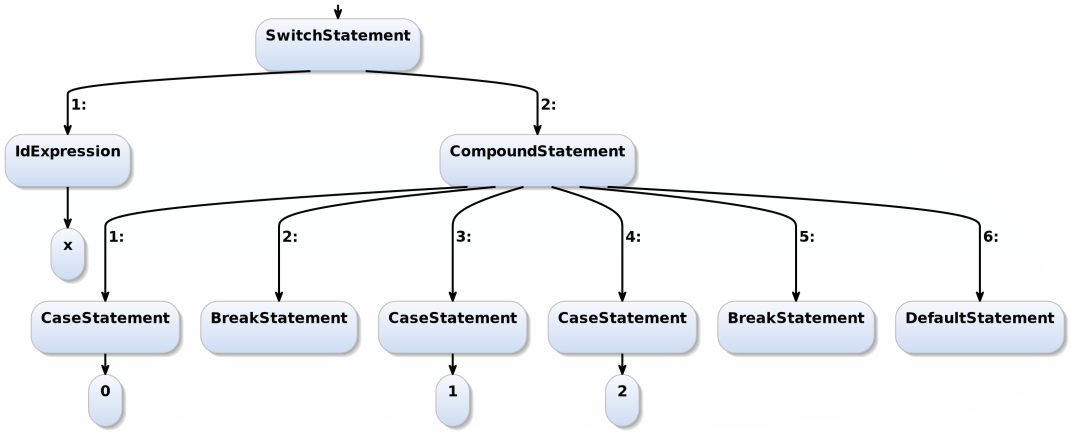
As mentioned in Section 2.1.1 SCCharts is a synchronous language that is different to most other synchronous languages and has some support for sequentiality but only under strict rules. A synchronous language is based on the synchronous hypothesis which says that each signal (or in this case variable/valued object) has only one certain value during one tick. SCCharts extends this model by the *initialize-update-read protocol*, which allows the value of a variable to be updated in a tick, even

```
1  switch (x) {
2  case 0:
3    //Statements
4    break;
5  case 1:
6    //Statements
7  case 2:
8    //Statements
9    break;
10 default:
11   //Statements
12 }
```

Listing (4.7) The source code of a `switch` statement



(a) The SCCharts representation of a `switch` statement

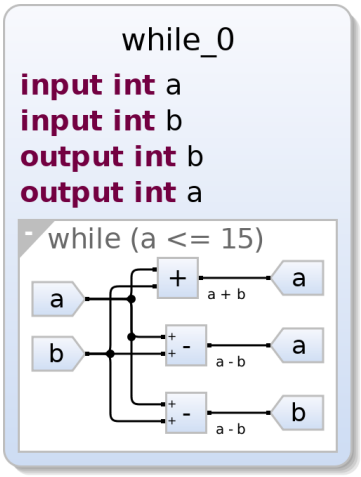


(b) The AST of a `SwitchStatement` node given by the CDT

Figure 4.8. A `switch` statement shown in code with the corresponding AST and the final representation in SCCharts dataflow

```
1 while (...) {
2   ...
3   a = a + b;
4   b = a - b;
5   a = a - b;
6   ...
7 }
```

Listing (4.8) The source code for a couple of sequential assignments



(a) The SCCharts representation following the so far established translation rules

Figure 4.9. The result of the translation of sequential assignments following the so far established translation rules

if it was already set to a certain value, if and only if this update occurs before the variable is read. This restriction makes sure that the update on the variable does not change the result of the read. A dataflow region in SCCharts, however, is calculated during one iteration and therefore it is not meant to have multiple writes and reads on the same valued object as it is possible in C code.

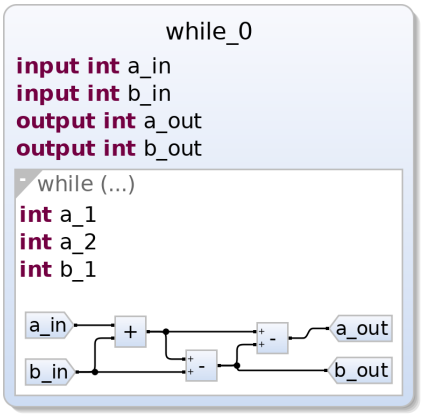
Listing 4.8 shows a `while` loop in C code that contains a small segment to swap the values of two integer variables a and b. It is a three step calculation, but the translation to SCCharts following the so far defined rules results in the visualization shown in Figure 4.9a. It can be seen that the sequentiality, which is needed for this, is not displayed. The first equation $a = a + b$ is visualized as expected. The two inputs a and b are connected with an add operator and the result is written to a again. The problem starts with the second equation $b = a - b$. Following the sequentiality of the source code here a should already be updated. However, since a is read in this equation the operator representing the equation is connected to the input valued object of a and not with the output of the add-operator. For the third equation, a and b should be updated thus the subtract-operator should be connected to the output of both previous operators but for the same reason as before it is again connected to the input valued objects. Another problem that is pictured in Figure 4.9a is that two operators are connected to the output valued object a, thus a is listed twice.

To overcome these complications the program is transformed into the so called Static-Single-Assignment (SSA)-form. For a program to be in SSA-form means that each variable has a valued assigned to it only once. To achieve this for a normal program, additional variables for each assignment have to be added and then the assignments in the source code have to be modified to use the new variables.

Listing 4.9 shows the modified source code of the example shown previously. The initial values for a and b are stored in the variables a_in and b_in and the new value for a after the first assignment is stored in a new variable a_1. From now on all reads on a are changed to reads on a_1. The result of the second assignment is then stored in a new variable b_1 which results in the third assignment consisting of only new variables where the equation $a_1 - b_1$ is assigned to a_2. The renaming of the

```
1 while (...) {
2   ...
3   a_1 = a_in + b_in;
4   b_1 = a_1 - b_in;
5   a_2 = a_1 - b_1;
6   ...
7 }
```

Listing (4.9) The source code for a couple of sequential assignments transformed into an SSA form



(a) The SCCharts representation after the translation with the use of SSA

Figure 4.10. The result of the translation of sequential assignments with the extension of SSA

inputs to *a_in* and *b_in* is done because, following the previous described translation rules, the in- and outputs of the states are created before the actual body is translated and in case that the same variable is read and written in one state, as in the shown example, the inputs get the suffix *_in* and the outputs get the suffix *_out* attached to their names.

Following these extensions the translation of the example is shown in Figure 4.10a. Now the wiring between the actors is done as expected. The input of the actor corresponding to the second equation is connected with the input *b_in* and the output of the actor corresponding to the first equation. The actor representing the third equation is then connected to the output of the other two operators.

The variables added for the SSA-form are declared in the dataflow region. They are handled as local variables and are not actually rendered into the graphic like the in- and outputs. The last step is to link each last used local valued object to its respective output valued object. The assignments $a_out = a_2$ and $b_out = b_1$ are added to the dataflow region to link the local values to the actual outputs of the state.

4.1.4 Not Supported Constructs

Some common functions of the C language are missing. First, pointers are not supported. The problem with pointers for this sort of visualization is that they do not fit in the assumed model of a dataflow circuit. A pointer represents the address of a certain value in the machines memory, but the dataflow graphic is meant to represent the use of the value itself. It could be tried to visualize the pointer by just visualizing the variable it is pointing on, but in C code it is also possible to perform arithmetic actions on pointers, which looses the link from one pointer to its initial variable. An approach to overcome such problems was not done by this thesis.

Second, arrays are missing translation rules in the presented extraction. The problem with this construct is the need for the SSA form. To correctly use an array the SSA principle explained in Section 4.1.3 would have to be established for every element of the array. Nevertheless it is possible to dynamically address a certain array element, for example by the use of an index variable *i*. It is not possible to determine based on the AST which of the array elements is used based on the AST.

Finally, the remaining features not supported are `struct`, `typedef`, and `union`. The representation of structs in SCCharts could also lead to difficulties with the SSA principle and the other complex types like unions or manually defined types can lead to similar problems.

4.1.5 Structflow Abstraction

Regardless to the fact that the basic extraction of dataflow does not support `structs`, an additional extraction was developed to represent algorithms which work on one central struct. This abstraction is based on the work of Janina Reuter [Reu19]. In her Bachelors Thesis, Reuter implemented different pitch tracking algorithms. These algorithms all follow the same scheme. All needed data is stored in one central struct, which is handed over to the functions calculating the algorithms themselves. This way a dataflow pattern is created with the difference that each of these functions has multiple struct elements as inputs and also multiple struct elements as outputs.

The first step for creating translation rules for this use case is to draw up guidelines the source code has to fulfil. As mentioned previously the algorithm is expected to be separated into different functions that all manipulate one central struct. To make it easier to recognize the struct it is expected that each of these functions takes it as the first parameter. The second important guideline states that within all of the functions the struct is referenced by the same name. The last important point to mention is that it is expected for the struct to not have another struct as element. Arrays and all basic types can be handled.

As soon as these guidelines are set, the entry point of the algorithm has to be found. Following the implementations of Reuter one central function is assumed that leads through the algorithm. To overcome the need of a fixed name for this core function it is decided to use a single marking comment for its identification. The comment will not be included by the resulting AST structure, but it is possible to retrieve a comment node, which eventually includes node locations to determine where in the given source code the comment is located. Since every node in the AST has such node location information it is possible to determine which function definition node follows the marking comment and therefore mark the extraction's entry point.

It is to mention that the abstraction has some differences to the basic dataflow extraction presented before. First, not every every used variable is considered but instead only the different struct elements are presented as valued objects. Every use of a variable that is not an element of the central struct will only be displayed by a simple string value representing the variable's name. This is done to reduce the displayed information on the used struct, to give the user a better overview about what struct elements are used in which function.

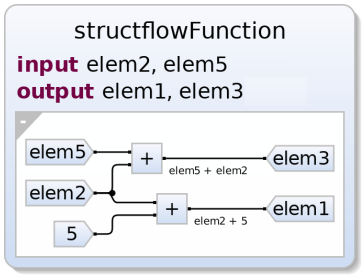
Second, the resulting function states differ as stated before. Since a function receives the central struct as its first argument, the function arguments are no sufficient source to determine the inputs for the function state and for the same reason the function will be able to change multiple elements of the struct and therefore the respective state has to have more than one output.

Listing 4.10 shows the source code definition of a function following the guidelines. The central struct is named `cStruct` and it is given to the function as its first parameter. The function body consists of two assignments on different struct elements, where the respective assigned expressions also consist of different struct elements or in one case of a literal value.

Figure 4.11a shows the resulting state after the extraction. In this state the central struct can not be found. Instead the used elements of the struct are listed. The elements used in the assigning expressions are listed as inputs and the elements an expression is assigned to are listed as outputs. To retrieve the information which elements are read and written, the same method is used as with the control statements in the basic extraction as explained in Section 4.1.2, with the single difference that

```
1 void structflowFunction(struct CentralStruct *cStruct)
2 {
3   cStruct->elem1 = cStruct->elem2 + 5;
4   cStruct->elem3 = cStruct->elem5 + cStruct->elem2;
5 }
```

Listing (4.10) The source code for an exemplary function of the structflow abstraction



(a) The SCCharts representation of the exemplary function

Figure 4.11. The representation of a function for the structflow abstraction

only read and writes to elements of the central struct are considered. As soon as the function states are created the extraction follows the same way as the basic one: translating control statements into states and assignments into SCCharts assignments, all reduced to only consider the central struct's elements.

4.2 State Machine Extraction

Following the extraction of the dataflow model, this section will describe the concept of extracting a state machine out of given C++ code. The basic difference for these two approaches is that the state machine extraction does not directly translate the behaviour statement for statement from the source code, but rather extracts an abstract model out of the it.

As already described in Section 1.2, a state machine consists of a set of states and another set of transitions to connect these states. The problem for the extraction of these elements out of plain source code is that there is no direct way to define them. Multiple approaches are possible to design a state machine in C++ code, but for the automatic extraction it is either necessary to know in advance which approach is taken or to find a complex way to dynamically determine which approach is taken. However, it is chosen to support two different extractions for two different ways to define a state machine in C++ code. Therefore this section first describes the two design patterns for state machines, before laying out how the extraction of these state machines is done.

4.2.1 State Machines in C++

The first step to define definition guidelines for state machines in C++ code is to show what elements are needed in the code. In mathematics a state machine is defined by the 5-tuple $(S, s_0, \Sigma, F, \delta)$, where:

- S represents the finite set of states ($s_0 - s_x$)
- s_0 represents the initial state (with $s_0 \in S$)
- Σ represents the finite set of inputs
- F represents the finite set of final states (with $F \subseteq S$)
- δ represents the transition function (with $\delta : S \times \Sigma \rightarrow S$)

This leads to the following list of needed elements: a set of states, with one of them being the initial state, a set of inputs, a potential empty set of final states, and a transition function to determine how the states are connected with each other. It is noted that both design guidelines for state machines that are presented in this section have no possibility to indicate a state as final.

The first described approach to define these elements with C++ code is a centralized one that defines the transition function with one central `switch` statement, which is taken from the ardumower project[1]. The ardumower project is an open source project for the development of a fully functional automatic lawnmower. It also contains a state machine which was taken as the example for one of the two definition guidelines. Due to the single `switch` statement to contain the transition function of the whole state machine, this approach will further be referred to as the centralized state machine.

The second approach for state machine definition handled by this thesis takes a decentralized and class-based approach by providing a single class for each state of the state machine. This approach will therefore be referred to as class-based state machine.

The Centralized State Machine

As shown in Listing 4.11 an `enum` is chosen to list all the states. This way one state is simply represented by an integer value that can be referenced by the state's name. This helps to give a better overview in the source code while the concept stays simple.

