C to Dataflow: Support for Further Language Constructs

Jette Petzold and Connor Schönberner

Master Project
October 12, 2021

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

Advised by
M. Sc. Niklas Rentz

Abstract

Andersen introduced the C to dataflow project into Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). It translates C source code to actor-based dataflow diagrams and was later improved by Rentz. C to dataflow aims to augment existing documentation of legacy code. We added further support for various language constructs: `break` and `continue`, pointer, multidimensional arrays, structs and unions, global variables. However, there are still features that need to be covered. These unsupported features are ignored and hence not visualized properly. A first evaluation of C to dataflow is conducted by means of open source software to analyse the usefulness with legacy code.

Acknowledgements

We would like to thank our advisor Niklas Rentz for his advice and constructive feedback on the implemented visualizations. Furthermore, we want to thank Prof. Dr. Reinhard von Hanxleden for his feedback and proposing further features, the scope, and focus of the project.

Contents

1 Introduction **1**

2 More C to Dataflow Features **3**
2.1 Aims . . . 3
2.2 Multiplexer . . . 4
2.3 While Loop . . . 5
2.4 If Statement . . . 5
2.5 Break and Continue . . . 5
2.6 Pointer . . . 8
2.6.1 Return Statement . . . 10
2.7 Arrays . . . 11
2.8 Structs and Unions . . . 11
2.8.1 Pointer Usage . . . 13
2.8.2 Nested Structs and Arrays as Fields . . . 14
2.8.3 Designated Initialization . . . 14
2.9 Global Variables . . . 15
2.10 Robustness . . . 16

3 Evaluation **19**
3.1 Comparing Struct Visualizations . . . 20

4 Future Work **23**
4.1 Further Loop Improvements . . . 23
4.2 Better Array Visualization . . . 23
4.3 Support for Preprocessor Macros . . . 25
4.4 Recursive Functions Break the Automatic Inlining . . . 25
4.5 Pointer Improvements . . . 25
4.6 Further Improvements . . . 25
4.7 Multi-File Support and Browsing Support . . . 26
4.8 C++ to Dataflow . . . 26
4.8.1 Object Orientation . . . 26
4.8.2 Anti-Pointer Features . . . 27
4.8.3 Further Features and Functional Features . . . 27

5 Conclusion **29**

Bibliography **31**

Abbreviations **33**

List of Figures

2.1 Difference between multiple small multiplexers and a big one, exemplarily illustrated for the output of an `if` statement. . . . 4
2.2 The old and the new visualization of the `while` statement. . . . 6
2.3 The old and the new visualization of the `if` statement. . . . 7
2.4 Visualization of the `break`, showing only the `while` body. . . . 8
2.5 Visualization of `break` in nested `if` statements, showing only the `while` body. . . . 9
2.6 Visualization of multiple `break` statements in one `while`, showing only the `while` body. 9
2.7 Visualization of a pointer. . . . 10
2.8 Visualization of a `return`. . . . 11
2.9 Improved array visualization compared to the old array visualization based on code of Reuter [Reu19]. . . . 12
2.10 Struct visualization that works with a struct modelling complex numbers. . . . 13
2.11 Visualization of nested structs and designated initialization. . . . 14
2.12 Visualization of a global variable. . . . 15
2.13 Visualization of an equation that could not be translated because `NULL` is not properly defined. . . . 16
2.14 Visualization of a declaration that could not be translated because `NULL` is not properly defined. . . . 16

3.1 Fully inlined visualization of a function taken from the code of Reuter [Reu19], not readable due to its size. . . . 20
3.2 A function using a struct visualized by the structflow extraction of Andersen [And19]. Graphics taken from [RSA+21]. The function originates from code of [Reu19]. . . . 21
3.3 A function using the struct visualization from Section 2.8. The function originates from code of [Reu19]. . . . 22

4.1 Proposed improvement for read and write operations on the same array. . . . 24

Chapter 1

Introduction

Maintaining or working with legacy code can be a non-trivial task. Adequate documentation of the latter can aid in this case [Par11]. It can offer insights into the overall architecture, connections between components, concepts behind the implementation, and usage of the project or its parts. However, documentation often tends to be outdated or poorly maintained in the course of a project [LSF03]. This is a consequence of the considerable time investment required for creating and maintaining a documentation properly. Due to the fact that software projects are often developed under time constraints, it can happen that a portion of the time for documentation is sacrificed in favour of implementing or improving features. This leads to inconsistencies and as a result in lack of trust in the documentation [Par11]. The resulting documentation's poor state aggravates working with the project's code for future developers.

Andersen introduced the C to dataflow project into KIELER [And19]. It translates C source code to actor-based dataflow diagrams and was later improved by Rentz. Actor-based dataflow models, shortened to actor models, are a specific type of dataflow models that find usage in model driven engineering as well as in commercial and academic tools [RSA+21].

In such a model components named actors execute and communicate with other actors. The interface of actors provides them with an internal state and a behaviour. For this reason, it states how the actor is allowed to interact with the environment. Communication of actors is conducted over ports, included in the interface. In addition, there are parameters configuring the operation of an actor. An actor model also includes communication channels that connect ports and are used to transport data between them. Communication only is allowed over such channels in actor models. As a result, actors can only interact with actors to which they are connected over channels [LNW03].

C to dataflow aims not to replace the manual creation process of documentation. Instead it is meant to augment existing documentation with automatically generated dataflow diagrams [RSA+21]. All in all, the previous contributions have implemented the core of C to dataflow and support for many core features of the C language. Building on those, we aimed to improve the existing support of language features and implement support for not yet supported language features.

Conceptual thoughts and the implementation of our additions are found in Chapter 2. Afterwards, Chapter 3 examines the usability of C to dataflow for real world code examples. Ideas and concepts for future work are described in Chapter 4. Eventually, Chapter 5 concludes with a summary of this project.

Chapter 2

More C to Dataflow Features

All implemented additions and conceptual thoughts to them are found in this chapter.

In general, our additions to C to dataflow strive to achieve a set of general aims that are outlined in Section 2.1. Implementing multiplexers that accept an arbitrary number of inputs was our first addition to C to dataflow. They are described in Section 2.2. They proved to be useful for several use cases in the course of our project. We transferred recent improvements for `if` conditionals to `while` loops, see Section 2.3. Those improvements are the original use case for the mentioned multiplexers. Following their first use, we also introduced them to `if` statements and adjusted the visualization of `if` statements to our improved visualization of `while` loops, see Section 2.4. Furthermore, we implemented support for `break` and `continue` statements that is described in Section 2.5. In addition, pointer support was added, including pointer operations, pointers in arguments of functions, and more, see Section 2.6. Section 2.6.1 describes improvements for the `return` statement that are a consequence of the previously listed additions such as pointer support and multiplexers. Advancing the visualization of arrays, C to dataflow now also supports multidimensional arrays, see Section 2.7. Pointer support as well as multidimensional array support are used for the newly added struct support which is presented in Section 2.8. Section 2.9 presents the implementation for support of global variables. It also covers our new additions such as pointers and structs. Eventually, Section 2.10 covers changes that improve the reliability and robustness of C to dataflow.

2.1 Aims

In the course of the project we followed three major aims that are the foundation of all implemented improvements.

1. Ultimately, the aim is to have a complete mapping for all language elements of C to actor-based dataflow models. This includes that, if feasible, all textual code should be mapped to dataflow models.

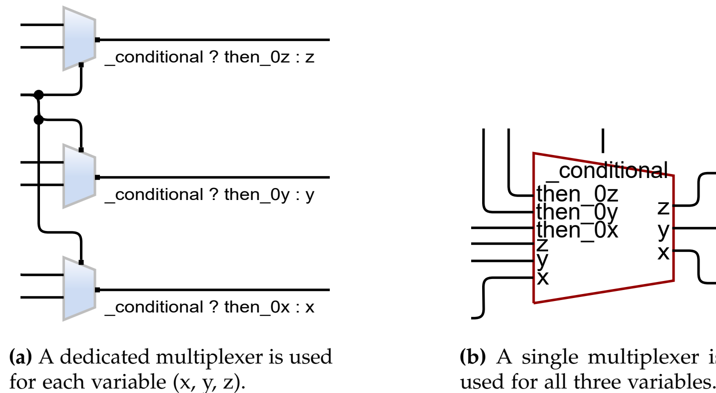
2. The actor-based dataflow models should remain readable. This includes being consistent, clear, and avoiding clutter.

3. Even if certain language elements are not supported yet, the diagram visualization should not lead to crashes.

The first two aims can conflict with each other: In some cases synthesizing the complete source code of a C program can lead to considerably big diagrams. Therefore, it might be a good idea to search for strategies to reduce the amount of elements of the diagram or even to not visualize certain elements. One example of such a problem is encountered for the conditions of `while` loops, see Section 2.3. In general, such cases need to be handled individually.

2.2 Multiplexer

When using a control statement, values of variables could be different depending on whether the statement is executed or not. In the visualization this is shown by using a dedicated multiplexer for each variable, which receives the condition of the control statement as input as well as the two possible values which the variable can be assigned to. The use of multiple multiplexers — one for each variable — leads to lots of visual clutter, conflicting with Aim 2 of Section 2.1. This is why they are replaced by one big multiplexer receiving all variables as input, as seen in Figure 2.1.



