Extracting Interactive Actor-Based Dataflow Models from Legacy C Code*

Niklas Rentz[1][0000−0001−6351−5413], Steven Smyth[1][0000−0003−2470−0880], Lewe Andersen[2], and Reinhard von Hanxleden[1][0000−0001−5691−1215]

[1] Department of Computer Science, Kiel University, Kiel, Germany
{nre,ssm,rvh}@uni-kiel.de
[2] Scheidt & Bachmann System Technik GmbH, Melsdorf, Germany
Andersen.Lewe@scheidt-bachmann-st.de

Abstract. Graphical actor-based models provide an abstract overview of the flow of data in a system. They are well-established for the model-driven engineering (MDE) of complex software systems and are supported by numerous commercial and academic tools, such as Simulink, LabVIEW or Ptolemy. In MDE, engineers concentrate on constructing and simulating such models, before application code (or at least a large fraction thereof) is synthesized automatically. However, a significant fraction of today's legacy system has been coded directly, often using the C language. High-level models that give a quick, accurate overview of how components interact are often out of date or do not exist. This makes it challenging to maintain or extend legacy software, in particular for new team members.
To address this problem, we here propose to reverse the classic synthesis path of MDE and to synthesize actor-based dataflow models automatically from source code. Here functions in the code get synthesized into nodes that represent actors manipulating data. Second, we propose to harness the *modeling-pragmatic* approach, which considers visual models not as static artefacts, but allows interactive, flexible views that also link back to textual descriptions. Thus we propose to synthesize actor models that can vary in level of detail and that allow navigation in the source code. To validate and evaluate our proposals, we implemented these concepts for C analysis in the open source, Eclipse-based KIELER project and conducted a small survey.

Keywords: Actor-Based Dataflow · Program Comprehension · Interactive Documentation.

1 Introduction

Precise and up to date documentation is a key aspect of quality maintenance of software systems [4, 19]. Good documentation does not only help the developer

* This work has been supported by the project Visible Code, a cooperation between Kiel University and Scheidt & Bachmann System Technik GmbH.

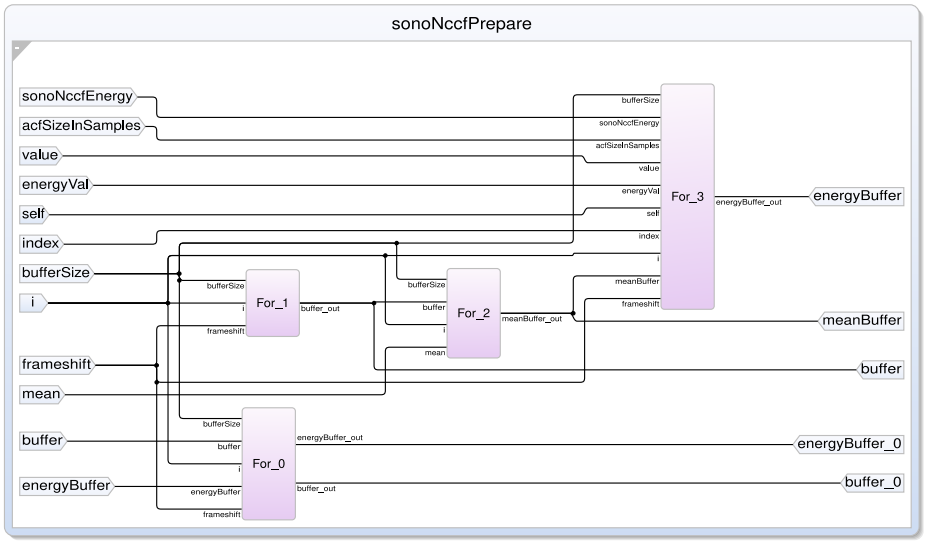


Fig. 1: An example actor model, illustrating the data flow in a signal processing component named sonoNccfPrepare.

to keep a better overview about the project, but also enables users to gain a perception about the usage, functionality, and connections inside the project.

Regardless of the advantages of a well-documented project, the documentation for many projects is outdated, as stated by different surveys, e.g., by Lethbridge et al. [14] and Singer [24]. Singer states that even if most developers appreciate good documentation, the time needed for its creation or maintenance leads to inconsistency, which further leads to a lack of trust from the developers.

We here propose to enhance the documentation of existing codebases by the usage of diagrammatic *actor-based dataflow models*, or actor models for short. Actor models are already commonly used in model-driven engineering and are supported by numerous commercial and academic tools, such as Simulink, LabVIEW or Ptolemy. Lee et al. [13] describe *actors* as components that can execute and communicate with other actors in a model. Their *ports* represent an interface for communication with other actors and their environment. For example, Figure 1 shows an actor named sonoNccfPrepare, with input ports sonoNccfEnergy, acfSizeInSamples, etc., and output ports energyBuffer, meanBuffer, etc. That actor includes other actors; For_1, for example, receives the inputs bufferSize, i, and frameshift from the environment, and provides output buffer to actor For_2.