Following the same principle, the input events are also defined by enlisting them with an `enum`, as shown in Listing 4.12. Further, an array is used to contain the information about which input is present and which input is absent in one tick as shown in Listing 4.13 line 2. The array in the example has two entries, one for each defined event. This way the calculation of the events can be excluded from the central `switch` statement into a separate function as shown in Listing 4.13 lines 19 to 21.

The other local variable is used to store the current active state, as shown in Listing 4.13 line 1. Further, at least two more functions are needed. The first one, as shown in Listing 4.13 lines 15 to 17, is a function used to indicate a state change by setting a new value to the previous mentioned current state variable. Using a function for the state change helps to make it easier for the extraction to indicate where a state change is meant to occur by finding a function call.

The second needed function is some sort of main function, thus a function that is called to start the state machine, for example shown in Listing 4.13 lines 4 to 13. This function starts with an initialization procedure. Part of this is the initial write on the current state variable to indicate the initial state, as shown in line 7. This would also be the place to initialize all other parts of the system that the state machine would communicate with, e. g., sensors for the read of inputs. After this initialization a loop is needed. Each loop iteration corresponds to one tick of the state machine. Inside the loop the first step is to read the new inputs and reset the old outputs. Thereafter the transitions have to be checked, whether a state change has to occur.

To check the transitions the central `switch` statement is used in this approach as shown in Listing 4.14. The switch is done on the current state variable. Therefore it has one `case` statement for each state so that the controlflow of the function will switch to the case body belonging to the state active in the current tick. The case body then checks if the events needed for a state change to occur are matched and then calls the state change function.

These constructs are enough to represent a simple state machine with C code. Besides these core state machine elements the source code still needs functionality to communicate the state machine's results. For the shown example, which is the same coffee vending machine as described in Section 1.2,

[1] `https://www.ardumower.de`

```
enum {
 STATE_0,
 STATE_1,
 STATE_2,
};
```

Listing (4.11) The `enum` to define the states

```
enum {
   EVENT_EURO,
   EVENT_HALF_EURO,
};
```

Listing (4.12) The `enum` to define the trigger events

```
int currentState;
int events[2];

int mainFunc() {

   //Initialization
   currentState = STATE_0;

   while(1) {
      //Iteration
      calculateEvents();
   }
}

void stateChange(int newState) {
   currentState = newState;
}

void calculateEvents() {
   //calculate Events
}
```

Listing (4.13) The local variables and functions needed excluding the central `switch` statement

```
switch (currentState) {

   case STATE_0:
      if (events[EVENT_HALF_EURO]) {
         stateChange(STATE_1);
      } else if (events[EVENT_EURO]) {
         stateChange(STATE_2);
      }
   break;

   case STATE_1:
      if (events[EVENT_HALF_EURO]) {
         stateChange(STATE_2);
      } else if (events[EVENT_EURO]) {
         stateChange(STATE_0);
      }
   break;

   case STATE_2:
      if (events[EVENT_HALF_EURO]) {
         stateChange(STATE_0);
      } else if (events[EVENT_EURO]) {
         stateChange(STATE_1);
      }
   break;

}
```

Listing (4.14) The central behaviour `switch` statement

Figure 4.12. All elements of the definition concept for the centralized state machine approach

some functionality would be needed to give out the coffee when a state change back to the initial state is triggered. This sort of information, however, is not considered by the extraction.

The state machine design of the ardumower project does not use the described pattern to define events. In their design scheme the test for a potential state change is often spread out over multiple layers of nested `if` statements, before separate function calls test additional conditions. Grouping all these conditions for one transition and displaying it in the resulting model could lead to unreadable long transition triggers in such a complex structure that the benefit of displaying them in the first place could get lost. To overcome this problem the proposed way is to exclude this conditional calculation and concentrate on transitions without a trigger.

Thus for the benefit of flexibility of the extraction, the use of the event scheme is optional. If it is not used and the trigger calculation is done as by the ardumower project, the resulting model will not contain transition triggers.

The Class-Based State Machine

A second possible way to define a state machine is based on the use of C++ classes. This approach seems to be more complex, but especially for larger state machines results in a more organized source code.

The basis of this definition is one class as shown in Listing 4.15. This class represents the whole state machine and contains all other definitions, which are again `enums` for the states and events, shown in lines 4 and 6, one abstract `StateBase` class, shown in line 8, and one class for each state that implements the state base, exemplary shown in line 10. Additionally, the main class has a constructor, shown in line 12. It also contains a map to map the `enum` values for the states to the corresponding state class objects, shown in line 13, and an iterator that points to the object of the currently active state, as shown in line 14. Finally, the main class has a state change function similar to the centralized state machine approach to indicate a state change shown in lines 16 to 19.

Listing 4.16 shows the `Statebase` class, which first defines the types for the map and the iterator used in the main class as shown in lines 4 and 5. Besides a constructor the class also contains an `evaluate` function. This function, shown in line 7, is used to define the transitions, for the different states. Each state class implementing this state base has to implement this evaluate function, which defines that states transitions. Further, the state base saves its parent state machine, the respective `enum` value, and a name.

The `enums`, as shown in Listing 4.17, are defined in the same way as in the previous design pattern, but in the class-based state machine approach the state first listed inside the `enum` becomes the initial state. Further, the extraction requires a certain naming pattern between the classes of the states and their respective definition in the `enum` statement. If the class definition of a state has the name `STATE_0`, the extraction expects the state to be named `aSTATE_0` in the `enum`. Since the state classes and the state `enum` are defined in the same namespace, they cannot have the same name. Therefore the described pattern is used to provide the extraction a way to link the state classes with the corresponding `enum` values. Another possible approach would have been to identify the state classes by the `enum` value they receive during their object creations. However, because of the benefit of code overview the `enum` representations of the states should have similar names to the classes corresponding to them. The requirement of a naming pattern was therefore not assumed to be a big constraint, while simultaneously simplifying the extraction.

The emaple implementation of the mentioned evaluation function is shown in Listing 4.18 for `STATE_0` of the previously explained coffee vending machine. The function receives an event as input which is checked by a `switch` statement. Inside the `case` statements the state changes can then be

```
class SM {

   public:
      enum State {...};

      enum Event {...};

      class StateBase{...};

      class State: public StateBase {...};

      SM();
      StateBase::Map mStates;
      StateBase::MapIt mCurState;

      void changeState(State nextState) {
         StateBase::MapIt newState = mStates.
              find(nextState);
         mCurstate = newstate;
      }


}
```

Listing (4.15) The core state machine class

```
class Statebase {

   public:
      typedef std::map<State, StateBase> Map;
      typedef Map::iterator MapIt;
      StateBase(SM& parent, State state, const
           std::string& name);
      virtual void evaluate(Event event){};

      SM& parent;
      State state;
      std:string name;

}
```

Listing (4.16) The abstract class inherited by each state class

```
enum State {
   aSTATE_0 = 0,
   aSTATE_1,
   aSTATE_2
};

enum Event {
   EVENT_HALF_EURO = 0,
   EVENT_EURO
};
```

Listing (4.17) The `enums` to list the state machines states and input events

```
void SM::STATE_0::evaluate(Event event) {

   switch (event) {
      case EVENT_HALF_EURO:
         changeState(aSTATE_1);
         break;
      case EVENT:EURO:
         changeState(aSTATE_2);
         break;
   }

}
```

Listing (4.18) The evaluate function of one of the state classes

Figure 4.13. All elements of the definition concept for the class-based state machine approach

invoked by calling the state change function. With this design pattern only one of all events can be checked during one tick. Thus it is not possible with this approach to design a state machine where multiple events are present in the same iteration. The modelled state machine class can then be used in another piece of code, which coordinates the event calculation and iterates over the state machine. Further, this concept can easily be enhanced, for example by adding an *onExit* function to the state base and the state class definitions, which will always be called by the state change function whenever a state is left or by adding a *during* function to the states which is called in each iteration for the active state to model *during-actions*.

This whole concept results in a more structured state machine, which helps with keeping a better overview especially for more complex state machines. On the contrary the centralized approach is simpler and thus more suitable for smaller state machines such as the coffee vending machine example.

4.2.2 Extract the Centralized State Machine

The extraction of a state machine defined with the centralized scheme explained in Section 4.2.1 is more complex than the direct extraction of dataflow, since besides the elements needed for the design of the state machine the source code could contain additional constructs such as functions or variables to define functionality for the state machine. Thus it is not only needed for the state machine to be designed in a certain pattern for the extraction to work, but it is also necessary to mark the needed elements in some way such that the extractor has an orientation point. One way to achieve this, would have been to require certain names for the elements, e. g., requiring that the `enum` for defining the state is called *State*. The problem with this solution is that requiring a naming pattern restricts the developer in his choice of names for the elements of his code. This becomes even worse if the extractor is meant to be used for an already existing project such as the ardumower project, because it could result in the need of renaming the state machine elements which then again could result in the need of renaming other elements of the code.

To avoid these problems it was chosen to use comments for the marking of the needed elements. For the extraction of the centralized state machine two to three of these comments are needed:

1. *//_EXTRACTOR_STATE_ENUM*
2. *//_EXTRACTOR_EVENT_ENUM*
3. *//_EXTRACTOR_BEHAVIOUR_SWITCH*

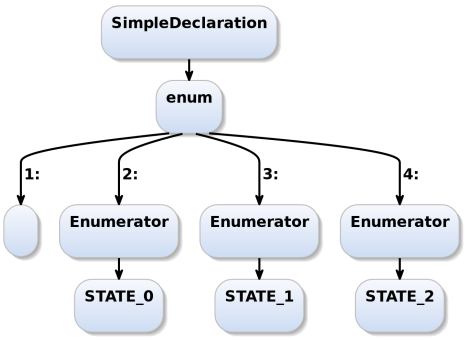
State Extraction

The first step of the extraction is to find the `enum` definition of the states. To achieve this it is required from the developer to mark this definition in his source code with a comment as shown in Listing 4.19. What can be seen in Figure 4.14a is that the AST built by the CDT to represent an `enum` has one `Enumerator` node for each listed element. Further, a node can be seen, which would contain the `enum`'s name, which is missing in the given example, since the `enum` is not named. Besides that it can be seen that comments are not connected to the tree structure, thus another way has to be found to match the comment with the corresponding AST node.

As explained in Section 4.1.5 the nodes are not only connected to represent the code structure but further contain information about each node's position inside the source code. It is possible to get a list of all comments from the AST even if they are not connected to the actual tree structure. Every comment node also has the information about the nodes location inside the source code. Thus

```
1 //_EXTRACTOR_STATE_ENUM
2 enum {
3  STATE_0,
4  STATE_1,
5  STATE_2,
6 };
```

Listing (4.19) The source code of the state `enum` with marking comment



(a) The AST of the state `enum`

Figure 4.14. The representation of states in the centralized state machine approach and its respective AST

following the principle that the marking node is inserted directly before the marked element, these node locations can be used to find the marked `enum` node in the AST tree structure.

The last step for extracting the states is then to create one state for each `Enumerator` node inside the `enum` and name them accordingly.

Event Extraction

Following the state extraction the input events are extracted the same way as the states, assumed there are any. For this purpose, another marking comment is used: *//_EXTRACTOR_EVENT_ENUM*.

Again, the first step is to look through all the comment nodes to find the corresponding node. If it cannot be found, it is assumed that the guideline to define events was not followed and the event extraction is skipped. If the comment is found, the corresponding `enum` can be found by the node locations. This time the found `Enumerator` nodes are not translated into states, but for each enumeration a boolean typed input is created and attached to the SCChart.

Transition Extraction

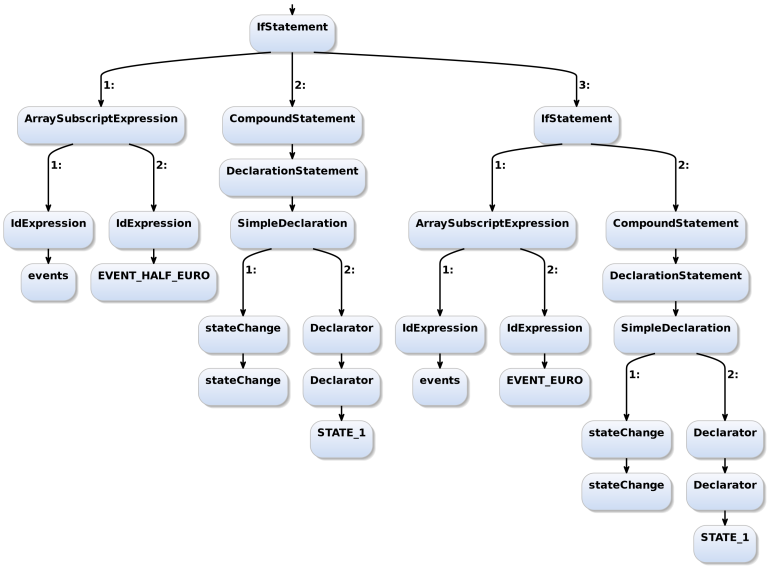
The third comment needed for the extraction marks the central switch statement. As shown in Listing 4.20, the comment is placed directly before the actual `switch` statement. As before, the comment is used to locate the node representing the central `switch` statement in the AST.

Figure 4.15b shows the AST representation of this `SwitchStatement` node and its children whereby the children of the `if` statements inside the `switch` `cases` are left out. What can be seen in the code and the AST is that in the control expression of the `switch` the variable to save the active state is listed. Thus no additional comment is needed to find the name of this variable.

