Model-based Compilation of Legacy C Programs

Stephan Lenga

Bachelorarbeit
2016

Institut für Informatik
Arbeitsgruppe Echtzeitsysteme und Eingebettete Systeme
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth

Eidesstattliche Erklärung

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

Kiel,

Abstract

As the development rate of software for nearly every sector of the industry is reaching new record highs, software engineers struggle to manually maintain the vast amount of legacy code. Therefore, it is of great interest to create a system which supports the maintenance of software and its legacy code. Even though software solutions already exist, none of them satisfies the numerous requirements which arise in practice.

This thesis aims at extending the functionality of the *model-based compilation of legacy C programs* of KIELER. It presents a way of converting non-concurrent, non-dynamic legacy C code to *Sequentially Constructive Charts* (SCCharts), which were developed by von Hanxleden et al., and focuses on the conversion and visual representation of control structures and function calls. The first goal of this thesis is the extraction of SCCharts that visualize the functionality of the legacy code in a more organized and intuitive fashion. These charts provide an intermediate format which facilitates software maintenance as well as source code modifications. The second goal is to extend the KIELER compilation process of extracted SCCharts. In particular, a compilation procedure for SCCharts which contain function calls is added to the KIELER software package. As a last step, the advantages of the extended functionality are assessed. Therefore, created SCCharts models are compared to the results of the previous prototype of the model-based compilation of KIELER. Structural changes of the code visualization are analyzed and the legacy C code is compared directly to the generated C code.

Acronyms

ASC	Acyclic Sequential Constructiveness
ASCET	Advanced Simulation and Control Engineering Tool
ASCET-MD	ASCET Model and Design
ASCET-SE	ASCET Software Engineering
AST	abstract syntax tree
CDT	Eclipse C/C++ Development Tooling
DebuKViz	KIELER Debug Visualization
ECU	electronic control unit
HTML	HyperText Markup Language
IDE	integrated development environment
iur	initialize-update-read
KiCo	KIELER Complier
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KlassViz	KIELER Class Diagram Visualization
KLighD	KIELER Lightweight Diagrams
M2M	model-to-model
M2T	model-to-text
MDSD	Model-driven software development
MoC	Model of Computation
MoCs	Model of Computations
PDF	Portable Document Format
RCP	Rich Client Platform

Acronyms

SCADE	Safety-Critical Application Development Environment
SCChart	Sequentially Constructive Chart
SCCharts	Sequentially Constructive Charts
SCG	Sequentially Constructive Graph
SC MoC	Sequentially Constructive Model of Computation
SLIC	single-pass language-driven incremental compilation
T2M	text-to-model
T2M2T	text-to-model-to-text
UI	user interface
UML	Unified Modeling Language
WTO	Write-Things-Once

Contents

List of Figures

Chapter 1

Introduction

In order to keep up with the rapid growth of the software industry and the constant change of user requirements, software development businesses need to respond quickly by adapting their already existing products or developing new software. As a result, it is speculated that the development of new software is outpacing the ability to maintain it. This alarming development is even labeled by Robert C. Seacord as the *Legacy Crisis* [SPL03, Chapter 1.3]. To maintain and modernize software, the legacy code needs to be modified and adapted. In the context of this thesis, *legacy code* is defined as a code that is maintained by someone other than the developer or which was written a longer time ago. Therefore, this task often causes difficulties especially when the legacy code of a complex system is not well-commented and consequently hard to understand. Thus, software modernization is costly and time consuming if it is done without the support of other technologies.

Model-driven software development (MDSD) is an approach which 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 provides a clear overview of general concepts, core functionality and structural composition of software. The representing 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 or the inner control 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]. For an introduction of important MDSD tools, the reader is referred to Chapter 2. The extracted models then serve as a medium to clarify and simplify complex processes as well as an intermediate format to generate source code from. The process from the model extraction from source code up to the generation of source code of the extracted models will be called text-to-model-to-text (T2M2T) engineering in this thesis.

With the help of model extraction, the legacy source code of a software is visualized by model extraction. The purpose of the created model is to support the readability and maintainability of the supposedly insufficiently commented code. After the model has been changed for maintenance or functionality enhancement purposes, source code can then be generated again. Therefore, T2M2T engineering enables the user to fast prototyping and template generation of software systems. But in order to generate source code that is semantically equivalent to the legacy code, no information can get lost during this process. Consequently, it does not suffice that the visualizing model only contains the main functionality or a rough overview of a system. It needs to represent all information that is in any way important

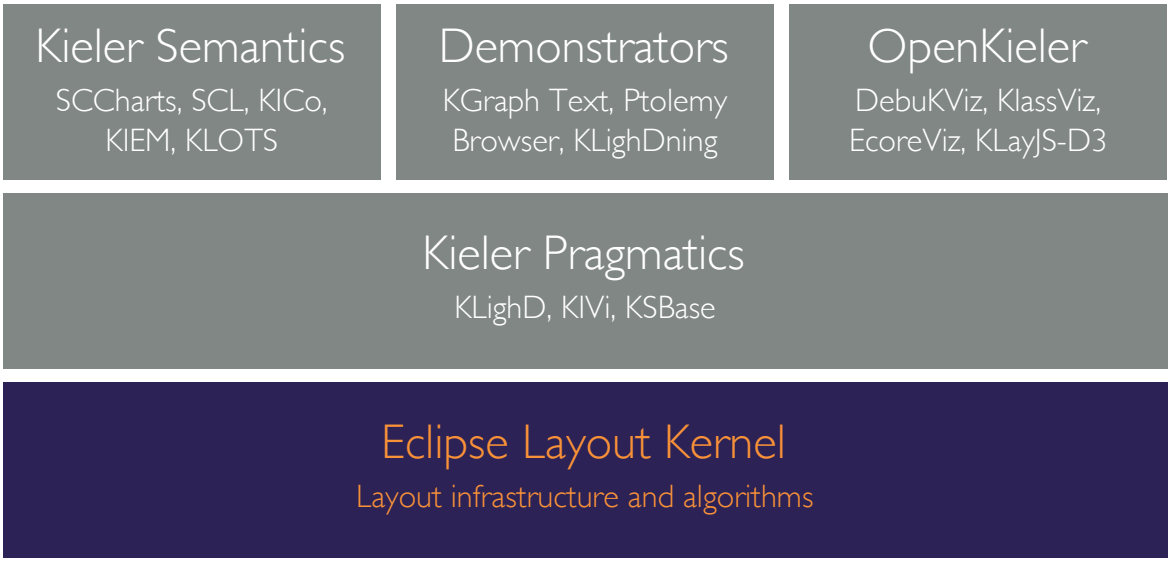


Figure 1.1. Overview of the KIELER[1]

to the functionality while still providing a clear and understandable model. Existing MDSD tools do not provide a satisfying solution for this challenge. Many tools either specialize on text-to-model (T2M) transformations which extract models from source code, or on model-to-text (M2T) transformations to generate source code from models. If a T2M2T feature is provided, it typically does not yield the desired results of generating semantically equivalent source code. The generated source code can only be seen as a code template which needs to be filled by the user rather than an executable source code. With the help of the results of this thesis, KIELER aims to provide a T2M2T feature which enables the user to generate semantically equivalent source code. As a result, the use of KIELER will lead to cost reductions of maintenance services and development processes.

1.1 Model-driven Software Development with KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is a research project which is developed by the Real-Time and Embedded Systems Group at the Kiel University, Germany. 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 development and maintainability. KIELER is divided into multiple parts, which is illustrated in Figure 1.1. One of the main tasks of the *Kieler Semantics* team is the definition of execution semantics for synchronous languages, such as SCCharts. The prototype of the model-based compilation of

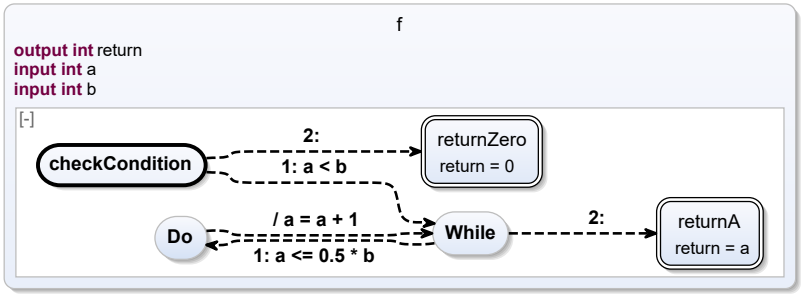
[1] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview

legacy C code [Ols16], which functions as a basis for this thesis, is also part of this work. The *Kieler Pragmatics* team provides means to visualize, edit and create models. The *Eclipse Layout Kernel* includes various layout algorithms and connects the graphical editors to them. The reader is referred to Chapter 3.2 for a more detailed presentation of the project structure of KIELER.

1.2 SCCharts

```
1  int f(int a, int b) {
2      if (a < b) {
3          while (a <= 0.5 * b) {
4              a = a + 1;
5          }
6          return a;
7      } else {
8          return 0;
9      }
10 }
```

(a) C code of a first example



(b) Manually created SCChart model which represents the C code of Figure 1.2a

Figure 1.2. Visual representation of variable declarations and assignments

Sequentially Constructive Charts (SCCharts) [HDM+14b] is a visual synchronous modeling language which is specialized in specifying safety-critical systems. This language uses a statechart notation [Har87] and provides determinate concurrency. The basis of SCCharts is formed by *Core* SCCharts which consist of a minimal set of constructs in order to model basic state machines. *Extended* SCCharts build on this foundation to provide more elaborate features and syntactical sugar and thus enhance the expressiveness and the readability of the created models. Extended SCCharts can be reduced to semantically equivalent Core SCCharts via model-to-model (M2M) transformations. By this, complexity is minimized. Furthermore, the Core SCCharts are used for compilation purposes. Figure 1.2b shows an introductory example of a Core SCChart which was manually modeled in KIELER. It represents the C code of Listing 1.2a. With the help of this example, basic functionalities and features of SCCharts are introduced. This naive approach is one of many ways of modeling the functionality of the given method f.

This SCChart consists of a *root state* named f. *Input* and *output variables* as well as *entry actions* are always noted below the name. They are called *interface declarations*. Entry actions are executed when the respective state is entered. The *state* f has one *region*. Furthermore, a state can have multiple regions. Each region can be seen as a thread which runs concurrently to others. The region of f contains four other states namely checkCondition, returnZero, While and returnA. checkCondition is the *initial state* of its region, which can be recognized by the bold border. It is mandatory that every region has exactly one initial state. It serves as a starting point for the control flow. *Transitions*, which are indicated by arrows, connect states and show

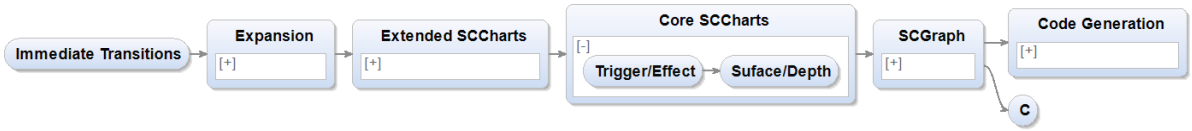


Figure 1.3. Overview of the KiCo compilation chain and its features

the possible control flow inside the region. Transitions can have *triggers* and *action* which are separated by a forward slash. The trigger of a transition needs to be fulfilled in order to enable the transition and to execute its action. A transition without a specified trigger is always enabled. In case 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. Solid-lined arrows represent *delayed transitions*. They are disabled in the same tick in which the source state got entered. Dashed arrows depict *immediate transitions*, which do not have this restriction. A *tick* is a signal from the outer environment of the SCChart that discretizes time. A state is left when all of its regions have reached a final state and are therefore terminated. A *final state* is characterized by a double-line boarder. Further information on additional features of SCCharts are presented in Chapter 4.1.

Starting at the initial state checkCondition, the condition of the if statement is checked. Since the priority of the bottom transition is higher than the upper transition, it is checked whether a is less than b. If this is not the case, the upper transition is taken. The final state returnZero is entered, the entry action return = 0 is executed and the region of the superstate f terminates. Provided that the condition of the if statement is satisfied, the state while is reached. Here, a is incremented as long as a is less or equal than half of b. If this does not apply anymore, the final state returnA is entered and the output variable return is set to the value of a.

1.3 Incremental Compilation of SCCharts

Part of KIELER is the KIELER Complier (KiCo) project which allows for a step-by-step transformation of SCCharts to C code via semantic-preserving M2M transformations. This approach is called *single-pass language-driven incremental compilation (SLIC)* [MSH14]. Figure 1.3 shows one possible compilation chain and its transformation features. This includes the feature *Immediate Transitions*, which is implemented in this thesis. This transformation step and its necessity is explained in Chapter 4.4. Multiple transformation features can be combined to *feature groups*, which are represented as rectangles. A group can be expanded to show the containing transformation features, as depicted by the group *Core SCCharts* in the figure above. Transformations of features may depend on prior transformations. SLIC lets the user choose which transformation steps are to be executed and then immediately displays intermediate results for validation and debugging. Dependencies are resolved automatically.

1.4 Problem Statement

The goal of this thesis is the improvement of the maintainability and the readability of legacy C code by converting it to an SCCharts model, which expresses the same functionality as the source code itself. This extracted model helps the user to understand and even to modify the program. An SCChart then functions as an intermediate format to generate executable code. In order to achieve a high scalability of information visualization, a structured graphical representation of C code needs to be developed. Especially the modeling of control structures and function calls present a serious challenge because the boundaries of those constructs can easily become unclear. As an addition, the representing SCChart does not only need to visualize the main functionality of the program, but it also needs to contain its every single information. Only then it can generate a semantically equivalent code later on. Having extracted a representing model from the code, the following step is an M2T transformation which generates executable C code from the SCCharts model. This job is done by the KiCo compilation chain that was presented in Chapter 1.3.

1.5 Outline of this Thesis

This section gives the reader a brief overview of all the intermediate steps, first from the development of the model-based compilation of legacy C code, to the implementation in KIELER, and finally to the evaluation of the results. Chapter 2 introduces related work that also provides means for model extraction as well as code generation. Each introduced technology is compared to the approach of this thesis and its necessity for development is discussed.

Chapter 3 describes technologies that are used throughout the implementation of the model-based compilation for KIELER. Hence, the KIELER project itself, which has already been outlined in Chapter 1.1, is looked at in a more detailed way. As KIELER is an Eclipse project, the Eclipse project[2] itself and its sub-project Eclipse C/C++ Development Tooling (CDT)[3] are discussed. Furthermore, the modeling language *Xtend*[4], which is used for M2M transformations, is introduced.

Chapter 4 presents concepts and ideas for the model-based compilation of non-dynamic, non-concurrent legacy C code. This chapter is split into two main parts: The first part takes a closer look on the syntax of SCCharts as well as the language itself and introduces the feature of *referenced* SCCharts. Next, the compilation of SCCharts down to C code is explored in more detail. This is necessary to recognize different possibilities and possible challenges of the visualization of source code.

The second part focuses on the process of visualizing C code. Therefore, this section of the chapter explains the extraction of an abstract syntax tree (AST) from the source code. Its information is used to create a representing model. First, the visualization of variable

[2] https://eclipse.org/
[3] https://eclipse.org/cdt/
[4] http://www.eclipse.org/xtend/

declarations and assignments is explained. Then, the conversion of control structures are discussed and the problem of representing function calls inside SCCharts and the correct conversion from and to C code is addressed. Consequently, solutions for the representation of recursive and non-recursive functions are developed, which are defined inside the .c-file we want to convert. Throughout this chapter, new ideas for the graphical representation of source code is compared to previous visualizations of the already implemented prototype.

Chapter 5 describes the essential parts of the implementation of the concepts which were introduced in Chapter 4. The focus lies on the changes and improvements of the *CDTProcessor*, which is responsible for the M2M transformations.

Once the theory and implementation has been covered, Chapter 6 evaluates the results and compares extracted models and generated code to those of the prototype. The evaluation focuses on the size and therefore on the number of states in a model and analyses the generated C code.

Chapter 7 concludes this thesis. It summarizes the findings and results and gives an outlook for future work.

Chapter 2

Related Work

This chapter introduces already established technologies and modeling tools which also provide code-to-model and/or model-to-code transformations. The necessity for the development of the model-based compilation of legacy C code in KIELER is emphasized by comparing each presented technology to the approach of the thesis.

Visual Paradigm

Visual Paradigm[1] 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. Therefore, this tool makes use of modeling languages such as Unified Modeling Language (UML)[2] to define specifications, create documentations and to assess requirements of the modeled system. Visual Paradigm supports several types of diagrams. Some of the more used types are class diagrams, activity diagrams and state machine diagrams [RLR+13]. Another important feature is the round-trip function for Java and C++ code. Round-trip engineering enables the user to generate a model from the source code and generate source code from a model.

