Electronic Communications of the EASST
Volume 74 (2017)

7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium, 2016

Model Extraction of Legacy C Code in SCCharts

Steven Smyth, Stephan Lenga, and Reinhard von Hanxelden

20 pages

Guest Editors: Anna-Lena Lamprecht
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Model Extraction of Legacy C Code in SCCharts

Steven Smyth, Stephan Lenga, and Reinhard von Hanxelden

Department of Computer Science
Faculty of Engineering
Kiel University

Abstract: With increasing volumes of developed software and steadily growing complexity of these systems, software engineers struggle to manually maintain the vast amount of legacy code. Therefore, it is of interest to create a system which supports the documentation, maintenance, and reusability of software and its legacy code. The approach presented here automatically derives SCCharts models from C code. These models can be used as visual documentation. By applying focus and context methods important parts of the model can be highlighted and may grant a better understanding of the overall software. Additionally, the models can be used as a source to create new state-of-the-art code for various languages and platforms, such as C code or VHDL, using code generators.

Keywords: Model Extraction, Model-based Engineering, SCCharts, Documentation, Code Generation

1 Introduction

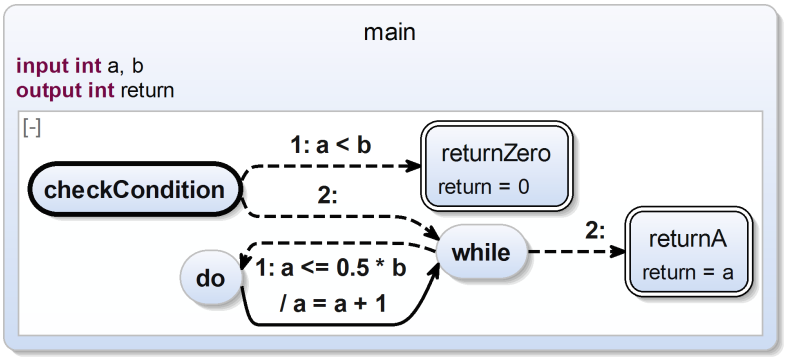
As the development rate of software in nearly every sector of the industry is reaching new highs, software engineers struggle to manually maintain the vast amount of legacy code. Seacord et al. labeled the alarming development that new software is outpacing the ability to maintain it the *Legacy Crisis* [SPL03, Chapter 1.3]. Therefore, it is of interest to create a system which supports the documentation, maintenance, and reusability of software systems and in particular of its legacy code.

Model-driven software development (MDSD) is an approach which besides other goals aims to support software modernization. Initially, it was created to develop new software, but recently it is also used for maintaining existing software systems [IM14]. MDSD makes use of graphical models to provide a clear overview of general concepts, core functionality, and structural composition of software. The represented models are created with the help of modeling languages. *Modeling languages* are visual programming languages which specialize in the specification of the requirements, the structure, the control flow, or the data flow of software systems. This is achieved by using a high level of abstraction. Sophisticated MDSD tools do not only enable the user to manually create models but also provide means for extracting models from source code [Sch06].

SCCharts The Sequentially Constructive Statecharts (SCCharts) language [HDM+14] is a visual synchronous modeling language which is specialized in specifying *safety-critical systems*.

```c
int main(int a, int b) {
  if (a < b) {
    return 0;
  }
  while (a <= 0.5 * b) {
    a = a + 1
  }
  return a;
}
```

(a) C code that serves as source model



(b) Example extraction of the C code depicted in Listing 1a

Figure 1: SCCharts extraction example

The language uses a statechart notation [Har87] and follows a synchronous Model of Computation (MoC) that provides determinate concurrency. While SCCharts contains various extended language features, which allow a compact representation of complex models, we here focus on the core features of the language, namely the *interface*, *states*, *transitions*, and *hierarchy*. To support the notion of *concurrency*, each state is divided into regions which can conceptually be seen as threads. However, since the approach presented here does not yet look into concurrent programs, all states include only one implicit *control flow region*. Additionally, states may contain actions. For instance, an *entry action* is executed whenever the state is entered. Figure 1b shows an example of a possible SCCharts representation of the C program shown in Listing 1a. It consists of a *root state* named main. *Input* and *output variables* are always displayed below the name. The state main contains five other states namely checkCondition, returnZero, while, do and returnA. checkCondition is the *initial state* of its region, which can be recognized by the bold black border. Transitions between states, which are indicated by arrows, show the possible control flows. The *trigger* of a transition needs to be fulfilled in order to enable the transition. It may be followed by a forward slash and one or multiple *actions* that are executed in sequential order if the transition is enabled. A transition without a specified trigger is always enabled. In this case, the transition label may directly begin with the slash and the action list. If a state has multiple outgoing transitions, *priorities* show the order in which the triggers of these transitions are checked. They are illustrated by a number in front of the trigger of the transition. The lower the number, the higher the priority. Time in synchronous languages is discretized into logical clock ticks. Solid-lined arrows represent *delayed transitions*. They are disabled in the tick in which the source state got entered and, hence, consume a logical tick. Dashed arrows depict *immediate transitions*, which do not consume time. A *final state*, which marks the end of the enclosing *region*, is characterized by a double-line border. Further information on all SCCharts features can be found elsewhere [HDM+14].

In Figure 1b, checkCondition is the initial state. The control flow of the program starts in this state. Since the priority of the upper transition is higher than that of the lower transition, it is checked first whether a is less than b. In this case, the final state returnZero is entered, the entry action return = 0 is executed. Additionally, the superstate main terminates. Otherwise, the

lower transition is taken because the empty trigger condition is always true. This leads to the while state. Here, a is incremented as long as a is less than or equal to half of b. Finally, if this condition is not satisfied anymore, the final state returnA is entered and the output variable return is set to the value of a.

All diagrams of models in this work are created and automatically layouted with the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). KIELER is a research project which is developed by the Real-Time and Embedded Systems Group at Kiel University. The primary objective is the enhancement of the model-based design of complex systems. By arranging graphical components with the help of automatic layout algorithms, and consequently freeing the user from redundant tasks, it improves the development process and maintainability [SSH13]. Besides being a platform and framework for researching and prototype development, KIELER also includes an SCCharts editor and compiler.

Contributions The approach presented allows to create visual representations of legacy C code automatically. On the one hand, one can take advantage of visual clues to help understand complex software, and on the other hand this integrates well into projects where some parts are already modeled in a graphical language. Especially in the automotive sector it is common to have for example a mix of Simulink models and C code.

Therefore, i) meaningful and understandable patterns to describe C programs must be found. The goal here is to enhance the overview and understandability of legacy systems. The presented transformation ii) generates SCCharts models, a novel Statechart dialect, especially developed for safety-critical systems. These models can be used for documentation and to generate new code for various platforms. Hence, the main application targets for, but is not limited to, legacy C code programs. We show iii) how we use the existing code generator of the KIELER SCCharts implementation to generate code for different platforms, such as C or hardware circuits, from the extraced models. Various other target platforms such as Java or VHDL, are also possible.