Unlike the MDE setting, where developers create visual (actor) models and textual code is synthesized from the models, we here propose to reverse the synthesis path and to automatically create models from existing code. We thus propose to extract documentation directly and automatically from the source, which has been shown to be helpful and wanted by developers [6]. Rugaber [21] states that program comprehension is the biggest bottleneck in development

time-wise and is mainly done manually, so automation and simplifications in the comprehension can cut down on that bottleneck. A key benefit of visual code comprehension tools is that users are not burdened anymore with the manual creation and maintenance of such visualizations, and that such visualizations are more up to date. This is not meant to completely replace manual documentation, as documentation extracted from code can only be a description and not define a specification or the thoughts that went into design decisions, as discussed by Parnas [19]. The generated visualizations proposed here are meant to complement and structure other documentation.

Contributions & Outline

- We propose an approach to automatically generate actor models from C programs, where the actors match the program structure and their interconnections reflect the data flow (Section 2).
- We have prototyped and integrated this model extraction and visualization in an open source, Eclipse-based modeling environment, which allows flexible diagram views that link back to the source code (Section 3).
- We have conducted a small user experiment that, for a given set of tasks, compares the effectiveness of source code analysis vs. the inspection of visual models (Section 4).

Section 5 discusses related work, we conclude in Section 6.

2 Actor-Based Dataflow Visualization

Most developers define the behavior of a program with imperative programming languages [15]. In programs split up into many different functions it may be unclear where data in functions come from and where they are used. Most commonly used IDEs provide some support for tracing the data, through highlighting or function usage trees, but in general they do not give an overview of the intraprocedural dataflow. We propose an actor-based dataflow view, akin to Ptolemy [5] and SCCharts dataflow [29]. This section describes how to visualize such a dataflow model for imperative programming languages. We illustrate this with the example of C code.

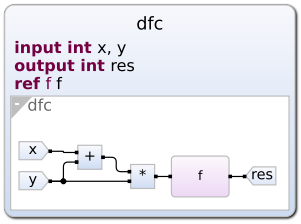
2.1 Actor-Based Dataflow

The example actor model shown in Figure 2b, which represents the code shown in Figure 2a, shows a possible mapping from elements of an imperative language to dataflow elements. We show this for the C language, the principles, however, are applicable to other higher-level imperative and functional languages as well. As they might use more complex constructs such as classes, generics, etc., these points have to be addressed in future work. The actor surrounding the dataflow model may have some declarations to define variables used as in-

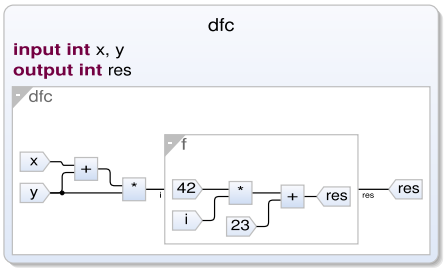
```
int f(int i) {
  return 42 * i + 23;
}

int dfc(int x, int y) {
  x = x + y;
  int multSum = x * y;
  int out = f(multSum);
  return out;
}
```

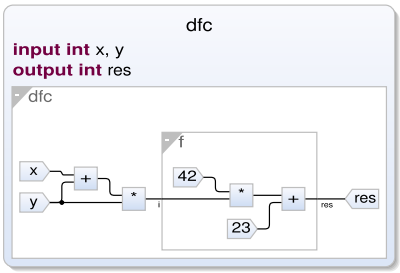
(a) Source code



(b) Actor model with collapsed actor `f`



(c) Actor model with expanded actor `f`



(d) Actor model with inlined actor `f`

Fig. 2: Automatically generated dataflow views.

or outputs for the main *dataflow region*. These declarations show the interface of the actor regarding the data it reads from and gives back to the environment. This is also visible in the dataflow region, as the in- and outputs of the actor are visualized as named flags on the left or the right of the dataflow, respectively.

The dataflow region itself visualizes a collection of *assignments*. All assignments connect their inputs to their outputs through *simple actors*, which are the basic operations available for numbers, such as $+$, $-$, $*$, etc., or *complex actors* with inner behavior, such as the actor `f`. We also make use of configurable *views*, to show the dataflow only at top-level, more detailed by expanding any complex actor, or completely inlined with connected interface, see Figures 2b, 2c and 2d.

2.2 Constructing Actor Models

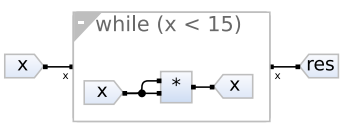
For the translation of the source code into an actor model, the Abstract Syntax Tree (AST) is analyzed and a view is presented based on its constructs. The following explains how we propose to visualize the different language constructs.

Assignment Statements To start with the basics, each assignment of an expression is represented by the connection of its simple or complex actors via an edge or *wire* in the view. These wires can connect to further assignments or other uses in complex actors.