This thesis also aims at providing a method for round-trip engineering C code. Handling C code is especially of interest when developing real-time and embedded systems. Besides, surveys have shown that C has been one of the most used programming languages for many years[3]. SCCharts as a modeling language has certain advantages over the state machine diagrams used in Visual Paradigm. It is easier to specify and design complex systems such as safety-critical reactive systems. The code generation of Visual Paradigm delivers a source code that is rather a scaffold code than an executable code. With the help of the model-based compilation, which is provided by this thesis, KIELER generates executable code that does not need further editing of the user.

ExplorViz

ExplorViz [FWW+13] is a monitoring tool for providing live trace visualization of the communication in complex software systems as well as performance analysis. It is developed by the research group Software Engineering at the Kiel University, Germany. A software system

[1] https://www.visual-paradigm.com/
[2] http://www.uml.org/
[3] http://www.tiobe.com/tiobe-index/

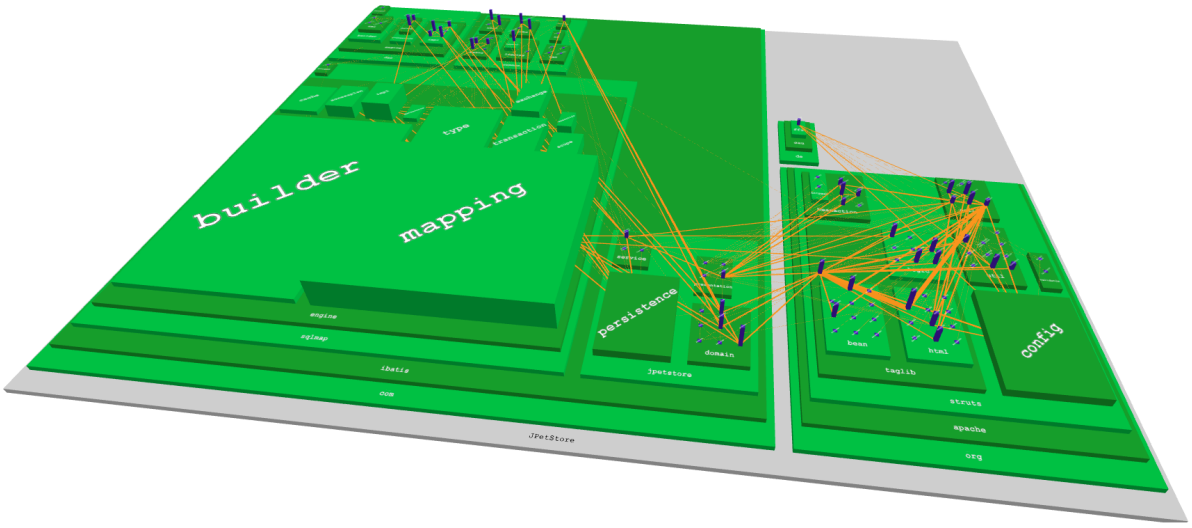


Figure 2.1. Generated landscape of a software system[4]

can be visualized as an interactive 3D city model which displays the packages of the system as boxes. These boxes can then be opened to reveal the inside, which can be other packages or classes. Classes are visualized by blue sticks. Their height is determined by the number of function calls of this class. Function calls are shown as yellow lines which connect two blue sticks. Figure 2.1 is an exemplary illustration of a generated city model of a software system.

ExplorVis 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 M2T 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.

Ptolomy II

Ptolomy II[5] is an open-source software for developing and simulating actor-oriented models. It is developed in the Center for Hybrid and Embedded Software Systems of the University of California, Berkeley [BHL+02]. Actors are defined as software components. They execute concurrently and communicate with each other through message passing via interconnected ports. Hence, a model is composed of hierarchical interconnections of its actors. Another particular feature is the introduction of a *director*. A director defines the semantics of the model rather than the framework itself and implements its Model of Computation (MoC). Ptolomy II provides a set of directors which support different types of models such as discrete-event models, continuous-time models and synchronous/reactive models. Each level of hierarchy in a model can have its own director. Therefore, multiple MoCs can be combined.

[4]https://www.explorviz.net/media.php
[5]http://ptolemy.eecs.berkeley.edu/ptolemyII/

Additionally, Ptolomy II enables C code generation from actor models. The .c-file is created by connecting specific template files for the different actors. These template files consist of code blocks. It is to be stated that there are only a limited number of actor types which are provided with supporting helper code. Secondly, only a small set of model types are supported. That is why there are many limitations to the code generation. The resulting C code serves as a template which needs to be filled with information rather than an executable code. Therefore, the code generation can be viewed as a concept demonstration.

ASCET

The Advanced Simulation and Control Engineering Tool (ASCET)[6] by ETAS GmbH is a product family for the model-based development of embedded automotive software which specialized on code generation. This product family consists of combinable software tools. Each of them focuses on a different set of tasks. Exemplarily, the tool ASCET Model and Design (ASCET-MD) is used to model physical systems while the tool ASCET Software Engineering (ASCET-SE) enables executable C code to be generated. Additionally, a number of static analysis and tests validate models. The main application area of ASCET is the modeling of and the code generation for safety-critical systems such as an electronic control unit (ECU) for vehicles. As a result, the correctness and robustness of the generated code is essential. Thanks to automatically included defensive coding checks in the generated code, run time errors are handled. To support the claim of high quality, the code generator of ASCET is certified by multiple norms including IEC 61508 and ISO/DIS 26262 which are norms for safety-critical systems.

While the standard of M2T code generation sets an example for the KIELER code generation, ASCET lacks the functionality of round-trip engineering. Therefore, the model-based compilation of legacy C code of KIELER provides a critical missing feature.

SCADE

The Safety-Critical Application Development Environment (SCADE) Suite[7] product by Esterel Technologies is an MDSD tool for developing safety-critical embedded software. It is based on the formally defined declarative and synchronous data-flow programming language Lustre [HCR+91] which is primarily used for critical control software of aircrafts, helicopters as well as nuclear power plants. SCADE features a graphical and textual editor for modeling data-flow and state machine charts. Furthermore, simulators and methods for formal verification of models ensure the quality of the created models. The certified code generator of SCADE then has the possibility of generating C code. Like ASCET, SCADE convinces with the quality of code generation but lacks the round-trip engineering.

[6] http://www.etas.com/de/products/ascet_software_products.php
[7] http://www.esterel-technologies.com/products/scade-suite/

Doxygen

Doxygen is a tool for generating software reference documentation. Therefore, the user must modify their source code by adding Doxygen documentation commands. These commands are then used to generate the program documentation. The documentation can either be of textual or graphical type. Multiple output types are supported like HyperText Markup Language (HTML), Portable Document Format (PDF) or LaTeX. It is also possible to extract the code structure from an undocumented source file in order to provide a general overview. Doxygen supports a multitude of programming languages including C, C++, Java, PHP and Python.

While Doxygen is used to provide an overview over a software system by using model extraction, the approach of this thesis focuses on providing a generation of executable source code. Furthermore, the user does not need to edit the legacy code in order to extract models from it.

Chapter 3

Used technologies

Before starting to explain main ideas of this thesis, the reader is introduced to the applied key technologies throughout the implementation and is given an overview of their main tasks and use cases. This should provide a better understanding of the presented solutions and their implementation. Section 3.1 introduces Eclipse, used plug-ins and the programming language Xtend. Thereafter, Section 3.2 explains the different main areas of the KIELER project.

3.1 Eclipse

The Eclipse Project[1] is an open-source software which was initially developed by IBM and released in 2001. Though it is mostly known for its Java integrated development environment (IDE), it also offers support for a number of other programming languages such as C, C++ and

[1] https://eclipse.org/

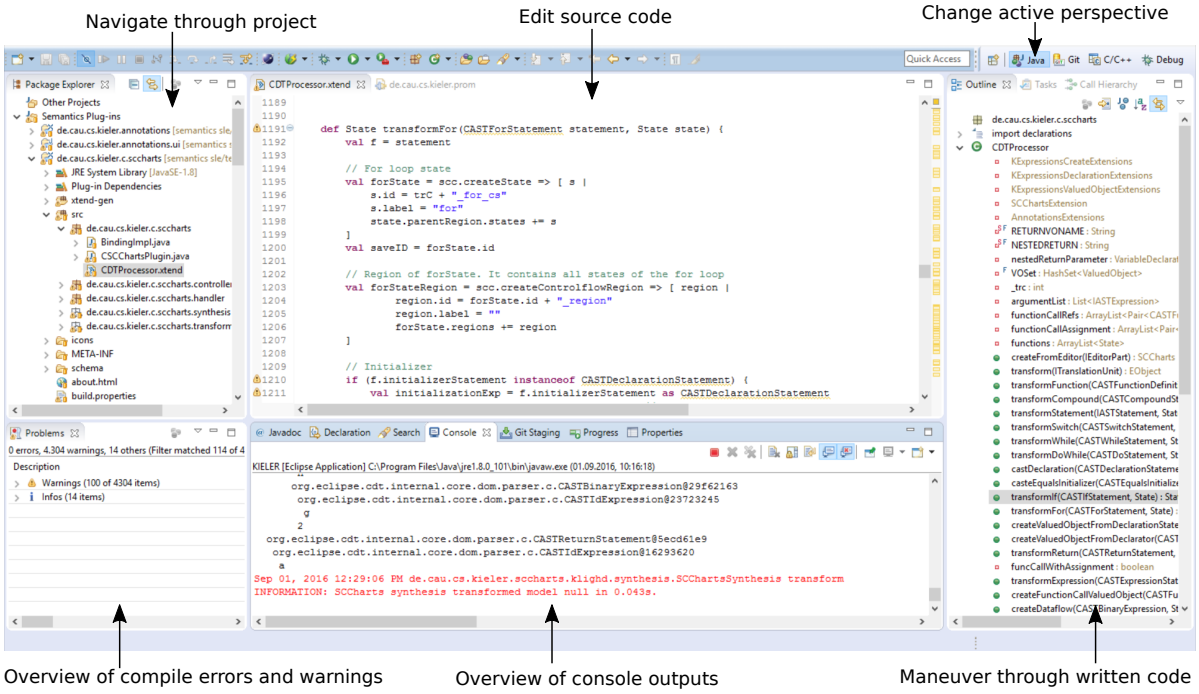


Figure 3.1. Overview of an Eclipse workbench and its different views

Haskell. Not only can it be used for programming, but also for modeling purposes. With its extensible plug-in system, the user has the possibility to customize the Eclipse environment to their liking. For that, Eclipse introduced the Rich Client Platform (RCP) for general purpose application development. It is a minimal set of plug-ins that are required to build a rich client application. These mentioned plug-ins manage generally needed tasks including booting Eclipse, running other plug-ins, managing the Eclipse Workbench with its views and editors as well as taking care of the file system. Hence, new plug-ins and Eclipse-based applications can be developed without spending resources on implementing main functionalities like the one just stated. The mentioned Eclipse workbench is the desktop development environment which combines multiple perspectives. A perspective defines the set and layout of views and editors which is aimed to accomplish a specific task. Only one perspective can be active at a time. An example of a workbench is shown in Figure 3.1.

3.1.1 Eclipse C/C++ Development Tooling

Eclipse C/C++ Development Tooling (CDT)[2] is an Eclipse plug-in which serves as a fully functional IDE for developing applications in C/C++ using the Eclipse. Not only does it provide a fully featured editor but it also supports services such as source code navigation, static code analysis, debugging and unit tests. The perspective of the model-based compilation in KIELER uses such a CDT Editor for the user to input the source code.

3.1.2 Xtend

Xtend[3] is a Java dialect which introduces a more compact syntax than Java and additional convenient features such as type inference, operator overloading and lambda expressions. The latter is commonly known from functional programming languages. All M2M transformations in this thesis are implemented with Xtend. Listings 3.1 and 3.2 give the reader a glimpse of the syntactical differences of Java and Xtend. This simple example presents a class `Greeter` which is used to greet a list of people. The method `greetABunchOfPeople` calls the method `sayHello` for each entry of a list of persons. The called method `sayHello` then prints a personalized greeting for the respective person. When comparing the two listings, several differences stand out. One variation to be noticed is the `def` to declare a method. Another innovation are extension methods such as *forEach*. They enable the programmer to add new methods to existing types without modifying them. Here, the type `List<String>` is extended by the method `forEach` which maps the function `println` to each entry of the list `people` with the help of a lambda expression. Additional features such as type inference, implicit returns and optional semicolons enhance the readability and give the source code a much leaner look. The given examples were taken from the official Xtend website[3].

[2] https://eclipse.org/cdt/
[3] http://www.eclipse.org/xtend/

```java
package my.company;
import java.util.List;

public class Greeter {

    public void greetABunchOfPeople(List<String> people) {
        for (String name : people) {
            System.out.println(sayHello(name));
        }
    }

    public String sayHello(String personToGreet) {
        return "Hello " + personToGreet + "!";
    }
}
```

Listing 3.1. Java code example

```
package my.company
import java.util.List

class Greeter {

    def greetABunchOfPeople(List<String> people) {
        people.forEach [
            println(sayHello)
        ]
    }

    def sayHello(String personToGreet) {
        "Hello " + personToGreet + "!"
    }
}
```

Listing 3.2. Java example translated to Xtend

3.2 KIELER

The introduction of this thesis 1.1 already gave a rough overview of the main structure of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project[4]. KIELER is a research project of the research group Real-Time and Embedded Systems at Kiel University, Germany. It is based on the Eclipse RCP and aims to enhance the graphical model-based design of complex systems. KIELER is an open source software and is licensed under the Eclipse Public License[5]. With the help of Figure 1.1, Chapter 1.1 introduced the main areas of KIELER. In the following, a closer look is taken on each of the different areas.

[4]http://www.rtsys.informatik.uni-kiel.de/en/research/kieler/
[5]https://eclipse.org/org/documents/epl-v10.php

Kieler Semantics

This thesis is associated with the area of Kieler Semantics which focuses on the execution semantics of systems, especially synchronous systems. It provides means to compile and simulate graphical modeling languages by using simulators based on C or Ptolemy[6]. Ptolemy is a platform for modeling and simulating of concurrent, real-time or embedded systems. For further details see Chapter 2. Simulators are integrated into KIELER by the KIELER Execution Manager (KIEM).

KIEM[7] [Mot09] is responsible for the connection between simulators as well as validation and visualization components on the one hand and the user interface of KIELER on the other hand. This link enables the simulation and execution of a given model. At this point it is emphasized that KIEM does not carry out simulations itself but rather connects all necessary components to each other.

The KIELER Complier (KiCo)[8] [MSH14], which is also part of the Kieler Semantics, allows for a step-by-step transformation of SCCharts to C code via semantic-preserving M2M transformations. Your own transformations can be written in Xtend or Java. They are then registered to KiCo by using provided extension points. After a transmission executes, the result is returned to the compiler. It can now be passed on to other transformations or the resulting model can be displayed to the user. This string of transformations which depends on the prior is called a transformation chain. Providing an example, the reader is referred to Figure 1.3 in the introduction of this thesis.

Kieler Pragmatics

Kieler Pragmatics develops solutions for supporting the daily work of a model developer. Therefore, the topics tackled are the visualization, the editing and the creation of models. Means are also provided to synthesize diverse views on the created models. The subproject KIELER Lightweight Diagrams (KLighD) [SSH13] offers a lightweight representation of models. A textual model is transformed to a lightweight diagram by a certain synthesis, which defines how the diagram should look like. As a next step, the created model is handed over to KIELER layout algorithms which automatically arranges the diagram components to free the user from the redundant task of positioning the components manually. Figure 3.2 clarifies the process of the visualization of a textual model.