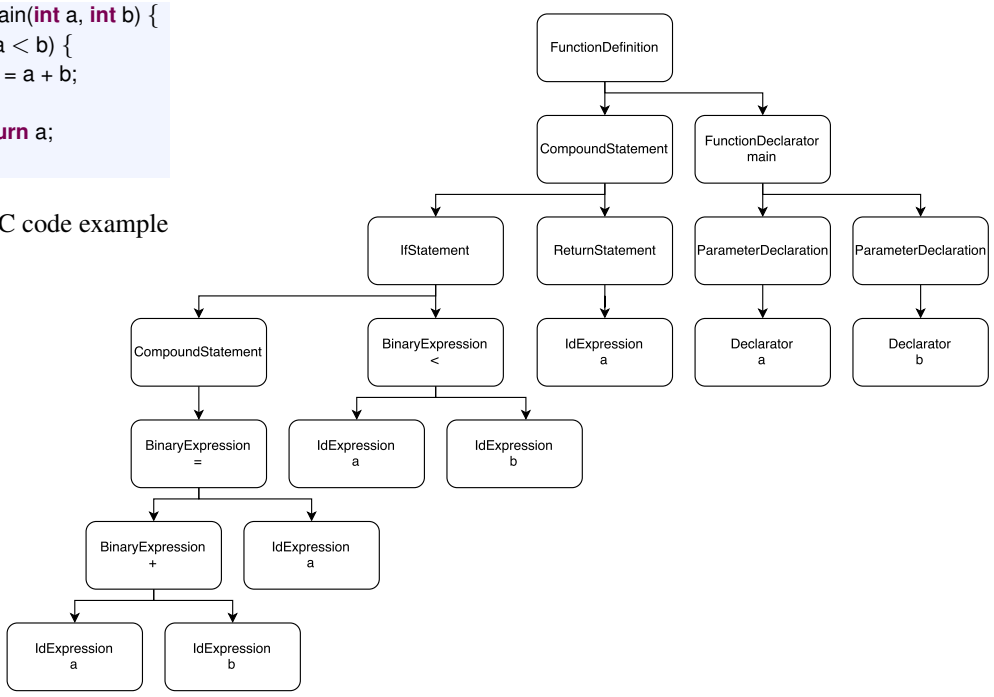
Outline Section 2 describes the model extraction mechanism. In particular, it shows how the various language features of C are mapped to the language features of SCCharts. Since the suitability of these mappings depends on the use case, we also discuss alternative ways of depicting the program in the abstract model. Thereafter, Section 3 presents the possibilities of the existing code generators and depicts preliminary results. We discuss related work in Section 4 and conclude in Section 5.

2 Model Extraction

In the first step, we evaluate the suitability of SCCharts to serve as model for C programs. Therefore, the C programs have to be parsed to create an *Abstract Syntax Tree* (AST) (see Section 2.1). The model extraction prototype uses model-to-model transformation techniques according to the single-pass language-driven incremental compilation approach (SLIC) [MSH14] in Section 2.2. Additionally, a few changes to the SCCharts visual language are adapted to the C programming language to facilitate understandability of the extracted models.

```
int main(int a, int b) {
  if (a < b) {
    a = a + b;
  }
  return a;
}
```

(a) C code example



(b) AST of the program shown in Listing 2a

Figure 2: Abstract Syntax Tree (AST) example

2.1 Parsing C Programs

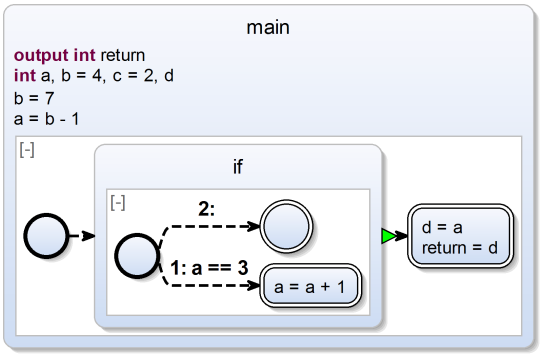
Since KIELER is an Eclipse framework, the Eclipse C Development Tooling (CDT)[1] is available. The CDT is an Eclipse project which serves as a fully functional IDE for developing applications in C/C++. It provides a fully featured editor and additional services such as source code navigation, static code analysis, debugging and unit tests. The CDT can parse files of C projects that are created or imported into the Eclipse environment. The CDT parser creates an Abstract Syntax Tree. The AST serves as the source model of the transformation. Commonly, each node of an AST represents a construct occurring in the source code.

For example, when the C program in Listing 2a is parsed by the CDT, the AST in Figure 2b is generated. First, the root node FunctionDefiniton defines the main function. It contains the FunctionDeclarator and its CompoundStatement. The FunctionDeclarator holds the name of the function, here main, and its parameter declaration. Each ParameterDeclaration defines one parameter of the function. In this example, the Declarators a and b are present. A CompoundStatement represents a code block surrounded by curly brackets. The main function contains one compound statement, which includes an *if statement* and a *return statement*. Both have corresponding nodes in the AST. The subtree of the IfStatement node also contains a compound statement and the *binary expression* that is used for evaluating the if statement. In all expressions, IdExpressions reference declared variables. The complete overview of possible nodes in the AST generated by

[1] https://eclipse.org/cdt

```
int main() {
  int a;
  int b = 4;
  b = 7;
  a = b − 1;
  if (a == 3) {
    a = a + 1;
  }
  int c = 2;
  int d = a;
  return d;
}
```

(a) C program illustrating declarations & assignments



(b) Extracted SCChart of the C program depicted in Listing 3a

Figure 3: Declarations and Assignments

the CDT can be found in the CDT manual[1]. As mentioned before, the AST serves as the source of the subsequent model transformation to SCCharts. Its structure and the provided information is used to identify the necessary model components and their positioning within the SCCharts model. Therefore, the different nodes of the AST are converted one after another by traversing the tree in depth-first order. Each node represents a model component that needs to be created. After every node has been visited, the text-to-model transformation process is complete. The resulting SCChart is a semantically equivalent visual representation of the original source code.

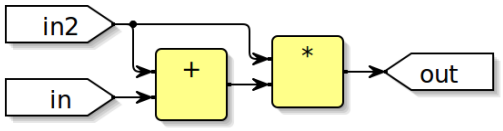
2.2 Transforming C Code to SCCharts

There may be several ways to represent the source program. Thus, the representations presented in the following subsections may in parts only be suggestions. To find a viable solution, several questionnaires were handed out to university staff and students. The provided answers were considered in the first prototype [Ols16]. However, we fine-tuned the representation of both the extracted abstract model and the visual representation of the source program to maximize overview and understandability. Nevertheless, SCCharts is a relatively new modeling language and still evolves in terms of language semantics, syntax and also compilation approaches. If the SCCharts language becomes more powerful, we expect that the number of C programs that we can transform automatically will increase. This will be an ongoing process and we anticipate that some the representations will change in the future. Further suggestions for improvements are discussed in Section 5.

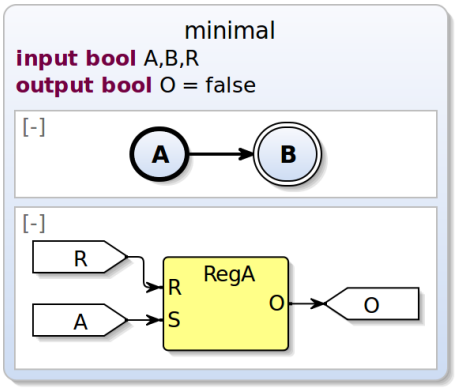
2.2.1 Functions

Every function of a C program is represented by a root state in the extracted SCChart. As described in Section 1 all states in our approach only use one implicit control flow region. These regions represent the compound statements of the functions and start with an initial state where

(a) Simple assignment in data flow notation



(b) Calculation in data flow notation



(c) Combination of control and data flow

Figure 4: Combining SCCharts with data flow notation [Uml15]

the control flow starts as soon as the enclosing state becomes active. The control flow represents that of the C program. If the function includes further control structures with compound statements, e.g., an if statement, these are represented by further superstates. Additionally, the arguments of the function are declared as input variables of the corresponding type. Analogously, if the function is returning a *return value*, an appropriate output variable is created. Local variable declarations of the function in C are also declared in the declaration interface of the SCChart. They are neither marked as input nor output.