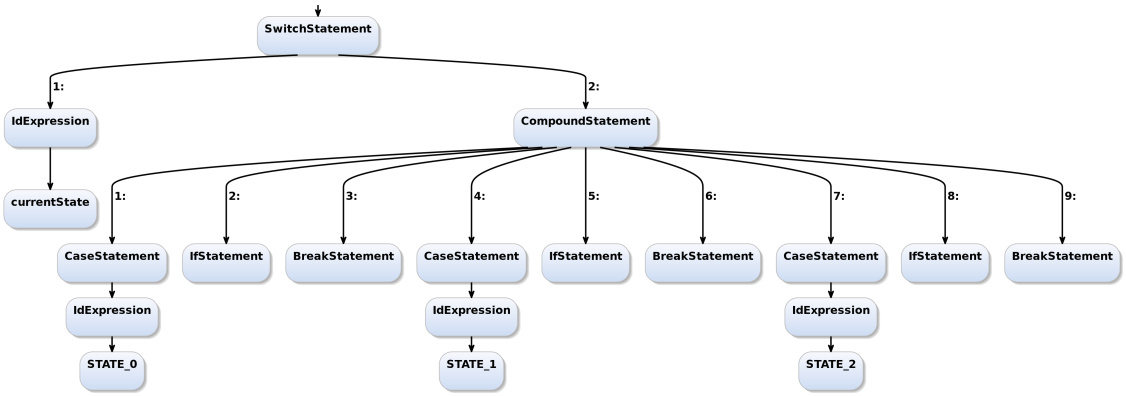
The next element needed for the extraction of the transitions is the state change function. The definition guideline shows that the current state variable is only written twice, first in the state change function and second when the initial state is set. A search for every binary expression representing an assignment on the active state variable on the whole AST is done. To determine which of the two assignments is in the state change function it is checked what expression is written onto the active state variable. For the initial state assignment a state representation from the state `enum` is written to

```
1  //_EXTRACTOR_BEHAVIOUR_SWITCH
2  switch (currentState) {
3
4     case STATE_0:
5        if (events[
              EVENT_HALF_EURO]) {
6           stateChange(STATE_1);
7        } else if (events[
              EVENT_EURO]) {
8           stateChange(STATE_2);
9        }
10    break;
11
12    case STATE_1:
13       if (events[
              EVENT_HALF_EURO]) {
14          stateChange(STATE_2);
15       } else if (events[
              EVENT_EURO]) {
16          stateChange(STATE_0);
17       }
18    break;
19
20    case STATE_2:
21       if (events[
              EVENT_HALF_EURO]) {
22          stateChange(STATE_0);
23       } else if (events[
              EVENT_EURO]) {
24          stateChange(STATE_1);
25       }
26    break;
27
28 }
```

Listing (4.20) The source code of the central `switch` statement



(a) The AST for one of the `if` statements in the central `switch` statement



(b) The AST for the central `switch` statement without the children of `if` statement nodes

Figure 4.15. The central `switch` statement with the corresponding AST representation

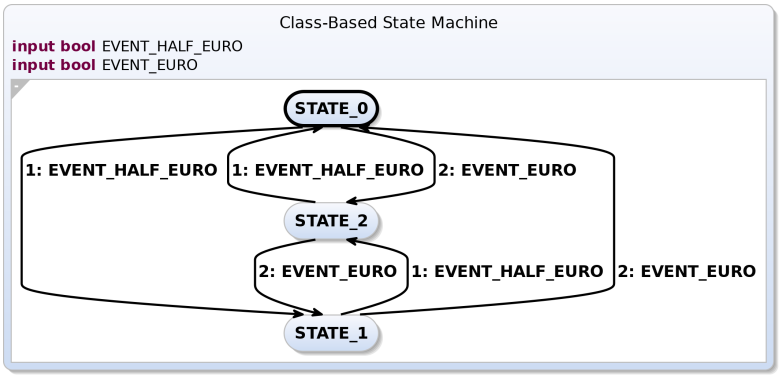


Figure 4.16. The extracted state machine graphic for the coffee vending machine example

the active state variable, for the function assignment a function parameter is written to the variable. With the assignment node found that containing `FunctionDefinition` node contains the name of the state change function. Besides this not only the state change function has been found but also the assignment for the initial state is located. Referenced by the `enum` value written to the active state variable the corresponding state can be found and marked as the initial state.

At this point all source code elements of the state machine are found and only the transitions are left for extraction. To achieve this, all calls of the change state function have to be found. The first step of this process is searching the source code for defined functions that contain a call of the state change function. This excludes the function containing the main loop with the central switch statement, which will be handled separately, and the state change function itself. Following the described scheme, the transitions should only be defined in the central switch statement, but for more flexibility possible function calls will be considered. For every function found, it is saved which states are reachable by it. Possible condition checks inside the functions are not considered. As stated before only conditions following the design scheme are included into the resulting model.

The next step is to analyse the central `switch` statement by iterating over its children. Every `case` statement represents a new potential source state, while every `break` statement removes all potential source states found so far. If the given state machine uses the proposed event handling scheme, the extractor expects a chain of `if` statements following a `case` statement. Figure 4.15a shows the AST for one link of such a chain. The `ArraySubscriptExpression` node, which is the first child of the `IfStatement` node, represents the condition expression from line 5 of Listing 4.20. This expression is converted into an SCCharts expression and will be used for the trigger of potential transitions in the then branch of the `if` statement. Said branch is then analysed for calls of the state change function or calls of one of the transition-containing functions mentioned earlier. These function calls give a list of target states. For each potential start state found so far a transition is created to each found target state with the created condition expression as its trigger. After that the potential else branch is processed in the same way. If the event scheme is not used, then the statements following a `case` statement are individually processed like the then branch to find the target states without creating a trigger.

The decision was made to only draw one transition from one state to another. If multiple transitions are declared in the source code the transitions are merged into one. If the event scheme is used, the additional trigger is combined with the other one by connecting them with an *OR* operator. This extraction results for the described coffee vending machine in the state machine shown in Figure 4.16.

4.2.3 Extract the Class-Based State Machine

The extraction of the class-based state machine follows the same basic method as the extraction of the centralized state machine. Four comments are used to mark certain definition elements such that the extractor can find them and build the resulting SCChart out of it:

1. *//_EXTRACTOR_STATE_ENUM*
2. *//_EXTRACTOR_EVENT_ENUM*
3. *//_EXTRACTOR_STATE_CHANGE_FUNCTION*
4. *//_EXTRACTOR_STATE_CLASS_EVALUATE_FUNC*

One more comment is needed to identify the state change function. The fact that no central `switch` statement is used removes the possibility to identify the active state variable like it was done with the centralized state machine. Therefore it is not possible to find the writes on this variable which also removes the possibility to automatically identify the state change function.

An additional comment for the active state variable is not needed, because, as described in Section 4.2.1, the initial state is determined by the first state listed in the state `enum`. Thus, for the extraction it is not needed to know the active state variable.

The beginning of the extraction is the same as with the centralized state machine. The `enums` for the states and events are identified with the respective comments and then translated into the states and inputs of the resulting SCChart with the only difference that the prefix *a* that is preconditioned by the naming principles is removed from the state names.

Transition Extraction

The difference between both approaches is the extraction of the transitions. The first step for the extraction of transitions out of the class-based state machine design is to identify the name of the state change function. Since this is done by the use of the respective comment, the next step is to find the name of the evaluation function that is used to define the transitions of a state. The comment to mark this evaluation function is used within the definition of the abstract Statebase class as shown in Listing 4.21 line 7.

As soon as the evaluate function's name is identified, the next step is to find the definitions of these function for each different state. Since the state names and therefore the class names of the state classes and the name of the evaluation function are known, the function definitions can be found by a search through all function definitions in the AST.

Following this, these evaluation functions are worked on state by state. Since each evaluation function is registered for one state, the source state of the resulting transitions is fixed. For the complete extraction the determination of the target states and trigger is still missing.

As shown in the design pattern the body of the evaluation function consists of one `switch` statement deciding on the given event parameter. Therefore, the `case` statements give the trigger for the transitions. Every time a `case` statement is found while checking the `switch` statement, the corresponding event is saved in a list. On the occurrence of a `break` statement this list is cleared, and when a call of the state change function is found, a transition is built to the state given as parameter for the state change function with all trigger events saved in the list combined with *OR* operators as the transition's trigger.

At this point the extraction of the class-based state machine is completed and the extraction of the coffee vending machine example in the class-based scheme leads to the same diagram as with the centralized scheme, which is shown in Figure 4.16.

```
class Statebase {

   public:
      typedef std::map<State, StateBase> Map;
      typedef Map::iterator MapIt;
      StateBase(SM& parent, State state, const std::string& name);
      //_EXTRACTOR_STATE_CLASS_EVALUATE_FUNC
      virtual void evaluate(Event event){};

      SM& parent;
      State state;
      std:string name;

}
```

Listing 4.21. The evaluation function is marked by a comment in the definition of the abstract `Statebase` class

4.3 Highlighting

This section explains a highlighting feature, which is provided additional the extraction of the described models. Besides the automatic creation of diagrams for the extracted SCCharts models, the KIELER tool is also able to create interactions between these diagrams and the original source code. To achieve this, a so called *ModelUpdateController* is used, which provides a *selectionChanged* function that is called every time a new object of the graphic shown in the diagram view is selected. The selected object can then be identified to provoke a reaction. This feature is used to establish a link between the extracted models to the given source code. It is possible to select for example a state of a created dataflow model which will result in the highlighting of the source code fragment responisble for the creation of this state as shown in Figure 4.17.

4.3.1 Highlight Annotations

As explained before, the integrated event handling makes it possible to react to a change in the selection and then determine the selected object. Thus, it is necessary to have information attached to the selected SCCharts, about the source code fragment corresponding to it. With that information the CDT provides functionality to select the given code fragment and scroll the editor onto the provided position.

Each AST node contains both the offset and the length of the source code fragment that it represents. Therefore the information that should be attached is present during the extraction. For the attachment, SCCharts annotations are used. Similar to a Map-entry, one of the possible types of SCCharts annotations consists of a *key* and a *value*. Thus, every time an SCCharts element is created during the extraction, two annotations are attached to it. The first has the key *Offset* and contains the offset of the source code fragment as value. The second has the key *Length* and contains the length of the fragment. For example the state of the `main` function, highlighted in the shown example in Figure 4.17, has the annotations *(Offset, 155)* and *(Length, 81)* attached to it. This means that the source code fragment representing the `main` function begins hat the position 155 of the given source code and contains 81 characters. These annotations can then be read in the *selectionChanged* function and the values are used to select the corresponding source code fragment by calling the corresponding CDT function.

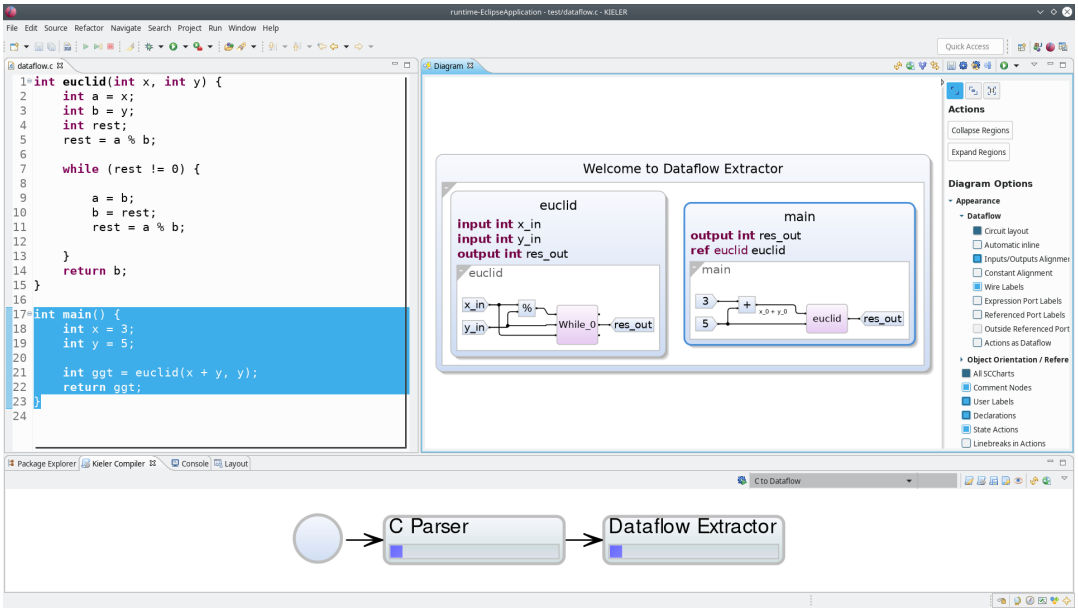


Figure 4.17. A screenshot of KIELER showing an example of the highlighting for a small dataflow model

4.3.2 Highlighting Dataflow

Every created state in the extracted dataflow model is linked to the corresponding statement that creates the state. For example, the state created for a function definition is linked to the location information attached to the corresponding `FunctionDefinition` node. Selecting that function state will therefore select the whole function, including its body, in the source code as shown in Figure 4.17.

The other created elements are the assignments. For highlighting an assignment it is chosen to attach the corresponding location information to the valued object the assignment is targeting. This way the selection of the wire in the diagram will result in the highlighting assignment this wire originates from.

4.3.3 Highlighting State Machines

The decision what source code fragments should be selected if a certain element of the state machine diagram is selected is not as simple as with the dataflow approach. In the state machine model a single object does not correspond to one single part of the source code. The decision was made to only attach annotations to the states of the resulting graphic.

In the centralized state machine approach the states are linked to the `case` in the central behaviour `switch` statement that correspond to this state as shown in Figure 4.18. In the class-based state machine approach a state is linked to the evaluation function of the class corresponding to this state as shown in Figure 4.19.