Eclipse Layout Kernel

This section comprises various layout algorithms and combines them to the graphical editors. By automatically applying the algorithms to every change the user makes, it completely relieves from the redundant and time consuming task of firstly developing a clear and

[6] http://ptolemy.eecs.berkeley.edu/
[7] https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=328095
[8] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Kieler+Compiler

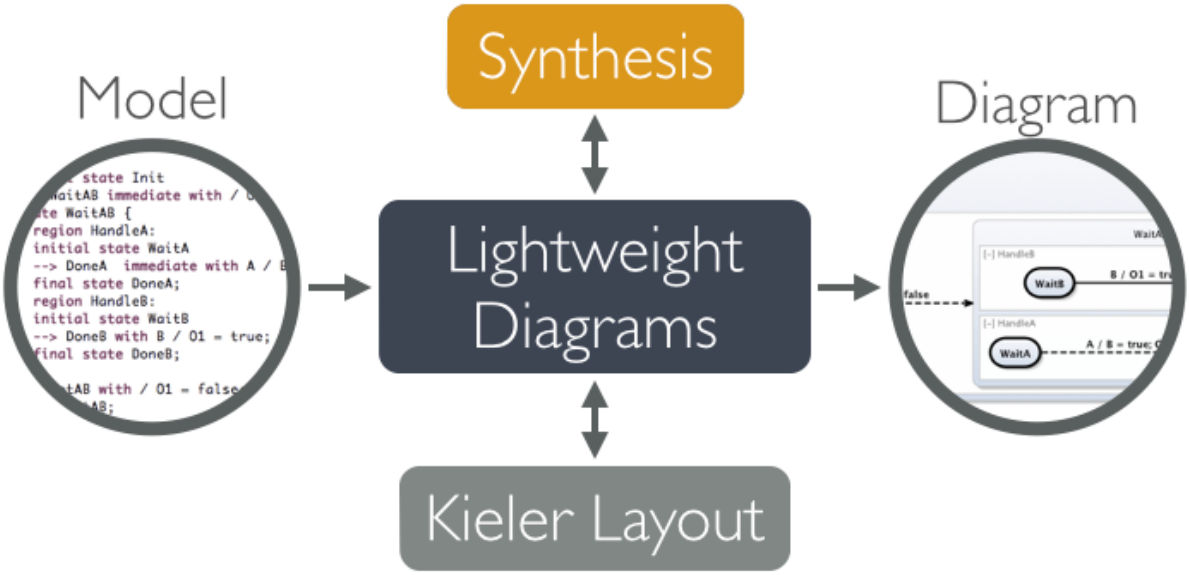


Figure 3.2. Overview of the visualization of a model in KIELER[9]

comprehensible layout and secondly positioning nodes and edges manually. Consequently, the user gets an immediate feedback of the changes he applied to the model.

Demonstrators

Demonstrators are the set of different editors of KIELER which can be used for the modeling input. This thesis makes use of a demonstrator, which is not part of KIELER, namely the CDT-Editor. This editor is provided by the Eclipse plug-in Eclipse CDT which was introduced in Chapter 3.1.1. It is used to register the C code input.

Open Kieler

The Open Kieler project is an open source project which is hosted on GitHub[10]. It encompasses a selection of demonstrator projects. One of the projects which is part of Open Kieler is KIELER Class Diagram Visualization (KlassViz)[11]. It provides a dynamic visualization of class diagrams using KLighD. Another featured project is KIELER Debug Visualization (DebuKViz)[12]. DebuKViz dynamically generates a graphical view of selected variables while debugging in Eclipse. At this point, the reader is referred to the documentations of the projects for further information as they are not relevant to this thesis.

[9] https://rtsys.informatik.uni-kiel.de/confluence/pages/viewpage.action?pageId=10751615
[10] https://github.com/OpenKieler
[11] https://github.com/OpenKieler/klassviz
[12] https://github.com/OpenKieler/debukviz

Chapter 4

Model-based Compilation of Legacy C Programs

The following chapter introduces the ideas and concepts of the newly developed model-based compilation of non-dynamic, non-concurrent legacy C programs. In order to be able to clarify the possibilities and challenges of the visualization of C code, the following topics are presented: Initially, the syntax of SCCharts is depicted and the features of this visual language, which are of importance to this thesis, are explained in Section 4.1. Thereafter, the basic concept of Sequential Constructiveness is presented and the compilation steps of SCCharts to C code are described. The subsequent Section 4.2 gives an in-depth examination of the extraction of a representing SCCharts model from C code. Possible visualizations of control structures and function calls are discussed and compared to the generated models of the already implemented prototype of Smyth and Olsson in Section 4.3. Finally, the code generation via the KiCo compilation chain is explained in Section 4.4.

4.1 SCCharts and their Compilation in KIELER

Section 1.2 already introduced the syntax and several features of the SCCharts language. Furthermore, Figure 4.1 provides an overview of additional features for *Core* SCCharts and *Extended* SCCharts. *Core* SCCharts provide the basis of this language with a minimal set of modeling constructs. *Extended* SCCharts enhance the expressiveness by adding additional features. Firstly, the available features of Core SCCharts are presented. Thereafter, a closer look is taken at the additions of Extended SCCharts. The top region of Figure 4.1 presents core components while the bottom region shows extended elements. The following part focuses on the features relevant for this thesis and explains them in detail. Core SCCharts may contain the following elements only:

States: As described by Charles André [And03], states define a condition or a status that may persist for a significant period of time. If this condition is met, the state is considered *active*. A state may be marked as *initial* or *final* and it may contain *regions* and *interface declarations*. The purpose of initial and final states is explained in the next item "*Regions*" of this itemization. Thereafter, the usage of interface declarations is clarified. A state is called a *superstate* if it encloses at least one region. Otherwise, it is labeled as a *simple state*. In Core SCCharts, states with inner behavior must not be marked as final. Additionally, it is prohibited for final states to have outgoing transitions. Furthermore, a state with no label

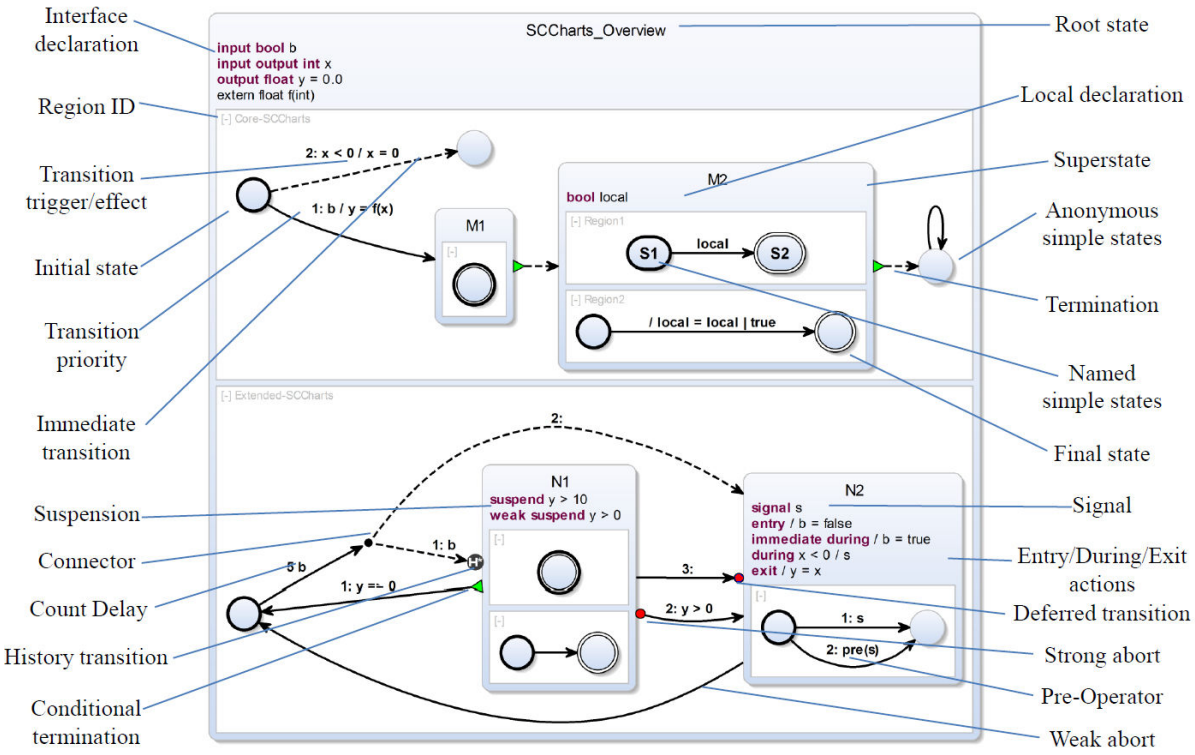


Figure 4.1. Overview of Core SCCharts of Extended SCCHarts features [HDM+14b]

is called an *anonymous state*. An SCChart consists of a set of states which are also called *root states*. These root states contain the functionality of the model.

Regions: Regions provide the user with the possibility of concurrency. Each region represents a thread which runs concurrently to others. A region must contain exactly one initial state and may contain multiple final states. When a state becomes active, all of its regions become active as well. Thus, the control flow starts at the initial state of each region. A region is considered to be *terminated* when a final state is reached. In case all regions of a superstate terminated, the superstate itself terminates as well.

Interface Declarations: Variables represent the main instrument for communication inside the SCChart. They are declared by the *interface declaration* of a state. Variables can be *inputs* from the outer environment of the SCChart, or *outputs* written to the environment. Furthermore, *inputoutput* variables combine the functionality of both other types. From this point on, when referring to inputs and outputs, this also includes inputoutputs. The environment initializes input variables at the beginning of a *tick* and reads output variables at the end of a tick. To clarify, a tick is concept that discretizes time [And03]. Unlike inputs and

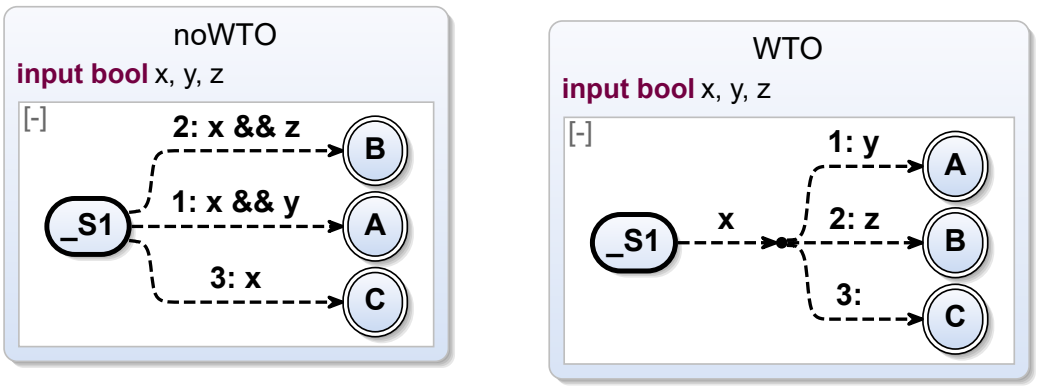
inputoutputs, outputs are not initialized at the beginning of each tick. However, they are persistent across the boundary of two ticks. It is also possible to declare *local* variables which are neither input nor output. Local variables are restricted to the boundaries of its respective state and can only be used by its inner behavior. As an example, consider the state `M2` of the SCChart shown in Figure 4.1. The local variable `local` is only known inside the state `M2` itself. Local variables are per default uninitialized. Therefore, the initial value is undefined. Once initialized, its value is persistent from one tick to another.

Transitions: Transitions connect two states with each other. They always have one *source* state and one *destination* state. A transition whose source and destination are the same is called a *self transition*. The optional *transition label* describes the *priority p* of the transition, a boolean expression called *trigger t* and an *action a*. Each element of the transition label is optional. The syntax of the label is described as "[*p*:] [*t*] [/ *a*]". A transition is activated when its source state is active and its trigger expression is satisfied. Thereupon, the transition gets taken, any action is executed and the destination state is entered. If a state has multiple outgoing transitions, their trigger conditions are checked in the order of their priorities. The lowest priority number is evaluated first. Core SCCharts differentiate between immediate and delayed transitions. *Immediate transitions* are represented by dashed arrows. They can become active in the same tick the source state is entered. An immediate outgoing transition without a trigger is called a *default transition*. Its source state is referred to as being *transient*. A transient state is always left in the same tick it is entered. Solid arrows indicate *delayed transitions*. Unlike immediate transitions, they are disabled in the same tick the source state is entered. Furthermore, Core SCCharts feature *termination transitions* in order to provide a possibility of preemption. Its visualization is a dashed arrow with a green triangle. In Core SCCharts, termination transitions may have a priority and an action but no trigger. It activates when the source superstate terminates.

Extended SCCharts provide the modeler with a more elaborate variant of the SCCharts language. By adding more complex features and syntactical sugar, Extended SCCharts enhance the expressiveness of the created model while simplifying the work of the user. Every Extended SCChart can be reduced to a semantically equivalent Core SCChart by M2M transformation. The following list contains all complementary features that are relevant to this thesis. For a complete introduction of all Extended SCCharts features and their transformation to Core SCCharts elements, the reader is referred to [HDM+13].

Connectors: A connector is a transient state and must therefore be left during the same tick it is entered. This construct can be used to enhance the readability of a model by linking multiple transitions to a single *compound transition*. Additionally, the usage of connectors promotes the Write-Things-Once (WTO) principle which encourages the programmer to reduce the repetition of information. Figure 4.2 contrasts the neglection and the usage of WTO. Figure 4.2a presents an SCChart with three input boolean variables `x`, `y` and `z`. Furthermore, its initial state `_S1` has three outgoing transitions. Each of these transitions

tests the input x. Figure 4.2b depicts a semantically equivalent SCChart which follows the WTO principle. The usage of a connector state divides the trigger checks into two parts. First, the input x is checked. Thereafter, the second condition check leads to the desired final state.



(a) Multiple evaluation of x without WTO

(b) Single evaluation of x due to WTO

Figure 4.2. The WTO principle

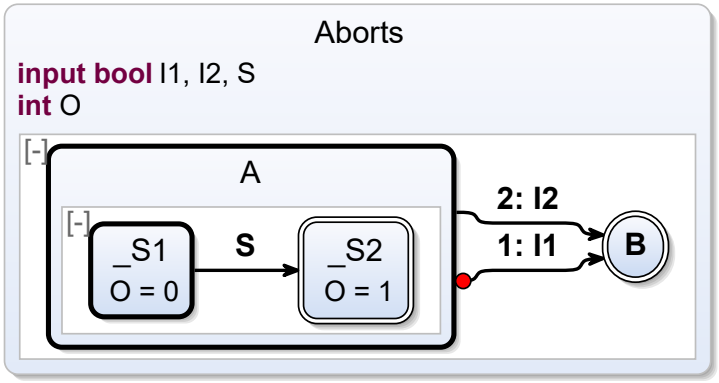


Figure 4.3. Example of strong and weak aborts

Complex Final States: Core SCCharts do not allow final superstates. If a final state activates, the corresponding thread of this region terminates. However, Extended SCCharts do provide this feature of complex final states. Therefore, superstates can be marked as final when using Extended SCCharts.

Entry Actions: An entry action of a state is executed immediately after the state is entered but before its inner body is able to react. If a state has multiple associated entry actions, they are performed in sequential order. A common use case of entry actions are the initialization of local variables.

Exit Actions: An exit action of a state is performed when leaving the previously active state due to either termination or aborts. Its exit action is executed after the inner behavior of the state had the possibility to react. Moreover, exit actions act before actions of outgoing transitions as well as immediate strong abort transitions. Like entry actions, multiple exit actions of a state are performed in sequential order.

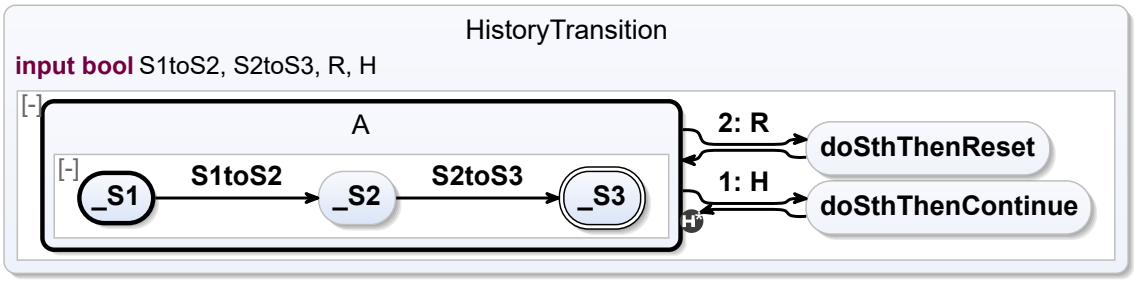


Figure 4.4. Comparing the behavior of history transitions and weak abort transitions

Aborts: Additionally to the already presented termination transition, the SCCharts language features two different transition types of aborts. *Strong aborts* are visualized by an arrow with a red circle connected to the source superstate. The trigger of a strong abort termination get tested before both the source state and its inner body have the possibility to react. *Weak aborts* gets tested after the body of the reaction of the state to be aborted. This type is represented by a plain arrow. The different behaviors of weak and strong aborts are illustrated in Figure 4.3. As a first example, assume that the input variables `I1` and `S` are true in the same tick. Since the trigger of the strong abort transition is true, the transition becomes active and enters the final state `B`. The content of the aborted state `A` does not get executed. Hence, the variable `O` does not get set to `1`. As a second example, let the inputs `I2` and `S` be true in the same tick. Therefore, the weak abort transition gets taken after the body of state `A` executed. Consequently, the value of the variable `O` equals `1` at the end of the tick.