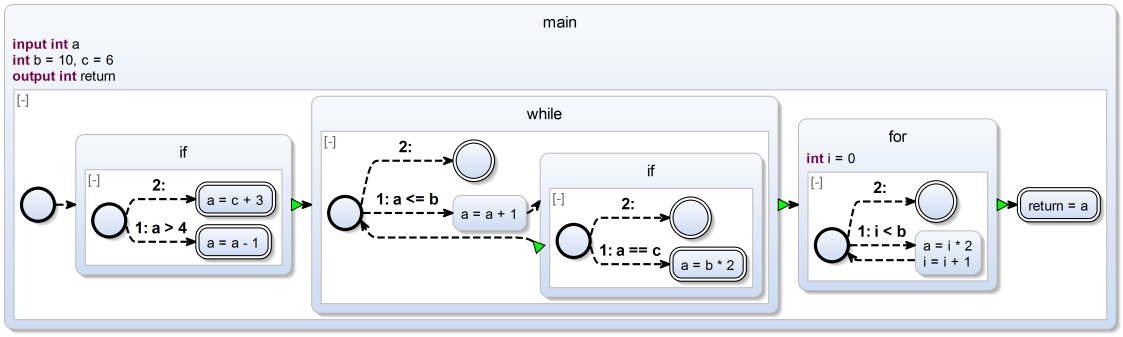
2.2.2 Assignments

There are different ways to assign values to variables in SCCharts. To stay as close as possible to the original code, SCCharts also uses *initializations* if the value is immediately assigned after the declaration. Additionally, it is possible to assign values to variables as transition actions or as state actions, e.g., entry actions that are executed whenever a state is entered. If the assignments take place at the beginning of a state, bundling these assignments to entry actions facilitates the readability. This reduces the number of overall states of the SCChart. Contrary to the standard syntax of SCCharts, which uses an extra keyword on entry actions, the *entry* keyword was omitted due to the fact that entry actions are the only actions that are used. This makes the visualization less verbose and increases the readability, because states with actions can be displayed in a more compact way than transitions with actions. Consider that the entry actions are executed in their sequential order, but after the initialization part of a declaration. Hence, dependencies between the assignments are preserved.

In the function and declaration example in Figure 3 you can see that the SCChart displayed in Figure 3b gets extracted from the program in Listing 3a. Hence, as described in Section 2.2.1, the function `main` creates the corresponding root state `main`. The four variable declarations `a`, `b`, `c`, and `d` are all visible in the interface of the root state regardless of where they were declared in the C function. Declarations with *initialization part* also preserve their *initialization*. Furthermore, the assignment `a = b - 1` is executed immediately after the function is called. Therefore,

```
1  int main(int a) {
2    int b = 10, c = 6;
3    if (a > 4) {
4      a = a − 1;
5    } else {
6      a = c + 3;
7    }
8    while (a <= b) {
9      a = a + 1;
10     if (a == c) {
11       a = b * 2;
12     }
13   }
14   for (int i = 0; i < b; i = i + 1) {
15     a = i * 2;
16   }
17   return a;
18 }
```

(a) Control statements example C code



(b) Extracted control statements from the C program shown Listing 5a

Figure 5: Control statements example

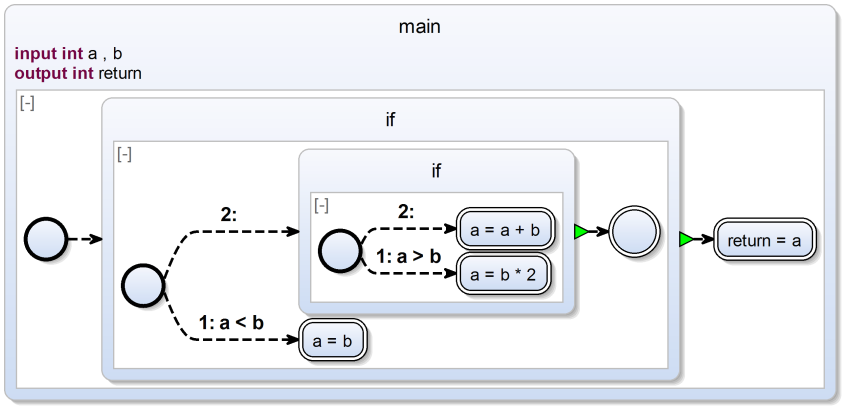
the assignment is displayed as an entry action. The control of the region passes from the anonymous initial state to the *if statement*. Here, the *condition* is checked before the control flow can proceed. After the compound statement of the *if statement* finishes, d and the *return value* are set. Contrary to the assignment of 2 to c during the initialization, a cannot be assigned to d beforehand, because this would neglect the potential increment of a in the *if statement*. Finally, the program terminates.

One might argue that the different notations for assigning values might get confusing. We chose this approach to stay relatively close to the original code and also to keep the models small with respect to the space needed for the diagram. However, it is of course possible to only use one notation for reasons of clarity which might result in more verbose models.

Even in a control flow orientated approach it is possible to use data flow like assignment syntax if the reader is more familiar with this notation. This is not implemented in the current version, but Umland [Uml15] showed how data flow notations can be combined with classical SCCharts (cf. Figure 4). In Figure 4a a simple assignment is depicted. The value in is written to out. Figure 4b shows the equation $out = (in + in2) * in2$. The combination of control and data flow semantics could happen on region level in SCCharts as can be seen in Figure 4c. The exact semantics of such combinations have yet to be defined. However, there already exist several examples of mixed semantics which can serve as models. Some are discussed in the related work in Section 4.

```c
int main(int a, int b) {
  if (a < b) {
    a = b;
  } else if (a > b) {
    a = b * 2;
  } else {
    a = a + b;
  }
  return a;
}
```

(a) if-then-else in C



(b) Extracted model of the C code listed in Listing 6a

Figure 6: if-then-else example

2.2.3 Control Statements

The control flow starts at the initial state of a compound statement and proceeds straightforwardly. To clarify the boundaries of control structures, every following control statement is embedded into its own superstate. Hence, the extracted model is organized according to the control statement structure of the original model. If desired, the corresponding line numbers of the C code can be added to all statements. However, in our experience this seems to be superfluous, at least for small examples. Moreover, user interactions with the graphical diagram can be configured such that the textual source of a specific element is selected automatically in the source code editor. The inner behavior of control structures, which is of no importance to the reader of the model, can be hidden by pressing the [-] symbol in the upper left corner of the region of the respective state. This can be of further benefit to understand the essence of the depicted function or program. Also, all control statements share similar layouts. As before, in each superstate the control starts at the initial state. Then, the body of the particular control statement follows and, finally, each structure has at least one final state. Optionally, a control structure may also declare local variables and actions. Control structures of the same type closely resemble each other in SCCharts. Here, only the enclosed components may differ, but can also be collapsed.