(a) A dedicated multiplexer is used for each variable (x, y, z).

(b) A single multiplexer is used for all three variables.

Figure 2.1. Difference between multiple small multiplexers and a big one, exemplarily illustrated for the output of an `if` statement.

The conditional input determines which of the inputs are selected as output. If the condition is true, the first half of the inputs are selected, which are in the example `then_0z`, `then_0y`, and `then_0x`. Otherwise the other three variables — `z`, `y`, and `x`, — are selected. The skin of the multiplexer is not properly integrated in the Sequentially Constructive Statecharts (SCCharts) style yet and should be adjusted in future work.

In order to create such a multiplexer, an actor is created and annotated with the appropriate skin-path. This actor gets an empty label to avoid overlapping of the port labels with the actor label. The skin is provided by a file which leads to the appearance of a multiplexer and has two input ports, a south port for the condition, and one output port. Additionally, the property `allowNonFlowPortsToSwitchSides` is set to true, so that the condition port can also be north if this leads to a better layout according to the used layout algorithm. If more than two input and one output ports are needed, additional ports are added dynamically. This is the case if more than one variable is present. Additionally, if a multiplexer has only one input both input ports are fused. Both cases have not been implemented by us. The actual condition input is connected to the condition's port by using the annotation `toPort <portName>`, whereby the condition's port name is always "in1".

Once the transformation for all nodes is finished, the number of a multiplexer's ports is known. As a result, the size of the multiplexer is adjusted properly. Furthermore, the ports of a multiplexer are sorted: First the ones that should be the output if the condition is true, and beneath them the ports for the values that should be the output if the condition is false. As a reoccurring concept, we use tag annotations to be able to identify specific objects later on. Following this, to determine the case a port belongs to, the valued objects that are connected to the ports receive tag annotations. If a valued object has the tag annotation "pos", it should be the output for the case that the condition is true, otherwise the tag annotation is "neg".

2.3 While Loop

So far, the `while` loop has been visualized by creating an actor that contains the loop body as dataflow and the loop condition as label as seen in Figure 2.2a. We argue that this contradicts the aim to have a complete translation of C code to dataflow diagrams. Consequently, the condition should be visualized and not written as a label.

The new visualization can be seen in Figure 2.2b. In order to separate the loop body from the loop condition, each of them gets its own actor in which it is visualized. Especially in large diagrams, the user could want the condition to appear as text and not as dataflow in order to increase the readability. Regions of actors can either be expanded, meaning they are displayed entirely, or collapsed, meaning their embedded actors are hidden. This behaviour is demonstrated in Figure 2.2, showing a `while` statement. In Figure 2.2b the actor of the condition is expanded, while it is collapsed in Figure 2.2c. Displaying the conditional of a loop as dataflow instead of a textual representation can result into a cluttered actor, conflicting with Aim 2. As a compromise, the label of the condition actor contains the condition in textual form, so that the condition is still stated, even if the actor is collapsed. This is one of the expressed conflicts of our two major aims. It is solved by the described compromise that is able to assure that both aims are considered. The result of the condition actor is then used to visualize that the execution of the loop body, and hence the output of the `while` actor, depends on whether the condition is true or false. Therefore, a multiplexer is used which has the result of the condition actor as the condition and as inputs the output variables of the `while`-loop and the values the variables have before entering the loop. Instead of adding a multiplexer for each output variable of the `while`-loop one big multiplexer is used as introduced in Section 2.2.

In general, it is desirable to indicate that a `while` body is repeated since this should increase the readability. Routing the outputs of a loop body back to its inputs would be a natural way to achieve this. However, this would add several additional wires to the diagram, increasingly cluttering it for an increasing amount of outputs. If `break` or `continue` statements are present, see Section 2.5, this approach would cause even more visual clutter due to needing additional visualizations for loop interruptions. Therefore, we chose to step away from this idea since it violates Aim 2.

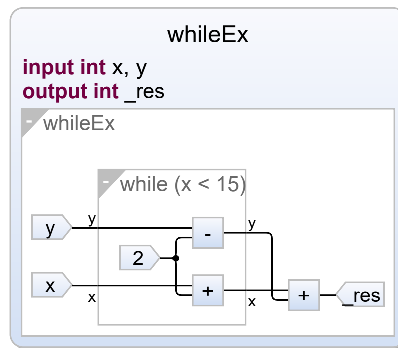
This visualization can be transferred to the `for`-loop and `do while` as well but is currently only implemented for the `while`-loop.

2.4 If Statement

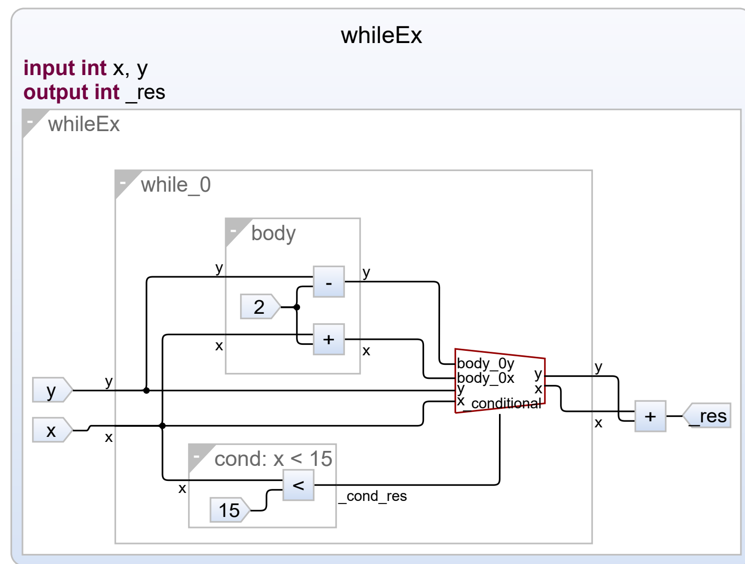
In contrast to the `while`-loop, the condition of the `if` was already visualized but did not have its own actor. In order to be consistent in the visualization, a actor for the condition is added. Additionally, multiplexers are used to determine the output of the `if` actor as well, so they got replaced by one big multiplexer. This can be seen in Figure 2.3 in which the condition `x < 15` has its own actor and the resulting condition is used for the big multiplexer. Both changes are supported by our Aim 2 for being a tool that improves the readability. A difference to the `while` actor is, that the input for the multiplexer could be empty. In this case it is not created.

2.5 Break and Continue

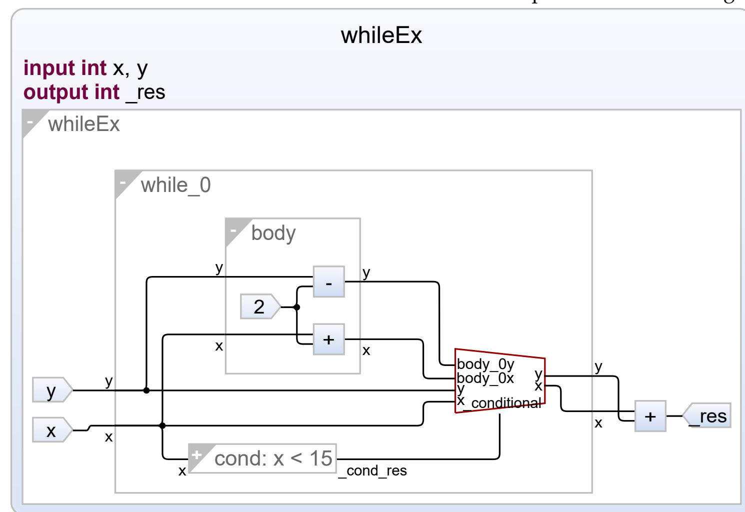
`Break` as well as `continue` statements interrupt or redirect the control flow as well as the dataflow of a program. As important language elements, it is necessary to include them in the mapping from C to dataflow, fulfilling Aim 1. `Break` and `continue` statements are semantically slightly different but



(a) The old visualization of the `while` statement.

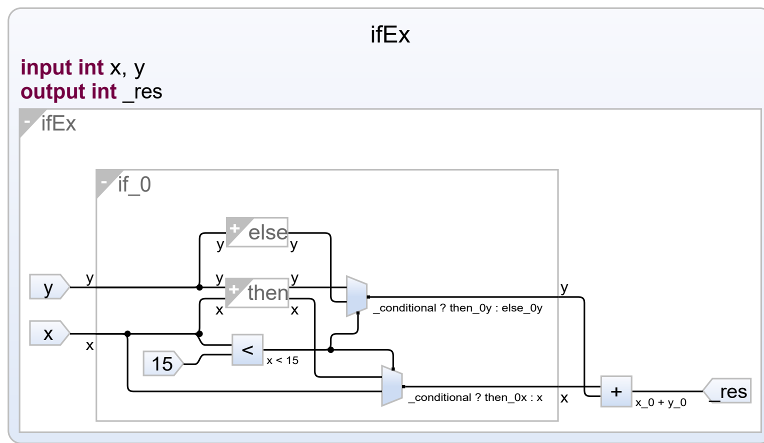


(b) The new visualization of the `while` statement with expanded condition region.