History Transitions: Usually, the initial state of a region is entered when its parent superstate becomes active. However, a history transition allows to re-enter a superstate at the last state it was left previously. If a superstate becomes active for the first time, it is entered through the initial state, as usual. A history transition is indicated by an arrow with a black circle next to its arrow head. Inside the circle is an *H*. The behavior of a history transition is exemplified by Figure 4.4. As a first example, state `_S2` of superstate `A` is active. As a next step, all input variables but `H` are of value false. Thus, the state `A` is left and the state `doSthThenContinue` is reached via the bottom outgoing transition. After this state is left again in the next tick, the history transition permits to reenter the superstate `A` and to continue at state `_S2` with its execution. As a second example, assume that the input variable `R` changes its value to true while all other inputs are of value false. Consequently, state `A` is left again and the upper transition is taken in order to reach state `doSthThenReset`.

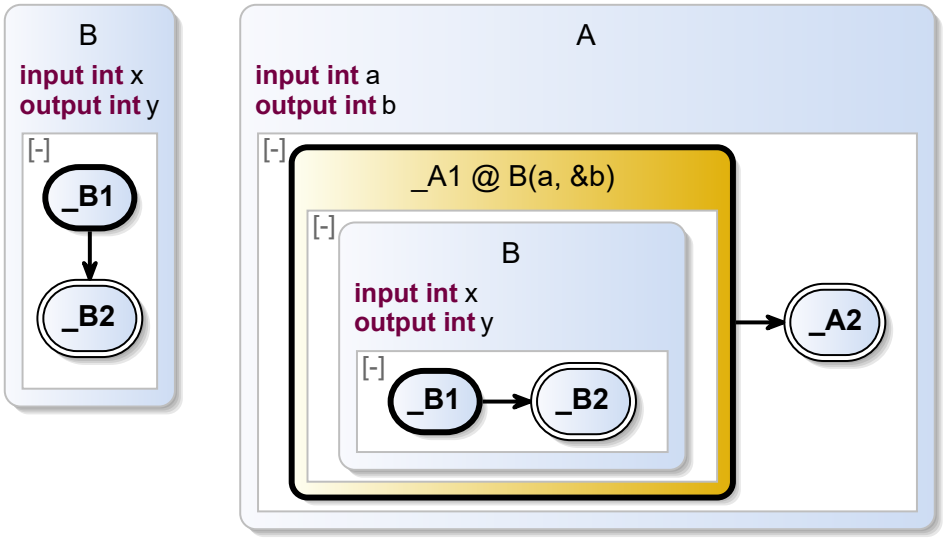


Figure 4.5. Reference states in SCCharts

As a next step, state `A` is reentered at the initial state `_S1` via a delayed transition. The previous active state `_S2` of superstate `A` is discarded. Even though history transitions are a convenient feature, they should be used with care as they greatly increase the overall state space of the created model. It does not suffice anymore to only remember the sub-states of active superstates. Additionally, the sub-states of inactive superstates need to be kept track of.

Reference States: Reference states, which are distinguished by their golden frame, enable the modeler to refer to other states of SCCharts which are contained inside the same project. In addition to the usual attributes and behaviors of a regular state, a reference state maps the input and the output variables of the referenced state to other specified variables. Furthermore, it is also possible to refer to reference states. For clarification, consider Figure 4.5. This example shows an SCChart with two root states `A` and `B`. The state `A` encloses a reference state `_A1` which refers to the state `B`. The reference state automatically contains all of the inner behavior of the referred state without further manual modeling. At this point, the reader may notice the different label structure of reference states. `_A1` is the name of the reference state. After an @-symbol, the name of the referenced state is located. The parentheses enclose the *parameter* variables to which the input and the output variables of the referred state are mapped to. An ampersand (&) in front of a parameter marks an output variable. In the given example, the variable `a` of state `A` is mapped to the input variable `x` of state `B`. Likewise, the variable `b` is mapped to the output variable `y`. The concept of reference states is very similar to function calls. Variables can be passed to other states. These states can use these inputs for internal computations. When the referred state terminates, it can return a value to the reference state which can be used for further computations.

4.1.1 Sequential Constructiveness

SCCharts is a visual language which specializes in specifying safety-critical systems. Hence, ensuring a determinate execution of concurrent threads is essential. Therefore, any occurrences of *race conditions* must be ruled out in order to prevent non-determinate behavior. Other synchronous languages like *Esterel* already provided means to ensure determinacy but they come with heavy restrictions. Specifically, multiple assignments of shared variables during the same tick are forbidden in Esterel's synchronous Model of Computations (MoCs). An MoC defines a set of allowable operations which are used in computation and their respective costs. In order to lift this limitation, the Sequentially Constructive Model of Computation (SC MoC) is introduced [HMA+13]. It enforces the initialize-update-read (iur) protocol which orders concurrent variable accesses. Moreover, it refines the *write-before-read* access protocol of previous synchronous languages. Prior the explanation of this protocol, several terms must be defined. The following definitions are taken from the paper "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications" by von Hanxleden et al. [HDM+14b].

Combination function: A function $f : X \times X \to X$ which satisfies the relation $f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$ for all $x, y_1, y_2 \in X$ is called a *combination function*.

Confluent variable access: A variable access of multiple concurrent threads is called *confluent* if the order in which the threads are executed is of no importance. As an example, consider two concurrent threads that assign the same value to the same variable. Naturally, the scheduling order does not matter and the variable always holds the correct value after both threads executed.

Absolute write access: An absolute write defines an *initialization* $x = e$ of a variable x and an expression e.

Relative write access: A relative write access defines an *update* assignment $x = f(x, e)$ for x of type f, where f is a combination function, x a shared variable and e an expression which does not reference x.

Read access: A *read* is a variable access that does not update the value of the variable.

For each variable, the iur protocol orders the non-confluent concurrent variable accesses which occur during the same tick. Firstly, all concurrent absolute writes are to be executed. They have to be confluent to each other. Otherwise, the program is not executable. Secondly, all concurrent relative write accesses must be scheduled after the initializations. In a final step, all read accesses can be scheduled after updates are done. A program is *sequentially constructive* if there exists an execution path of program while complying with the iur protocol with exception to confluent writes. Additionally, every execution of this control path must generate the same deterministic output. Since the examination of this problem is of co-NP complexity

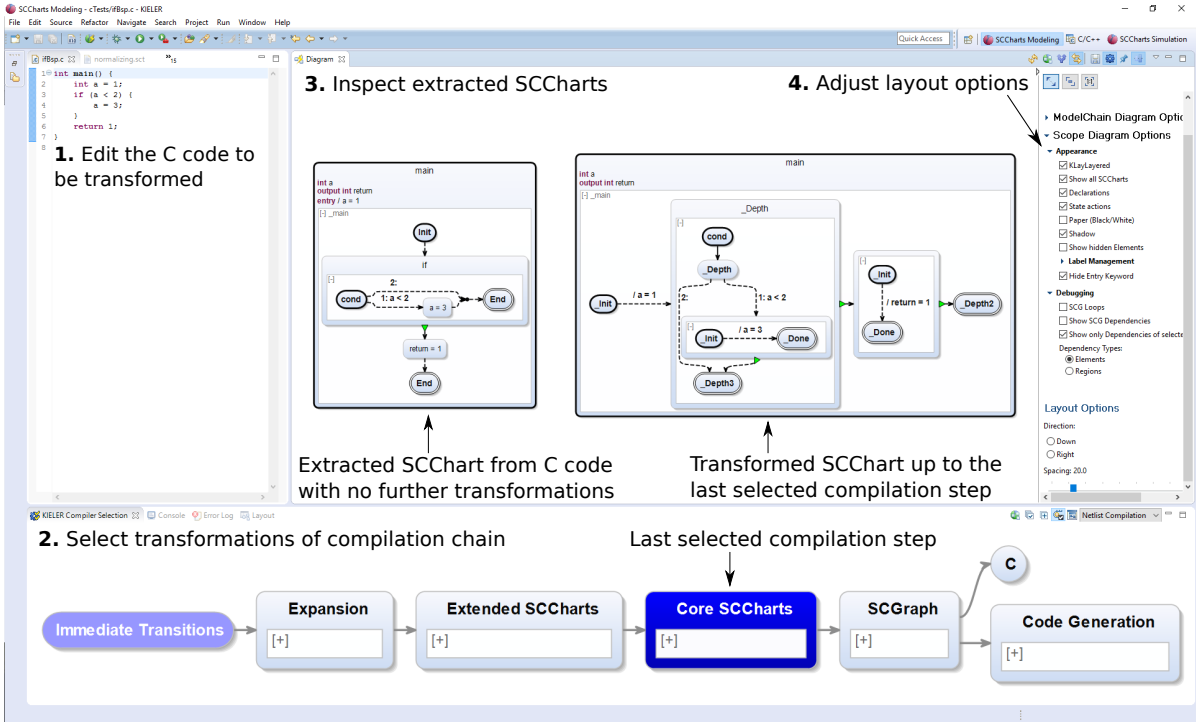


Figure 4.6. The KIELER workbench including additional annotations for the user story for the interactive compilation

[HDM+13], the approximation Acyclic Sequential Constructiveness (ASC) is introduced. A program is *ASC schedulable* if there exists an instantaneous acyclic control flow path through the execution of the program. Furthermore, it is differentiated between iur acyclic and data-flow acyclic programs. An *iur acyclic* program does not allow instantaneous cycles when all of its iur edges involve run-time concurrent accesses. *Data-flow acyclic* programs do not allow any non-confluent accesses.

4.1.2 The Interactive Compilation of SCCharts in KIELER

This section presents another innovative component of KIELER – the *single-pass language-driven incremental compilation (SLIC)* [MSH14]. SLIC enables the user to effectively validate and debug models by inspecting intermediate compilation results. These intermediate results are provided by a *step-by-step compilation*. Since the improvement of the validation and debug process is of utmost importance, this feature greatly enhances the quality of the created product. This especially applies for the development of safety-critical systems. The incremental model-based compilation is realized by a *compilation chain* consisting of a set of M2M transformation features. A *feature* may depend on prior features. If a transformation is selected by the user in the compilation chain, the compilation result of this transformation,

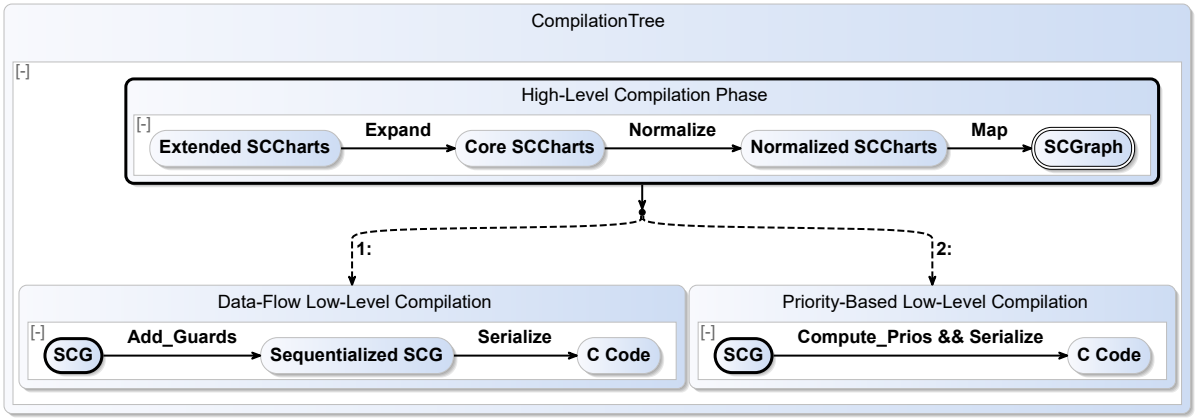


Figure 4.7. The compilation tree from Extended SCCharts to C code is grouped into a high-level phase and two different low-level phases [MSH14]

including all other transformations on which the selected feature depends, is displayed. Dependencies are resolved automatically.

Figure 4.6 illustrates the user story of the interactive model-based compilation of C code to SCCharts of KIELER. It presents one possible workbench layout for the model-based compilation in KIELER. The user interface (UI) of this tool is divided into four main views. After entering the C code in the editor in the top left corner, the user may select desired transformation steps by clicking on the feature groups of the KiCo compilation chain in the bottom view. All contained transformation steps inside the selected feature groups are executed, as well as all prior features. Alternatively, individual transformation steps can be selected by expanding the respective feature group. In this given example, the feature group *Core SCCharts* is selected. Thereafter, the middle view of the interface lets the user compare the intermediate compilation results on the right to the original extracted SCChart on the left. The created models are automatically arranged by integrated layout algorithms. The user can customize the layout of the graphical components by selecting provided *layout options* on the right hand side of the UI. Additional options for changing the appearances of SCCharts are located there as well.

After a general understanding of the usage of the incremental compilation feature has been given, it is necessary to take a closer look at the KiCo compilation chain itself. The transformation process from SCChart models to source code is split into two main parts, namely the *high-level compilation phase* and the *low-level compilation phase*. This segmentation is illustrated by Figure 4.7. The *high-level compilation phase* functions as a preprocessor for converting SCCharts into an intermediate format called the Sequentially Constructive Graph (SCG). This compilation phase is broken down into three main segments. Firstly, all Extended SCCharts features are *expanded*, resulting in a semantically equivalent Core SCCharts model. Secondly, the complexity of the transitions is reduced by *normalizing* the Core SCChart. The upper row of Figure 4.8 summarizes all permitted components of a normalized Core SCCharts. Other constructs than the ones presented are not allowed. A region may contain a thread

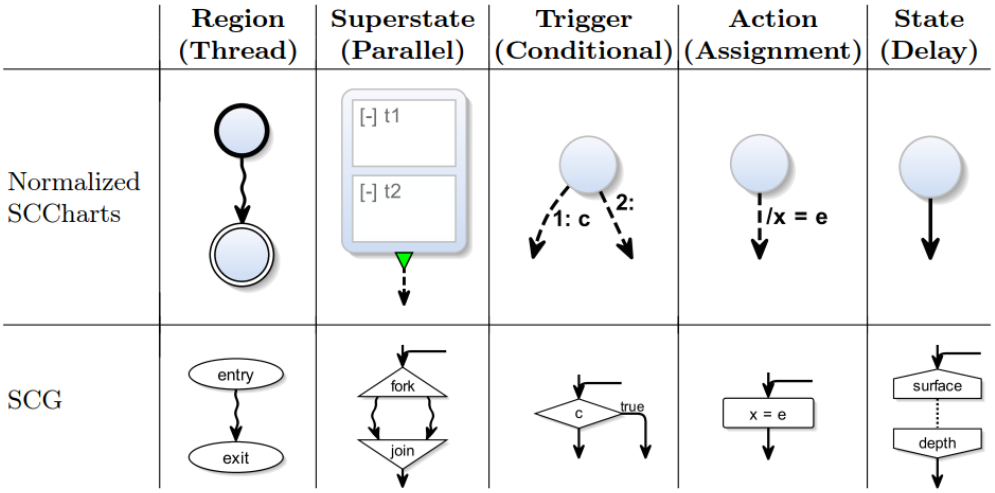
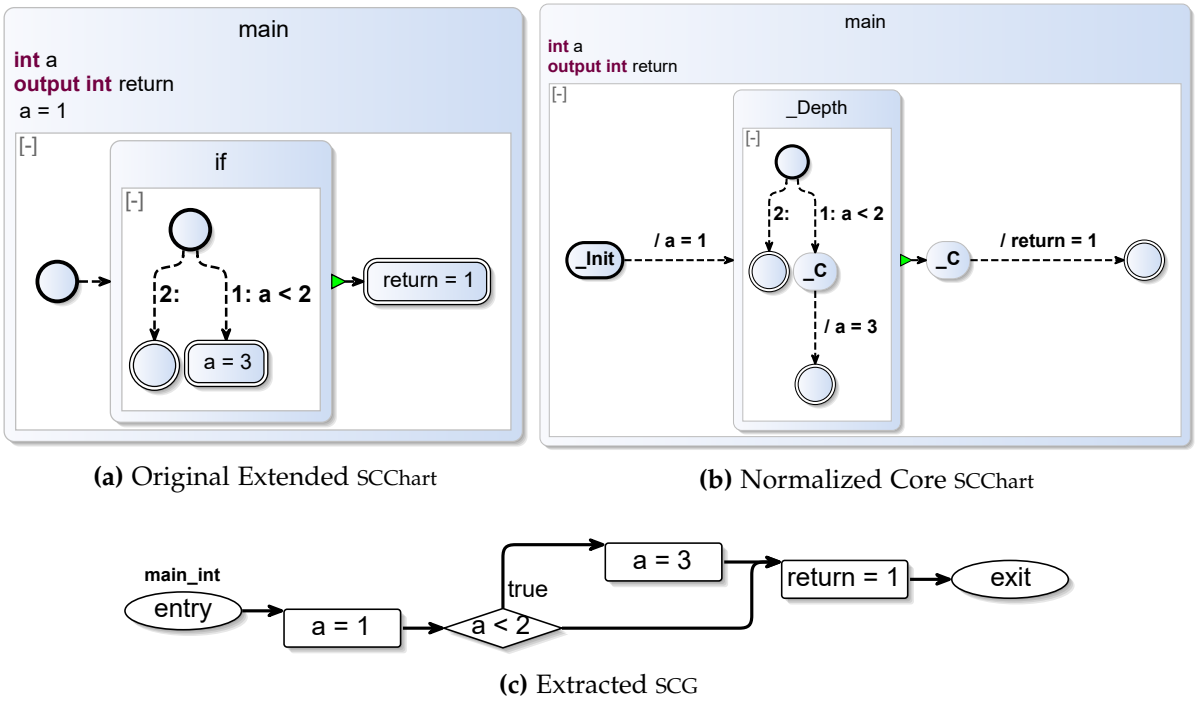


Figure 4.8. Overview of normalized SCCharts and SCG components [MSH14]

which consists of an initial state, a final state and connected states in between. Additionally, superstates with multiple parallel regions, including an associated termination transition, are allowed. A conditional if statement may be modeled as a state with two outgoing immediate transitions. The higher priority transition checks whether the condition c of the if statement is true. The lower priority transition represents the else-statement. Moreover, an assignment of an expression e to a variable x is handled by an immediate transition with no trigger. At last, a pause ensures the delay of following actions until the next tick.