As an introductory example for control statements, Figure 5b presents an SCChart visualizing the C code of Listing 5a. Again, the function and its declarations create a new SCCharts root state. Furthermore, the function contains several control statements which also include compound statements, namely if, while, and for. All of these get transformed to superstates with their own scope and are executed in sequential order. The *while statement* also comprises another *if statement*. You can see the nested states in the diagram in Figure 5b. We will discuss the different control statements in detail in the following paragraphs.

if-then-else Statement The first embedded control statement in the SCChart in Figure 5b is the if-then-else statement. The path the control flow follows is determined by the condition in the statement. If the condition holds, the *then branch* is taken. Otherwise, the control follows

the *else branch*. Therefore, in the SCChart, the condition of the if statement is used as trigger of the first outgoing transition. This corresponds to the true branch. If the expression evaluates to true, the transition is taken. The second transition does not have an explicit trigger and is implicitly true. Since it is has a lower priority than the first transition, it is always taken in the *else case*. The targets of both transitions are marked as final, so the control statement superstate is left immediately in both cases.

Figure 6b shows the extracted diagram from the C code shown in Listing 6a. Here, the if-then-else statement is extended by another *else-if* structure. This corresponds to another nested if statement. So, if the *else branch* is taken, the control switches to the nested if statement which is represented by another superstate. Subsequently, the second if condition is checked and the control flow proceeds accordingly.

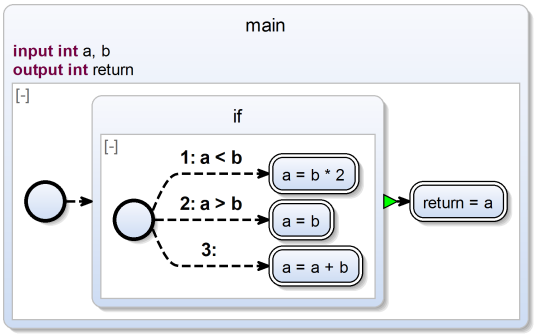


Figure 7: if-then-else alternative

As an alternative, Figure 7 shows another *if-else* variant. Here, the nested superstates are flattened. The different *if-else* cases get transformed to transitions with priorities according to the order their conditions get tested. This demonstrates that you do not have to adhere to the structure of the AST in all cases. As the *if-then-else* alternative is usually more compact and also well readable, we prefer this depiction.

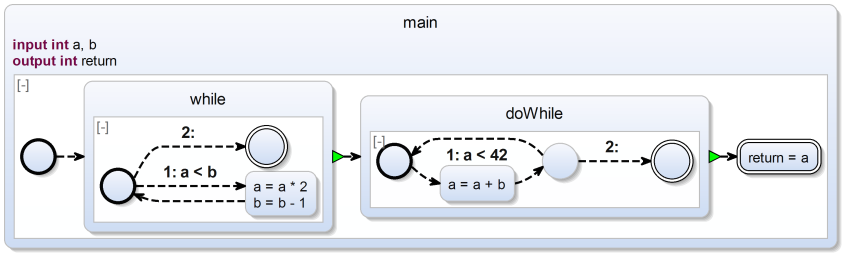
while Statement The *while statement* from the example in Figure 5 also becomes a superstate. The condition of the while loop is checked in the initial state. If the condition evaluates to true, the loop continues. Otherwise, the superstate is left immediately. In the while loop in Listing 5a, a is incremented by one before the nested *if statement*, also represented by a superstate, is checked. If it would contain another nested control statement, the nesting would also continue in the SCChart. Remember that the user can hide inside behavior if desired. After the *if superstate* finishes, the control returns to the check of the while condition.

The difference between the while statement and the do-while statement can be seen in Figure 8b which is an SCCharts representation of the C code listed in Listing 8a. In the main function of the C code, two loops, one while loop and one do-while loop, are executed sequentially. The corresponding root state in the SCChart consists of two embedded superstates for the two loops as described before. The difference between the two while loops, and the behaviors of their two superstates, is the point in time when the loop condition is checked. In the simple while loop the condition is checked immediately after the control enters the superstate. Contrary to this, in the do-while loop, the body gets executed immediately and the condition is checked afterwards. In terms of the SCCharts model, only the outgoing transition from the initial state decides whether or not the loop is a while loop or a do-while loop.

for Statement The last control statement shown in Figure 5 is the *for statement*. The standalone variant is depicted in Figure 9b. As in the other control statements, a *for statement* also becomes

```
int main(int a, int b) {
  while (a < b) {
    a = a * 2;
    b = b - 1;
  }
  do {
    a = a + b;
  } while (a < 42);
  return a;
}
```

(a) while loops in C

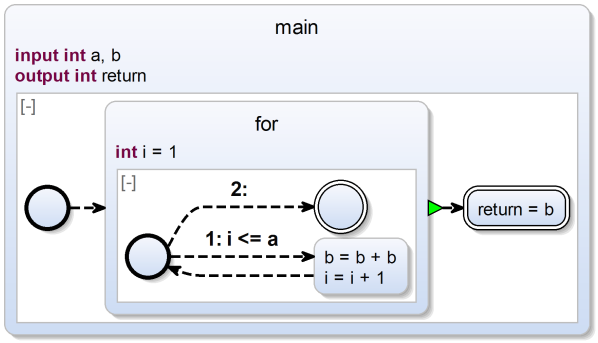


(b) Extracted model of the C code depicted in Listing 8a

Figure 8: while example

```
int main(int a, int b) {
  for (int i = 1; i <= a; i++) {
    b = b + b;
  }
  return b;
}
```

(a) for loop in C



(b) Extracted model of the C code shown in Listing 9a

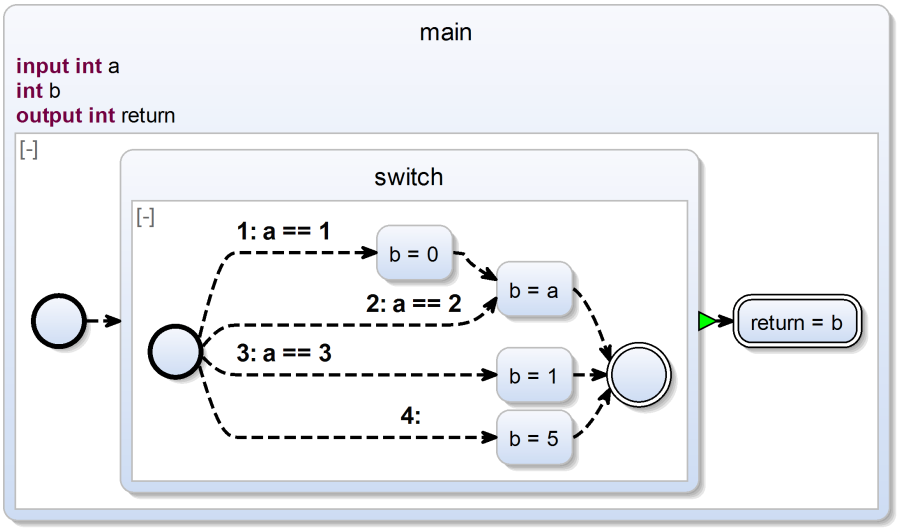
Figure 9: for example

a superstate and resembles a while loop. However, the for loop gets a dedicated counter variable which is declared at the beginning of the superstate and initialized with the corresponding value. The condition of the for loop is checked similarly to the condition of the while loop. Additionally, the last assignment in the body of the for loop modifies the counter variable. Again, the superstate is left as soon as the condition evaluates to false.

switch Statement The switch statement allows to combine cases of a conditional expression. However, due to the infamous *fall-through semantics* this is also a common C *pitfall*. Single switch cases may be merged inadvertently if a break statement was forgotten. When extracting the C code presented in Listing 10a in the actual approach, the SCCharts representation is the one depicted in Figure 10b. The *fall-through* from case 1 to case 2 becomes obvious. All other cases use the break statement to leave the switch at once. Even if the example gets bigger with arbitrary break statements, this is a nice example how the automatic graphical representation of the program helps to understand the semantics of the legacy C code.

```
int main(int a) {
  int b;
  switch(a) {
  case 1:
    b = 0;
  case 2:
    b = a;
    break;
  case 3:
    b = 1
    break;
  default:
    b = 5;
  }
  return b;
}
```