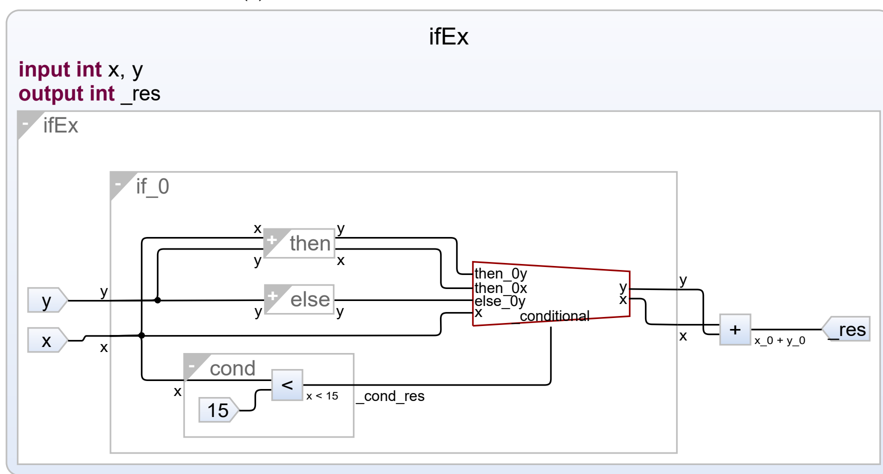


(c) The new visualization of the `while` statement with collapsed condition region.

Figure 2.2. The old and the new visualization of the `while` statement.



(a) The old visualization of the `if` statement.



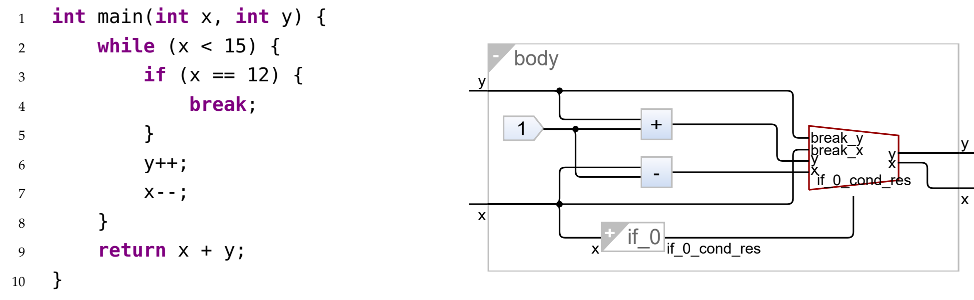
(b) The new visualization of the `if` statement.

Figure 2.3. The old and the new visualization of the `if` statement.

are visualized in the same way because we do not visualize when the loop body is repeated. In the following, both statements are meant although only the `break` statement is mentioned.

Semantically speaking, whether a `break` is executed or not decides which of two possible values of a variable is part of the result of a loop. It also affects whether a loop is executed again but this is not visualized, as mentioned beforehand. Consequently, it has the same effect as a multiplexer. This is the reason why multiplexers are used to visualize `break` statements as seen in Figure 2.4. We decided to locate this multiplexer not in the `if` actor corresponding to the `if` in which the `break` is defined. Instead we place it in the surrounding `while` actor that corresponds to the `while` that is interrupted by the `break`. This choice follows the semantics of a `break` since it determines the output of the `while` actor. Placing it in a potentially nested `if` would lead to a more complex visualization and would reduce readability, violating Aim 2.

The input for the case that the `break` is executed is set as soon as the `break` occurs. Thereby, the current values of all variables that will be changed if the `break` is not executed, are considered. After the translation of the remaining `while` body is finished, the condition and the other inputs are set. For



```
int main(int x, int y) {
    while (x < 15) {
        if (x == 12) {
            break;
        }
        y++;
        x--;
    }
    return x + y;
}
```


Figure 2.4. Visualization of the `break`, showing only the `while` body.

each variable that the `break` actor already received as input for the case that the `break` is executed, it needs an input for the case that the `break` is not executed. This input is the current value of the variable, which means the value at the end of the `while` body. For instance in Figure 2.4 this procedure is observable for the variables `x` and `y`. If the `break` in line 4 is executed the unchanged values of the variables, named `break_x` and `break_y`, are selected by the multiplexer. Otherwise, the decrement of `x` and increment of `y`, computed by the current iteration of the `while` body, are selected.

The condition of the `break` statement's surrounding `if` is reused as condition for the `break`-multiplexer. Syntactically, a `break` does not require a surrounding `if` statement. However, if it is not surrounded by one, the code after the `break` is never reached, hence it is dead code. Consequently, it does not make sense to use a `break` without a surrounding `if`. In the case that the `break` occurs in nested `if` statements, the conditions of these statements are combined with the *AND* operator and the result is used as the condition for the `break`-multiplexer. In Figure 2.5 this leads to the combination of both `if` statement conditions with an *AND*. Hence, the condition for the `break`-multiplexer is `x < 15 && x !=1`.

If a `while` contains multiple `break` statements, for each `break` such a actor is created. The output variables of the last `break` actor are used as input for the negative case in the preceding `break` actor, provided it needs the variables as input, and so on. In Figure 2.6 this applies for the `x` value of the right `break`-multiplexer which represents the first `break` statement. In general, the visualization of `break` statements works for various variants of `if` statements and also in combination with `return` statements.

2.6 Pointer

For now, we focus on pointers of variables. Pointer arithmetic is currently not implemented. For code using pointers without pointer arithmetic, the address a pointer points to is usually not relevant. Instead, the value at the address is of interest. This is the reason pointers are visualized the same way as other variables except in function calls. Figure 2.7 demonstrates this with the function `pointerFunc` called by the function `pointer`. The pointer `p`, pointing to the variable `sec`, is passed to `pointerFunc`, thus it is added to its input. Since `p` is used for changing the value of `sec`, `p` is also set as output of `pointerFunc`. Afterwards, `p` is wired to the result variable `res` of the function `pointer` because it contains the value of the return variable.

In contrast to primitive types, if a function has a pointer as parameter and changes the value the pointer points to, then the value also changes outside of the function. In order to visualize this, all functions that have pointers as parameters and change their values receive them as output variables.

```
1  int nestedBreak(int x, int y) {
2      while (x < 15) {
3          if (x != 1) {
4              if (y == 42) {
5                  break;
6              }
7              x++;
8          }
9      }
10     return x + y;
11 }
```

Figure 2.5. Visualization of `break` in nested `if` statements, showing only the `while` body.

```
1  int breaks(int x, int y) {
2      while (x < 15) {
3          if (x == 0) {
4              break;
5          }
6          x++;
7          y++;
8          if (y == 0) {
9              break;
10         }
11         x++;
12     }
13     return x + y;
14 }
```

Figure 2.6. Visualization of multiple `break` statements in one `while`, showing only the `while` body.

```
1 void pointerFunc(int *p) {
2     *p += 20;
3 }
4
5 int pointer(int sec) {
6     int *p = &sec;
7     pointerFunc(p);
8     return sec;
9 }
```

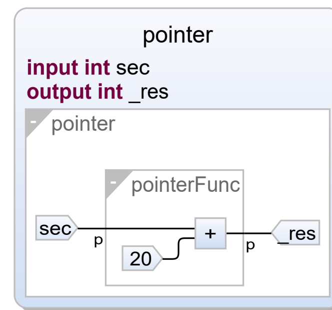


Figure 2.7. Visualization of a pointer.

These outputs can then be wired with the pointer outside of the function actor or directly with the variable that is stored at the address to which the pointer points. If the function call is an unknown function, then it is assumed that the arguments that are pointers are changed and hence are outputs of the function. This procedure ensures that function calls with declared pointer variables or pointers of the form `&variable` as arguments are correctly visualized. In general this means: Whenever possible, the original variable instead of the pointer is used, especially in the statement `return *pointer`.

Currently, there is no visualization for changing the address that is stored in a pointer variable. If the stored address is changed then the visualization should not indicate that the pointer still points to the variable at the location of the old address. This is only partly implemented at the moment. It requires more time investment and testing for assuring that all edge cases work.

2.6.1 Return Statement

So far, only `return` statements of non-void functions were considered. If multiple `return` statements are used in one function, their return values were linked. Now that pointers are supported, they must be considered in `return` statements as well because depending on whether a `return`, that is not at the end of a function, is executed, the value of pointers could be different. For non-pointers this is irrelevant since their values cannot be accessed outside of the scope of their functions, but if a pointer is passed as an argument to a function, the pointer can still be used after the function call.

Similar to the visualization of `break` & `continue`, a `return` is represented by a multiplexer. It is added to the function's actor since only the output of the function depends on the execution of the `return`. In Figure 2.8 the values of the variables to which the pointers `p` and `y` points is changed below the `return`. Since `p` is a parameter and thus the value to which it points could be used outside of the function, `returnEx` has `p` as output. The value of `p` depends on whether the `return` is executed: It takes on either the unchanged or the changed value. This is determined by the multiplexer representing the `return`. However, this does not apply for the pointer `y` because it is only local and not a parameter, and also not for `x`, since the value is passed to the function by call-by-value.