As a last step, the resulting model is mapped into an SCG. The SCG represents the control-flow of an SCChart and allows for elaborate analyses such as data dependencies of shared variables between two threads. Figure 4.8 presents the mapping of each normalized component to the appropriate SCG component. An initial state of a thread is represented by an entry node while its final state is converted to an exit node. A fork node splits the control flow into parallel running threads, one for each region of a superstate. When all threads have terminated, they are joined together by a join node. Furthermore, an SCG denotes a conditional statement as a diamond-shaped node with its condition c inside. A variable assignment is depicted as a rectangle. Finally, a pause is illustrated as a dashed line connecting a surface node and a depth node inside an SCG. The following Figure 4.9 exemplifies the process of transforming an Extended SCChart to an SCG. As a first step, the Extended SCChart of Figure 4.9a is expanded and subsequently normalized into a Core SCChart, shown in Figure 4.9b. Thereafter, the resulting model is mapped to an appropriate SCG of Figure 4.9c.

The *low-level compilation phase* uses the SCG as the basis of the code generation process. For this phase, the KIELER project introduced two alternative approaches, the *data-flow* compilation approach and the *priority-based* compilation approach. The *data-flow* compilation approach is currently implemented and used in KIELER and is therefore also being used in this thesis.



(a) Original Extended SCChart

(b) Normalized Core SCChart

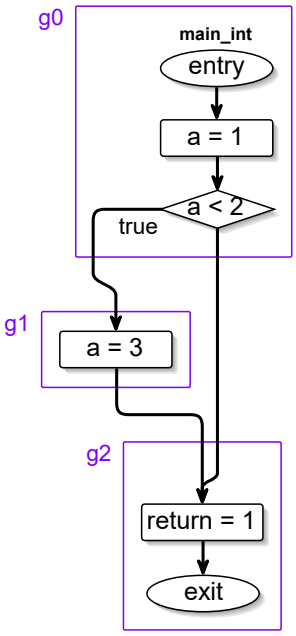
(c) Extracted SCG

Figure 4.9. Transforming an Extended SCChart to a Core SCChart and thereafter to an SCG

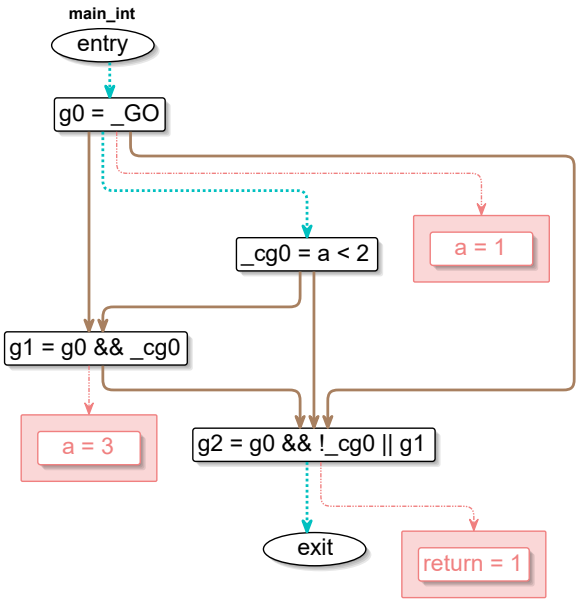
This approach requires the SCG to be data-flow acyclic. The goal of the data-flow approach is to generate a *netlist*, a description of the connectivity of an electronic circuit, where all components are always active. This netlist can then be simulated in software or hardware. For that, *guards* are introduced in order to regulate the control flow. Additionally, nodes are grouped together into *basic blocks*. A basic block may only be entered if the boolean value of its guard is true. The following Figure 4.10 continues the previous example of Figure 4.9. The SCG from Figure 4.9c is divided into three basic blocks. The first basic block represents the initialization of the variable `a`. The second basic block compounds the body of the if statement while the third and last basic block aggregates the end of the program. Each basic block has its own guard `g0`, `g1` and `g2`. As a next step, appropriate expressions are assigned to each of the guards. This assignment is shown in Figure 4.10b. The guard `g0` of the first basic block depends on the `_GO` start signal which marks the beginning of a program. The second guard `g1` is satisfied if `g0` and `_cg0` is true. The variable `_cg0` denotes the condition `a < 2` of the if statement. Consequently, the second basic block activates when the first basic block was entered and the conditional statement is true. The final basic block is either entered if `g0` is true but the condition of the if statement is not true or it is entered if `g1` is true.

In the final step of the M2M transformations, the scheduler orders the previously created guards in order to create a *sequentialized* SCG, which is shown by Figure 4.10c. Finally, the sequentialized program is ready for deployment and can be translated into various

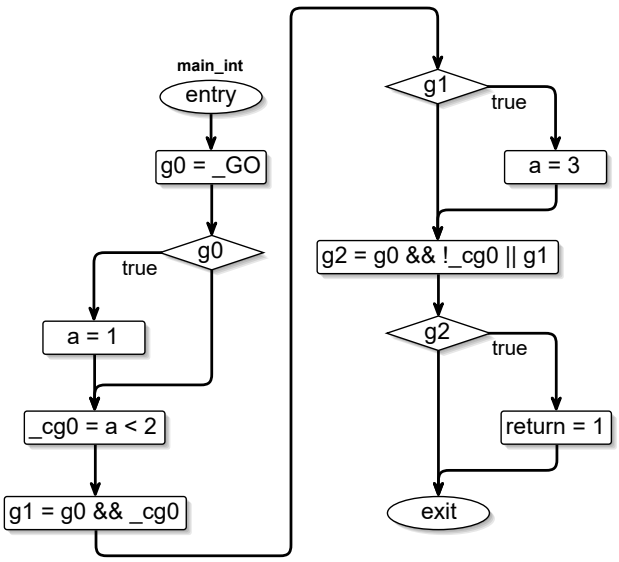
programming languages such as C. For a more detailed presentation of each step of the data-flow compilation approach, the reader is referred to [SMH15].



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



(b) Define guards of basic blocks



(c) Sequentialized SCG

Figure 4.10. Division of the SCG of Figure 4.9c into basic blocks with guards. Thereafter, the SCG is sequentialized.

The *priority-based* compilation approach specializes in generating software and lifts the restriction on the SCG to be data-flow acyclic. The reaction time of the software produced with the help of the data-flow approach is proportional to the size of the SCChart. In contrast, the reaction time of the priority-based approach only depends on the components which are active within a single tick. Consequently, this approach scales better with the increasing size of the models. However, the priority-based approach rejects some programs that the data-flow approach accepts [HDM+14b]. Since the priority-based approach is not part of this thesis, the reader is referred to [HDM+14b] for a more in-depth examination.

4.2 Generating ASTs from C code

Up to this point, all the necessary features were introduced which serve as the basis for this and the following sections. In summary, important SCCharts components were presented and illustrated such as different state and transition types. The sequential constructiveness was explained in order to understand the limitations of the low-level data-flow compilation approach which was discussed thereafter.

This section focuses on the T2M transformation of extracting an SCChart from C code. The approach of this thesis makes use of the Eclipse C/C++ Development Tooling (CDT) which was already presented in Section 3.1.1. First, the user enters the to be transformed C code into the provided CDT-Editor which is integrated into the workbench of the KIELER SCCharts tool. Then, the source code is parsed and an abstract syntax tree (AST) is created. This AST serves as the basis for the model extraction process. Each node of the AST represents a construct occurring in the source code. As an example, the listing in Figure 4.11a presents the C code corresponding to the previous SCChart shown in Figure 4.9a. With the help of the CDT plug-in, an appropriate AST is generated from this source code. In order to clarify the concept of ASTs, Figure 4.11b presents a graphical representation of the generated AST. The entire textual description of the AST can be found in the appendix, Figure A.1.

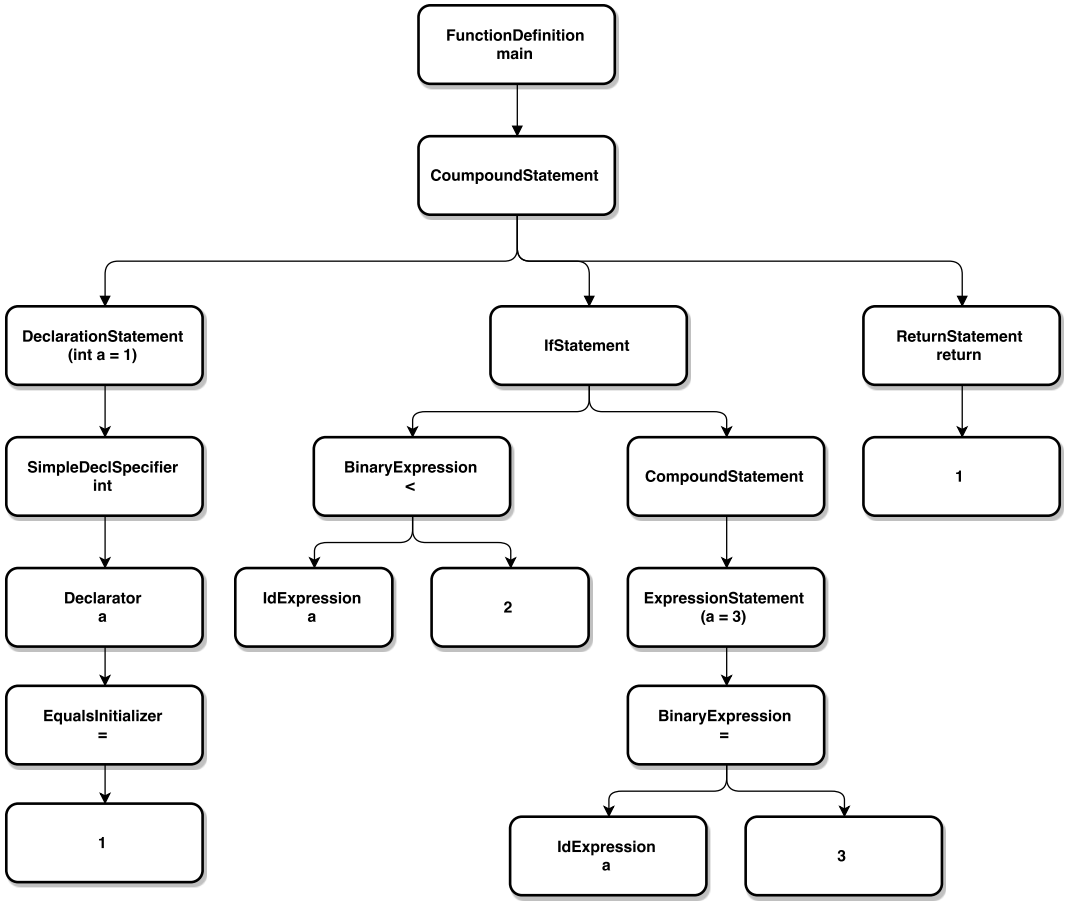
The AST is composed of different node types. First, the *FunctionDefinition* defines the `main` function which comprises a *CompoundStatement*. A CompoundStatement represents a code block surrounded by curly brackets *{ }*, in this case the body of `main`. At this point, the AST branches out into three sub-trees. The *DeclarationStatement* node initiates the declaration of the integer variable `a`. The *IfStatement* node parents a *BinaryExpression* node and another CompoundStatement node. The BinaryExpression forms the condition of the if statement, together with an *IdExpression* and an integer node. An IdExpression references an already defined variable. Next, the CompoundStatement defines the body of the if statement. Inside, an *ExpressionStatement* updates the value of the variable `a`. The final branch of the upper CompoundStatement is a *ReturnStatement*. Here, the integer value `1` is returned. This example shows only an excerpt from all AST node types. For a complete overview consider the CDT API[1]. As mentioned before, the AST serves as the basis of the model creation. Its structure

[1] http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html

and the provided information is used to identify the necessary model components and their positioning within the model. Therefore, the different nodes of the AST are converted one after another by traversing the syntax tree in depth-first order. Each node represents a model component that needs to be created. After every node has been visited, the T2M transformation process is complete. Thereafter, the resulting SCChart is a semantically equivalent visual representation of the original source code.

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

(a) C code example



(b) Visualization of the generated AST from the C code of Listing 4.11a

Figure 4.11. Representing C code with the help of ASCs

4.3 Creating SCCharts from an AST

After a model can be generated from an AST of a C program, the next objective to be addressed is to determine the way of visualizing source code. This section suggests ways of visualizing functions, variable declarations and assignments, as well as control structures and function calls, since these components form the basis of most of the C programs.

4.3.1 Functions

Each function defined inside the to be converted C code is represented by a root state of the extracted SCChart. The root state of a function f contains a *control flow region*. Inside this region, a mandatory anonymous initial state and at least one final state is located. Outgoing from this initial state, the model is created. The arguments of a function are declared as input variables. Additionally, non-void functions declare an output variable return of appropriate type.

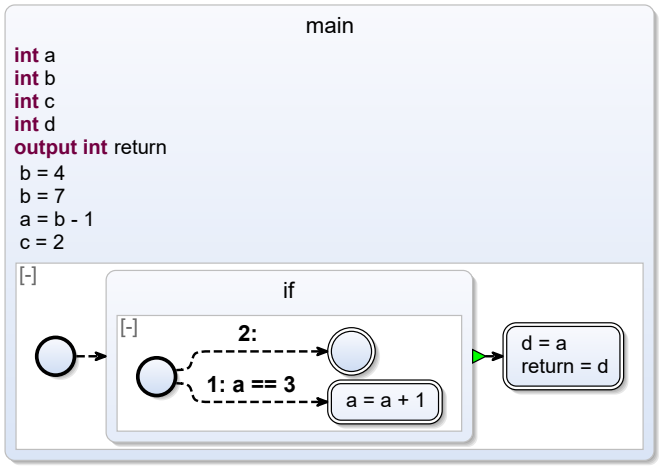
4.3.2 Variable Declarations and Assignments

Variable declarations are defined in the declaration interface of the appropriate root state. It is also possible to declare local variables inside nested superstates of the root state. Hence, the local declaration is assigned to the declaration interface of the parent superstate. The declaration of local variables is considered in more detail in the next subsection.