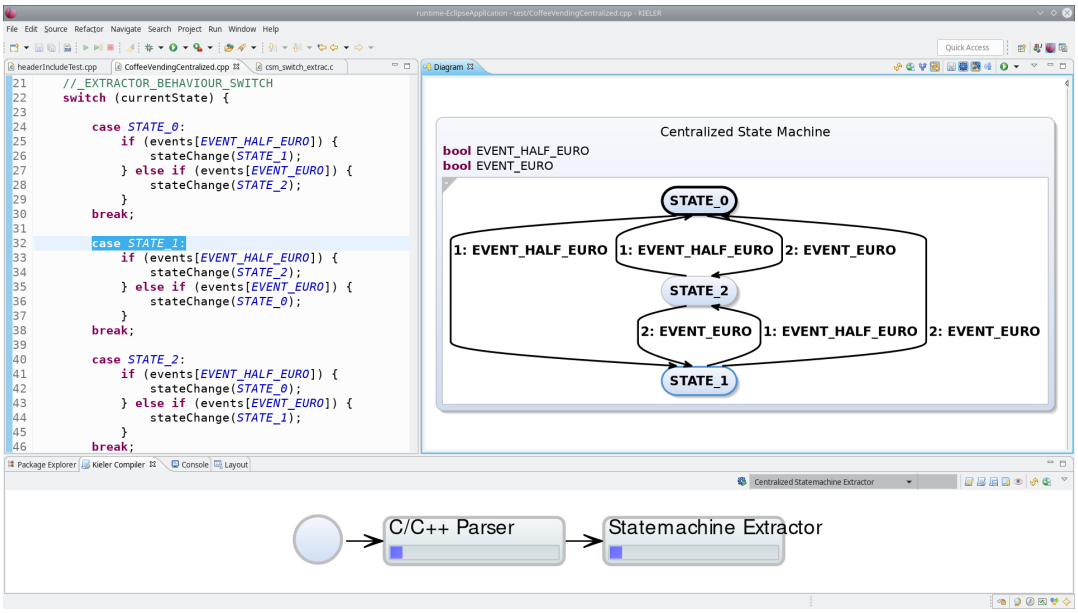


Figure 4.18. A screenshot of KIELER showing an example of the highlighting for the centralized state machine extraction

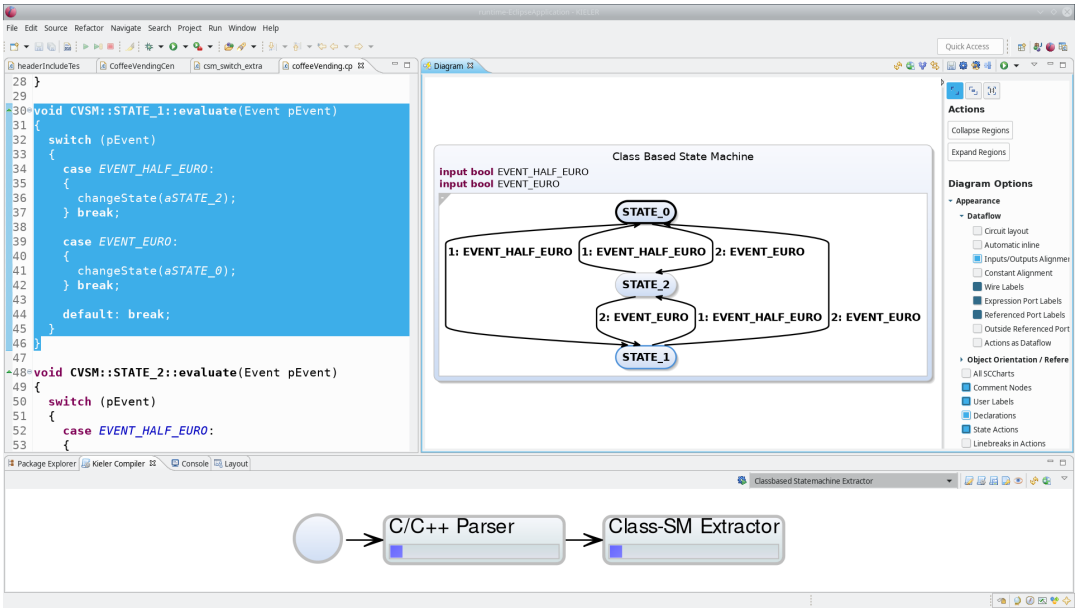


Figure 4.19. A screenshot of KIELER showing an example of the highlighting for the class-based state machine extraction

Chapter 5

Implementation

Following the description of the concept in Chapter 4, this chapter gives insight in the implementation of the described concepts. This implementation is done with the programming language Xtend, which is therefore the language the code examples are presented in.

First, a short description of the parser is given, to describe how the CDT is invoked to create the AST. Second, the implementation of the dataflow extraction is described. The translation of the statements such as functions, compound, or while is explained, before the translation of expressions is described and an overview about the handling of the valued objects with achieving the SSA form as described in Section 4.1.3 is given. Third, the implementation of the state machine extraction is explained. It is shown how the marking comments are used to find the corresponding AST elements. Thereafter, it is explained, how the missing elements of the centralized state machine are found, followed by a description about how the central `switch` statement is processed. Finally, it is explained how the class based state machine is extracted after the comment-marked elements are found, by describing how the state's evaluate functions are found and processed.

5.1 C Parser

The parsing of the C/C++ source code into an AST is done by the first processor of each KiCo created for each extraction. It takes the source code, invokes the CDT tooling to create the AST and then hands it over to the next processor defined in the system. Listing 5.1 shows the implementation of the `process` function of the parser processor.

The first step for the parsing is to get an object of the source file that is meant to be parsed, which is done in lines 3 and 4. The active environment is used to receive the absolute path of the file. This path is then used to create the `FileContent` object needed to invoke the CDT.

Thereafter, in lines 8 to 13, other objects are created that provide extra information for the parsing such as include paths which are stored in the `ScannerInfo` object in line 10. Besides that the `opts` integer can be used to give information about excluding different parts of the source code from the resulting AST. The value 0 is used to exclude nothing.

The `IncludeFileContentProvider` gives `FileContent` objects to the parser for files that are imported in the file that is meant to be parsed. The function `getSavedFilesProvider` gives such a provider for all locally stored included files. This option is needed to also include the definitions of the header file of the given source file into the AST. For example the in Section 4.2.2 mentioned ardumower project defines the `enum` representing the states inside the header file. Without the header file being part of the AST it would not be possible to extract them.

Finally, in lines 16 and 17, the actual parsing is invoked and the resulting AST is represented by the `IASTTranslationUnit`. This object is then set as the current model, which is needed to hand it over to the next processor for extracting the wanted model out of it.

```
override process() {
   // Create a FileContent out of the source file active in the editor
   val infrastructure = ProjectInfrastructure.getProjectInfrastructure(environment)
   val fileName = infrastructure.modelFile.absolutePath
   val FileContent fileContent = FileContent.createForExternalFileLocation(fileName)

   // Create assisting objects for the parsing
   val definedSymbols = new HashMap
   val String[] includePaths = #[]
   val ScannerInfo info = new ScannerInfo(definedSymbols, includePaths)
   val IParserLogService log = new DefaultLogService()
   val int opts = 0
   val IncludeFileContentProvider includes = IncludeFileContentProvider.getSavedFilesProvider()

   // Invoke the parsing of the CDT to receive an AST
   var IASTTranslationUnit translationUnit =
      GPPLanguage.getDefault().getASTTranslationUnit(fileContent, info, includes, null, opts, log)

   setModel(translationUnit)
}
```

Listing 5.1. The `process` function of the parser processor

```
1 var ifCounter = 0;
2 var swCounter = 0;
3 var forCounter = 0;
4 var whileCounter = 0;
5 var doCounter = 0;
6
7 var functions = <String, State> newHashMap
8 var VOs = <State, HashMap<String, ArrayList<ValuedObject>>> newHashMap
```

Listing 5.2. The global variables used for the dataflow extraction

5.2 Dataflow Extraction

The extraction of dataflow from one source code file is separated into the translation of each of its functions into one SCCharts state. To assist these translations some global variables are used, as shown in Listing 5.2. The counter variables of lines 1 to 5 are used for the naming of the control statement states as mentioned in Section 4.1.1. The `HashMap` named `functions` of line 7 is used to find the state of a function by its name, which is needed to reference a function's state if it is called inside another function's body. The `HashMap` named `VOs` of line 8 is used to organize the valued objects representing variables. The valued objects are separated by the states they are created for. Thereafter the name of the variable identifies the valued object list, which includes all valued objects needed to represent this variable in the SSA form. On this way no problems occur if different variables in different scopes use the same name, because these different scopes—as for example two sequential `while` loops—are represented by two different states, thus the respective valued objects are also separated inside the `VOs` `HashMap`.

```
1  def State buildFunction(IASTFunctionDefinition func) {
2     // Determine function name
3     val funcDeclarator = func.getDeclarator as CASTFunctionDeclarator
4     val funcName = funcDeclarator.getName.toString
5
6     // Create the state
7     val State state = createState(funcName)
8     state.label = funcName
9     ...
10 }
```

Listing 5.3. The creation of a function state with its name

With these variables set the `process` function of the dataflow extraction processor starts with the creation of an SCChart, followed by the creation of the root state and a controlflow region for it. As soon as these basic elements are created the given AST is processed by analysing the top level child nodes and invoking the function translation for each function definition node found.

5.2.1 Function Extraction

A function definition is represented in the CDT AST by the `IASTFunctionDefinition` interface. The translation of the function definition into SCCharts begins as shown in Listing 5.3. The translation function takes only the AST node representing the function definition as parameter and begins with retrieving the function declarator node to extract the function name in lines 3 and 4 and then creating the actual state for the function and naming it in lines 7 and 8.

After the state is created, the `HashMap` for its valued objects is initialized and attached to the global `HashMap VOs`. With that ready the next step is the extraction of the function inputs, as shown in Listing 5.4. First the parameters, which are child nodes to the function declarator, are retrieved and then processed one for one. The first step in line 5 is to create a variable declaration to represent the parameter. The type of the declaration is set to the SCChart type corresponding to the type of the parameter and the input flag of the declaration is set to true, before it is attached to the function's state.

With the declaration set up, the valued object to represent the parameter variable is still missing. Thus, in line 13 the name of the input variable is retrieved and a variable list to follow the concept for the SSA form, as stated in Section 4.1.3, is created in line 14.

With this information a valued object for the parameter is created and attached to the respective declaration in line 17, whereas the name gets the suffix "_in" to state its role as an input variable. Finally, the valued object is stored as the first element to its variable list, which is then stored in the state variable's `HashMap` of this function's state.

As soon as the inputs are set, the next step is to create the function output if there is any. Listing 5.5 shows the extraction of the function output. In the AST of a function definition the only information about what the function returns is given by the declaration specifier. Thus, the first step for the extraction of the state's output is to retrieve this information and translate it to an SCCharts type in lines 1 to 3. If the specified type is a non `void` basic type, an output is created in a similar way to the creation of an input.

So in lines 6 to 9 the corresponding variable declaration is created, this time the output flag is set to true instead of the input flag. Then, in lines 12 to 21 the respective valued object is created, paired

```
// Create inputs for the functions parameters
val parameters = funcDeclarator.getParameters
for (par : parameters) {
   // Determine parameter type
   val decl = createVariableDeclaration
   val declSpecifier = par.getDeclSpecifier
   if (declSpecifier instanceof IASTSimpleDeclSpecifier) {
      decl.type = declSpecifier.type.CDTTypeConversion
      decl.input = true
      state.declarations += decl
   }
   // Determine parameter name
   val varName = par.getDeclarator.getName.toString
   var varList = <ValuedObject> newArrayList

   // Create valued object for the input
   val VO = decl.createValuedObject(varName + "_in")
   VO.insertHighlightAnnotations(par)

   // Attach the valued object to its list and the list to the map
   varList.add(VO)
   stateVariables.put(varName, varList)
}
```

Listing 5.4. The extraction of the inputs for the function state

```
val declSpecifier = func.getDeclSpecifier
if (declSpecifier instanceof IASTSimpleDeclSpecifier) {
   val type = (declSpecifier as IASTSimpleDeclSpecifier).type.CDTTypeConversion
   if (type !== null) {
      // Determine output type
      val retDecl = createVariableDeclaration
      retDecl.type = type
      retDecl.output = true
      res.declarations += retDecl

      // Set output name to res
      var varName = "res"
      var varList = <ValuedObject> newArrayList

      // Create valued object
      val resVO = retDecl.createValuedObject(varName + "_out")
      resVO.insertHighlightAnnotations(declSpecifier)

      // Attach valued object to the list
      varList.add(resVO)
      stateVariables.put(varName, varList)
   }
}
```

Listing 5.5. The extraction of the output for the function state

with a corresponding variable list and both are stored in the `stateVariables` `HashMap` of the function's state. This time the valued object's suffix is "_out" to indicate its role as output variable.

The last step to complete the function state is to translate the function body. As noted before the body is represented in the AST by the `CompoundStatement` node. For the translation of it a separate function is defined, because as explained in Section 4.1.1 the bodies of the other control statements are also represented as `CompoundStatement` nodes and can therefore be translated by the same function.

5.2.2 Compound Statement Extraction

The `CompoundStatement` node has one child for every statement in it. Thus the function to translate it takes the AST node as input paired with the state it creates the dataflow region for. In the first step the empty dataflow region is created, before every statement node that is a child node to the `CompoundStatement` invokes a separate translation.

Depending on which kind of statement node is found a different translation is triggered. A `DeclarationStatement` node is translated into a new variable declaration added to the dataflow region. An `ExpressionStatement` node or `ReturnStatement` node leads to the translation of the included expression into an SCCharts expression. A control statement's node leads to the creation of a new state, which is referenced in the created dataflow region, which is described in more detail in the following.