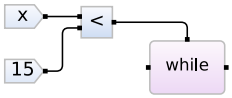
Compound Statements The translation of a compound statement is the core of the dataflow extraction. It represents the body of a function, but it is also

```
int main(int x) {
  while (x < 15) {
    x = x * x;
  }
  return x;
}
```

(a) Code



(b) While actor with textual representation of control expression, with expanded body and with context



(c) While actor with dataflow representation of control expression, with collapsed body and without context

Fig. 3: Alternative actor views of a `while` statement.

used to represent the body of control statements such as the `if` statement. To translate this into dataflow, all statements within are distinguished as detailed in this section and visually added to a dataflow region.

Function Definitions Figure 2a shows the definitions of the functions `f` and `dfc`. The representation of the function `dfc` in actor-based dataflow is shown in Figure 2b. It shows the function as an actor which can be referenced by other actors to represent function calls. The parameters are represented by variable declarations. Furthermore, the actor declares an output variable matching the return type of the function, shown with the name `res`. The compound statement of the function is then represented by the dataflow region of the actor.

Function Calls In Figure 2a, `out = f(multSum)` is an assignment statement that makes a function call. A complex actor is created that references the actor for the called function. The parameters of the function and its return value are linked to the corresponding variables in the dataflow region. This results in the diagrams shown in Figures 2b to 2d. If the called function is also defined in the given source code—and is not for example a library function—the referenced actor is expandable to show the behavior of the called function. This way, the resulting diagram can be navigated without showing every detail from the beginning, so that the users can choose by themselves which details they want to see.

While and Do-While Statements Control statements, such as loops, have no direct representation in classical dataflow views. However, they are an important part of the program structure that we want to preserve in our visualization. We therefore propose to represent each control flow statement as a complex actor referenced in the dataflow region, as it was shown for function calls. That complex actor shows control flow in an abstract way or in a state machine fashion to keep the main focus on the dataflow while giving the control flow a natural counterpart. We explain the loop representations in detail for the `while` statement only, as `do-while` and `for` statements are similar.

A `while` loop contains a conditional expression that controls how often the loop is repeated, and the loop body that represents its behavior. An example

`while` loop in C code is presented in Figure 3a. Figure 3b shows a corresponding actor, as currently implemented in our tooling. The loop body is translated with the rules for compound statements. Additionally, this region has a label to represent the control expression in plain text.

Alternatively, as illustrated in Figure 3c, one might choose to not show the control expression textually but to also synthesize it including any computation and side effects into actors, with the result connected to a port of the `while` actor. We have not implemented this option yet since it might lead to many additional elements in the resulting graphic. However, for future extensions it might be worth considering to offer this alternative visualization as an option.

The definition of the in- and outputs of the `while` actor is done by searching for any variable defined outside of the `while` loop that is read or modified inside the loop. In the resulting dataflow region, each read variable is connected with the inputs, and each written variable is connected with the outputs of the actor, as in Figure 3b.

For Statements The translation of the `for` statement is very similar to the other loop statements. The difference is that it does not only contain one expression for the condition. The `for` statement contains two more expressions for an initialization and an update of the loop, also shown as text.

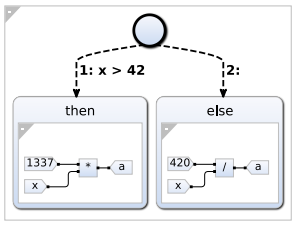
If and Switch Statements These control flow statements are again visualized as complex actors with the analysis of read and written variables. The control flow can be modeled and displayed as simple state machines. This is different to other approaches, as discussed in Section 5, since we use state chart visuals to represent the branching control flow that is not directly translatable into traditional dataflow views. Keeping this as a dataflow-only view with combination actors similar to multiplexers would be another viable approach allowing for non-trivial control expressions that we have not implemented yet.

The `if` statement shown in Figure 4a can be translated as shown in Figure 4b, with a branching initial state that hands the control either to the `then` or the `else` branch, depending on whether the condition expression results as `true` or `false`. The compound statements of both branches are then put into the dataflow regions of their respective states as described above. The numbers next to the transitions to the branches indicate their priority, where the lower number stands for a higher execution priority. So the `then` transition with the higher priority 1 will only be taken if the condition is true, otherwise the always-true transition with the lower priority 2 is taken, matching the semantics of the `if` statement. The connection of this actor to the outside is, as for all complex actors, via the ports for all in- and outputs as shown before and omitted here.

The `switch` statement follows a very similar strategy, as it corresponds to multiple chained up `if` statements. The code and resulting visualization are shown in Figure 5. Each `case` statement is equivalent to an equality check of the variable in the `switch` with the value in the `case` statement. The translation to the state chart visualization therefore is as for the `if` translation, where now

```
int main(int x) {
  int a;
  if (x > 42) {
    a = 1337 * x;
  } else {
    a = 420 / x;
  }
  return a;
}
```