A variable initialization is either handled by an entry action of the root state or an entry action of a separate state. Note that the entry action label entry is not displayed for entry actions. This aims to enhance the clearness and readability of the model. Variables that are initialized at the very beginning of the program, more precisely immediately after the initial

```
1  int main() {
2      int a;
3      int b = 4;
4      b = 7;
5      a = b − 1;
6      if(a == 3) {
7          a = a + 1;
8      }
9      int c = 2;
10     int d = a;
11     return d;
12 }
```

(a) C code example for variable declarations and assignments



(b) Representing model of variable declarations and assignments

Figure 4.12. Visual representation of variable declarations and assignments

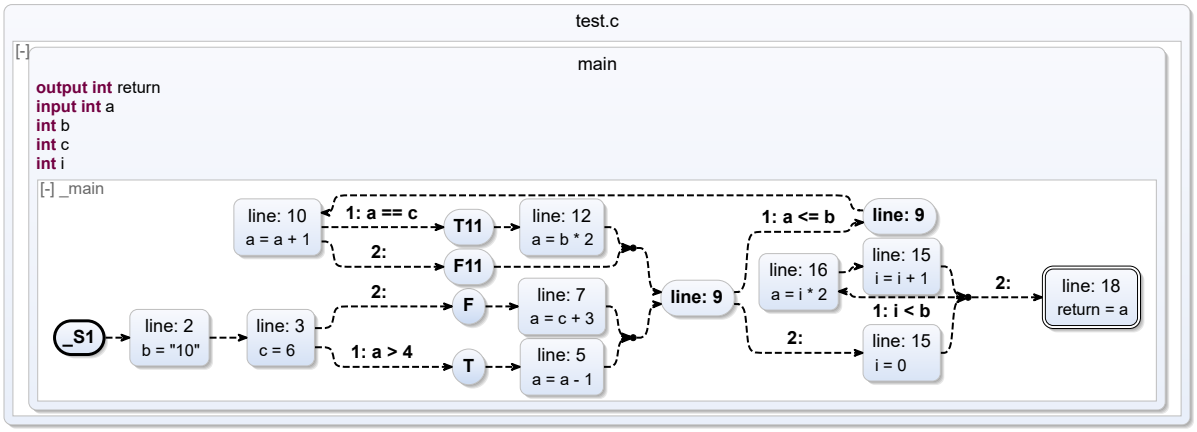
state of the region, are handled by introducing entry actions in the declaration interface of the root state. Thereby, this procedure reduces the number of overall states of the SCChart. Comparing the listing in Figure 4.12a to Figure 4.12b, the declarations and initializations of line 2 to line 5 are modeled as entry actions of the root state. Since multiple entry actions are performed in sequential order, the variable `b` is first initialized with the value of `4` and is then updated to `7`. Therefore, the entry action `a = b - 1` assigns the correct value of `6` to the variable `a`. If an initialization occurs after a state other than the initial state, an additional state is created and an appropriate entry action is assigned to its declaration interface. An exception to this proceeding is a non-initial variable declaration with an immediate non-dependent initialization, all in one statement. Therefore, the code of line 9 `int c = 2` can be transformed to an entry action of the root state. Since the declaration of line 10 `int d = a` depends on the value of `a`, it cannot be declared as an entry action of the root state since the initial value of the variable `a` could change during the execution of the program. Here, this is the case when `a` gets incremented inside the if statement.

4.3.3 Control Structures

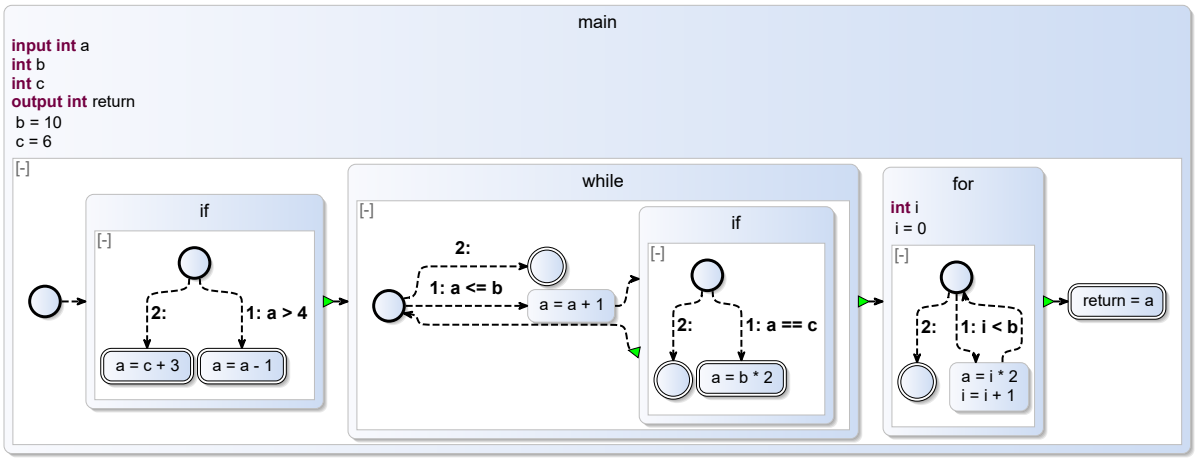
As an introductory example, Figure 4.13b presents an SCChart visualizing the C code of the listing in Figure 4.13a. This SCChart was extracted by the prototype of the model-based compilation of legacy C code by Smyth and Olsson [Ols16]. The SCChart below of Figure 4.13c shows the result of the overhauled model extraction process of this thesis. This comparison intents to emphasize the importance of developing a clear and structured visualization of control structures. The representation of the prototype does not yet support the reading flow of the model as intended and obscures the functionality of the converted source code. Furthermore, the reader can get a first impression of the structural changes of the visualization of SCCharts. When taking a closer look at the former visualization of Figure 4.13b, the control flow of the program is clear when new variables are initialized. Furthermore, the beginning of the first if statement can be identified easily. However, when reaching the node labeled `line: 9`, the reader of the model needs to take a closer look in order to follow each step further of the program. Since the boundaries of control structures are not clear on first sight, the visualization of the prototype makes it difficult to understand the purpose or functionality of the converted C code. Especially multiple nested control structures can obscure the structure of the SCChart. In order to clarify the bounds of control structures, each of them is embedded into its own state. This way, the extracted model becomes better organized. Thereafter, one can clearly identify what the states and transitions are associated with. The inner behavior of control structures, that 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 in order to understand the essence of the depicted function or program. On closer inspection, the reader may notice that control structure states share a similar layout. Every control structure has an initial state, at least one final node and additional states which represent the body of the control structure. The transitions leaving the initial state check whether the condition of the control structure is met.

```c
int main(int a) {
    int b = 10;
    int c = 6;
    if (a > 4) {
        a = a − 1;
    } else {
        a = c + 3;
    }
    while (a <= b) {
        a = a + 1;
        if (a == c) {
            a = b ∗ 2;
        }
    }
    for (int i = 0; i < b; i = i + 1) {
        a = i ∗ 2;
    }
    return a;
}
```

(a) Introductory C code example of control structures



(b) Extracted SCChart model of the old prototype



(c) Extracted SCChart model with the new visual representation of control structures

Figure 4.13. Comparison of different visualizations of C code

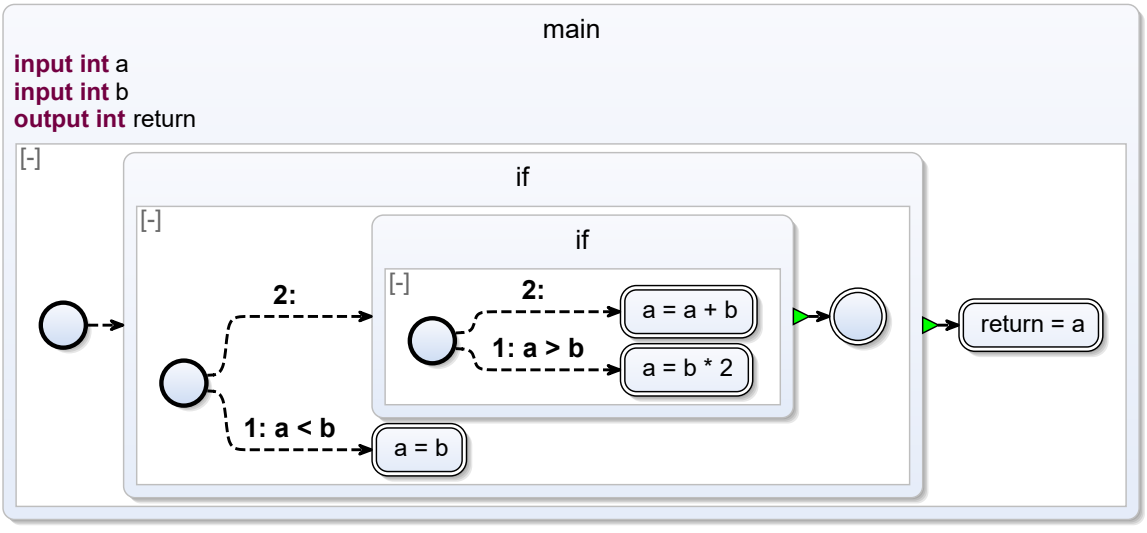
The condition is represented by the trigger of the respective transition. If the condition is met, the transition is taken and the body of the control structure is executed. If the condition is not satisfied, the final state of the superstate is reached in order to leave the control structure. A termination transition ensures that the following states are only reached after the state of the control structure terminates. Therefore, a correct sequence of events is assured. The exact structure of each different control structure type is presented in the following paragraphs.

The if-then(-else) Statement

The first control structure to look at is the if-then(-else) statement. Figure 4.14b visualizes the C code of Listing 4.14a which consists of two condition checks `a < b` and `a > b`. Immediately after entering the initial state of the first if state, the condition `a < b` is checked. If the condition is satisfied, `a` is set to `b`. The next line to be executed is `return a`. Consequently, the state of this variable assignment is final and the control structure is left via a termination transition.

```
1  int main(int a, int b) {
2      if (a < b) {
3          a = b;
4      } else if (a > b) {
5          a = b * 2;
6      } else {
7          a = a + b;
8      }
9      return a;
10 }
```

(a) C code example of nested if statements



(b) Extracted SCChart model

Figure 4.14. Visual representation of if-then-else control structures

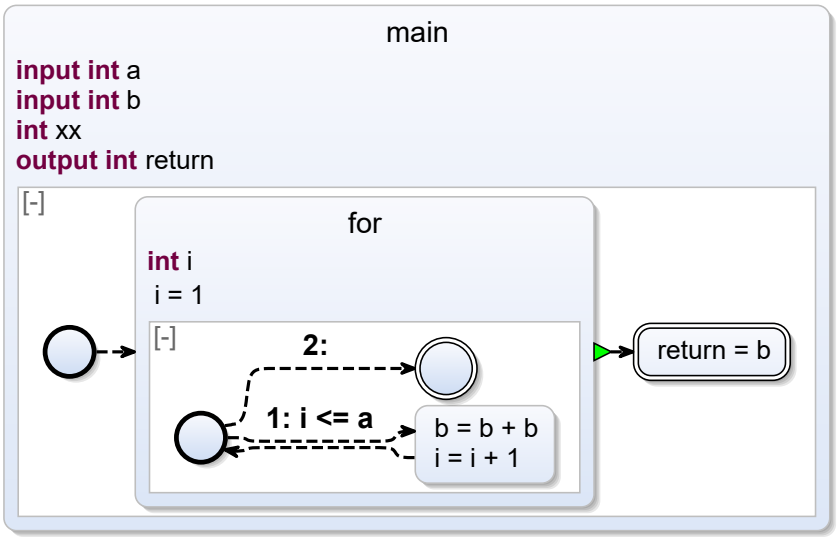
If the first condition is not met, the else state becomes active. Since the else state consists of an additional if statement, namely if (a > b), a second if state is nested inside the first. Here, the proceeding is repeated. After the inner if state terminates, the outer if state terminates as well. Finally, a is returned and the main state has finished its execution.

The for Statement

Next, the model creation of a for loop is introduced and presented by Figure 4.15. Additionally to the similar layout of every control structure, a local counter variable i is declared and is set to its initial value when entering the for state. Note that the declaration is not added to the declaration interface of the root state, but to the parent state of the loop. Therefore, the local variable i is only known inside the loop state. If the loop condition is satisfied, the body of the loop is executed. Having completed these measures, the counter variable i is incremented before returning to the initial state again. When the condition is not met anymore, the transition to the final state is taken and the loop state terminates. Afterwards, the rest of the program is executed.

```
1 int main(int a, int b) {
2
3     for (int i = 1; i <= a; i = i + 1) {
4         b = b + b;
5     }
6     return b;
7 }
```

(a) C code example of a for loop



(b) Extracted SCChart model

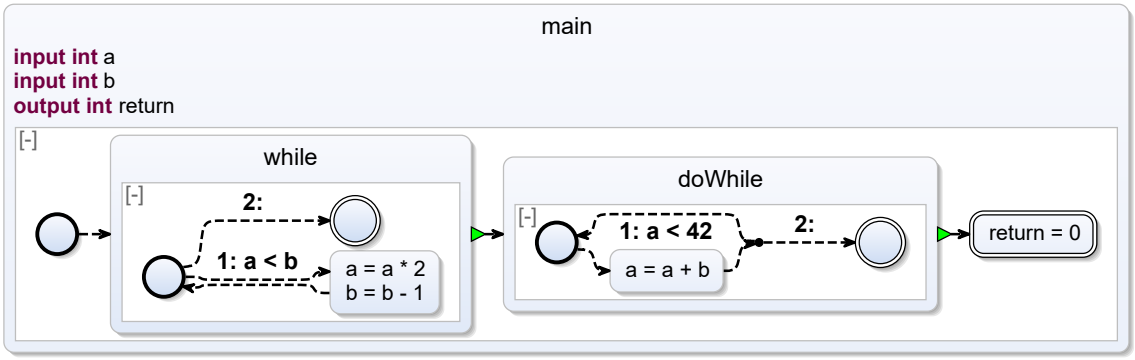
Figure 4.15. Visual representation of a for loop

The (do-)while Statement

The while statement as well as the do-while statement behave similarly to the for loop. The main difference is the absence of a typical counter variable which is declared and initialized as an entry action of the loop. Consequently, we do not require a mandatory increment or decrement when returning to the initial state. Figure 4.16 shows an example of a representation of a while, as well as a do-while loop.

```
1  int main(int a, int b) {
2      while (a < b) {
3          a = a * 2;
4          b = b - 1;
5      }
6      do {
7          a = a + b;
8      } while (a < 42);
9      return 0;
10 }
```

(a) C code example of a while and a do-while loop



(b) Extracted SCChart model

Figure 4.16. Visual representation of a while loop and a do-while loop

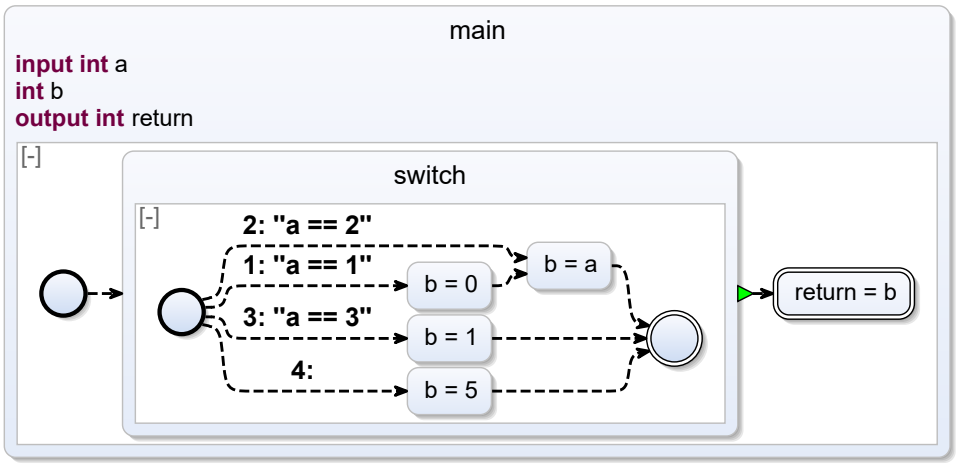
The switch Statement

The already implemented prototype of the model extraction in KIELER offers a possibility to convert switch-case statements to SCCharts and the other way around [Ols16, pp.31-32]. However, every case of the code to be converted requires a break line. The implementation of this thesis allows a fall-through from one case to another. A fall-through occurs when a break statement is missing at the end of a case. The control flow falls through onto the next case without checking its condition. This behavior is exemplified by Listing 4.17a. If the variable `a` equals `1` then `b` is set to `0`. Due to a missing break statement, the second case is executed right after the first case. Therefore, the variable `b` gets overwritten with the value of `a`. At the end

of case 2, a break statement prevents further fall-through and causes the switch statement to be left. The program ends with the return of the value of b. This fall-through can also be seen in Figure 4.16b. For each case, a transition with the corresponding condition leaves the initial state. After entering the switch state, each case transition is checked in order of the C code with the help of appropriate transition priorities. The transition leaving the initial state without a condition represents the default case. The fall-through is visualized by the unconditioned transition connecting the body of the first case to the body of the second case. Consequently, the second case is executed right after the first case. After a case is executed and no fall-through happens or in case of a default case, the last state of the case is connected to a connector state. The final state is reached via this connector state in order to leave the switch state.

```
1  int main(int a) {
2      int b;
3      switch(a) {
4          case 1:
5              b = 0;
6          case 2:
7              b = a;
8              break;
9          case 3:
10             b = 1
11             break;
12         default:
13             b = 5;
14     }
15     return b;
16 }
```