5.2.3 Control Statement Extraction

Following the concept for extracting the control statements, as described in Section 4.1.2, the process is very similar for each of the statements. Therefore the basics will be explained on the example of the `while` statement, before a deeper insight is given for the more complex `switch` statement.

While Statement Extraction

The function for the translation of the `while` statement is shown in Listing 5.6. The function takes the AST node of the statement as input, paired with the state and dataflow region that represent the code body the `while` statement was found in.

The first step in line 3 is to create the state that will represent the statement and name it after the stated guideline. After that, the state needs to be referenced in the given dataflow region. To achieve this a reference declaration is created, attached to the region and a respective valued object to represent the state is created, as shown in lines 5 to 8.

Following that, the needed variables of the referenced state need to be attached. Thus, in line 11 a function is called to set the inputs of the state and in line 12 a function is called to set its outputs. Thereafter, in lines 14 and 15 the condition expression of the statement is extracted and written to the state's label. Finally, in lines 17 to 19, the body of the loop is translated into a dataflow region which is connected to the statement's state and labelled following the guideline defined in the concept.

The in- and outputs

To connect for example the newly created while state with its used and written variables multiple steps are needed. It has to be analysed, which variables are needed. For this purpose—in respect to the inputs—the function shown in Listing 5.7 is used.

The main idea of this process is to recursively check every child node of the `WhileStatement` node to find `IdExpression` nodes which represent variable uses. Thus as soon as such an expression is found

```
1  def buildWhile(IASTWhileStatement whileStmt, State rootState, DataflowRegion dRegion) {
2     // Creating the state
3     val whileState = createState("While_" + whileCounter)
4     // Reference the new state in the dataflow-region of the containing compound statement
5     val refDecl = createReferenceDeclaration
6     dRegion.declarations += refDecl
7     refDecl.setReference(whileState)
8     val whileObj = refDecl.createValuedObject("While_" + whileCounter)
9     whileCounter++
10    // Create the in- and outputs and link the respective valued objects
11    setInputs(whileStmt, rootState, whileState, dRegion, whileObj)
12    setOutputs(whileStmt, rootState, whileState, dRegion, whileObj)
13    // Set the condition Expr of the while as label
14    val condExpr = createKExpression(whileStmt.getCondition, whileState, dRegion)
15    whileState.label = "while " + condExpr.serialize
16    // Build the body of the while
17    val whileDBodyRegion = buildCompound(whileStmt.getBody as IASTCompoundStatement, whileState)
18    whileState.regions += whileDBodyRegion
19    whileDBodyRegion.label = whileState.label
20 }
```

Listing 5.6. The function to translate a `while` statement into SCCharts dataflow

in line 4 and the variable is not locally declared inside the `while` loop, the respective variable name is attached to the resulting set of inputs.

Two other statements are interesting. First, a `BinaryExpression` node could represent an assignment, which means that the `IdExpression` corresponding to the variable the assignment writes to is not an input and should therefore not be found, which is checked in lines 10 and 11. If the expression is not an assignment, both operands are checked. Second, a `FunctionCallExpression` node does not only contain `IdExpression` nodes for potential variables used as arguments, but it also contains one to represent the called function's name. That `IdExpression` node however should not be read as an input either, which is checked in lines 17 to 23. Finally, if another statement is found, each of its child nodes are checked in lines 24 to 26.

The method for finding outputs works on the same principle with the difference, that only the `IdExpression` of the first operand of an assigning `BinaryExpression` is needed. Thus, every node not being a `BinaryExpression` is skipped.

With the complete sets of in- and outputs for each entry a respective variable declaration is created and attached to the corresponding state. Further, for each declaration a valued object and a list to organize the SSA form is created. The lists are attached to the `HashMap` of the statement's state variables. With the last step the newly created valued objects are attached to their corresponding counterparts of the dataflow region representing the code body the translated statement was found in. For example, if a `while` statement is translated, which reads a variable x, then the valued object representing the input value for this variable in the state created for the `while` statement is connected to the last used valued object representing the variable x, before the `while` statement was found. If the statement does write to a variable y, then—in respect to the SSA form—a new valued object is created in the dataflow region containing the `while` and the output object is assigned to the newly created one.

```
1  def HashSet<String> findInputs(IASTNode stmt, State parentState) {
2     var inputs = <String> newHashSet
3        // Add a found non local variable use
4        if(stmt instanceof IASTIdExpression) {
5           val varName = stmt.getName.toString
6           if (VOs.get(parentState).containsKey(varName)) inputs += varName
7        // Consider only variables that are not target of an assignment
8        } else if (stmt instanceof IASTBinaryExpression) {
9           val operator = stmt.getOperator
10          if ((operator >= IASTBinaryExpression.op_assign) && (operator <= IASTBinaryExpression.
                 op_binaryOrAssign)) {
11             inputs = findInputs(stmt.getOperand2, parentState)
12          } else {
13             inputs = findInputs(stmt.getOperand1, parentState)
14             inputs += findInputs(stmt.getOperand2, parentState)
15          }
16       // Consider only arguments of a function call
17       } else if (stmt instanceof IASTFunctionCallExpression) {
18          val arguments = stmt.getArguments
19          for (argument : arguments) {
20             inputs += findInputs(argument, parentState)
21          }
22       // Test all children of other statements
23       } else {
24          for (child : stmt.children) {
25             inputs += findInputs(child, parentState)
26          }
27       }
28    inputs
29 }
```

Listing 5.7. The function to find all read variables of the given statement

Switch Statement Extraction

The extraction of the `switch` statement does, until the translation of the body, follow the same principle as the extraction of the `while` statement, with the only difference, that as mentioned in Section 4.1.2 the region of the switch state is a controlflow region containing an initial state, a final state, and states for each `case` and `default` statement of the `switch` statement's body. Therefore the statements in its body have to be translated differently. Listing 5.8 shows the basic setup for the translations of this body. Three variables are needed: First, `noBreakCase` stores the state of the last already processed `case` that was not closed by a `break` statement. Second, `activeCase` stores the state for the `case` statement that is currently processed. Third, `activeDRegion` stores the dataflow region of the state that is currently processed. After that the statements of the body are checked one by one.

On the occurrence of a `case` or `default` statement, a state will be created, which will be discussed later on. If a `break` statement is found, as shown in lines 12 to 14, an outgoing transition from the active state to the final state is created before setting the `activeCase` variable to `null`. If any other statement is found it is translated the same way as in any other compound statement and the created translation is added to the dataflow region of the currently processed state.

The actual creation of the case states is shown in Listing 5.9. First, if a new `case` or `default` statement is found the old active state is written to the `noBreakCase` variable, as shown in line 2. Second, in lines

```
// Stores a case state if it is not ended by a break
var State noBreakCase
// Stores the currently processed case state
var State activeCase
// Stores the dataflow region of the currently active state
var DataflowRegion activeDRegion

for (child : swBody.children) {
   // If a case or default statement is found a state needs to be created
   if (child instanceof IASTCaseStatement || child instanceof IASTDefaultStatement) {
      [...]
   // The break statement creates the leaving transition and clears the active state
   } else if (child instanceof IASTBreakStatement) {
      activeCase.createTransitionTo(finalState)
      activeCase = null
   // Every other statement is part of the cases body and is translated to its dataflow region
   } else {
      buildStatement(child as IASTStatement, swState, activeDRegion)
   }

}
```

Listing 5.8. The translation of the `switch` statement's body without the translation of `case` and `default` statements

```
// Store the last active state to non break - if a break occurred activeCase is null
noBreakCase = activeCase
activeCase = cRegion.createState("")

activeDRegion = createDataflowRegion("")
activeCase.regions += activeDRegion

// Create a transition from the last case to the new if no break was found
if (noBreakCase !== null) {
   noBreakCase.createTransitionTo(activeCase)
   noBreakCase = null
}
// If the state is for a case the respective transition needs a trigger
if (child instanceof IASTCaseStatement) {
   val caseTransition = initState.createTransitionTo(activeCase)
   caseTransition.trigger = child.getExpression.createKExpression(swState, dRegion)
   activeCase.label = controller.serialize + " == " + caseTransition.trigger.serialize

} else {
   // The default state does not need a trigger
   initState.createTransitionTo(activeCase)
   activeCase.createTransitionTo(finalState)
   activeCase.label = "default"
}
```

Listing 5.9. The creation of the `case` and `default` states inside the `switch` statement

3 to 6, the new state and its dataflow region is created. Third, it is checked if the previous state was not closed by a `break` statement. If a `break` statement had been found before the `activeCase` variable was written to the `noBreakCase` variable, it would have been `null`. Thus only if no `break` statement was present the condition in line 9 would be true and a transition from the old to the current state would be created.

Last needed is the transition leading to the newly created state from the initial one. It has to be distinguished, if the starting statement was a `case` or a `default` statement, since a `default` statement has no condition. Thus, if the created state represents a `case` statement, the condition expression of the `CaseStatement` node is translated into an SCCharts expression and becomes the trigger of the transition in line 16, if the state represents a `default` statement no trigger is used, but an additional transition is created from the default state to the final state.

5.2.4 Expression Extraction

The extraction of an expression follows a similar principle as the extraction of a statement. Depending on the kind of expression meant to translate a different approach is taken.

For the translation of an `IdExpression`, the valued object corresponding to the attached variable is retrieved and a reference expression to this object is returned. For a `CastExpression` the cast itself is skipped and the expression which was meant to be cast is translated recursively. For the `LiteralExpression` an SCCharts value corresponding to the litral's value is returned.

`BinaryExpression` nodes and `UnaryExpression` nodes both consist of an operator paired with operands, where the operands again can be expressions. For the translation the SCCharts version of an operator expression is created with the corresponding operator type. These operator expressions can contain sub references referring to their operands. Thus, for each operand of the original expression an additional expression translation is done and the result is added as sub reference to the operator expression.

The more complex expressions are the `ConditionalExpression` and the `FunctionCallExpression`. A `ConditinalExpression` in its core is just a simpler version of an `if` statement. Therefore the translation follows the same rules. A state is created to represent the expression, the read variables in the expression are used as the state's inputs and it has a single output representing the result of the expression. Further a reference to the valued object representing this state's output is the return value of the translation of the `ConditionalExpression`.

The translation of a `FunctionCallExpression` follows a similar route, as shown in Listing 5.10. First in lines 2 to 4 a reference declaration is created and added to the dataflow region the expression belongs to. Then, in lines 5 and 6, the state corresponding to the called function is retrieved. In lines 7 to 9 the reference is set to the function's state and a representing valued object is created. Following this, in lines 12 to 14 a reference to the valued object corresponding to the function's output is retrieved, which will be the translations output. If this function call for example is used to assign the functions result to a variable, the reference to the function state's output valued object is needed for the assignment to the valued object representing the variable.

Finally, to complete the translation of the function call, the argument expressions have to be attached to the inputs of the function state. For each argument of the function call the respective input valued object is retrieved in line 19 and the argument expression is translated into an SCCharts expression in line 20, before an assignment assigning the expression to the valued object is added to the dataflow region.

```
1  // Create the reference to the function's state and create the valued object
2  val funcCall = expr as IASTFunctionCallExpression
3  var refDecl = createReferenceDeclaration
4  dRegion.declarations += refDecl
5  val funcName = (funcCall.getFunctionNameExpression as IASTIdExpression).getName.toString
6  val refState = funcName.findFunctionState
7  refDecl.setReference(refState)
8  refDecl.annotations += createTagAnnotation("Hide")
9  val funcObj = refDecl.createValuedObject(funcName)
10
11 // Create the reference to the function's output valued object
12 expression = funcObj.reference => [
13    subReference = refState.declarations.filter(VariableDeclaration).map[ valuedObjects ].flatten.filter
          [ name == "res_out" ].head.reference
14 ]
15
16 // Convert the argument expressions and attach them to the input valued Objects of the referenced state
17 var i = 0
18 for (argument : funcCall.getArguments) {
19    val funcObjArg = refState.declarations.filter(VariableDeclaration).map[ valuedObjects ].flatten.get(
          i)
20    val connectExpr = argument.createKExpression(funcState, dRegion)
21    dRegion.equations += createAssignment(funcObj, funcObjArg, connectExpr)
22    i++
23 }
```

Listing 5.10. The algorithm to translate a `FunctionCallExpression`

5.2.5 Valued Object Handling

The last part to be described for the implementation of the dataflow extraction is how the handling of valued objects is done. All valued objects used for the SSA form to represent one variable are stored in one `ArrayList`. Within these lists, the first valued object is the one representing the input value, if there is any, and the last valued object is representing the output value, if there is any. Further, every time a new assignment onto this variable is processed, a new valued object is created and added at the end of the list, right before the output object. Therefore, the algorithm to retrieve a valued object by its variable's name is separated into retrieving the last used valued object corresponding to the named variable and creating a new valued object if needed. The first part is shown in Listing 5.11.

The first step, which can be seen in lines 3 to 6, is the quick approach of taking the last valued object out of the list as first candidate for the resulting object. If this valued object is not an output value and no write is indicated by the function parameter `writing`, then this first candidate will be returned.