(a) switch statement in C



(b) Extracted model of the C code depicted in Listing 10a

Figure 10: switch example

2.2.4 Function Calls

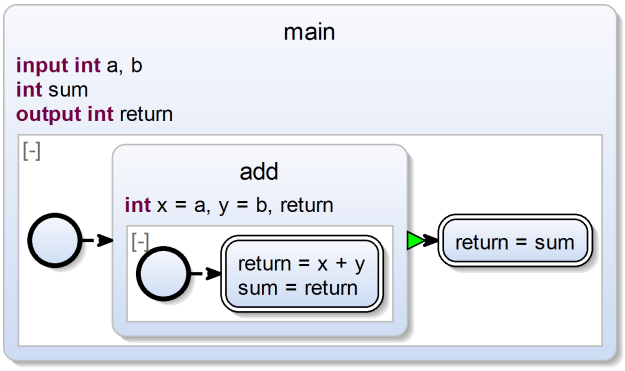
Representing *function calls* in SCCharts may depend on the compilation approach. Consider the C program listed in Listing 11a. A first straightforward approach performing a *macro expansion* is depicted in Figure 11b. Similar to the control statements presented before, the function call gets its own superstate. When the state of the called function add is entered, the arguments a and b are copied to local variables x and y. These variables are used for internal computations. After the sum of the two integer values is assigned to the local return variable, the value is copied to the variable sum. Even though the visual feedback for the user shows the detailed behavior of the program, the approach leads to problems with multiple calls of the same function as each function call will be translated to its own superstate. All of these states would contain the same information which would lead to an inflated SCChart. As a result, the generated C code from the SCChart would also contain duplicated code blocks. Consequently, this approach does not scale. Also, the actual semantic information that a call to a function happened is lost in this representation.

The second approach presented in Figure 11c makes use of reference states. A reference state in SCCharts references another SCChart and becomes active under the usual rules. However, the selected compilation approach decides how the referenced state is handled in the downstream compilation. For example, referenced SCCharts can also be expanded as explained before. Nevertheless, if the selected compilation approach supports actual function calls (e.g., when generating code for software) then the code generation is also able to treat the called SCChart as a separate function without duplicating its code. The function add is added as a second root state to the SCChart as already suggested in Section 2.2.1. The reference state Call maps the arguments a and b to the input variables x and y of the referenced state. Additionally, the variable

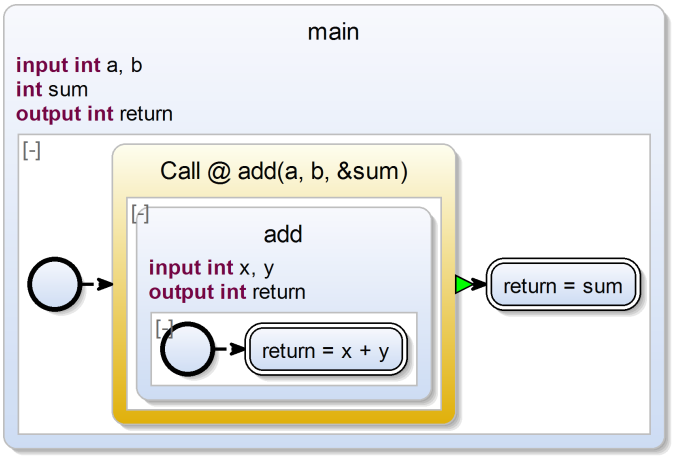
```
int main(int a, int b) {
  int sum;
  sum = add(a,b);
  return sum;
}

int add(int x, int y) {
  return x + y;
}
```

(a) Example of a function call in C



(b) Function call included via macro expansion

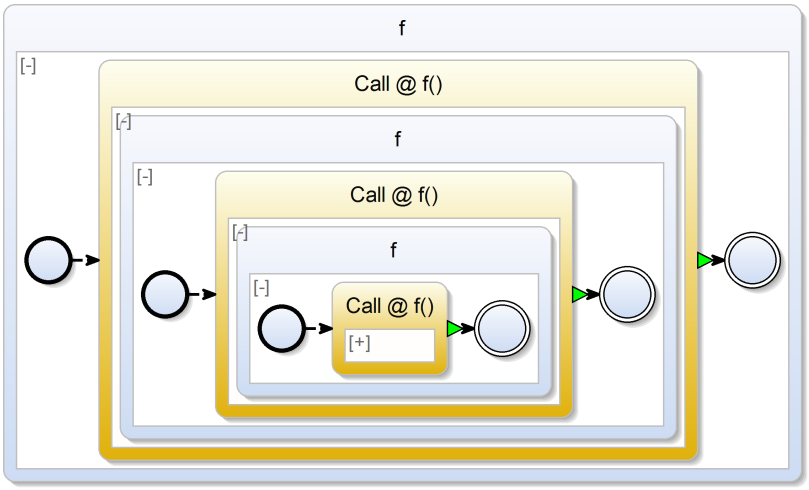


(c) Referenced call notation in SCCharts

Figure 11: Function call example

sum is mapped to the output return of referenced state add. Here, we use the SCCharts *function call* syntax, which resembles the *call-by-reference* syntax of C with the ampersand prefix (&). However, since SCCharts uses explicit inputs and outputs in interfaces, the variable binding is determined by the parameter order. The ampersand prefix merely serves as visual hint. The return value of the function is also given back in this way. Moreover, this solution does also support the expand/collapse features of common superstates. Even though each function call still has its separate state, its inner behavior can be hidden if desired. Hence, collapsing chains of function calls can also help to increase the overall overview.

By default, SCCharts generated programs do not acquire more memory dynamically at runtime. Hence, this approach cannot handle arbitrary recursive function calls. However, if the recursive depth is obtainable statically or if the generated code is able allocate more memory during runtime, this approach works in principal. Also, the visualization of referenced SCCharts could be used to inspect such programs as can be seen in Figure 12a. Nevertheless, the recursive nature may be expanded infinitely.



(a) Possible recursive call visualization

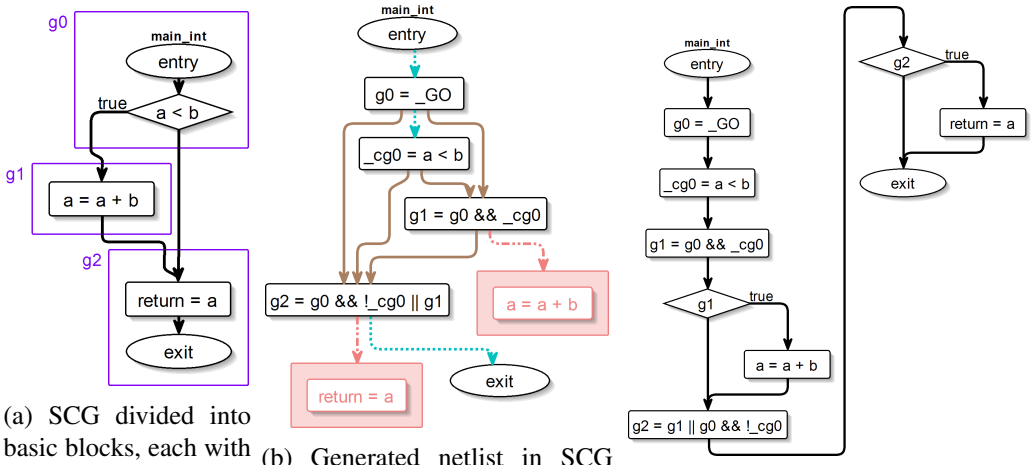
Figure 12: Example of a recursive function call

3 First Results

In the second step, we used the extracted models and the already existing KIELER SCCharts tool chain to automatically generate C code. To validate the feasibility of the approach, regression tests were executed and compared to the source models. Section 3.1 briefly introduces the used code generation method. Section 3.2 illustrates the approach based on a Fibonacci program.