(a) Code

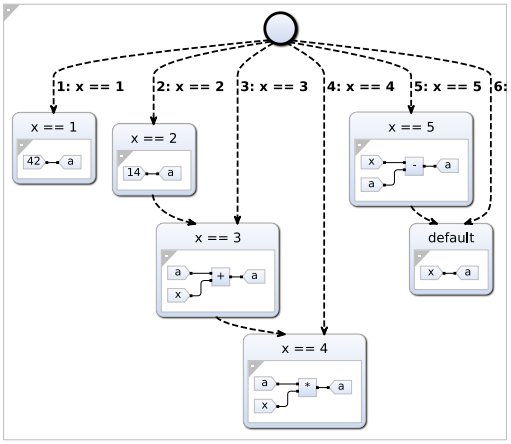


(b) View

Fig. 4: The actor representation of an `if` statement.

```
int main(int x) {
  int a = 0;
  switch (x) {
    case 1:
      a = 42;
      break;
    case 2:
      a = 14;
    case 3:
      a = a + x;
    case 4:
      a = a * x;
      break;
    case 5:
      a = x - a;
    default:
      a = x;
  }
  return a;
}
```

(a) Code



(b) View

Fig. 5: The actor representation of a `switch` statement.

instead of a single `then` branch a conditional transition to a `case` state is added for each `case` statement with decreasing priority. The `else` branch is equal to the `default` case, as it will be the transition with the lowest priority and no condition attached to it. Additionally, every `case` gets an outgoing transition into the `case` state with the next lower priority if it is missing the final `break` statement, as the control flow will continue into the next `case` block in that case.

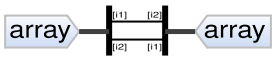
2.3 Data Types

Many use cases are already covered when using primitive data types such as `int`, `float`, and so on. These represent single values and are shown to be held by wires in the view. Other data types, however, are also widely used in C and other imperative languages. Their representations are presented here.

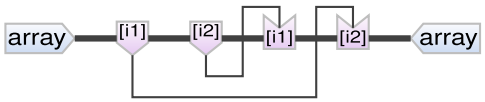
Arrays Arrays are used in most languages, and we propose two visual counterparts. A code example swapping two array indices with their possible views are

```
void swap(int array[],
  int i1, int i2) {
  int temp = array[i1];
  array[i1] = array[i2];
  array[i2] = temp;
}
```

(a) Source code



(b) Dataflow view dividing the array



(c) Dataflow view with actors for modification

Fig. 6: Possible representations for arrays, which could also be used for structs.

shown in Figure 6a. In the first proposed view in Figure 6b, which we have opted for in our prototype, the thicker wire representing the array is divided and split into further wires for each read operation, labeled with the read index, and then combined back into the thicker array wire for each write operation, labeled with the written index. This view is compact and shows all array accesses in a fashion where only the data is relevant. Another possible view, not implemented yet but illustrated in Figure 6c, has a continuous thicker array wire and visualizes every array access as an own actor. This representation may be less tidy, due to additional edge crossings, and has a strict dependency on the order of operations on the array, even if they may be interchangeable in the dataflow sense.

Structured Data Structured data such as in `structs` in C or classes in other languages can be visualized similar to arrays, where the thick wire represents said structured data and its field accesses map to the index accesses in Figure 6.

Pointers Pointers to structured data and pointers in general can be visualized as wires as described above, where the wires carry the data that is pointed to. However, this only applies to pointers which are not modified. Pointer arithmetic can be used in C, though that disconnects the pointer variable from the data it points to and cannot be visualized easily with the concepts shown here anymore, as it is may depend on actual runtime data and cannot be analyzed statically as described.

Structs A common practice to pass data around among functions is to collect these as structured data such as a `struct` in C and to pass a pointer to that data to the functions. We propose an alternative way to show each field of a central `struct` like individual variables together with the usual variables to flow through the algorithm. An example of a dataflow graphic using this *struct flow abstraction*, based on signal processing code provided by an industrial partner, can be seen in Figure 9.

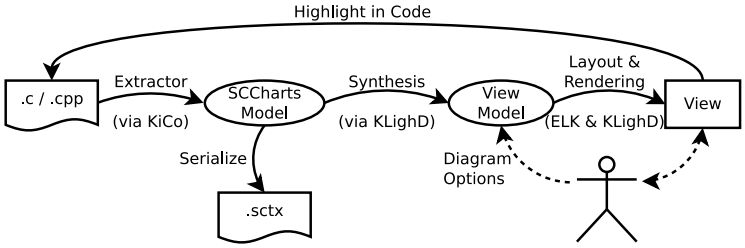


Fig. 7: The workflow for visualizations in KIELER.

3 Implementation and Validation

For the extraction and visualization of dataflow from source code we make use of pre-existing technologies for model-based design from the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. The typical workflow and user interaction in the Eclipse-based tool explained in this section is presented in Figure 7 and described below. A key aspect is the separation of *model* and customizable *views*, which serve as abstract documentation and navigation aid, as advocated in *modeling pragmatics* [7].