However, if this candidate has the suffix "_out" in its name, it is stated by the naming conventions that it is the output value. Since the output values have to be the last valued object written to in the chain of valued objects representing a variable, the returned object must not be the output valued object. Therefore, it has to be checked if another candidate exists. If the list has another valued object stored the preceding one is selected in lines 12 to 14. If no preceding object exists, a new one has to be created, what is handled in lines 15 to 31. First, it is saved that the list only had the output value stored. Second, the variable declaration of that output value is retrieved and its type is stored. Third, in lines 22 to 27 a new variable declaration is created with the same type and attached to the dataflow region,

```
def ValuedObject findValuedObjectByName(State state, String name, boolean writing, Region dRegion) {
   // Retrieve the last valued object of the variables list
   var ValuedObject vo
   val varList = VOs.get(state).get(name)
   vo = varList.get(varList.length - 1)
   var varName = vo.getName()
   // If the last object is the "_out" object the preceding valued object has to be used
   var onlyOut = false
   var hasOut = false
   if ((varName.contains("_out")) && (varName.lastIndexOf("_out") === (varName.length - 4))) {
      hasOut = true
      if (varList.length > 1) {
         vo = varList.get(varList.length - 2)
         varName = vo.getName()
      // If the list only contains the "_out" valued object, a preceding one has to be created
      } else {
         onlyOut = true
         var varDecl = vo.getVariableDeclaration
         varName = varName.substring(0, varName.length-3) + "0"
         val type = varDecl.type
         // If the original variable declaration was attached to the state the new one should be
              attached to the dataflow region
         if (!dRegion.declarations.contains(varDecl)) {
            varDecl = createVariableDeclaration
            varDecl.annotations += createTagAnnotation("Hide")
            varDecl.type = type
            dRegion.declarations += varDecl
         }
         // Create the valued object and add it to the variable list, before the "_out" object
         vo = varDecl.createValuedObject(varName)
         varList.add(0, vo)
      }
   }

   [...]

   vo
}
```

Listing 5.11. The first part of the algorithm to retrieve a valued object by its variable's name

what is done to hide the declarations of the valued objects only created for the SSA-form. Fourth, in lines 29 and 30 the new valued object is created and added to the respective list, positioned before the output object.

If the requested valued object is stated to be used for a reading purpose the current candidate can be returned, but if a write to the valued object is indicated, than a new object has to be created. If this creation was already done, because only the output object was present, this step can be skipped. The creation of this valued object follows the same principle as the previously described one. The only difference is that the name of the new object is determined in respect to the name of the previous candidate. If the suffix of the previous candidate was "_in" the new created valued object receives the suffix "_0". If the old suffix was a number, this number is increased by 1 and becomes the suffix of the new created valued object.

5.3 Statemachine Extraction

To begin the extraction of a state machine out of one of the definition patterns described in Section 4.2.1 the marking comments for identifying the different state machine elements have to be used. Even if the comment nodes are not part of the AST structure, it is still possible to retrieve a list containing all comment nodes of the given source code. The order of this list follows the order of the comments in the given source code and each comment node has information about its source code location attached to it.

Listing 5.12 shows the recursive algorithm that is used to identify the AST node following a given comment node. In the beginning, the location information of the given comment is received in lines 4 and 5. After that, the child nodes of the given AST node are checked. If there is a node, whose offset is smaller than the comments offset, but the sum of the nodes offset and length is larger, then the comment is inside the currently checked node. This also means, that the searched node is inside of it. Therefore, the search is restarted for the children of this node, as shown in lines 16 to 20. If no such comment containing node can be found before a node is checked with an offset larger than the comments offset, this means that the searched node is found, because as with the comment nodes, the child nodes of an AST nodes are ordered following their occurrence in the source code. Thus, the first node with a larger offset than the comment node, is the first node following the comment in the source code.

As soon as the comments are used to identify the needed elements, the states and events are created for the resulting SCChart and the resulting objects are saved in `HashMaps` for later use. From this point on, the extraction of both state machine patterns varies and will therefore be explained separately.

5.3.1 Centralized Statemachine

To continue with the extraction of the centralized state machine pattern additional elements have to be found: the current state variable, the initial state, and the state change function. The current state variable is found with the central `switch` statement. Following the design pattern this `switch` statement uses the current state variable as its controller expression, thus the variables name can be extracted out of the `SwitchStatement` node's children.

With the state variable identified, the whole AST is checked via a depth-first-search for binary expressions with an assignment operator, where the first operand is the state variable. Following the design guideline, there are only two such expressions. The example code shown in Listing 4.13 lists these expressions as `currentState = STATE_0` and `currentState = newState`. One of these expressions assigns one of the state enumerators (`STATE_0`) to the current state variable. This assignment sets the

```
def IASTNode getCommentFollowingNode(IASTNode ast, IASTComment comment) {
   var IASTNode node
   // Get comment location information
   val commentContainingFilename = comment.getContainingFilename
   val commentOffset = comment.getNodeLocations.get(0).getNodeOffset
   // Iterate thourgh the ast childs to find the following node
   var found = false
   for (var i = 0; i < ast.children.length && !found; i++) {
      // Get the node location information
      val child = ast.children.get(i)
      val childFilename = child.getContainingFilename
      val childOffset = child.getNodeLocations.get(0).getNodeOffset
      val childLength = child.getNodeLocations.get(0).getNodeLength
      // Check if the node is in the same file as the comment
      if (childFilename.equals(commentContainingFilename)) {
         // Check if comment is inside the node
         if (childOffset < commentOffset && (childOffset + childLength) > commentOffset) {
            // Following node has to be inside the current node
            node = getCommentFollowingNode(child, comment)
            found = true
         // First comment with greater offset is the comment following node
         } else if (childOffset > commentOffset) {
            node = child
            found = true
         }
      }
   }
   node
}
```

Listing 5.12. The algorithm to find the AST node following the given comment node in the source code

initial state. Thus the enumerator used in it indicates the initial state. The other assignment is used inside the body of the state change function and can therefore be used to identify this function, by looking up the parent nodes until a `FunctionDefinition` node is found.

The last preparation step, before the transitions can be created, is to check all defined functions of the given source code, if they contain a call of the state change function. The results of this analysis are then stored in a `HashMap`, mapping the function names to a list of potential target states.

Building Transitions

As soon as all elements of the state machine are identified and created, the central `switch` statement is analysed to extract the transitions. The algorithm for this is shown in Listing 5.13. Before the actual statement is worked on an `ArrayList` is set up in line 2, which is used to store possible start states for the transitions during the extraction. Following that, each child statement of the `switch` statement is processed in the `for` loop in lines 4 to 35.

If the current statement is a `case` statement, a new potential start state is found. At this point it is not known, if the previous `case` segment was ended with a `break` statement. Therefore the `ArrayList` for start states is used. Thus, in lines 7 to 15 the state name used in the case expression is used to find the respective state, which is then added to the start states list. If the current statement is a `break`

```
def buildTransitions(IASTSwitchStatement sw) {
   val startStates = <State> newArrayList
   val swCompound = sw.getBody
   for (var i = 0; i < swCompound.children.length; i++) {
      var stmt = swCompound.children.get(i)
      // Add the state of the case-stmt to the starting states
      if (stmt instanceof IASTCaseStatement) {
         val caseStmt = stmt as IASTCaseStatement
         val expression = caseStmt.getExpression
         if (expression instanceof IASTIdExpression) {
           val caseStateName = (expression as IASTIdExpression).getName.toString
           val caseState = states.get(caseStateName)
           caseState.insertHighlightAnnotations(stmt)
           startStates.add(caseState)
         }
      // Clear all starting states
      } else if (stmt instanceof IASTBreakStatement) {
         startStates.clear
      } else {
         // If the event handling is used build the transition with trigger
         if (eventHandlingUsed) {
            if (stmt instanceof IASTIfStatement) {
               transitionFromIf(stmt, startStates)
            }
         // Without event handling build the transition without trigger
         } else {
            val targetStates = findTargetStates(stmt)
            for (var j = 0; j < targetStates.length; j++) {
               for (var k = 0; k < startStates.size; k++) {
                  buildTransition(startStates.get(k), targetStates.get(j), null)
               }
            }
         }
      }
   }
}
```

Listing 5.13. The algorithm to extract the transitions out of the central behaviour `switch` statement

statement, the list of starting states is cleared, as shown in lines 17 and 18. If the statement is neither a `case` nor a `break` statement, the potential of a state change is given. Depending on whether the optional event pattern is used or not the statement is analysed. If if is not used, then the function `findTargetStates` is used to analyse the statement. This function returns a list of all target states reachable through the given statement. If the statement is a call of the state change function itself, then the returned list contains the state given as parameter to the function call. The other possibility is that the statement is a call of a function that contains state changes. These functions were analysed after the marking comments, therefore the already prepared list can be returned. As soon as the start and target states are known, the transitions between them are created in lines 28 to 32.

If the event handling is used, the guideline expects an if-else-chain to check the transition triggers. The function called in line 23 extracts the condition expression from the `if` statement and converts it into an SCCharts expression. It then creates transitions from each start state to the target states reachable from the then branch of the given `if` statement, with the extracted expression as its trigger. Then it analyses the else branch, if there is any. If the else body again contains an `IfStatement` node the function is recursively called for this node, otherwise transitions are created for the possible target states of the else branch with the negated expression.

5.3.2 Class Based State Machine

The elements marked by comments in the class based state machine approach are the states, events, the evaluation function of the `Statebase` class, and the state change function. To find the transitions for this pattern it is still needed to identify the evaluation function of each state, followed by analysing them to build the transitions.

To find the evaluate functions a search for all function definitions is done. With the name of the state and the name of the evaluation function known, the needed functions can be filtered out of that list. With the evaluation functions found, for each state the `buildTransitions` function, shown in Listing 5.14, is called.

The already known elements are retrieved in lines 2 to 7. Following that, a list containing the currently active events of the processed `switch` statement are stored in line 8, before each statement in the body of the `switch` is processed.

If a `case` statement is found, this means a new event is checked, and therefore the event is added to the currently active event list, as shown in lines 11 to 18. The list is used, to be able to remember already checked event cases that were not closed by a `break` statement. If a `break` statement is found, processed in lines 20 to 21, the event list is cleared.

If another statement is found, it has to be checked whether a transition has to be created. Therefore, the statement is analysed to find all its target states, by recursively searching for all possible calls of the state change function or transition containing functions in the statements tree structure. For each found target state, a transition has to be created. Depending on the event list, the trigger is set. If the list contains at least one event, an expression is created in line 29. For each additional event, the so far created expression is combined with the next event by the use of a *logical OR* operator in lines 32 to 38. The created expression is then set as the trigger of the created transition in line 39.

```
def buildTransitions(String state) {
   val sourceState = states.get(state)
   val evalFunc = stateEvals.get(state)
   var IASTStatement transitionSwitch
   for (child : evalFunc.getBody.children) {
      if (child instanceof IASTSwitchStatement) transitionSwitch = child.getBody
   }
   var evts = <String> newArrayList
   for (child : transitionSwitch.children) {
      // Add a new event if a case statement is found
      if (child instanceof IASTCaseStatement) {
         val condExp = child.getExpression
         if (condExp instanceof IASTIdExpression) {
            var expName = condExp.getName.toString
            if (events.keySet.contains(expName)) {
               evts.add(expName)
            }
         }
      // Clear the event list if a break statement is found
      } else if (child instanceof IASTBreakStatement) {
         evts.clear
      // Build transitions otherwise
      } else {
         val targetStates = findTransitionTargets(child)
         for (targetState : targetStates) {
            val transition = sourceState.createTransitionTo(targetState)
            if (evts.length >= 1) {
               var opType = OperatorType::LOGICAL_OR
               var Expression op1 = events.get(evts.get(0)).reference
               var Expression op2
               // If more than one event is active connect them with OR
               for (var i = 1; i < evts.length; i++) {
                  op2 = events.get(evts.get(i)).reference
                  val tmp = opType.createOperatorExpression
                  tmp.subExpressions += op1
                  tmp.subExpressions += op2
                  op1 = tmp
               }
               transition.trigger = op1
            }
         }
      }
   }
}
```

Listing 5.14. The algorithm to extract the transitions out of the evaluation function of a state's class

Chapter 6

Evaluation

The following chapter evaluates the results of the presented methods to extract dataflow and state machines out of C/C++ code. The first two sections demonstrate the result of the described methods executed on larger projects. Further, a small survey was done to evaluate the benefit the extracted models have over the source code to answer certain questions about the given projects. The results of the survey are presented in Section 6.1.3 and Section 6.2.3.

6.1 Dataflow Extraction

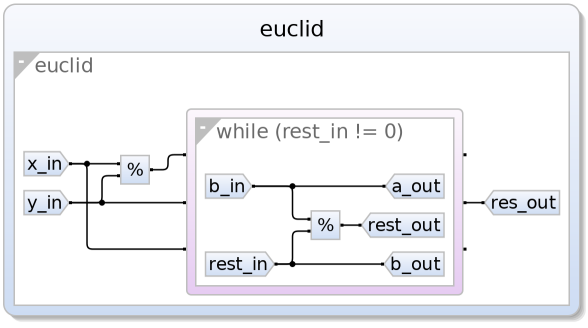
To review the presented basic dataflow extraction a small function implementing the euclidean algorithm was extracted. To review the structflow abstraction, one of the algorithms developed by Janina Reuter as part of her Bachelor's Thesis[Reu19] was used as input.