The condition of the `return`'s multiplexer is the condition of the surrounding `if` or in the case of nested `if` statements, the combination of their conditions. Only pointers that are parameters of the function can qualify as input for the multiplexer because only these can be accessed outside of the function. Of those pointers only the ones that are changed after a `return` is not taken are set as input for the multiplexer, since the values of the other ones do not depend on the statement.

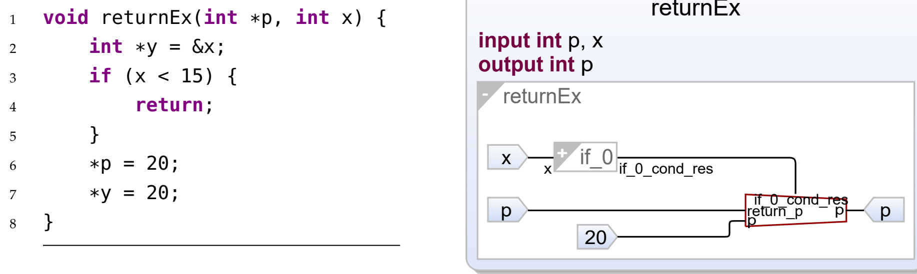


Figure 2.8. Visualization of a `return`.

2.7 Arrays

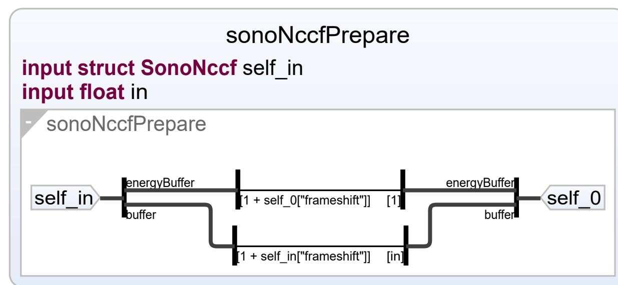
All in all, the existing array implementation already covered most use cases. Nonetheless, support for multidimensional arrays was missing. Implementing it not only helped us achieving Aim 1 but they are also needed for our struct support that builds upon the array visualization, see Section 2.8. The now added implementation for multidimensional arrays builds upon the multidimensional array support of SCCharts. The latter is the foundation for the visualization of the C to dataflow synthesis. It is based on the fact that an array-like valued object is treated as an n-dimensional array by the SCCharts synthesis if you add n indices to it for $n \in \mathbb{N}$. Furthermore, operations on n-dimensional arrays are visualized like chained operations on one dimensional arrays in SCCharts. The described visualization of multidimensional arrays is depicted in Figure 2.11. This figure is technically meant to demonstrate how nested structs are visualized, compare Section 2.8.2. However, since structs are in general represented by arrays and use the same visualization, nested structs are equivalent to multidimensional arrays.

In order to make the visualization of arrays more readable, see Aim 2, the logic for generating new variants for valued objects of arrays has been improved. Beforehand, a new valued object for an array was created after a write following a read operation. Now this is only done when the value at a read index is updated. Additionally, when accessing the value at an index i of an array, then the variant of the array is chosen for the visualization that has received the most recent update of the value at index i. The effect on the array visualization can be seen in Figure 2.9. In it the utilized struct `self` is visualized by using an array. Read and writes are distinguishable by the position of the index next to the lane for array access. A positioning left of the lane indicates a write and a positioning right of the actor indicates a read. Line 8 updates the array `buffer`, which is a field of `self`, resulting in the creation of a new variant because it is the first write to the array of `self`. In the old visualization this newly created variant is used to update the array `energyBuffer`, despite the fact that it has not been changed. Additionally, this update creates a new variant of `self`, albeit `energyBuffer` has not been changed beforehand. As a result, the both array updates are wired in row leading to a less readable visualization, see Figure 2.9b. With the improved handling of arrays both writes update the same variant of `self`, leading to a parallel wiring, as seen in Figure 2.9a. In addition, only one new variant of `self` is created.

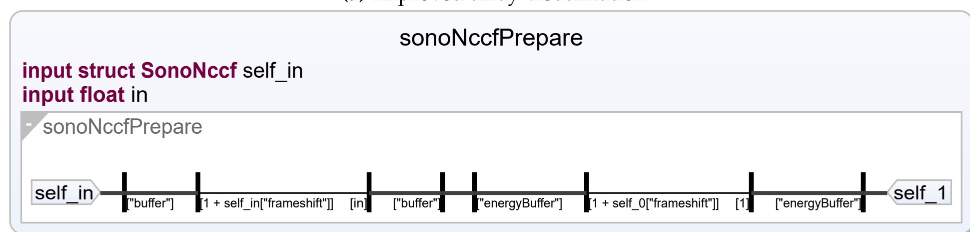
2.8 Structs and Unions

Struct support is introduced by relying on the visualization of arrays. This is accomplished by creating array-like valued objects when structs are declared or expressions or statements operate on the fields of structs. We did not use SCCharts classes respectively its structs because structs cannot be

```
1  struct SonoNccf {
2      int frameshift;
3      float buffer[];
4      float energyBuffer[];
5  };
6  static void sonoNccfPrepare(
       struct SonoNccf self, const
        float in)
7  {
8      self.buffer[in] = self.
           buffer[1 + self.
           frameshift];
9      self.energyBuffer[1] = self.
           energyBuffer[1 + self.
           frameshift];
10 }
```



(a) Improved array visualization



(b) Old array visualization

Figure 2.9. Improved array visualization compared to the old array visualization based on code of Reuter [Reu19].

```
1  struct Complex {
2      float real;
3      float imag;
4  };
5
6  void addNumbers(struct Complex
        c1, struct Complex c2,
        struct Complex *result)
7  {
8       result->real = c1.real + c2
            .real;
9       result->imag = c1.imag + c2
            .imag;
10 }
```

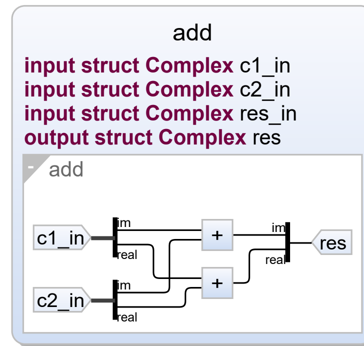


Figure 2.10. Struct visualization that works with a struct modelling complex numbers.

passed in the current version of SCCharts. Our implementation does not require struct definitions to be included in the textual representation. In order to simulate a field access or a write operation to a field, the field's name is passed as an index to the array-like valued object of the struct. As a result, an assignment to a field is represented by the pattern `structName["fieldName"] = value` in the SCCharts model. SCCharts seems to not prohibit this syntax, albeit strictly speaking it violates the semantics of arrays. A semantically correct type for structs is set via the `host` type feature of SCCharts. It follows the pattern `struct <StructType> <nameOfStruct>`. The valued objects of structs are annotated with a tag annotation identifying them as structs. Without such a tag annotation, they are not easily distinguishable from arrays if the equation synthesis should handle them differently. Currently, it is noticeable that the struct support reuses the visualization of arrays, meaning some labels still look like they are meant for array accesses. As a first improvement, the parentheses and quotations mark around accessed and updated variable names on the ports of the array accesses in the visualization have been truncated.

Additionally, SCCharts' arrays seem to have an inconsistency: If values from arrays are read and used in a binary operation, then the operator is only labelled with the array name with no reference to which indices were read. This is caused by labelling the valued objects of arrays. The valued objects of structs do not receive such labels, thus they do not have the same inconsistency. However, the flaw arises in both cases when the multidimensional array visualization is used, see Section 2.8.2.

The implemented handling of structs is reused for supporting unions. For this use case the host type of the valued object is set to `union <UnionType> <nameOfUnion>`. Due to the fact that unions can only hold information in one field at a time, a new variant of the valued object is created in each update.

2.8.1 Pointer Usage

It is common to pass only the pointers of structs instead of the whole struct to functions. For this use case the C standard includes the shorthand `structPointer->field` for dereferencing a struct pointer and addressing one of its fields. This shorthand allows to access the field's value as well as assigning new values to it. Ergo, it can appear on the left side of an assignment as well as on the right side. Due to its nature of being a shorthand, the visualization of the arrow notation is equivalent to the point notation.

The visualization of the arrow notation is shown by Figure 2.10 in the function addNumbers using instances of the struct declaration Complex. Its result is stored in the two fields of the output parameter *result, which are accessed using the arrow notation in lines 9 and 10. In the diagram the resulting writes to both fields are visualized again by using the array visualization. Our visualization even supports the usage of multiple arrows in one expression, which is useful for handling nested structs.