KIELER uses the model-based framework KIELER Compiler (KiCo) [26] to create compiler chains to and from modeling languages, executable code, visual models, or intermediate models in multiple configurable steps, so-called *processors*. We use this compiler framework to build a new compiler chain compiling from source code, in this case C code, to the visual modeling language SCCharts. Our synthesis chain can also parse C++ code, and can, e.g., synthesize state machines from a state pattern based on C++ templates, as discussed further in Andersen's thesis [2]. We use the C/C++ Development Tooling (CDT) for parsing the source code and a novel extraction to generate SCCharts models from the parsed code.[3]

The Sequentially Constructive StateCharts (SCCharts) language presented by von Hanxleden et al. [9] provides determinate concurrency using a graphical statechart notation and also supports the use of dataflow. We use it as a modeling language for its support of modeling dataflow combined with its possibility to create configurable and interactive visuals using KLighD.

The KIELER Lightweight Diagrams (KLighD) framework, as presented by Schneider et al. [22], generates interactive views from arbitrary models using an abstract *view model* to describe node-link diagrams. As SCCharts was designed as a visual language using KLighD to automatically visualize modeled instances, the use of SCCharts as our basis creates a shortcut to generating views. This provides filtering and configurability of the view out of the box that are useful for our use case.

[3] The code for the extraction is available at https://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/semantics/browse/plugins/de.cau.cs.kieler.c.sccharts?at=refs%2Ftags%2Fdiagrams21

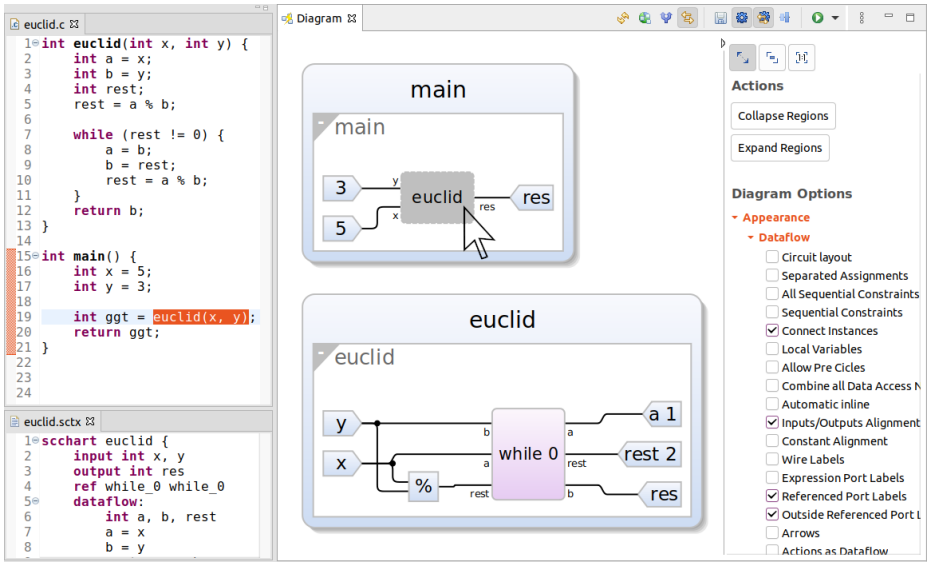


Fig. 8: Automatic highlighting from a dataflow view to the source code and interaction in the tool.

All figures shown of dataflow use the KIELER infrastructure with SCCharts. Here we use the visual aspect of the language, ignoring the semantics of the language itself. To provide more than static diagrams, this infrastructure allows the user to browse the models freely and interactively. The user can expand and collapse regions and actors, such as shown in Figures 2b and 2c. One may also use *diagram options* to show or hide labels for the inputs and outputs of referenced complex actors, as also shown in Figures 2b and 2c, and other visual options. Fuhrmann and von Hanxleden [7] presented how view management is conceptualized and implemented in KIELER and can be used for SCCharts.

Furthermore, when having the extracted diagram and the source code of an algorithm next to each other, we allow the user to interactively trace the origin of diagram elements. For example, clicking on a complex function call actor leads the user directly to the function call expression in the source code for investigating the diagram and the source code itself in parallel. An impression on how this looks in the KIELER environment is shown in Figure 8.

Figure 7 visualizes how to combine these modes of interaction with the tool to get from the source code to a visualization and other artifacts. KIELER employs the Eclipse Layout Kernel (ELK) and KLighD to compute concrete views from the view models. One aspect not to be underestimated when using visual models is how the proper usage of *secondary notation* affects readability [20]. To that end, we employ the *layer-based layout* computed by ELK [23], which facilitates a natural left-to-right reading direction; for dataflow diagrams, this means that information flows from inputs to outputs.

Figure 8 also shows the user interface in KIELER. There are two open editors, one with the base C source code and one with the extracted and serialized SCChart model, together with the view of that SCChart model with tools for interactivity, such as a side bar for the diagram options, and the possibility to highlight and navigate to elements from the view back to the source code.