6.1.1 Basic Extraction Example

The euclidean algorithm calculates the Greatest Common Divisor (GCD) of two given integers, by modulo calculations. A possible implementation of the algorithm in C code is shown in Listing 6.1. This implementation assumes that the greater of the two parameters is given to the first parameter. It then calculates the modulo of the first parameter to the second one and stores it in a local variable in line 5. After that a loop is used to calculate the modulo of the lower value of the last calculation to the modulo of the last calculation as long as the calculated modulo is not zero in lines 7 to 11.

```
1  int euclid(int x, int y) {
2    int a = x;
3    int b = y;
4    int rest;
5    rest = a % b;
6
7    while (rest != 0) {
8      a = b;
9      b = rest;
10     rest = a % b;
11   }
12   return b;
13 }
```

Listing (6.1) The source code of the `euclid` function



(a) The extracted diagram of the `euclid` function

Figure 6.1. The code for the `euclid` function with the extracted representation

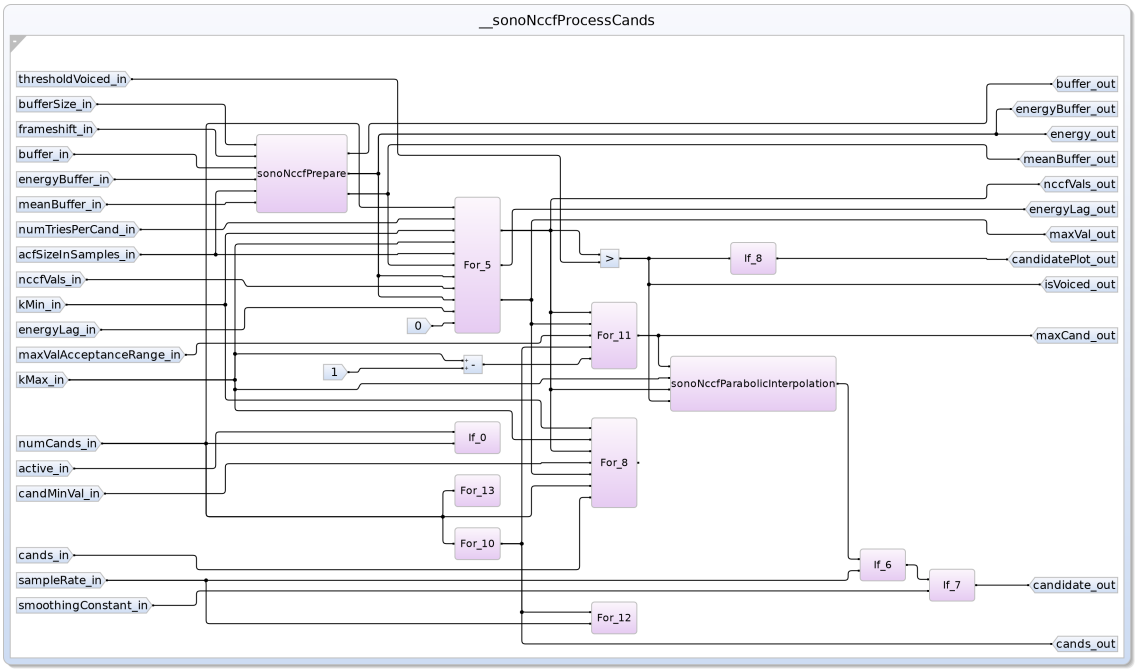


Figure 6.2. The extracted diagram of the structflow from one of the pitch tracking algorithms from Reuter

The extracted diagram of this function is shown in Figure 6.1a. It displays both the function parameters and the function output and connects them via two actors. The first actor resembles the modulo calculation of the inputs and the second one corresponds to the `while` loop. In the shown figure the actor for the loop is expanded to show the extracted loop body. Here can be seen how the values of the local variables are changed during the loops execution. The variable `a` receives the value variable `b` had before the loops execution, variable `b` receives the value variable `rest` had before the loop, and the new value of variable `rest` is calculated by a modulo operation.

The time needed for the extraction of model out of the presented source code averages on an *AMD Ryzen 5 2500U* CPU with a clock frequency of 1.5 GHz on 4 ms and therefore beats every possibility to manually create the model. Further, the diagram depicts exactly the information it was created for. It can be seen what variables are used and how they change during the calculation of the depicted function.

6.1.2 Structflow Extraction Example

The source code used for the structflow extraction example consists of 330 lines of commented code which contain eight different functions following the structflow scheme. The resulting model consists of one state to represent the marked function and references 34 other states to represent the containing functions and control statements. The average time needed for the extraction of the model out of the source code's AST on an *AMD Ryzen 5 2500U* CPU with a clock frequency of 1.5 GHz is 17 ms and therefore excessively faster than the manual creation of an equivalent model.

Figure 6.2 shows the result of the extraction. To reduce the diagram's size, both the declarations of the state's in- and outputs and the port labels for the in- and outputs of the referenced states are hidden.

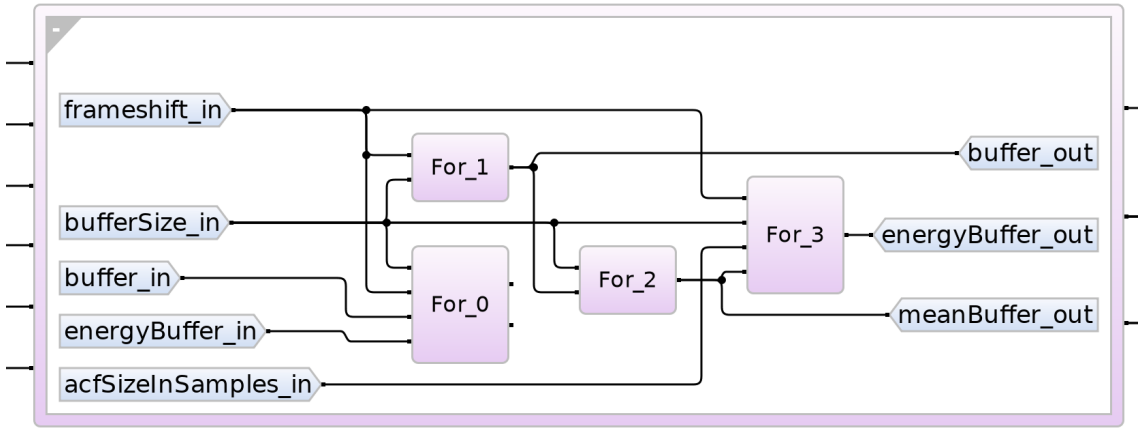


Figure 6.3. A cut-out of the extracted graphic of one expanded function state

The name of the displayed state shows that a function called `__sonoNccfProcessCands` is displayed. As explained in Section 4.1.5 the used abstraction is focused on displaying the elements of one central struct, which are used in the extracted function. It can be seen that 19 different elements of the struct are read during the calculation of the algorithm and 12 are written. Further, the diagram gives direct information about the statements used inside of the function. It can be seen that four `if` statements and six `for` statements that use elements of the central struct are used in the highest layer of the function. The diagram also shows that the functions `sonoNccfPrepare` and `sonoNccfParabolicInterpolation` are called. It displays, which elements of the central struct are read by each of these statements and which are written. Due to these connections it stands out, that for example the `if` statement labelled as `If_0` does no write to one of the struct elements, even if it reads two of them. This means, that assignments inside this `if` statement only assign values to variables that are not elements of the central struct, which are not displayed due to the filtering of the extraction.

Further information can for example be retrieved by expanding the state of the function `sonoNccfPrepare`. As shown by the cut-out Figure 6.3 the updated diagram shows the body of the expanded function. Here can be seen that the expanded function uses multiple `for` statements to manipulate the values of three different elements of the central struct.

A problem that stands out is that the outgoing ports of the `for` statement labelled as `For_0` indicates that struct elements are written to inside of this statement, but the ports are not connected to any of the outgoing elements. This occurs since the respective struct elements are arrays. The presented extraction represents an array only as a single valued object and even if some of the array's elements are written in said `for` statement, other elements are written in the `for` statement `For_1` of `For_3`. The first write gets lost in the diagram, since for every write a new valued object is created. The valued object created to represent the first write is never read before a new one is created to represent the second write. Therefore, the first valued object is not further connected in the diagram. Thus, these kinds of problems occur every time a variable is written two times without reading the first written value. Nevertheless, the highlighting explained in Section 4.3 helps the developer to quickly find the corresponding lines in the source code, which helps to quickly find the source for these kinds of problems.

Table 6.1. The results of the survey to evaluate the extracted dataflow diagram

Task	Code	Diagram
Identify used `for` statements	16 s	3.2 s
Identify read struct elements	24.7 s	7.6 s
Identify last write	27.7 s	8.2 s
Identify written elements	126 s	5.2 s

6.1.3 Diagram Evaluation

To evaluate the benefit of the resulting dataflow diagram a small survey with the PhD students of the Real-Time and Embedded Systems Group from the department of computer science at the Kiel University was done. During the survey the participants had four different tasks to retrieve certain information about the algorithm used as structflow example. One half of the participants was asked to retrieve said information out of the source code and the other half got the extracted diagram to work with. The source code was presented in the CDT editor and all IDE features were usable for the participants. The diagram was presented in the KIELER diagram view. Since the extracted model was meant to document only a certain subset of information about the given source code, the tasks created for this survey are directed to retrieve information out of the same subset. There certainly is information that the created model does not include and that is therefore also not included in the given tasks.

The average results of the survey are shown in Table 6.1. The first task was to identify all `for` statements that were used by the marked function and it can be seen in the results that the participants using the diagram were on average five times faster than the participants using the source code. The second task was to identify the number of struct elements used by a certain function different to the marked function. In this case the participants using the diagram were on average more than three times faster than the source code group. In the third task the participants were asked to identify the function that does the last write to a certain element of the struct and the use of the diagram was again more than three times faster. The last task was to identify all elements of the central struct that are written to during the calculation of the marked function. For solving this task the participants using the diagram were more than 24 times faster than the participants using the source code. Further, no participant of the source code group was able to find the correct solution. They stated they would have additionally required at least a piece of paper to take notes and that the IDE functionalities to highlight the central structs name was essential for the task. These results show that the extracted model can reduce the time needed to find information about source code by a significant amount.

6.2 Statemachine Extraction

To review the state machine extraction two examples are presented. For the centralized approach the already mentioned ardumower project is extracted and for the class-based approach an industry example is described.

6.2.1 Centralized State Machine Extraction Example

The source code containing the ardumower state machine[1] in the version from june 2019 consist in total of 2378 lines of commented code. The defined state machine consists of 26 states, but counting the number of transitions leads to a problem. As mentioned in Section 4.2.2, the definition does not use a clear scheme to define the conditions for the different transitions. In the ardumower project testing for these conditions is not only done inside the central switch statement, but in many cases additional functions are used, which again test several different conditions before a function state is called. This leads to an excessive amount of potential transitions and gives the assumption that during the development of the project no consistent plan for the state machine was followed. As explained, the extraction tries to consider every one of them and at the same time provide a diagram that is still comprehensible. Due to the lack of a manageable event scheme, the different condition tests are neglected and every potential transition is considered with the only restriction, that not two transitions with the same source and target state are created. The extraction of this state machine out of the AST with an *AMD Ryzen 5 2500U* CPU with a clock frequency of 1.5 GHz takes an average of 75 ms, which also exceeds the possibilities of a manual creation in a significant way.

Figure 6.4 shows the results of said extraction. It can be seen that even after constricting the extraction to display only one transition if multiple transitions between the same states exits, the amount of transitions is enormous. Nevertheless the diagram gives information about important parts of the state machine. The state set can be distinguished and it can be seen how these states are connected.

What stands out apart from this is that apparently multiple states, as for example `STATE_ROLL_WAIT` in the top left corner of the diagram, are not even connected to any other states. This happens since the extraction only considers state changes that occur from within the central `switch` statement. The ardumower project however does also trigger state changes before this statement. These state changes can not be considered, because the `cases` of the central `switch` statement are the only source for the extractor to retrieve information about the source state of a state change. Thus, these transitions are not included. Another point that could lead to the unreachable states is the state change function. In the ardumower project that state change function not only writes a new value to the current state variable, but also checks different conditions and depending on them possibly overwrites the new state that is written. These kinds of transitions are also not included.

Thus, the main statement this example gives for the evaluation is that the lack of a well ordered event scheme in the ardumower project leads to an extracted model, which does offer some information, but could be misleading due to the incomplete transition set.

6.2.2 Class-Based State Machine Extraction Example

The source code for the class-based state machine example consists of 747 lines of commented code and describes a state machine with 9 different states which are connected with a total of 25 transitions. Different to the centralized approach the class-based state machine has to have a proper event scheme, due to the way the evaluation functions are defined, which leads to a more consistent extracted model. The execution time for this extraction averages on the *AMD Ryzen 5 2500U* CPU with a clock frequency of 1.5 GHz on only 10 ms.