2.8.2 Nested Structs and Arrays as Fields

In C it is allowed to define fields as structs or arrays. Nested fields can be accessed by multiple usage of the point or arrow notation. If a field of a nested struct is accessed, this is visualized by using the multidimensional array visualization, as seen in Section 2.7. The assignment of a value to a field of a nested struct is exemplary modelled by structName["nameOfNestedStruct"]["fieldName"] = val. The visualization of nested structs is shown in Figure 2.11. In it structs of type struct Segment have two fields of type struct Point. The fields of Point instances can be handled in the same way like fields of not nested structs. In addition, the code to the figure shows how writes and reads of nested structs are visualized. Fields with array type are handled and visualized analogously.

```
struct Point {
    int x;
    int y;
};

struct Segment {
    struct Point begin;
    struct Point end;
};

int main() {
    struct Segment d1 =
        {.begin = {.x = 0, .y = 0},
         .end = {.x = 5, .y = 5}};
    struct Segment d2 =
        {.begin = {.x = 6, .y = 1},
         .end = {.x = 7, .y = 6}};

    int bsp = d1.begin.x
        + d2.begin.x;
    return bsp;
}
```

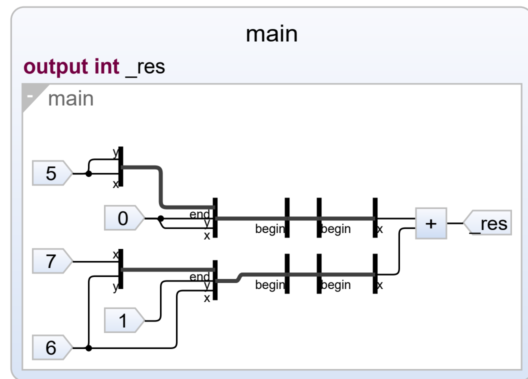


Figure 2.11. Visualization of nested structs and designated initialization.

2.8.3 Designated Initialization

Similar to arrays it is possible to initialize structs via designated initialization in C. This was introduced in C99. For example struct <StructType> <structName> = {.<field0> = <value0>, .<field1> = <value1>}; initializes two fields of a newly declared struct. Beforehand, designated initialization was already implemented for arrays. However, this implementation cannot be reused for structs since it translates the InitializerList that is included in the AST for designated initializations to a

VectorValue. Those VectorValues are then translated to the array visualization. This conflicts with our struct visualization since it uses the names of fields as indices, rather than natural numbers. Instead of reusing the solution for arrays, our one recognizes when a struct should be initialized via designated initialization, sequentially adding an assignment for each initialized field to the passed dataflow region. This process also works for recursive usage of designated initialization, which is useful for initializing nested structs. Similarly, the initialization of array fields and designated initialization of arrays in the designated initialization of structs can be visualized by our solution. An example for nested structs is depicted in Figure 2.11.

2.9 Global Variables

Global variables are supported by adding them as inputs to the actors that use them. Additionally, they are added as outputs to the actors that changes the values of them, so that functions which use them afterwards receive the correct value. In order to connect these outputs and inputs, the surrounding actors — of actors using global variables — also receive them as input/output. An example usage of global variables can be seen in Figure 2.12. In it the global variable `global` is declared and used in the function `useOfGVar` but not in the function `noUseOfGVar`. Consequently, the global variable is an input of `useOfGVar`'s actor, so that it can be used in its region, but is not passed to `noUseOfGVar`.

```
1  struct Distance {
2      int feet;
3      float inch;
4  } dist;
5
6  int global = 42;
7
8  int useOfGVar(int x);
9  int noUseOfGVar(int x);
10
11 int globVar(int x, int y) {
12     return useOfGVar(x + y);
13 }
14
15 int useOfGVar(int x) {
16     return noUseOfGVar(x) +
           global;
17 }
18
19 int noUseOfGVar(int x) {
20     return 2 * x;
21 }
```

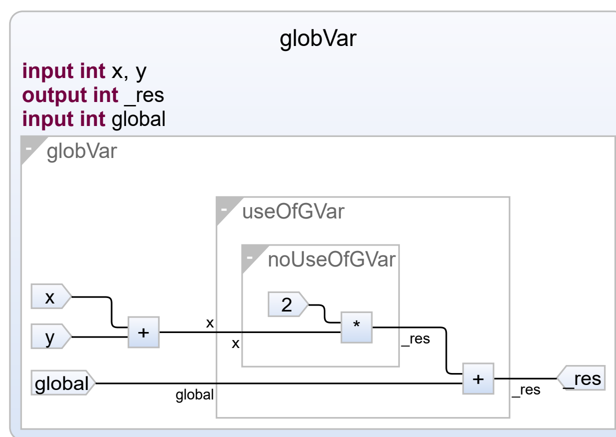
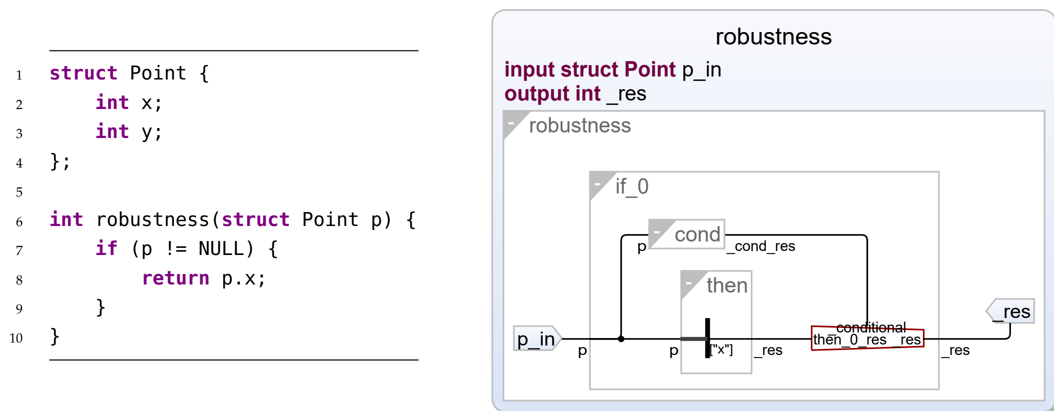


Figure 2.12. Visualization of a global variable.

This visualization works independent of the order of the function definitions in the editor and hence also for endless recursion. The type of a global variable can be a primitive one as well as a struct or pointer.

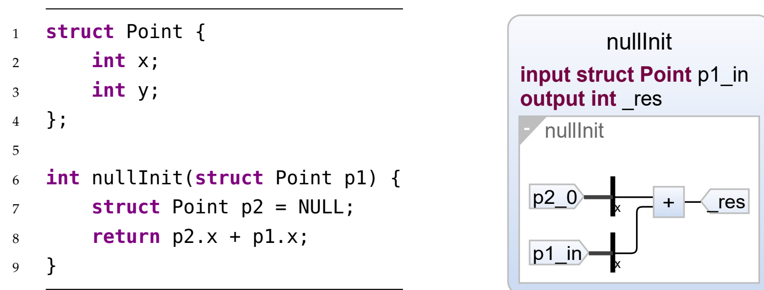
2.10 Robustness

Even with the features introduced here, there are several more expressions that are not supported yet, such as preprocessor macros and function pointers, and cases that previously led to errors during the generation of the diagram. In order to prevent that only a failure message is shown, these cases are ignored such that the remaining code that can be translated to dataflow is still visualized. This implements our Aim 3. In detail this is achieved by only adding equations of which all components are correctly set. In this context, correctly set means no needed parameter is null or that a component has a subcomponent that is null. Figure 2.13 shows the function `robustness` in which the `if` condition could not be visualized because `NULL` is not properly defined. This leads to a visualization in which the condition is not shown as dataflow since the equation could not be translated. Aside from that, it is checked whether valued objects are null. This can be the case when for example `NULL` is used but not properly defined. An example can be seen in Figure 2.14 where `NULL` is used to initialize a variable but is not properly introduced. That is why the initialization is not visualized.



```
1  struct Point {
2      int x;
3      int y;
4  };
5
6  int robustness(struct Point p) {
7      if (p != NULL) {
8          return p.x;
9      }
10 }
```


Figure 2.13. Visualization of an equation that could not be translated because `NULL` is not properly defined.



```
1 struct Point {
2     int x;
3     int y;
4 };
5
6 int nullInit(struct Point p1) {
7     struct Point p2 = NULL;
8     return p2.x + p1.x;
9 }
```


Figure 2.14. Visualization of a declaration that could not be translated because `NULL` is not properly defined.

Usually types that are used should be known either by including the correct header or because it is a primitive type. However, for the case that a type is not known the visualization is still correctly displayed by utilizing the *host* type. Casts with unknown types, such as `(type)(expr)`, are interpreted as function calls and hence visualized as unknown functions.

Additionally, using brackets on the left side of an assignment does not lead to a crash anymore. Moreover, function names can now be in parentheses without changing their semantics or provoking

a crash of the visualization. A small fix now assures that assignments of the pattern `array0[i] = array1[j];` can be visualized.

With these arrangements a visualization is shown which only contains the parts that could be translated to dataflow. At the moment `println` statements are used to show that parts of the original C code could not be translated. However, this comes with the disadvantage that the `println` statements are not visible to users. Thus, it is not always clear when a diagram is not correctly visualized. As a replacement, some form of visual warnings should be used for this task. One good candidate for this purpose are the already implemented warnings of SCCharts.

Chapter 3

Evaluation

Until now, we have presented our improvements for C to dataflow on the basis of small examples. However, it is meant to aid in the context of legacy code. Therefore, it should also be examined for real world code examples. We tested our improvements with code provided by Reuter [Reu19]. C to dataflow still does not support all features of C. Chapter 4 examines what is still missing in C to dataflow. Consequently, progress for fulfilling Aim 1 was made but it was not reached. Especially, the missing support for preprocessor commands hinders C to dataflow to be utilized for many files in open source projects such as Redis[1], GCC[2], and xfwm4[3]. Nonetheless, our improvements allow C to dataflow to process Reuter's code and allow first insights into other projects.