For practical validation, we implemented a prototype following the concepts as described in the previous sections. The dataflow view does not fully support the pointers as discussed in Section 2.3 yet, the basic functionality for passing pointers and writing to them in function calls, however, is already implemented. The struct flow abstraction is used as a proof of concept of the view for `structs`. In it, we require the algorithm to be separated into different functions that all may manipulate one central struct. If they manipulate the struct, it is expected that each of these functions take it as the first parameter. Finally, as the struct flow abstraction does not support arrays as the default dataflow view does, those elements are not wired up in Figure 9 used for the survey, leaving the complex actor named `For_8` unconnected in the view. Having full support for structs as described above in our main dataflow view and adding that to the visual model will be part of future work. This will also allow us to present the tool with larger programs than the examples used here and in part improve the scalability and understandability. For example, Figure 9 relies heavily on the use of structs and would get clearer.

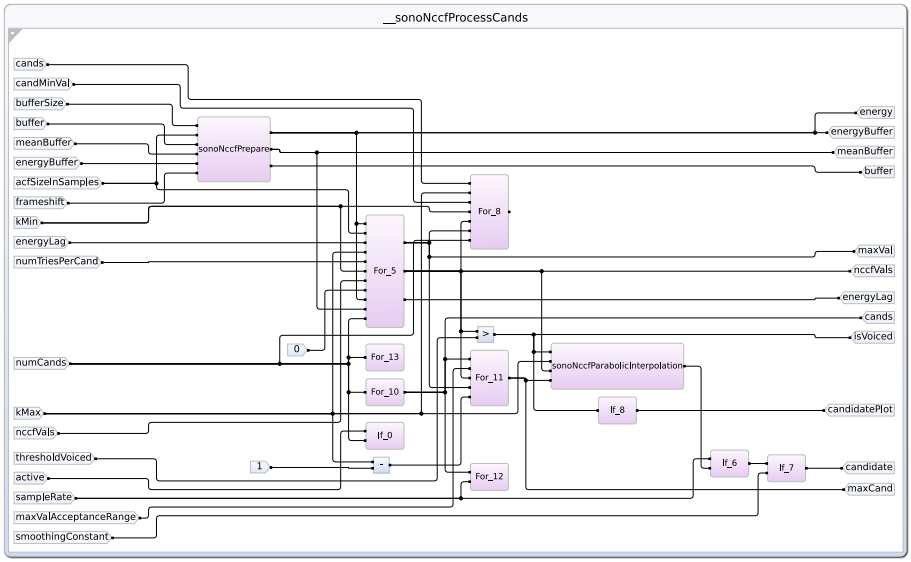


Fig. 9: Top level struct flow view filtered to display the main struct. Hierarchies are all collapsed to show a first overview.

Table 1: Survey results for the tasks.

Task	Code	Diagram	p-value
Used `for` statements	13.6 s	3 s	0.020
Read struct elements	23.7 s	5.6 s	0.039
Last write	22.4 s	7.6 s	0.054
Written elements	125.9 s	5.1 s	< 0.001

4 Survey

As a first assessment of the benefit of the resulting dataflow diagrams, we conducted a preliminary survey with six PhD students from the working group. The participant group was split in half, with each group being asked to complete the same tasks regarding the dataflow example in Figure 9, utilizing only the core concepts. Half of the participants got to view the source code in an Eclipse IDE with the installed CDT for the tasks for the dataflow example. The other group had the same tasks, using the extracted diagram in the interactive view in KIELER. All participants had previous knowledge of the C programming language, the usage of the Eclipse IDE, and the diagram view with SCCharts, thus mitigating any skill-based bias in the results. They were allowed to use all features of the IDE and the interactive diagram freely to simulate real workflow.

The *first task* was to identify all `for` statements that were used by the main function of the program. The *second task* asked for the number of elements of the main struct that were used by the `sonoNccfParabolicInterpolation` function. In the *third task* the participants were asked to identify the function that does the last write to the `buffer` element of the central struct. Finally, the *fourth task* was to identify all elements of the central struct that are written to during the calculation of the main function. We note that the dataflow graphic used the struct flow abstraction and thus was already filtered to only show the flow of the struct elements alone, so other local variables were not shown.

To compare the efficiency of the tasks in both scenarios, the time the participants needed for the completion was measured. Everything was already set up for them, so the measured time only spans from their first look at the diagram or code until they gave the answer. We then took the times between the participants and did a two-sample *t-test* assuming equal variances on the logarithm of the measured times for testing the significance of better results using the diagram. The logarithmic mean times and the p-value of equal means are presented in Table 1. The results show a significant increase in the ability to quickly solve these tasks. The p-values indicate to reject the hypothesis of equal means at confidence intervals of over 90% each, meaning that completing these specific tasks is most likely faster. In this sample all of these tasks were solvable on average more than three times as fast when using the extracted diagram than without, while almost all tasks were solved correctly. Only in the fourth task none of the participants was able to give the correct answer given the code alone and one participant even gave up and stated that there was no way to solve the task in

the two minutes that were suggested as a soft time limit. They said that they needed at least a piece of paper to take notes and that the IDE's functionality to highlight the name of the central struct was essential to make it even possible.