(a) C code example of a switch statement

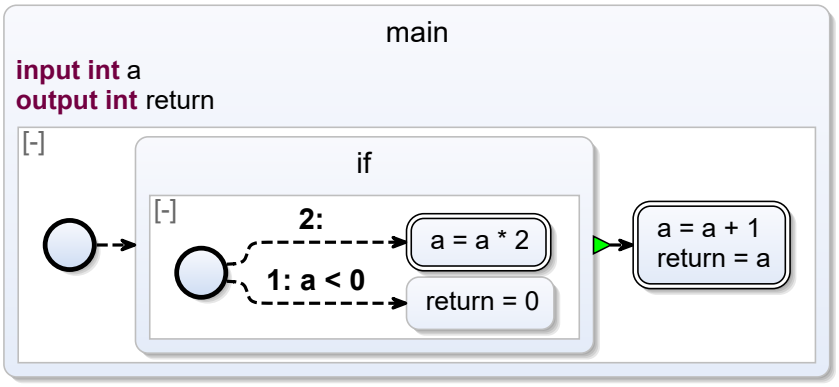


(b) Extracted SCChart model

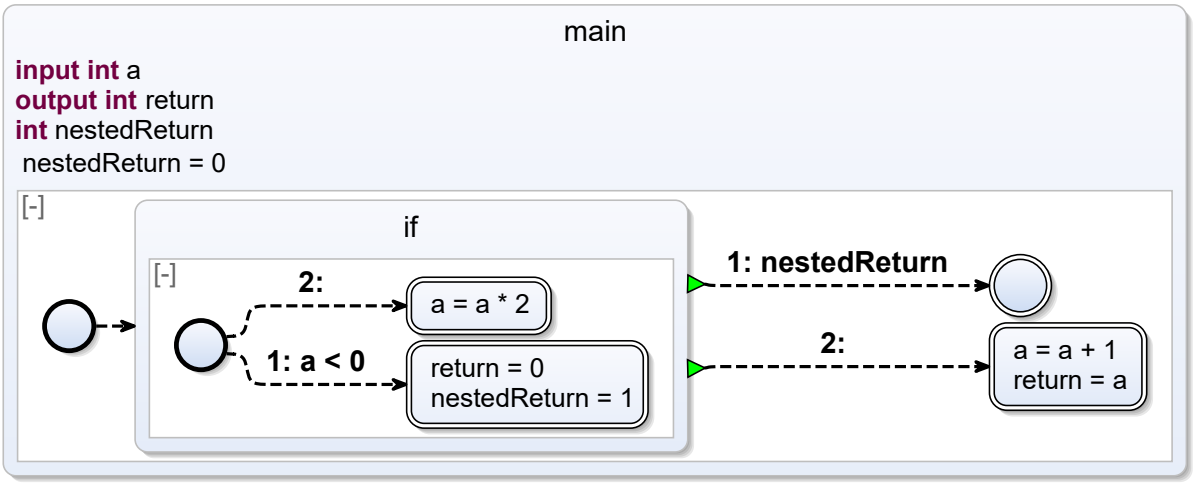
Figure 4.17. Visual representation of a switch statement

```c
int main (int a) {
    if (a < 0) {
        return 0;
    } else {
        a = a * 2;
    }
    a = a + 1;
    return a;
}
```

(a) C code example of a nested return statement



(b) Extracted SCChart model with an incorrect handling of nested return statements



(c) Extracted SCChart model that preserves the semantics of the program

Figure 4.18. Visual representation of nested return statements

Nested Return Statements

Since this approach of visualizing C code introduces possible multiple hierarchy levels when dealing with control structures, nested return statements inside these control structures need to be handled. For example, consider Figure 4.18b. If `a` is less than `0`, the program should return `0` before terminating. Hence, no other statement should be executed. Therefore, a single final state, which sets the output variable `return` to `0`, is not enough. This way, only the if state is left, `a` is incremented by one. Thereafter, the final state is reached and the `return` variable is overwritten with the value of `a`. Figure 4.18c presents a way of visualizing the code that preserves the semantics of the program. In case of a nested return statement, a integer variable `nestedReturn` is declared and is set to `0` when entering `main`. Inside the state of the nested return statement, an additional entry action sets the variable `nestedReturn` to `1`. In order to terminate not only the control structure state but also the entire `main` root state, an alternative exit transition is added. After leaving the state in which the nested return statement is located, this alternative exit transition can only be taken if the nested return statement was reached before. Otherwise, the lower-priority transition is taken in order to continue the execution of the program.

4.3.4 Function Calls

The process of developing a representation of function calls went through multiple iterations. Initially, the prototype of the model-based compilation of legacy C programs handled function calls by using host code. There was no visual information about the behavior of called functions. Therefore, the user did not get sufficient information to fully understand the effect or use of a called function.

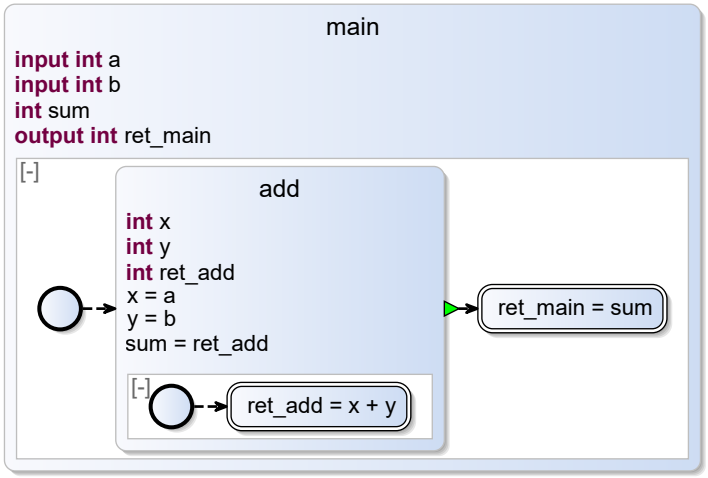
Figure 4.19 visualizes the first solution approach. The idea was to nest the called function inside a separate state within the state of the caller. When the state of called function `add` is entered, the passed arguments `a` and `b` are copied to local variables `x` and `y`. These variables are used for internal computations. Additionally, the original values of `a` and `b` do not get lost for further calculations inside the main state. After the sum of the two integer values is assigned to the local return variable `ret_add`, an exit action copies this returned value to the variable `sum`. This approach solves the problem of the missing visual information of the behavior of called functions. However, the suggested approach leads to problems when multiple calls of the same function occur. A separate state would be created for each function call. Moreover, 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 redundant code blocks. Consequently, this approach is not viable.

A second approach outsources the state of the called function into its own state located outside of the caller state. Figure 4.20 shows an SCChart that represents the code of Listing 4.19a as well. The first difference to be noticed is that the `main` state is not the root state anymore, but it is nested inside a newly introduced root state. The reasoning behind this change is the necessity for global variables that are known to both the `main` state and the `add` state. This

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

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

(a) C code example



(b) Extracted SCChart model

Figure 4.19. Nested visual representation of a function call

includes all variables that are used for a function call. These variable definitions are contained inside the declaration interface of the new root state. Furthermore, a variable executeFuncCall is declared which signals the beginning of the function call. When the function call occurs, the action of the transition leading to the state funcCall sets executeFuncCall to 1. This activates a weak abort transition in order to enter the add state. It is of great importance that the outgoing transition of funcCall is delayed. Hence, the transition cannot become active in the same tick in which the state funcCall is entered. This ensures that the weak abort transition to the state add gets taken. After the called state terminates, the result is written to the global variable sum and a history transition returns the control flow to the state funcCall. Additionally, the variable executeFuncCall is set to 0 to prevent further unwanted function calls. Finally, the final state is entered and the output variable is set.

This approach reduces the inflation of the SCChart in the case of multiple function calls while still preserving the visual information of the behavior of the called function. Every function call still has its own state inside the caller's state though without any inner behavior. Furthermore, the generated C code does not contain duplicate code blocks. However, history transitions greatly increase the overall state space of the model since all sub-states of every inactive state need to be remembered. Therefore, the history transition should be avoided if possible to provide a light-weight SCChart model. It is especially essential when modeling systems with limited memory capacity such as embedded systems.

The final approach is presented in Figure 4.21 and makes use of reference states. The state of the function add is added as a second root state to the SCChart. The reference state Call maps the passed arguments a and b to the input variables x and y of the called state. Additionally, the variable sum is mapped to the output return of state add. This solution provides the visual representation of the inner behavior of the called functions, as well as keeps the inflation of the SCChart to a minimum. Even though one function call still has one separate state, its

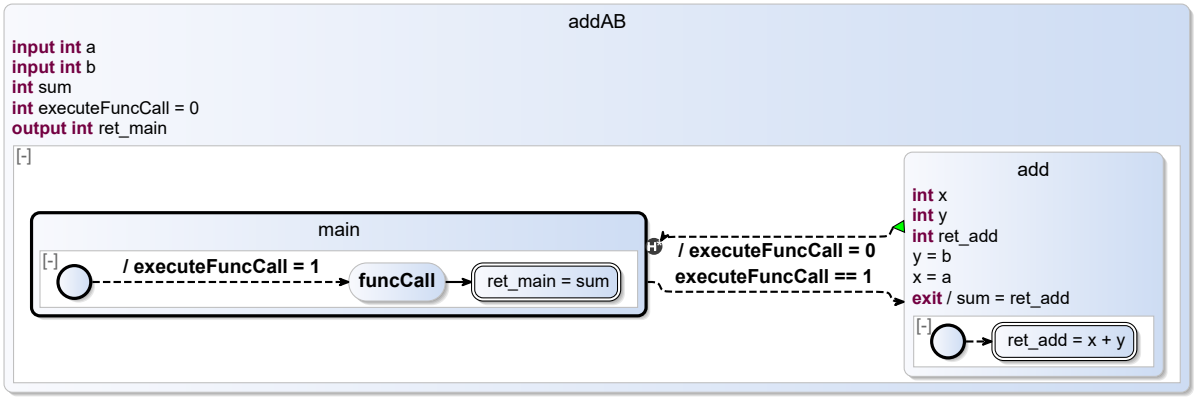


Figure 4.20. Visual representation of a function call with an outsourced called function state

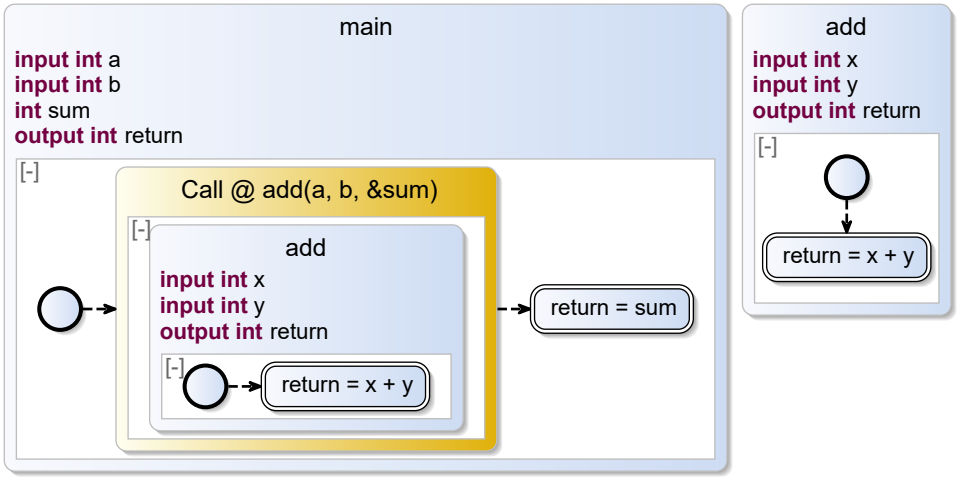
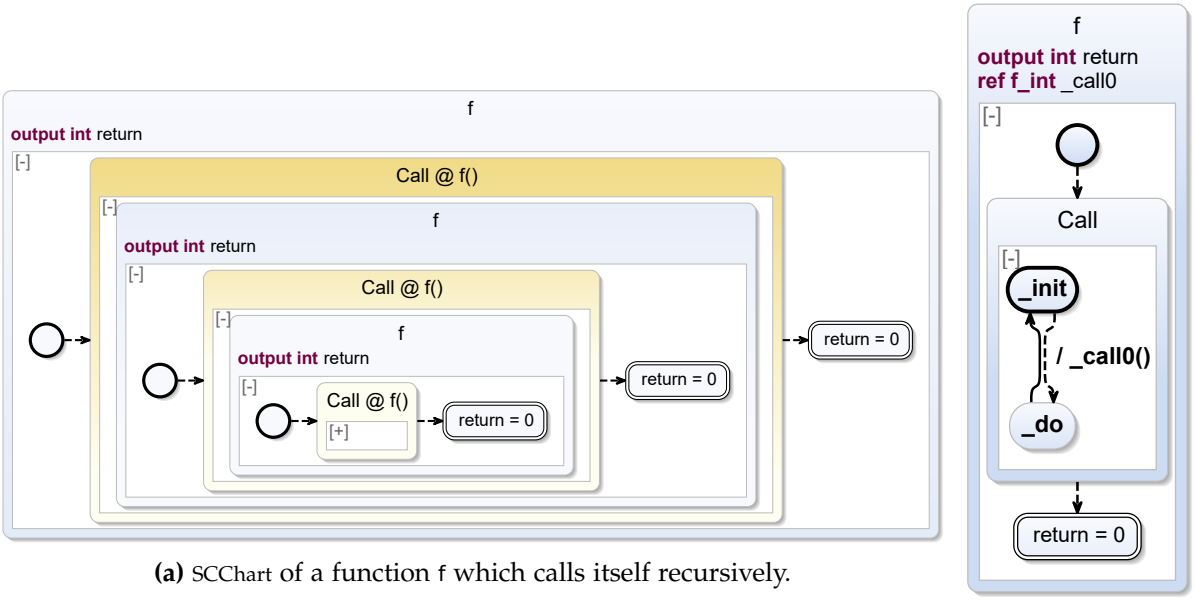


Figure 4.21. Visual representation of a function call by using a referenced state

inner behavior can be hidden without losing any information. The additional root state of the function still provides the necessary details. When the SCChart is transformed into C code, the referenced function state is expanded only once, even when it is called multiple times. Consequently, the generated C code defines functions only once as desired and the creation of duplicate source code is prevented.

The main problem with this approach is the inability to handle recursive function calls. As an example, consider Figure 4.22a. The SCChart contains a function `f` that recursively calls itself again. The final state is never reached since each reference state contains another reference state. However, after the M2M transformation step *Expansion* of the KiCo compilation chain, only one function call remains and the infinite loop of recursive calls is discarded. The resulting SCChart is displayed in Figure 4.22b. The reason for this is the single expansion of a reference state. Consequently, the single expansion feature, which optimizes the generated source code greatly while not dealing with recursive functions, prevents the transformation



(a) SCChart of a function f which calls itself recursively.

(b) Corresponding Expanded SCChart

Figure 4.22. Single expansion of a recursive reference state

of recursive functions to work as intended. This challenge of adjusting the transformation for recursive function calls remains future work. Additionally, the concept of using a weak abort as the outgoing transition of the reference state only supports called root states that do not contain any delayed transitions. Using an outgoing termination transition solves this issue. However, the usage of a termination transition leads to another critical problem when expanding the reference state. In this scenario, the outgoing transition of the state Call of Figure 4.22b is a termination transition. Consequently, the state Call is never left since no final state is created. Future work should solve this problem and provide a support of function calls with delayed transitions.

Figure 4.23 shows a first idea of how to handle recursive function calls. By falling back onto the usage of history transitions, a recursive call can be modeled like a while loop. In this given example, the accumulator technique is used in order to store the result of one recursive iteration. However, the usage of history transitions is impracticable, this way of transforming is not implemented and should only serve as an impulse for further development.