When C to dataflow is used for real world code it tends to generate considerably large models. Generating those takes a perceptible amount of time. On a lower zoom level, the diagram does not offer much information about the underlying code since actors are hardly distinguishable and labels not readable, as seen in Figure 3.1. Consequently, it is hard to deduce the structure and the connections in the diagram. In order to gain more insight, you need to increase the zoom level for inspecting parts of the diagram and reduce it afterwards to see its context. Smart zoom features and advanced browsing features would improve this workflow. Other current projects explore and implement features improving the browsing experience in the KIELER project such as the parallel running project Google Maps for Models. Those improvements could also be used for C to dataflow. Collapsing actors, removing edge labels, and deactivating other visualization features can lead to a slightly better readable diagram in some cases, as can be seen in Figure 3.3.

The still continuing pandemic aggravated to survey our improvements with peers. In a later stage, it should be conducted whether the improvements are understandable and whether they improve C to dataflow. For this we propose a small test run. At first, the test subjects should gain hands-on experience with C to dataflow. This could start with 15 minutes of trying out C to dataflow on code using our improvements. Depending on the existing knowledge of the participants explanations of the visualizations should be provided. Afterwards, a small test could follow that asks the subjects to assign the right diagram to a code snippet and the other way round. Following this, a short poll could ask quantitative and qualitative questions about the implementations. For example: "Does this parameter affect the output/an early return/ a break" or "Do you think that the diagram is clear or too cluttered?" The tests should include assessing whether our changed visualizations for `while` and `if` statements are actually perceived to be better than the prior implemented variants. One addition in the newly implemented variant is that conditions of `while` and `if` statements are included as diagrams as well as label in the header of the condition's state. Reflecting this, the subjects should be asked which option they prefer.

[1] https://github.com/redis/redis

[2] https://github.com/gcc-mirror/gcc

[3] https://gitlab.xfce.org/xfce/xfwm4

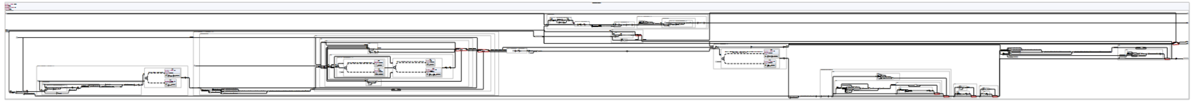


Figure 3.1. Fully inlined visualization of a function taken from the code of Reuter [Reu19], not readable due to its size.

3.1 Comparing Struct Visualizations

While not included in the current version of C to dataflow, Andersen provided an implementation for visualizing structs named structflow extraction. Its core idea is to extract all fields of a struct instance and add them as local variables to the scope in which they are used. Hereby, the structflow extraction focuses on the variables of the fields, hiding all other local variables of the scope. As a result, the structflow extraction violates our Aim 1 and necessitates an implementation such as our struct visualiziation, compare Section 2.8, that includes structs and all other variables of a scope. Figure 3.2 shows the actor of the function `_sonoNccfProcessCands`, using the structflow extraction of Andersen. As mentioned, the structflow extraction only displays struct fields as variables, hiding all other local variables of a scope.

Our struct visualization avoids this weakness by visualizing the structs as arrays. At the end of a function operating on a struct, our implementation outputs the latter in form of a variable. Therefore, it is not noticeable which fields of the variable were changed without retracing the whole usage of the struct in the function's actor. The structflow extraction does not share this problem since it handles all fields as single variables. For example, the function `_sonoNccfProcessCands` in Figure 3.2 calls the function `sonoNccfPrepare`, passing it the fields `buffer`, `energyBuffer`, `meanBuffer`. Subsequently, `sonoNccfPrepare` outputs those variables since it updates their values.

Similarly, the mentioned function `_sonoNccfProcessCands` is visualized using our struct visualization in Figure 3.3. At the end of the function, `self` is part of the output variables. As described, it is not directly visible which of its fields have been updated at this point of the function. In order to deduce this, one would need to retrace the utilization of `self` in the function. It can be noticed that the generated diagram is considerably wider than the diagram generated by the structflow extraction. This has several reasons: On the one hand, C to dataflow did not support arrays and some ports were not wired despite having a dataflow dependence in the version that was used for creating Figure 3.2. On the other hand, most programs have linear dataflow through their methods and functions. The layout in each graphic was automatically created by the Layered Layout of Eclipse Layout Kernel (ELK). It creates a layout with directed flow, default left to right, that assigns each actor to a layer. Applied on a chain of method calls, each node of the chain needs to be assigned to a different subsequent layer. This is the reason why linear dataflow in programs leads to wide diagrams.

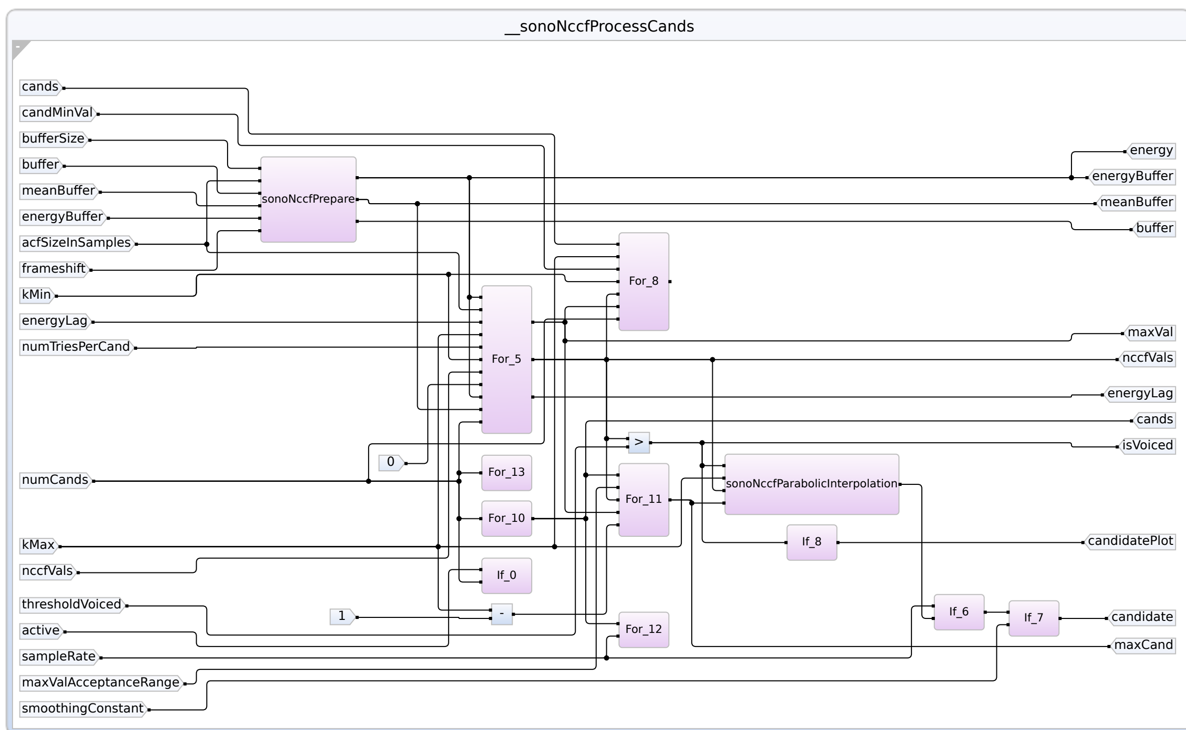
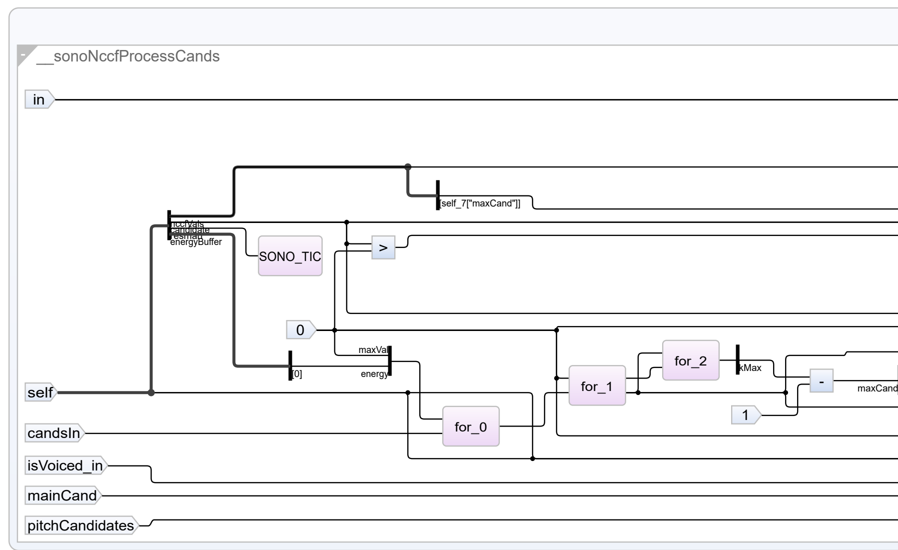
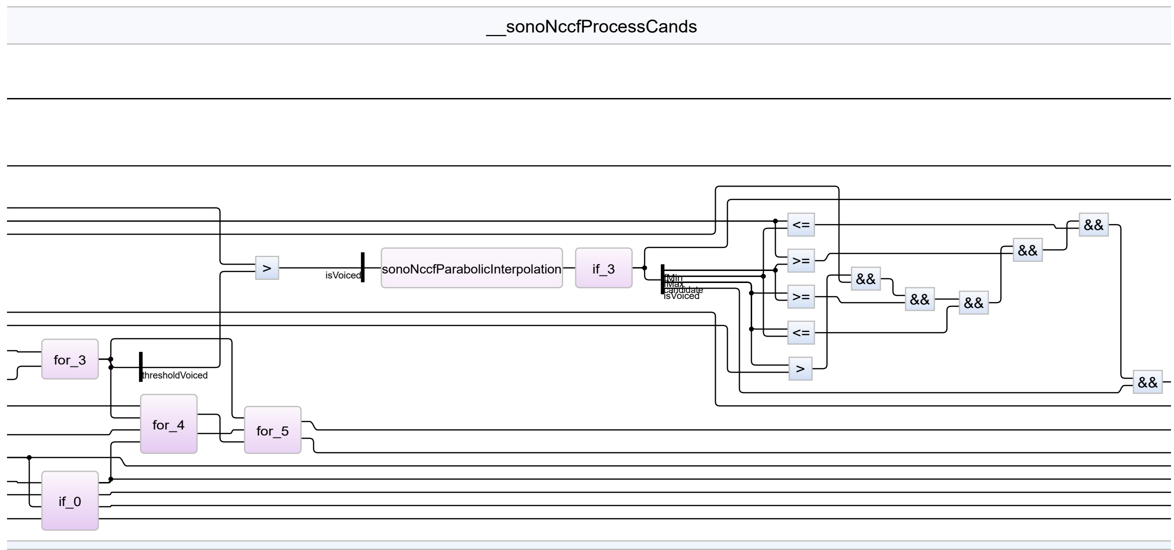