This indicates that the generated diagrams will make these tasks quicker and easier to solve than with the code and usual IDE tooling alone. Surely, these tasks are constructed in such a way that they should be solvable faster with the diagram, for example showing what was asked for in the questions on a single screen in the first task compared to needing to scroll through the code, but that confirms the assumption that a diagram like this can present information in a more accessible way than typical IDE features alone, and that can be harnessed by using the extraction and interactivity presented in this paper. Furthermore, the small sample size, the current state of the tool, and the tailored questions only allow for preliminary study results that cannot be generalized and need to be validated with a more complete tool. But it is not the focus to replace the code, but rather to provide a tool to help the developer comprehend the code alongside it, so these results already show potential to improve specific tasks.

5 Related Work

Analyzing dataflow is common practice, often used for low level compiler optimizations [10,17]. However, these are typically low-level analyses, and not meant to help program comprehension. To quote Ishio et al. [11], *while developers have to investigate dataflow paths, existing source code viewers focus on method calls as a main relationship.* In this paper, however, we focus on a higher level analysis for the comprehension of developers. Ishio et al. [11] have investigated interprocedural dataflow for Java programs. They propose Variable Data-Flow Graphs (VDFGs) that represent interprocedural dataflow, but abstract from intraprocedural control flow. Unlike our work, their analysis ignores sequential control flow, thus in program fragments like `x = y; y = z;` it (falsely) assumes that `x` depends on `z`. For method calls, they assume that these produce data only via their return value. They provide an interactive viewer, but their interaction consists of selecting specific nodes (e.g., a variable) in the VDFGs, for which then the neighboring nodes are indicated. Our work instead aims for providing high-level overviews of whole program (regions) that can be explored interactively.

Namballa et al. [18] use VHDL and create a control and dataflow graph (CDFG) for the program, which is an integral part during the synthesis from the behavioral specification of a program into an electronic circuit using logic gates. This CDFG described by Amellal and Kaminska [1] focuses on a control flow graph with hints for the variables used in the dataflow. These graphs are not extracted to help programmers comprehend the code but are specific to the hardware synthesis. Furthermore, the graphs focus on showing the concrete control flow, while we focus on displaying the concrete dataflow, while hinting at the control flow for non-linear flow of data.

Beck et al. [3] and Gèvay et al. [8] present methods to execute imperative control flow on dataflow machines and graphical representations for those trans-

lations. The use of their visualizations as a program comprehension tool is restricted as they mainly focus on the execution on parallel dataflow machines. Moreover, they do not describe any interactive features for the diagrams.

There are further tools and frameworks to reverse engineer diagrams from C code. CPP2XMI is such a framework as used by Korshunova et al. [12] for extracting class, sequence, and activity diagrams, or MemBrain for analyzing method bodies as presented by Mihancea [16]. UML class models are extracted from C++ by Sutton and Maletic [27], and another framework for the analysis of object oriented code is presented by Tonella and Potrich [28]. All these tools and frameworks show the importance of reverse engineering and presenting views to programmers, whereas these do not cover dataflow like in our approach.

Commercial tools such as the McCabe tool suite or SCITools' Understand also focus on visualizing code metrics for an easier comprehension, but they do not include such specific means of visualizing intraprocedural dataflow. The DMS Software Reengineering Toolkit is another tool supporting some dataflow analysis framework, that, however, also concentrates on the control flow and the statements themselves, with data dependencies between each statement added to the view, similar to the graphs shown in Namballa et al. [18].

Smyth et al. [25] implemented a generic C code miner for SCCharts. The focus was to create semantically valid models from legacy C code, which can then be compiled to modern code for various platforms. The paper only considers a small subset of C, which is translated into control flow constructs. The work also tried to find appropriate means for visualizing common C patterns in control flow, which still is a difficult question for larger models.

6 Conclusions and Outlook

As argued by others before, visual models may be a valuable documentation of textual programs. Diagrams leverage the human perception capabilities in ways that program text typically does not use, and visual models typically entail some level of abstraction. Thus visual models are—at least here—not meant to replace code, but rather to augment them. In terms of visual documentation of source code, a main novelty presented here is the synthesis of actor-based dataflow diagrams. Not all C language features are fully supported yet our implementation, but in our experience, the dataflow and struct flow visualizations appear to be valuable for the user and to provide an easy way to analyze the flow of data within functions. A first user experiment indicated the advantages of the visual models over the original program text for answering certain questions.

The synthesis of diagrams for code documentation also shows the broad applicability of visual models such as SCCharts. They can be used for programming in the SCCharts language directly, but also for the automatic generation of informative graphics. A compiler framework such as KiCo in the IDE makes the mapping to a visual model easy to complete.

As future work SCCharts and the dataflow extraction could be extended with support for the remaining constructs of the C language to enlarge the set of

programs that can be shown in dataflow in their entirety. With that, evaluating the views and their scalability on production codebases is a next step in verifying this approach. Also, further filtering could be applied to the elements shown in the visualization, adding to the existing filtering possibilities in SCCharts. Finally, formalizing the notation and a more representative survey would be worthwhile future work.