3.1 Netlist-Based Code Generation

Figure 13 illustrates the downstream netlist-based compilation approach [HDM+14, SMH15] of the KIELER SCCharts implementation. When extracting the model from the AST example in Listing 2a the existing code generation approach of the KIELER SCCharts implementation then generates a Sequentially Constructive Graph (SCG) which is a control flow graph representation of the given program. The SCG is partitioned into basic blocks as can be seen in Figure 13a. From these blocks a new netlist in form of an SCG can be synthesized. The netlist of the example is depicted in Figure 13b. The assignments in the black nodes correspond to the activating guards of the basic blocks, here g0, g1, and g2. _GO is the *start signal* of the program and is true in the first clock tick. The assignments that are guarded by the blocks are depicted as red nodes. The different colored arrows illustrate different kinds of dependencies. All dependencies enforce an order: the source of a link must precede the target. The colors decode the semantic reason of these orderings. Brown colored links mean that this expression depends on the result of another expression. Red colored links say that this node is guarded by the source of the link. Blue colored links depict sequential control flow dependency. Finally, when generating sequentialized code, a schedule obeying these dependency constraints can be found if the given program is sequentially *constructive*, meaning it does not have any causality issues under the sequentially constructive MoC (SC MoC) [HDM+14].



(a) SCG divided into basic blocks, each with separate guards

(b) Generated netlist in SCG form

(c) Sequentialized SCG

Figure 13: Illustration of the netlist-based code generation approach of the KIELER SCCharts implementation.

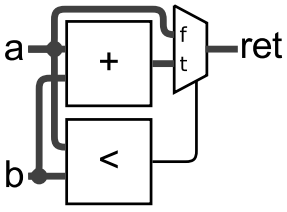


Figure 14: Generated circuit (optimized)

The netlist-based approach can be used to synthesize software and also hardware [MSH14, RSM+16]. The corresponding circuit can automatically be created and is displayed in Figure 14. However, the netlist-based approach does not handle instantaneous loops, unlike the priority-based approach [HDM+14]. In fact, the size and structure of the model and the choice of target platform usually determine which compilation method promises best results. However, in the case of model generation, the choice of the compilation approach also determines parts of the model. For example, a loop may not be generated instantaneously if the target of the compilation chain is a circuit. Here, the synchronous MoC provides tools which allow us to create circuits, but it requires that at least one transition in the loop must be marked as delayed to consume time. For future SCCharts versions we could imagine a dynamic transition type that decides if it is delayed or immediate during compile time after the compilation approach is set.

3.2 Full Example

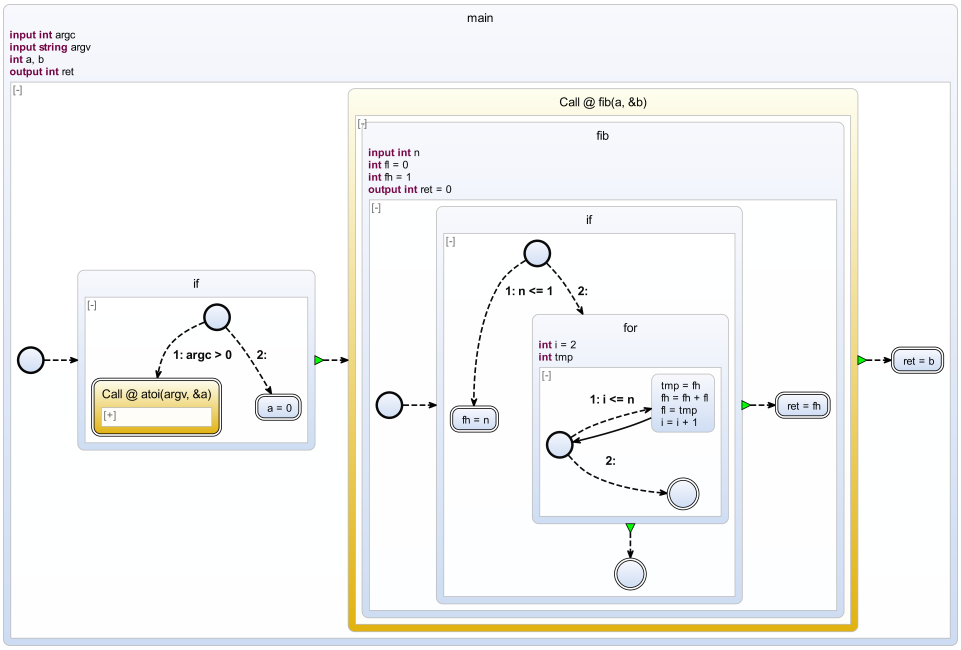
We validated the correctness of functionality within a discretized timing environment typical for synchronous languages [PEB07]. We are able to generate (simulated) circuits out of C programs with the restriction that loops are not instantaneous. However, the transformation automatically marks transitions as delayed if required. This section discusses the extraction and code generation process to both, software and hardware.

The C program in Listing 15a calculates the n^{th} Fibonacci number. It was extracted according to the approach explained in Section 2. The extracted SCChart is shown in Figure 15b. At first, the argument check is performed. If an argument is provided, the function calls the atoi system

```c
int main(int argc, char** argv) {
  int a, b;
  if (argc>0) {
    a = atoi(argv[0]);
  } else {
    a = 0;
  }
  b = fib(a);
  return b;
}
```

```c
int fib(int n) {
  int fl = 0, fh = 1;
  if (n<=1) { fh = n; }
  else {
    for (int i=2; i<=n; i++) {
      int tmp = fh;
      fh += fl;
      fl = tmp;
  }}
  return fh;
}
```

(a) Fibonacci in C



(b) Extracted SCChart of the Fibonacci C program in Listing 15a

Figure 15: Full Fibonacci example

function and converts the string into an integer which is then stored in a. Otherwise, a is set to 0. Subsequent to the argument check, the Fibonacci function fib is invoked. As described before, this reference is expandable and the structure of this function can be explored immediately. fib consists of an *if statement* and a *for loop*. As explained earlier in Section 3.1, the transition in the *for loop* may be delayed or immediate depending on the selected compilation approach. Eventually, the program returns the requested Fibonacci number.