Figure 6.5 shows the extracted state machine diagram. It displays all 9 states of the state machine and this time depicts all of the transitions. It can be seen that none of the states are unreachable and each transition is labelled with the event necessary for the state change to occur. What stands

[1] `https://github.com/Ardumower`

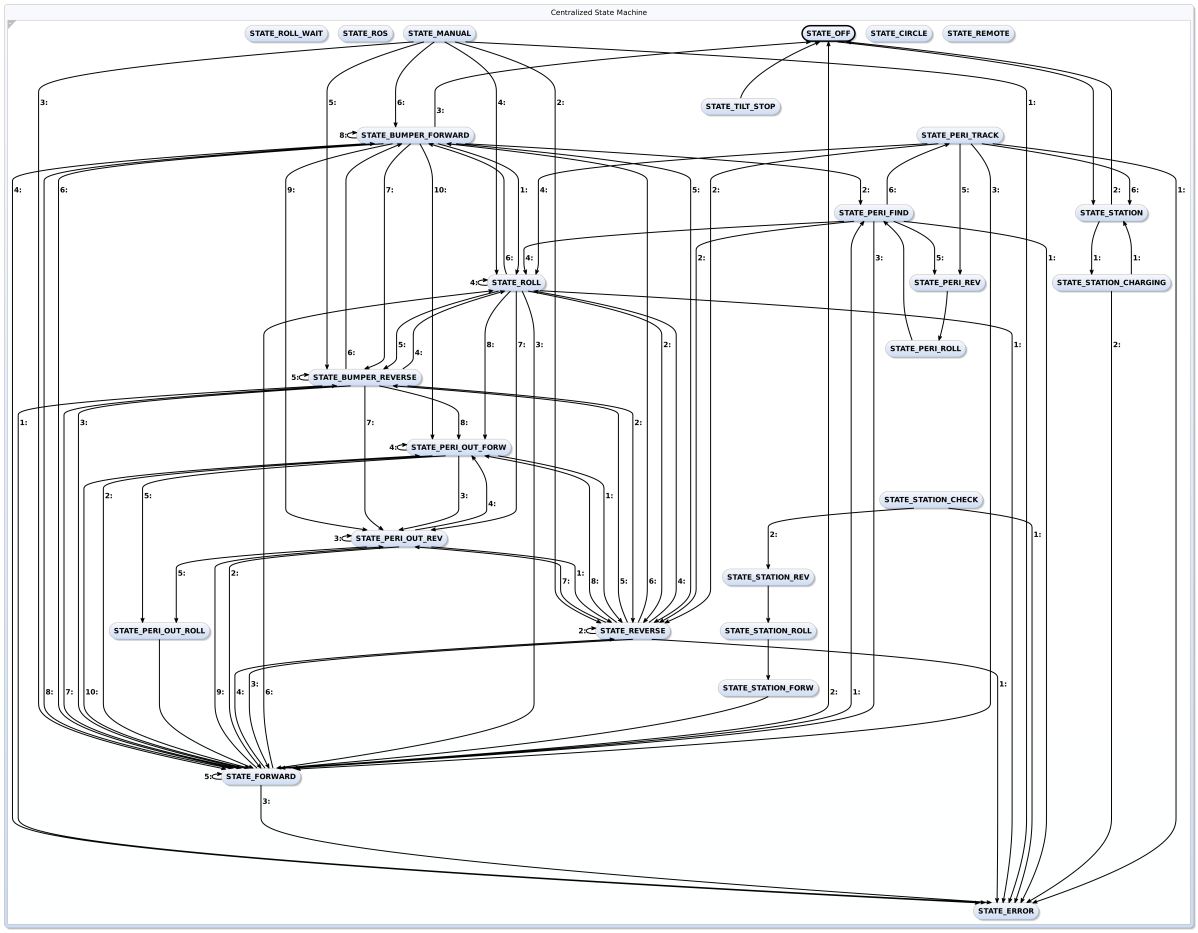


Figure 6.4. The extracted state machine from the ardumower project

out is that multiple transitions lead from the state `StateReady` to the states `StateZwischBemStopp`, `StateDienstUeb`, `StateRandBemStopp`, and `StateReading`. This indicates that the respective calls of the state change function are in different `case` blocks of `StateReady` that are separated by a `break` statement. In the source code this separation is needed due to different side effects that occur inside these `cases`. Even if the extracted model does not depict these side effects, the fact that these transitions did not get merged—as for example the transitions leading from `StatePaper` to `StateInit`—indicates their existence.

6.2.3 Diagram Evaluation

The survey introduced in Section 6.1.3 was extended to also evaluate the extracted state machine graphics. Therefore, the participants had again four different tasks to retrieve information that the model was meant to present out of the state machine diagram from the class-based example. It was chosen to use the class-based example since the centralized example has no event scheme and therefore does not contain all wanted information. Table 6.2 shows the average results of the survey. The first task was to determine how many states the state machine contains. The results show that

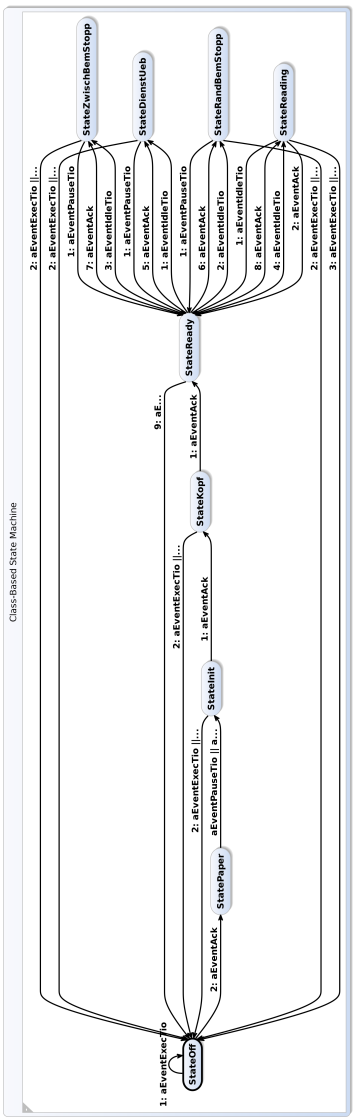


Figure 6.5. The extracted state machine from the class-based industry example

Table 6.2. The results of the survey to evaluate the extracted state machine diagram

Task	Code	Diagram
Identify stateset	11.3 s	4.3 s
Identify path	97.3 s	9.3 s
Identify transitions	56 s	13.3 s
Identify unreachable state	83.3 s	5.2 s

even for this small task, the participants using the diagram were more than twice as fast as the participants using the source code. The second task was to find a path from the state `StateOff` to the state `StateZwischBemStopp`. To solve this task, the diagram group was more than ten times faster than the source code group. The third task was to determine how many incoming transitions the state `StateOff` has, which was solved more than four times faster by the diagram group. For the fourth and final task the transition leading from `StateOff` to `StatePaper` was changed to lead from `StateOff` to `StateReady` to take every incoming transition from `StatePaper`, which makes `StatePaper` an unreachable state. The last task was to identify which of the nine states was not reachable. The results show that the participants using the diagram were 16 times faster than the participants using the source code. This shows, that the extracted diagram helps significantly to keep an overview about a state machine's structure.

Chapter 7

Conclusion

This thesis introduces methods to extract information out of given C/C++ code to create SCCharts models out of it. These models can then be visualized by the KIELER layout tool chain to create software documentation graphics. Concepts are presented for two different kinds of documentation.

First, a method is described to extract a dataflow model out of the given source code. Is is shown that the automatic creation of this model does not only save a lot of time in comparison to a manual creation, but also results in a diagram that gives a better overview to an user about the variables used in an algorithm and the algorithms structure compared to the plain source code.

Second, methods to extract a state machine defined in the given source code is presented. The resulting model includes the defined states of the state machine and shows how they are connected with each other. Again, it is shown, that this automation does not only save a lot of time, but also results in a diagram that helps with answering structural questions about the state machine.

The presented concept shows that the visualization of SCCharts can not only be used for the actual development of SCCharts models, but it can also be used to automatically create informative graphics. The single step needed for this is to add a specialized KiCo to the already existing collection of compilers in KIELER, which maps the wanted information to different SCCharts elements. This opens a complete new topic of possible use cases for SCCharts and their visualization. Besides the extension of the introduced methods of this thesis, it could be possible to find even more aspects of source code documentation that are representable with SCCharts.

7.1 Future Work

The extractions introduced by this thesis create SCCharts models to document different kinds of information about the given source code. However, for both extractions there are also possible improvements or extensions to attach more information to the resulting models or create more flexibility to the extractors. Some of these extensions will be listed in the following.

7.1.1 Missing Features for the Dataflow Extraction

As stated in Section 4.1.4, not every feature of the C language is supported by the introduced extractor: pointers, arrays, and complex types such as structs and manually defined types. Each of these features are often used by developers and supporting them would significantly enlarge the set of translatable programs.

7.1.2 Filtering

The abstraction introduced in Section 4.1.5 does support at least a constrained version of pointer or struct uses and also filters the presented information. The basic dataflow extraction considers every statement of the given source code, where the abstraction only considers statements that include a use

of at least one struct element. Still the resulting model presented in Section 6.1.2 can be overwhelming at the first inspection. The introduction of possibilities to filter the conceivable information could help to find the needed information even better.

There are multiple possibilities to manage this filtering. It could be possible to introduce a way similar to the marking comments to indicate the information that should be filtered before the extraction is done. This way, several different models filtered for different information can be created. Another option for filtering would be to introduce special diagram options to the diagram view displaying the resulting model, to filter it interactively after the extraction is completed.

7.1.3 Additional State Machine Functionality

So far the introduced extraction supports two different state machine patterns. Both of them are sufficient to design simple state machines. Besides states and transitions it is possible to mark one state as the initial state and to define input events which are used as triggers for the transitions.

However, these design patterns are missing possibilities to define other state machine elements that could also be of interest. For example the presented coffee vending machine in Section 4.2 is meant to give out coffee every time the needed amount of one and a half Euro was paid. This information is not included in the resulting model, because the design pattern has no option to define output events.

Outputs are only one possible element that could be included in the design patterns. Some other possible elements would be entry, exit, and during actions, final states or hierarchy.

7.1.4 Work on State Machine Patterns

Besides the extension of the so far introduced state machine patterns it would be helpful to introduce additional patterns. So far even if a state machine is defined in a slightly different way as one of the two patterns requires, it would not be able to extract it. Therefore the introduction of additional patterns would increase the flexibility of the extractor.

A problem that could rise by the introduction of additional state machine patterns is that for two similar patterns it could become complicated for a user to determine which of these patterns is used by his defined state machine. Therefore, it would be helpful to make it possible for the extractor to determine the used pattern dynamically during the extraction.

Another benefit this could be that a dynamic approach could have another way to find the state machine elements in the source code such that the marking comments would not be needed any more.

7.1.5 Secondary Notation

So far it is not possible for the extraction to include any secondary notation into the resulting model. A manually created graphic could contain such information. An algorithm for example could conceptually be divided into a preparation and a calculation step or for a state machine a whole subset of the states could correspond to a certain phase of the state machine's execution while the other states relate to another phase.

Such information cannot be included in the extracted model, but would most likely be depicted in a manually designed graphic. Therefore, introducing a way to include it to the extracted models would increase the quality of the result.

7.1.6 Combining the Extractions

The last point to note is the possibility to combine both extractions. A state machine defined in source code often includes side effects, such as calculations or function calls which are triggered if a certain state is reached. However, the so far introduced state machine extraction only considers structural information about the state machine and therefore does no include these algorithmic side effects. A possibility to include them could be to use the dataflow extraction to translate the side effects to SCCharts and find a way to attach these models to the extracted state machine.

This list of potential extensions for the extraction is most certainly not complete. The field of software documentation is enormous and the presented methods are only a beginning to simplify this whole process.

Bibliography

[Bd91] F. Boussinot and R. de Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1293–1304. DOI: 10.1109/5.97299.

[CPP17] J. Colaço, B. Pagano, and M. Pouzet. "SCADE 6: A formal language for embedded critical software development (invited paper)". In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. Sept. 2017, pp. 1–11. DOI: 10.1109/TASE.2017.8285623.

[DK82] A. L. Davis and R. M. Keller. "Data Flow Program Graphs". In: *Computer* 15.2 (Feb. 1982), pp. 26–41. ISSN: 0018-9162. DOI: 10.1109/MC.1982.1653939. URL: http://dx.doi.org/10.1109/MC.1982.1653939.

[FKH17] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. "Software landscape and application visualization for system comprehension with ExplorViz". In: *Information and Software Technology* 87 (2017), pp. 259–277. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.07.004. URL: http://www.sciencedirect.com/science/article/pii/S0950584916301185.

[FL02] Andrew Forward and Timothy C. Lethbridge. "The Relevance of Software Documentation, Tools and Technologies: A Survey". In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. DocEng '02. McLean, Virginia, USA: ACM, 2002, pp. 26–33. ISBN: 1-58113-594-7. DOI: 10.1145/585058.585065. URL: http://doi.acm.org/10.1145/585058.585065.

[Gio95] J. -. Giomi. "Finite state machine extraction from hardware description languages". In: *Proceedings of Eighth International Application Specific Integrated Circuits Conference*. Sept. 1995, pp. 353–357. DOI: 10.1109/ASIC.1995.580747.

[GKR+14] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. "Textbased Modeling". In: *CoRR* abs/1409.6623 (2014). arXiv: 1409.6623. URL: http://arxiv.org/abs/1409.6623.

[Gri19] Lena Grimm. "From Lustre to Graphical Dataflow Programs". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf. Diploma thesis. Kiel University, Department of Computer Science, May 2019.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. URL: http://www.sciencedirect.com/science/article/pii/0167642387900359.

[HCR+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320. DOI: 10.1109/5.97300.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[Len16] Stephan Lenga. "Model-based compilation of legacy C programs". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sle-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.

[LSF03] T. C. Lethbridge, J. Singer, and A. Forward. “How software engineers use documentation: the state of the practice”. In: *IEEE Software* 20.6 (Nov. 2003), pp. 35–39. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1241364.

[NRE04] R. Namballa, N. Ranganathan, and A. Ejnioui. “Control and data flow graph extraction for high-level synthesis”. In: *IEEE Computer Society Annual Symposium on VLSI*. Feb. 2004, pp. 187–192. DOI: 10.1109/ISVLSI.2004.1339528.

[Reu19] Janina Reuter. “Real-Time Pitch Tracking Algorithms in C to Test Model Extraction”. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2019.

[Sin98] J. Singer. “Practices of software maintenance”. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. Nov. 1998, pp. 139–145. DOI: 10.1109/ICSM.1998.738502.

[SMS+13] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y. Guéhéneuc. “An Empirical Study on the Efficiency of Graphical vs. Textual Representations in Requirements Comprehension”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. May 2013, pp. 33–42. DOI: 10.1109/ICPC.2013.6613831.

[SSH18] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. *Watch Your Compiler Work — Compiler Models and Environments*. Technical Report 1806. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018.

[SSv13] C. Schneider, M. Spönemann, and R. von Hanxleden. “Just model! — Putting automatic synthesis of node-link-diagrams into practice”. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.