References

1. Amellal, S., Kaminska, B.: Scheduling of a control and data flow graph. In: 1993 IEEE International Symposium on Circuits and Systems. vol. 3, pp. 1666–1669. IEEE (1993)
2. Andersen, L.: Dataflow and Statemachine Extraction from C/C++ Code. Master thesis, Kiel University, Department of Computer Science (Dec 2019), https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lan-mt.pdf
3. Beck, M., Johnson, R., Pingali, K.: From control flow to dataflow. Journal of Parallel and Distributed Computing **12**(2), 118–129 (1991)
4. Cook, C., Visconti, M.: Documentation is important. CrossTalk **7**(11), 26–30 (1994)
5. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE **91**(1), 127–144 (Jan 2003)
6. Forward, A., Lethbridge, T.C.: The relevance of software documentation, tools and technologies: A survey. In: Proceedings of the 2002 ACM Symposium on Document Engineering. pp. 26–33. DocEng '02, Association for Computing Machinery, New York, NY, USA (2002)
7. Fuhrmann, H., von Hanxleden, R.: On the pragmatics of model-based design. In: Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development, Revised Selected Papers. LNCS, vol. 6028, pp. 116–140. Springer, Budapest, Hungary (2010)
8. Gévay, G.E., Rabl, T., Breß, S., Madai-Tahy, L., Markl, V.: Labyrinth: Compiling imperative control flow to parallel dataflows. CoRR (2018), http://arxiv.org/abs/1809.06845
9. von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., O'Brien, O.: SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). pp. 372–383. ACM, Edinburgh, UK (Jun 2014)
10. Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier Science Inc., New York, NY, USA (1977)
11. Ishio, T., Etsuda, S., Inoue, K.: A lightweight visualization of interprocedural data-flow paths for source code reading. In: Beyer, D., van Deursen, A., Godfrey, M.W. (eds.) IEEE 20th International Conference on Program Comprehension (ICPC). pp. 37–46. IEEE, Passau, Germany (Jun 2012)
12. Korshunova, E., Petković, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: Reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In: 13th Working Conference on Reverse Engineering (WCRE'06). pp. 297–298. IEEE Computer Society, Benevento, Italy (Oct 2006)
13. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers (JCSC) **12**(3), 231–260 (2003)

14. Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use documentation: The state of the practice. IEEE Software **20**(6), 35–39 (2003)
15. Meyerovich, L.A., Rabkin, A.S.: Empirical analysis of programming language adoption. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13). pp. 1–18. Association for Computing Machinery, New York, NY, USA (2013)
16. Mihancea, P.F.: Towards a reverse engineering dataflow analysis framework for Java and C++. In: Negru, V., Jebelean, T., Petcu, D., Zaharie, D. (eds.) 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 285–288. IEEE Computer Society, Timisoara, Romania (Sep 2008)
17. Muchnick, S.S., Jones, N.D.: Program flow analysis: Theory and applications. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA (1981)
18. Namballa, R., Ranganathan, N., Ejnioui, A.: Control and data flow graph extraction for high-level synthesis. In: IEEE Computer Society Annual Symposium on VLSI. pp. 187–192. IEEE (2004)
19. Parnas, D.L.: Precise documentation: The key to better software. In: The Future of Software Engineering, pp. 125–148. Springer (2011)
20. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM **38**(6), 33–44 (Jun 1995)
21. Rugaber, S.: Program comprehension. Encyclopedia of Computer Science and Technology **35**(20), 341–368 (1995)
22. Schneider, C., Spönemann, M., von Hanxleden, R.: Just model! – Putting automatic synthesis of node-link-diagrams into practice. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13). pp. 75–82. IEEE, San Jose, CA, USA (Sep 2013)
23. Schulze, C.D., Spönemann, M., von Hanxleden, R.: Drawing layered graphs with port constraints. Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout **25**(2), 89–106 (2014)
24. Singer, J.: Practices of software maintenance. In: Proceedings of the International Conference on Software Maintenance (Cat. No. 98CB36272). pp. 139–145. IEEE (1998)
25. Smyth, S., Lenga, S., von Hanxleden, R.: Model extraction for legacy C programs with SCCharts. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '16), Doctoral Symposium. Electronic Communications of the EASST, vol. 74. Corfu, Greece (Oct 2016), with accompanying poster
26. Smyth, S., Schulz-Rosengarten, A., von Hanxleden, R.: Towards interactive compilation models. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018). LNCS, vol. 11244, pp. 246–260. Springer, Limassol, Cyprus (Nov 2018)
27. Sutton, A., Maletic, J.I.: Mappings for accurately reverse engineering UML class models from C++. In: 12th Working Conference on Reverse Engineering (WCRE'05). pp. 175–184. IEEE Computer Society, Pittsburgh, PA, USA (2005)
28. Tonella, P., Potrich, A.: Reverse Engineering of Object Oriented Code. Springer Science+Business Media, Inc., New York, NY, USA (2005)
29. Wechselberg, N., Schulz-Rosengarten, A., Smyth, S., von Hanxleden, R.: Augmenting state models with data flow. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. pp. 504–523. LNCS 11200, Springer International Publishing (2018)