Following the code generation approach explained in Section 3.1, a sequentialized netlist is generated for the Fibonacci program. Hence, two SCGs for both functions, main and fib, are created. The netlist representation of both SCGs can be seen in Figure 16. The SCG for the

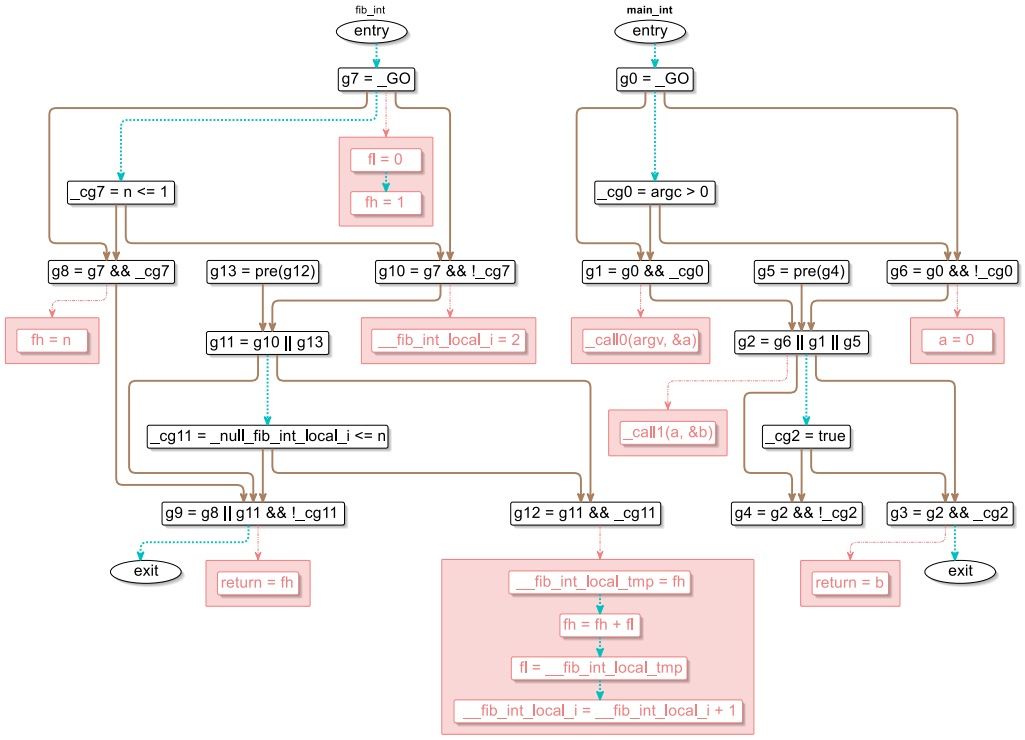


Figure 16: Generated netlist of the extracted SCChart depicted in Figure 15b

main function calls the SCG of the fib function. As the netlist-based approach is designed for both, software and hardware, the computation is logically clocked, meaning that the actual computation of the Fibonacci integer n requires n clock cycles. Nonetheless, it is possible to create both new code (see Figure 17a) and a circuit (see Figure 17b) with the same approach. The clocked computation of the Fibonacci example can be seen in Figure 18. We chose C as target language, because we wanted to compare the source and the target code directly. Of course, other languages are possible as well. Additionally, if not interested in the hardware synthesis, the previously mentioned priority-based approach can also be used for code generation. Usually, this code is more readable than the generated netlist.

In the hardware circuit the guard expressions are enclosed in the same manner as the basic blocks in the SCG. Activated blocks enable the calculations of the assignments they guard. To preserve the values of the variables between clock ticks, values must be saved in registers. Additionally, a *static single assignment* version of the SCG has to be created for the circuit to allow multiple variable assignments within one clock tick. The different versions of a variable are indexed in Figure 17b. The _TERM is set to true permanently as soon as the calculation terminates. Alternatively, it is also possible to only send the _TERM signal for a single clock tick. This would require fewer hardware gates. As before in the SCCharts visualization, the graphical diagram of the model also gets synthesized and layouted automatically instantaneously.

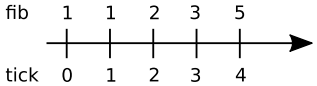


Figure 18: Clocked computation of the Fibonacci number

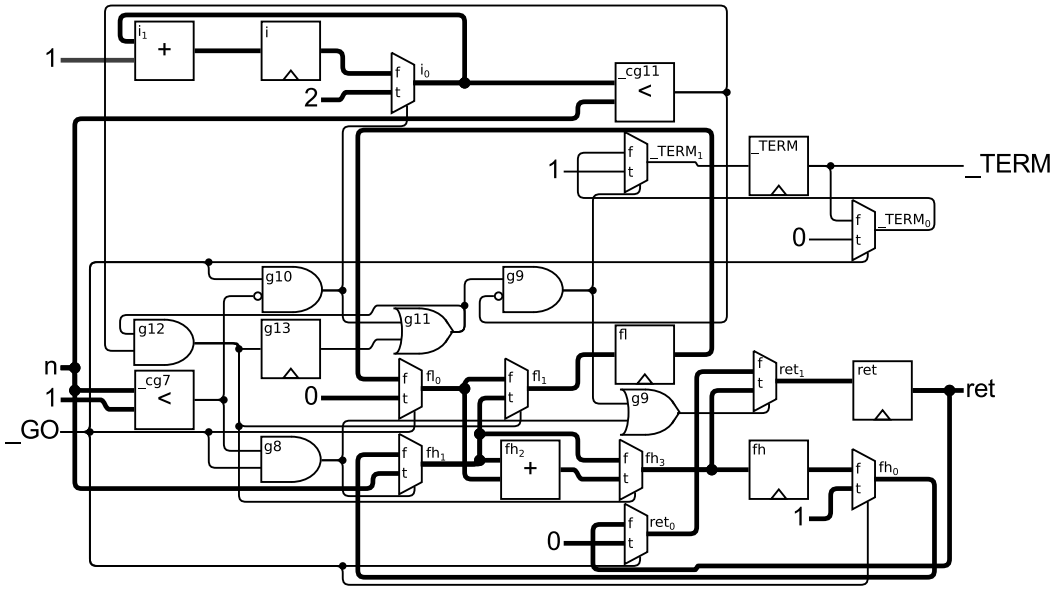
```
typedef struct {
  char _GO;
  char g7;
  ...
} TickData1;

void reset1(TickData1 *d) {
  d->pg12 = 0;
  d->_GO = 1;
  d->_TERM = 0;
}

void tick1(TickData1 *d) {
  tickLogic1(d);
  d->_GO = 0;
  d->pg12 = d->g12;
}

void tickLogic1(TickData1 *d) {
  d->g7 = d->_GO;
  if (d->g7) {
    d->fl = 0;
    d->fh = 1;
  }
  d->_cg7 = d->n <= 1;
  d->g8 = d->g7 && d->_cg7;
  if (d->g8) {
    d->fh = d->n;
  }
  d->g13 = d->pg12;
  d->g10 = d->g7 && !d->_cg7;
  if (d->g10) {
    d->__fib_int_local_i = 2;
  }
  d->g11 = d->g13 || d->g10;
  d->_cg11 =
    d->__fib_int_local_i <= d->n;
  d->g12 = d->g11 && d->_cg11;
  if (d->g12) {
    d->__fib_int_local_tmp = d->fh;
    d->fh = d->fh + d->fl;
    d->fl = d->__fib_int_local_tmp;
    d->__fib_int_local_i =
      d->__fib_int_local_i + 1;
  }
  d->g9 = d->g11 &&
   !d->_cg11 || d->g8;
  if (d->g9) {
    d->ret = d->fh;
    d->_TERM = 1;
  }
}
```

(a) Generated C code of the netlist in Figure 16 (excerpt of the fib function)



(b) Generated hardware circuit of the netlist in Figure 16

Figure 17: Possible out-of-the-box compilation targets for the Fibonacci example in Figure 15 in the KIELER SCCharts tools

4 Related Work

There are several model-based development or documentation tools that cover parts of our topic.