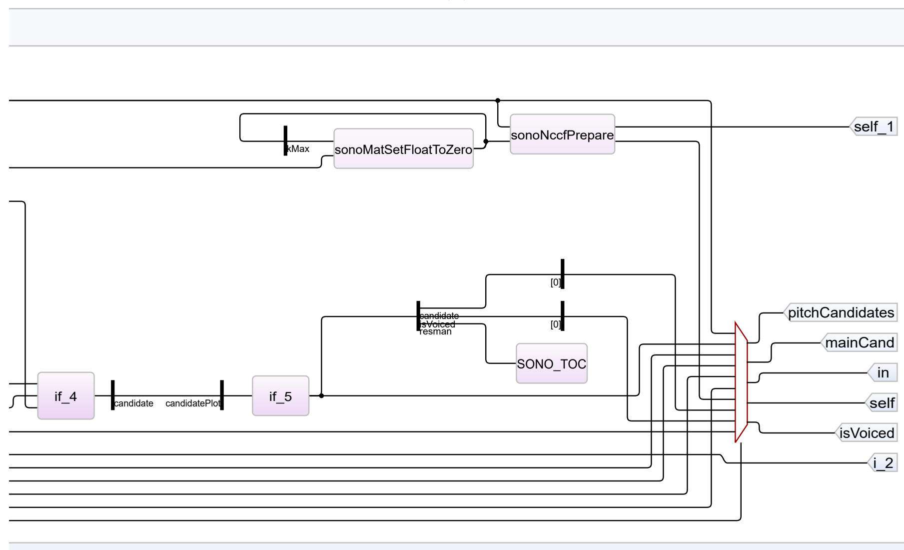
Figure 3.2. A function using a struct visualized by the structflow extraction of Andersen [And19]. Graphics taken from [RSA+21]. The function originates from code of [Reu19].



(a)



(b)



(c)

Figure 3.3. A function using the struct visualization from Section 2.8. The function originates from code of [Reu19].

Chapter 4

Future Work

This project considerably extends the support of C to dataflow. Nonetheless, there are still a lot of possible improvements and some features of C that are not supported in its current state.

Section 4.1 outlines further improvements for loops such as transferring the changes for `while` loops to `do-while` and `for` loops. Arrays will probably need more attention in the future of C to dataflow, the reasons for this are explained in Section 4.2. Preprocessor macros are yet to be supported by C to dataflow like Section 4.3 states. Subsequently, Section 4.4 examines the problems of automatic inlining recursive functions. Section 4.5 describes some open problems of pointers. A set of open problems is collected in Section 4.6 and Section 4.7 covers the open problem of multi-file support. Eventually, C to dataflow could be expanded to C++ to dataflow. This idea is explored by Section 4.8.

4.1 Further Loop Improvements

In the current state of C to dataflow `while` loops have received additional features. This includes conditionals that are mapped to actor-based dataflow and multiplexers that fuse the formerly used binary multiplexers, see Section 2.3. These improvements could and probably should be transferred to `do-while` and `for` loops.

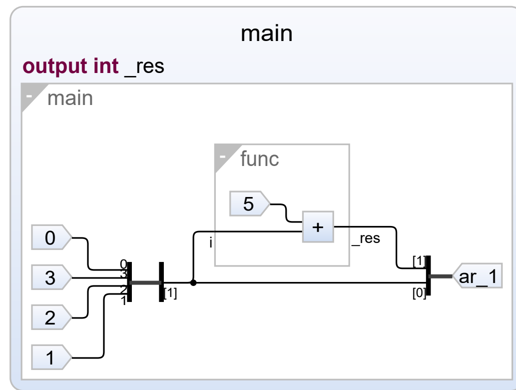
Additionally, it could be worthwhile to conceptualize how to visualize that loops repeat themselves. Currently, loops and `if` statements are only differentiable by looking at their labels. Such a change would make loops better distinguishable from `if` statements. One way could be to create an own styling for loop actors that includes a symbol somewhere in the actor or dataflow region symbolizing the repetition. However, such changes threaten to violate Aim 2 by adding additional visual elements to the diagram. In order to preserve Aim 2, one needs to find a solution that is as non-invasive as possible. One easy way could be to utilize different colouring for this. Following this idea, the contour of actors for `if` statements could keep the standard colour, the contour of loops could be coloured differently.

4.2 Better Array Visualization

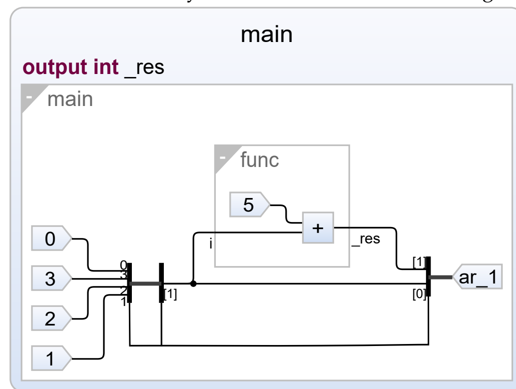
C to dataflow uses the unchanged array visualization of SCCharts. However, this visualization has some flaws. Applying a binary operation on the same array element gives the binary operations only one input wire. Another label bug occurs when using the multidimensional arrays of SCCharts: If one uses accessed elements from multidimensional arrays, for example in binary operations, then the label of the operation only shows the name of the arrays. Thus, it is not easily deducible on which data the operation is applied.

A fundamental problem is that the current array visualization does not visualize situations well in which values are read from and written to the same array. In fact, it does not differentiate whether the same array or different arrays are accessed and updated. Such details currently need to be deduced by

the users by looking at the labels. One way in improving this could be to wire the "lanes" for read and write operations if they are performed on the same array. This is depicted in Figure 4.1. It shows several consecutive operations on the integer array `ar`. As proposed, an additional wire connects the array access lanes of `ar` to symbolize that the operations work on the same array in Figure 4.1b. The current visualization without this improvement is shown by Figure 4.1a.



(a) The current array visualization leads to this diagram.



(b) Wiring the array lanes is an option to imply identity.

Figure 4.1. Proposed improvement for read and write operations on the same array.

In imperative languages array subscripts accept arbitrary expressions. However, in the current state all array subscript expressions are visualized as labels. This contradicts our Aim 1 to map all source code of a C project to actor-based dataflow diagrams. Nonetheless, it is an open question where to locate such translations of array subscript expressions without creating clutter.

When the array visualization is used to visualize structs, unions, and enums, a different styling could improve the readability of the diagram. For example, the array visualization could have a different color for structs, unions, and enums.

4.3 Support for Preprocessor Macros

Until now preprocessor macros have been neglected in C to dataflow. However, they are an often used feature of C in open source projects such as xfwm4[1], the window manager of xfce. Simple macros that only replace markers by literals or insert functions could be supported by introducing them silently to the diagrams. Should a macro be not resolvable, this could be handled by introducing valued object bearing the name of the macro. Then users could easily recognize which macros could not be resolved. Furthermore, it is not obvious how to handle case dependant macros.

4.4 Recursive Functions Break the Automatic Inlining

If recursive functions are present, then automatic inlining breaks since it runs into an infinite loop while trying to visualize all states. In other words, the automatic inlining needs to be aware when a function calls itself recursively and should not expand the state of the called function. Instead it should inline an empty function state bearing the same name as the function, similar to how unknown functions are handled. A more complex case would be to recognize alternating recursion. One other way of dealing with recursion could be dynamic loading of the next recursion levels depending on the current zoom level as a part of advanced browsing support, also see Section 4.7.

4.5 Pointer Improvements

It is not obvious how pointer arithmetic should be handled from a dataflow orientated view. While it makes sense to visualize the operations on the pointers, addresses are often times not interpretable without context.

Currently, we do not check whether a pointer is manually set to an arbitrary address after pointing to a variable. This should be checked and looked into. Furthermore, function pointers need to be implemented. They probably need their own data structures to remember to which function state a pointer refers. In addition, void pointers do not work entirely. They are shown without a type in the diagram and casting a pointer to a void pointer seems to cause problems. While testing the new implementations, see Chapter 3, we encountered that open source projects tend to use pointers to fields of structs. This is not yet supported in our implementation but does not result into a failure to display a diagram.

In order to be able to differentiate pointers and other variables in the dataflow visualization, pointers could be visualized slightly differently such as it is already done for arrays.

4.6 Further Improvements

There are several problems for C to dataflow, not belonging to a bigger topic.

When using `switch` statements, return statements do not work. Multiplexers should receive a style that fits better to SCCharts. Furthermore, the two input port groups of the multiplexer should have some distance between them. This could be done by implementing a property for grouping ports in the ELK. Creating such a property would allow ELK to incorporate groups of ports in the diagram's layout. Consequently, this should be a better solution than manually grouping the ports, which could be done by fixing the position of ports in the synthesis. Besides, the labels of multiplexers need further

[1] `https://gitlab.xfce.org/xfce/xfwm4`

improvements since some of them tend to be misplaced. In general, label placement is a problem in some cases. The label of conditional states, for example, tends to be too close to its content.

Currently, C to dataflow supports structs and unions but enums are missing. They should receive support in the future and their visualization could be implemented similarly to structs. Section 2.10 describes features improving the robustness of C to dataflow. However, currently problems and errors are only reported as prints to the console, thus they are not visible for users. Instead visual warnings like the warnings and errors of SCCharts should be used. Left-clicking on a diagram element should normally highlight the code to which it belongs. Furthermore, it needs to be examined whether `typedef` usage works in our implementation. Additionally, casts can lead to problems. It should also be thought about whether and how casts should be visualized in the diagram.

Above all, some investigation needs to be conducted for understanding underlying logic and how to adjust it for the needs of C to dataflow. For example, ports are added to states without demand of them, compare Section 2.2, and fuses input ports, when a variable with the same name is used multiple times as input.

4.7 Multi-File Support and Browsing Support

In the current state, C to dataflow is only able to translate one C file to dataflow diagrams. Being able to visualize whole projects and to browse in them could improve the usability of C to dataflow. However, this is the very moment in which C to dataflow should receive advanced browsing support. The diagrams should automatically filter which parts of the diagrams should be visible at a particular moment. Otherwise C to dataflow is on the verge of becoming unusable either because of very long loading times or because of overwhelming its users with an unbearable amount of diagram elements. This is partly already the case for single C files including a considerably large amount of code, see Chapter 3.

4.8 C++ to Dataflow

The concepts of C to dataflow are meant to be abstract and usable for general imperative programming languages. Therefore, it makes sense to expand C to dataflow to C++ to dataflow in the future. This step would be eased by the fact that C++ is a superset of C. C++ programmes that only contain C features already work with C to dataflow. As a consequence, only C++ specific features need to be implemented for C++ to dataflow.

4.8.1 Object Orientation

Supporting object orientation is interesting since many imperative languages such as Java, C#, Python, and Javascript are object orientated.

For objects it needs to be conceptualized whether their fields should be handled similar to structs or whether they require a different type of visualization, encapsulating all their members. The latter is probably needed since classes also often also serve as a tool for structuring code.

Handling classes as structs respectively as arrays in C++ to dataflow could be obfuscating and would hide important parts of the program. Another point making struct-like visualization unattractive is the differentiation of class and object members. In addition, mapping project classes to classes of SCCharts is probably problematic, similar to implementing structs in Section 2.8. One problem that

would need to be solved for this way of implementing classes is that classes of SCCharts cannot be passed to other SCCharts.

Generics allow imperative languages to define functions and classes with more general types. In C++ generics are supported via templates that are widely used by bigger C++ projects. Consequently, they are a must-have feature for C++ to dataflow.

Other dataflow solutions already provide support for classes. The functional dataflow orientated language Prograph [MP85] organizes classes in an own classes window. In it each class is symbolized by a hexagonal icon and relations between parent and child classes are indicated by edges between them. Two further symbols included in the class icon represent that each class can have attributes and methods. Attributes and methods are again visualized in own windows [CGP89].

The tool VISSION merges data flow modelling with object orientated modelling in single abstraction that Telea calls metaclasses. Metaclasses extend C++ classes with dataflow semantics by adding a dataflow interface to them. This interface includes inputs, outputs, and an update method. The mentioned elements of a metaclass are delegated to the interface of its C++ class. Metaclasses are represented by an own icon that is an rectangle. Input ports are on its north side, output ports on its south side. Each instance of a metaclass is named following the pattern `<metaclassName>.<instanceName>`. In general, Metaclasses are visualized in a network editor window in VISSION [Tel99].

4.8.2 Anti-Pointer Features

In the course of its development the developers of C++ have recognized that the usage of pointers, storage allocation, and pointer arithmetic is not trivial. Thus they introduced features such as references and smart pointers to C++ offering similar options while being easier to use. It needs to be examined how to support both important C++ features in C++ to dataflow. References are probably supportable in the same way like our current pointer support, compare Section 2.6, since references always point to variables or data structures.

4.8.3 Further Features and Functional Features

Additionally, it needs to be examined whether important collection-like datatypes such as lists, vectors, and sets should be visualized in an abstract way as objects or receive an own visualization.

In the meantime, C++ and many other imperative programming languages have adopted functional constructs such as lambda expressions. Due to their nature lambda expressions are not always easy to understand and not easy to debug. Having dataflow-diagrams visualizing them could be an appropriate aid. Exceeding the scope of this documentation, there are probably many more features and details that need to be considered for extending C to dataflow to C++ to dataflow.

Chapter 5

Conclusion

Visual models are seen as valuable means of documentation for textual programs. Usually such models lift the represented code to a higher level of abstraction and profit from the fact that humans percept and process visual elements differently than text. As mentioned, C to dataflow is not meant to replace, but to augment code and textual documentation. Consequently, this approach combines the advantages of textual and visual documentation.

The C language contains lots of language constructs that need to be supported. Anderson [And19], Rentz [RSA+21], and we added support for a subset of them but there are still various unsupported constructs left.

When deciding on the visualization of a language construct, it is difficult to find a compromise between Aim 1 (complete mapping) and Aim 2 (readability). Whether the chosen visualizations in this project are helpful still needs to be evaluated in a survey. In order to test the features in large C projects, it would be helpful to support even more features such as macros since they are used quite often, and in the best case C to dataflow should provide smart zooming.

Bibliography

[And19] Lewe Andersen. "Dataflow and state machine extraction from C/C++ code". Master's Thesis. Universität zu Kiel, Dec. 2019. 96 pp.

[CGP89] P.T. Cox, F.R. Giles, and T. Pietrzykowski. "Prograph: a step towards liberating programming from textual conditioning". In: *[Proceedings] 1989 IEEE Workshop on Visual Languages*. 1989, pp. 150–156. DOI: 10.1109/WVL.1989.77057.

[LNW03] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. "Actor-oriented design of embedded hardware and software systems". In: *Journal of Circuits, Systems, and Computers* 12 (2003), pp. 231–260.

[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.

[MP85] S. Matwin and T. Pietrzykowski. "Prograph: a preliminary report". In: *Computer Languages* 10.2 (1985), pp. 91–126. ISSN: 0096-0551. DOI: https://doi.org/10.1016/0096-0551(85)90002-5.

[Par11] David Lorge Parnas. "Precise documentation: the key to better software". In: *The Future of Software Engineering*. Ed. by Sebastian Nanz. Berlin, Heidelberg: Springer, 2011, pp. 125–148. ISBN: 978-3-642-15187-3. DOI: 10.1007/978-3-642-15187-3_8.

[Reu19] Janina Reuter. "Real-time pitch tracking algorithms in C to test model extraction". Bachelor's Thesis. Universität zu Kiel, Sept. 2019. 106 pp.

[RSA+21] Niklas Rentz, Steven Smyth, Lewe Andersen, and Reinhard von Hanxleden. "Extracting interactive actor-based dataflow models from legacy C code". In: *Lecture Notes in Artificial Intelligence (LNAI)*. Diagrams: 12th International Conference on the Theory and Application of Diagrams. Vol. 12909. 2021.

[Tel99] A. Telea. "Combining object orientation and dataflow modelling in the vission simulation system". In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No.PR00275)*. 1999, pp. 56–65. DOI: 10.1109/TOOLS.1999.778999.

Abbreviations

ELK	Eclipse Layout Kernel
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
SCCharts	Sequentially Constructive Statecharts