The SCCharts language, which is a dialect of Harel's Statecharts [Har87], was specifically developed for safety-critical systems. The KIELER (Kiel Integrated Environment for Layout Eclipse Rich Client) lets the user edit SCCharts with a textual description, from which KIELER then automatically creates a visual representation in the familiar Statechart syntax. This auto-

matic layout allows the user to concentrate on the modeling problem without the need to handle tedious layouting tasks. The automatic layout is also an enabler for the model synthesis from C code presented here.

The Advanced Simulation and Control Engineering Tool (ASCET)[2] by ETAS GmbH is a product family for the model-based development of embedded automotive software. The main application area of ASCET is the modeling of and the code generation for safety-critical systems such as electronic control units (ECUs) for vehicles. The EHANDBOOK[3] also allows to extract visual models from C code and employs automatic layout for that. However, their representation is rather low level, data-flow oriented and based on a flat program dependence graph [FOW87]. In comparison, our visualization based on SCCharts is more control oriented and preserves high-level control constructs such as loops by making use of SCCharts hierarchy.

Doxygen[4] is a tool for generating software reference documentation. Doxygen can also produce graphs that give an overview of certain software aspects, such as call graphs and inheritance graphs. However, it cannot generate full models that capture the detailed software behavior.

Visual Paradigm[5] by Visual Paradigm International is a cross-platform modeling and management tool for IT systems. Its range of application reaches from modeling of software and databases to code generation and up to creating business process models. It also supports round-trip engineering for Java and C++ code which is also one of our goals.

ExplorViz [FWWH13] is a monitoring tool for providing live trace visualization of the communication in software systems. It is able to give the user an overview of the flow of communications, but does not grant insight to the actual behavior of a function. A model-to-text code generation is not possible. Hence, the model-based compilation of KIELER is more suitable for maintaining the legacy code of a software system, rather than monitoring it.

A number of modeling environments allow to synthesize code from visual models. The Safety-Critical Application Development Environment (SCADE) Suite[6] product by Esterel Technologies is a model-driven software development tool for creating safety-critical embedded software. It is based on the formally defined synchronous data-flow programming language Lustre [HCRP91] and offers both control-oriented and data flow modeling styles. SCADE offers a certified code generator. However, it cannot synthesize graphical models from textual code.

Ptolemy II[7] is an open-source software for developing and simulating actor-oriented models. Actors are defined as software components. They execute concurrently and communicate with each other through message passing via interconnected ports. Additionally, Ptolemy II enables C code generation from actor models. The .c-file is created by connecting specific template files for the different actors. The resulting C code serves as a template for further processing.

Edwards [Edw05] presents a comprehensive overview of the challenges of synthesizing hardware from C and discusses a number of approaches. He also mentions the difficulty of handling loops. As discussed, the synchronous model of computation, as embodied by SCCharts, allows to handle loops rather naturally by making use of the notion of a synchronous tick. In the absence of concurrency, we merely have to ensure that each control flow loop contains at least one delayed transition.

[2] http:/www.etas.com/ascet
[3] http:/www.etas.com/ehandbook
[4] http://www.doxygen.org
[5] https://www.visual-paradigm.com
[6] http://www.esterel-technologies.com/products/scade-suite
[7] http://ptolemy.eecs.berkeley.edu/ptolemyII

5 Conclusion

To summarize, the automatic extraction of C programs to models in a statechart notation is feasible. The extracted models can be used for technical documentation. However, finding "a best" visual representation for these models does not seem to be trivial even when considering only existing elements of the statechart dialect. Even when only considering function calls, assignments, and a couple of control statements, there are many different possibilities to display the model. A good balance between compactness, overall overview, and simplicity has yet to be found and may be impossible in a general manner. Nevertheless, extracted models of complex legacy code can help to understand and maintain these systems.

We implemented a *C extractor* for the synchronous language SCCharts. Hence, besides using the automatically generated and layouted diagrams for documentation, the already existing code generation chain can be used to create new code of the legacy program for different platforms, including hardware circuits. Again, deciding for one *best-fit* may not be possible and depends on the use case. This also influences the graphical notation of the statechart if one wants to preserve the semantics of the language as the different transition semantics of SCCharts show.

Future Work As future work, using alternative code generation approaches such as the priority-based compilation approach of the KIELER for software, the timing restrictions on loops should not be necessary anymore. However, this approach cannot be used to create hardware circuits [HDM+14]. Additionally, at the moment supported C programs are limited to ANSI C without *pointers* and *structs*. Lifting these restrictions to process a significantly wider range of legacy C programs is work in progress. Moreover, to validate the correctness and to evaluate the efficiency of our approach, we want to compare original C programs with their corresponding automatically generated counter-parts. Therefore, we need to extend our code base and the KIELER implementation to accept arbitrary test cases. Furthermore, as the KIELER tool suite and the included downstream compilation improves (e.g., more target platforms and improved language and development features), all advantages automatically become available for already extracted and yet to be extracted models. Finally, to evaluate the graphical representation and the ability to serve as a suitable documentation language, further user case-studies must be conducted. Especially when extracting larger programs, the readability of SCCharts and the usability of the existing tools should be studied further.

Bibliography

[Edw05] S. A. Edwards. The Challenges of Hardware Synthesis from C-Like Languages. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. DATE '05, pp. 66–67. IEEE Computer Society, Washington, DC, USA, 2005.

[FOW87] J. Ferrante, K. J. Ottenstein, J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Trans. on Programming Languages and Systems* 9(3):319–349, 1987.

[FWWH13] F. Fittkau, J. Waller, C. Wulf, W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proc. of the 1st IEEE International Working Conference on Software Visualization (VISSOFT'13)*. Pp. 1–4. Sept. 2013.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3):231–274, June 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE* 79(9):1305–1320, Sept. 1991.

[HDM+14] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, O. O'Brien. SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, Edinburgh, UK, June 2014.

[IM14] J. L. C. Izquierdo, J. G. Molina. Extracting models from source code in software modernization. *Software and Systems Modeling* 13(2):713–734, sept 2014.

[MSH14] C. Motika, S. Smyth, R. von Hanxleden. Compiling SCCharts—A Case-Study on Interactive Model-Based Compilation. In *Proc. of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. LNCS 8802, pp. 443–462. Corfu, Greece, Oct. 2014.

[Ols16] L. Olsson. Modellextraktion aus C Code. Bachelor thesis, Kiel University, Department of Computer Science, Mar. 2016. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/leo-bt.pdf.

[PEB07] D. Potop-Butucaru, S. A. Edwards, G. Berry. *Compiling Esterel*. Springer, May 2007.

[RSM+16] F. Rybicki, S. Smyth, C. Motika, A. Schulz-Rosengarten, R. von Hanxleden. Interactive Model-Based Compilation Continued – Interactive Incremental Hardware Synthesis for SCCharts. In *Proc. of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. LNCS 8802, pp. 443–462. Corfu, Greece, Oct. 2016.

[Sch06] D. C. Schmidt. Model-Driven Engineering. *Computer* 39(2):25–31, Feb. 2006.

[SMH15] S. Smyth, C. Motika, R. von Hanxleden. A Data-Flow Approach for Compiling the Sequentially Constructive Language (SCL). In *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörtschach, Austria, 5-7 Oct. 2015.

[SPL03] R. C. Seacord, D. Plakosh, G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison Wesley, Boston, Massachusetts , USA, 2003.

[SSH13] C. Schneider, M. Spönemann, R. von Hanxleden. Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. Pp. 75–82. San Jose, USA, 2013.

[Uml15] A. Umland. Konzept zur Erweiterung von SCCharts um Datenfluss. Diploma thesis, Kiel University, Department of Computer Science, Mar. 2015. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aum-dt.pdf.