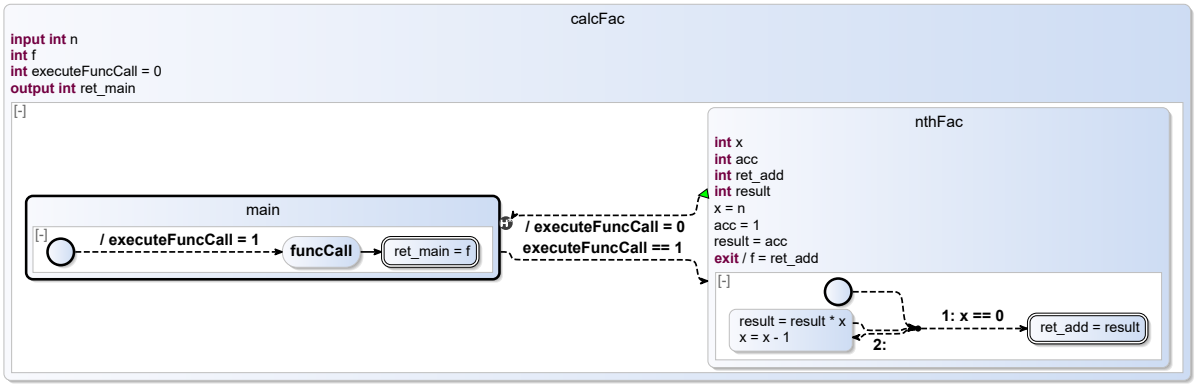
4.4 Compiling the extracted SCChart

After the model mapping of C code has been determined, the extracted SCChart is ready for further compilations by the KiCo compilation chain. Earlier, it was already stated that this thesis uses the data-flow low-level compilation approach. Since this approach does not allow

```c
int main(int n) {
    int f;
    f = fac(n, 1);
    return f;
}

int fac(int x, int acc) {
    int result = acc;
    if (x == 0) {
        return result;
    } else {
        result = fac(x − 1, x ∗ acc);
        return result;
    }
}
```

(a) C code example of a program which calculates the n-th factorial



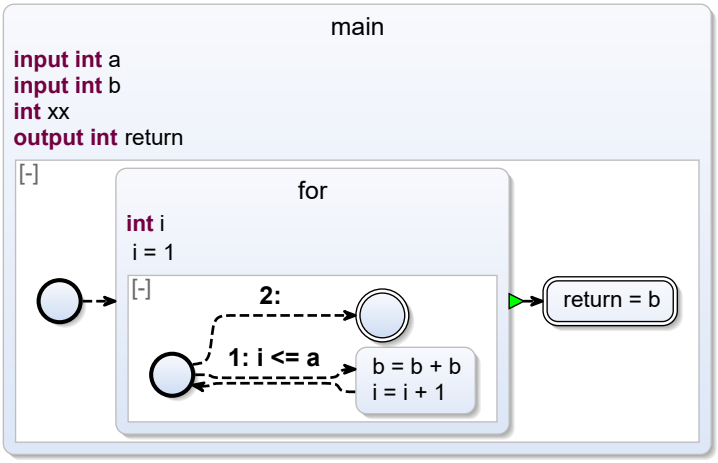
(b) Manually created representing SCChart model

Figure 4.23. Concept idea for handling recursive function calls

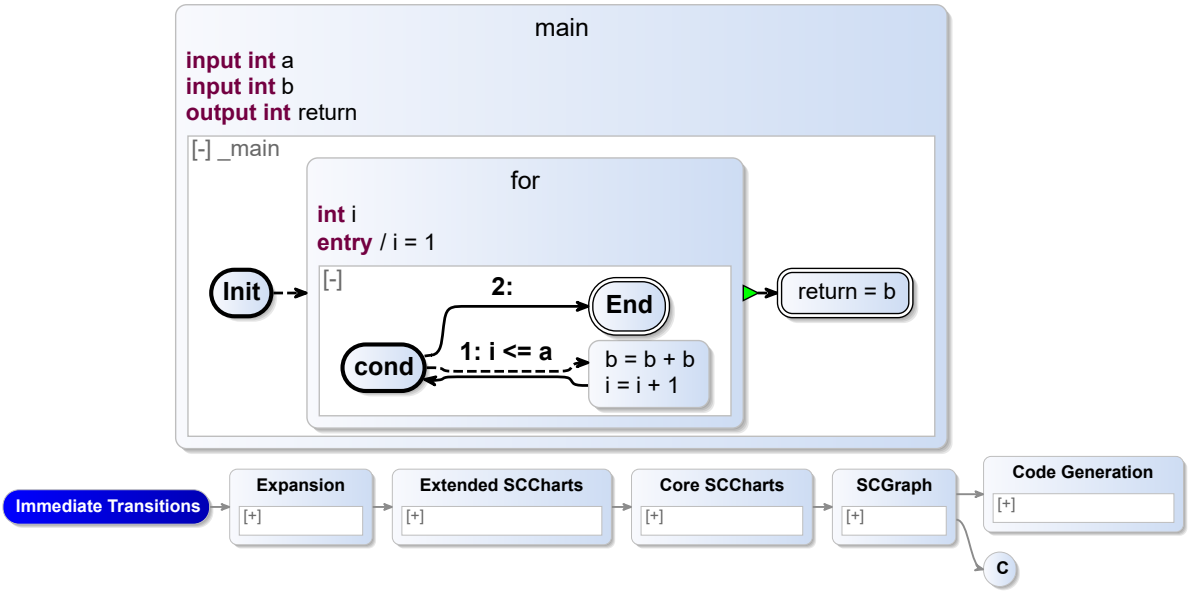
for instantaneous loops, the created SCChart needs to be checked for and freed from them before continuing.

Loops occur when the control flow of the program jumps from one code line to another previous line. Instantaneous loops are loops where one entire pass happens in an instant. Consequently, multiple passes through a loop happen instantaneously as well. This implies that an infinite amount of iterations of a loop can happen during one tick. This nature of an instant, non-time-consuming reaction is realized by an immediate transition inside an SCChart. Consequently, a loop is instantaneous if it only consists of immediate transitions.

As a next step, it is clarified when loops occur in the control flow of the program and therefore occur in the extracted SCChart. One possible source of loops are control structures such as for or while. The control flow always jumps back to the point where the exit condition of the loop is checked. In the SCChart this point is defined as the initial state of a loop.



(a) The extracted SCChart model still contains an instantaneous loop



(b) The instantaneous loop has been resolved by the *Immediate Transitions* transformation

Figure 4.24. Using the *Immediate Transitions* transformation step in order to avoid instantaneous loop

By delaying the incoming transitions which is taken when the condition for the next iteration is checked, it is guaranteed that each pass of a loop contains at least one delayed transition. In case of a do-while loop, the outgoing transitions of the initial state is delayed. This suffices in order to ensure non-instantaneousness. As a result, instantaneous loops are avoided for each of the control structures presented earlier. As already mentioned, when using the data-flow compilation approach for the code generation, the SCChart models need to be checked for and freed from instantaneous loops. This is realized by subjoining an addi-

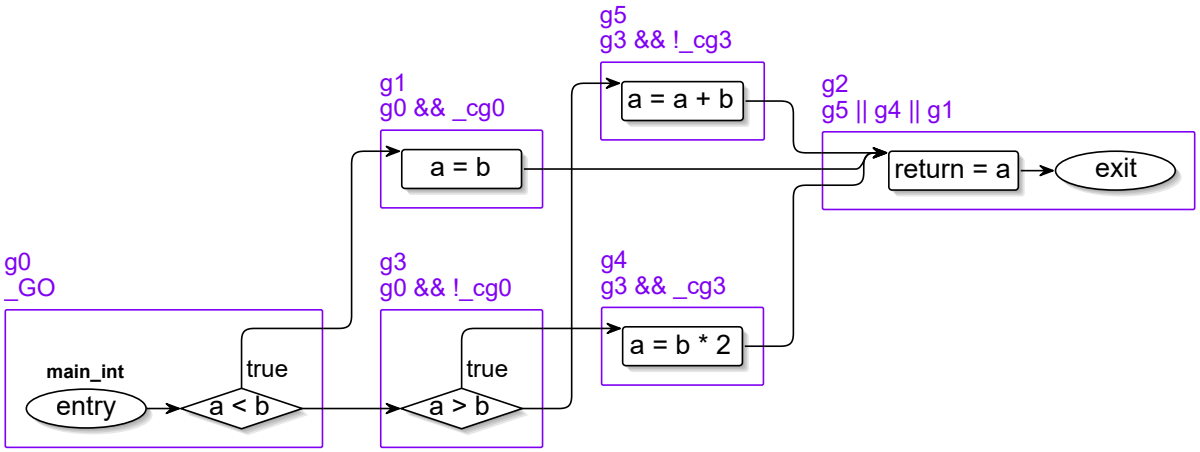
tional compilation step to the beginning of the KiCo compilation chain. This transformation delays marked immediate transitions and therefore, resolves instantaneous loops. Figure 4.24 provides an example which shows the effect of the newly developed transformation step Immediate Transitions. After its activation, the instantaneous loop inside the for loop is resolved.

Even though delaying transitions is necessary, it changes the semantics of the SCChart and therefore the semantics of the generated C code. Hence, the data-flow approach is not optimal when looking at the immediate execution of a program. However, it is possible to optimize the execution time of loops at the expense of required storage space. Techniques like *Loop Unrolling* reduce the number of iterations by replicating the body of the loop a number of times. A loop with n iterations can be completely unrolled by copying the body n-times. Naturally, the number of iterations must be know prior to its utilization. As an alternative, a priority-based low-level compilation approach is currently being implemented by the Real-Time and Embedded Systems Research Group into KIELER. This approach does not need the additional *Immediate Transition* transformation since it allows instantaneous loops. Since the Immediate Transitions transformation is a separate link in the compilation chain, it can be easily removed when it is not needed anymore.

4.4.1 Extracted C Code

After the high-level and low-level compilations are completed, the resulting C code can be examined by the user. Listing 4.25b shows an excerpt of the C code which was generated from the SCChart shown in Figure 4.14. Its generated SCG is presented in Figure 4.25a. The complete generated source code can be found in the appendix in Figure B.1a and Figure B.1b. For each root state of the SCChart, a struct TickData, a function tick, reset and tickLogic is generated. The generated data type TickData contains all necessary guard variables, input as well as output variables for its execution. The reset function sets the _GO variable to 1 to signal the beginning of the program. Additionally all *pre-variables*, which store the value of an associated variable from the previous tick, are initialized with 0. Next, the tick function simulates one tick and calls tickLogic. Afterwards, _GO is reset to 0 and the pre-variables are updated after the tick. The function TickLogic contains the execution logic of the program. When taking a closer look at the generated C code in Listing 4.25b, one can see the declarations of the guards from line 2 to line 6, line 10, line 14 and 18. The if statements in line 7, 11, 15 and 19 control the access to the respective basic blocks.

Function calls are realized by calling the tick function of the called SCChart. Listing 4.26 shows an excerpt from the generated C code of the SCChart shown in Figure 4.21. In addition to the guard variables and input and output variables, the data type TickData of the main function also contains a TickData1 variable _call0. TickData1 resembles the generated struct for the root state of the function add. The function call occurs in line 22 to line 25. Firstly, the passed arguments a and b of the function call are copied to the variables x and y of the called function add. Afterwards, the tick1 function of the callee is executed in order to compute the result of the expression a + b. Finally, the result is copied to the specified return variable sum



(a) The generated SCG of the SCChart shown in Figure 4.14

```
1  void tickLogic(TickData *data) {
2    data->g0 = data->_GO;
3    data->_cg0 = data->a < data->b;
4    data->g3 = data->g0 && !data->_cg0;
5    data->_cg3 = data->a > data->b;
6    data->g4 = data->g3 && data->_cg3;
7    if (data->g4) {
8      data->a = data->b * 2;
9    }
10   data->g1 = data->g0 && data->_cg0;
11   if (data->g1) {
12     data->a = data->b;
13   }
14   data->g5 = data->g3 && !data->_cg3;
15   if (data->g5) {
16     data->a = data->a + data->b;
17   }
18   data->g2 = data->g1 || data->g5 || data->g4;
19   if (data->g2) {
20     data->ret = data->a;
21   }
22 }
```

(b) Excerpt from the function tickLogic of the generated C code of the SCChart shown in Figure 4.14

Figure 4.25. Generated SCG and C code from the SCChart shown in Figure 4.14

of the main function. In case of multiple function calls, the appropriate tick function is called again.

```
typedef struct {
  char g0;
  char _GO;
  char g3;
  char g4;
  char pg3;
  char g1;
  TickData1 _call0;
  char _cg1;
  char g2;
  char ret;
  char sum;
  char a;
  char b;
} TickData;

void tickLogic(TickData *data) {
  data->g0 = data->_GO;
  data->g4 = data->pg3;
  data->g1 = data->g0 || data->g4;
  if (data->g1) {
    data->_call0.x = data->a;
    data->_call0.y = data->b;
    tick1(&data->_call0);
    data->sum = data->_call0.ret;
  }
  data->_cg1 = 1;
  data->g2 = data->g1 && data->_cg1;
  if (data->g2) {
    data->return = data->sum;
  }
}
```

Figure 4.26. Excerpt from the generated C code of the SCChart shown in Figure 4.21

Chapter 5

Implementation of the Model-based Compilation of Legacy C Programs

Based on the concepts introduced in Chapter 4, this chapter presents the implementation of the model-based compilation of legacy C programs of this thesis in the KIELER project. After an overview of the project structure is given to the reader, the two main parts of the implementation are discussed. Firstly, the *CDTProcessor* and its M2M transformations are presented in Section 5.1. Thereafter, the *Immediate Transitions Transformation* described in Section 5.2. The implementation is located in the project folder `de.cau.cs.kieler.c.sccharts`. The project is divided into five sub-packages:

`de.cau.cs.kieler.c.sccharts`: This package contains the centerpiece of the model-based compilation, namely the file *CDTProcessor.xtend*. It defines the M2M transformations for the code visualization and creates the extracted SCChart model from the C code. The next Section 5.1 will explain its components in detail.

`de.cau.cs.kieler.c.sccharts.controller`: This package contains the *CDTUpdateController* which was implemented by Lars Olsson. It is already used for the prototype of the model-based compilation of KIELER. This class realizes a listener for the CDT-Editor which is used to receive the user's C code. After the user input is saved, a *CEditor instance* is created and is passed to the transformation step in order to generate an AST. Afterwards, this AST is used to construct the representing SCChart.

`de.cau.cs.kieler.c.sccharts.handler`: This package contains the *CFileTransformHandler*. Like the CDTUpdateController, this Java file was also part of the model-based compilation prototype. As the name already suggests, its task is to handle the transformation of the input C code. After the model is extracted from the source code, this handler manages the resource allocation and the data storage.

`de.cau.cs.kieler.c.sccharts.synthesis`: This package contains the *HideEntryKeywordHook*. This synthesis, which was implemented by Alexander Schulz-Rosengarten, is used to hide the label of an entry action.

`de.cau.cs.kieler.c.sccharts.transformation`: This package contains the *CbasedSCChartFeature* and the *ImmTransTransformation* files. Together, they implement the newly developed transformation step *Immediate Transitions* which is added to the KiCo compilation chain. This M2M transformation step checks every transition of the created SCChart and delays

marked transitions in order to avoid instantaneous loops. The implementation is presented in detail in Section 5.2.

5.1 CDTProcessor

The CDTProcessor is the core element of the implementation of the model-based compilation. It comprises the creation of an SCChart from a generated AST. Additionally, it defines the way of visualizing code structures and of conjoining them. This sections presents key features of the CDTProcessor and their functionality.

As an introductory example, consider the code excerpt shown in Listing 5.1. After the AST of the input source code has been generated, each child of type CASTFunctionDefinition is passed to the transformFunction (line 6). This function is responsible for the model creation of functions. The state it returns represents this transformed function and is added to the list of root states of the SCChart (line 8). To enable the referencing of functions for function calls later on, each transformed function is stored temporarily in a list called functions, together with its associated VOSet list (line 7). VOSet contains valued objects like parameters and return variables.

```
1  // For each function definition add a new rootState to the rootSCChart.
2  ast.children.forEach [ func |
3      VOSet.clear
4      if (func instanceof CASTFunctionDefinition) {
5          val rootFunctionDefinition = func as CASTFunctionDefinition
6          val model = rootFunctionDefinition.transformFunction
7          functions.add(new Pair(model, VOSet.clone))
8          rootSCChart.rootStates += model
9      }
10 ]
```

Listing 5.1. Code excerpt from the transform function of the CDTProcessor

The method transformFunction creates a new state for a given function definition and extracts the return type as well as the parameter declarations. Thereafter, the compound body of the function is transformed with the help of the function transformCompound. A compound consists of multiple sub-trees of the extracted AST. Each node of these sub-trees represents a separate statement. Hence, the function transformCompound calls transformStatement. Each statement is transformed one after another until no unhandled statements are left. Listing 5.2 shows an excerpt from the described function which presents a selected number of implemented transformation functions. The type of the passed argument statement is checked and the appropriate transform function is called. The previously created state of the SCChart model is passed to the transform functions in order to continue the creation process from the correct position. Each statement type needs a separate transform function. Note that not every possible statement type has been supported yet. For instance, the conversion of goto

statements and pointer statements have not been implemented so far. This remains future work. For a complete list of statement types, the reader is referred to the CDT API[1].

```
1  def State transformStatement(IASTStatement statement, State parent) {
2
3      var actualState = parent;
4      if (statement instanceof CASTIfStatement) {
5          actualState = statement.transformIf(actualState)
6          checkForNestedReturn(actualState)
7      }
8      else if (statement instanceof CASTReturnStatement) {
9          actualState = statement.transformReturn(actualState)
10     }
11     else if (statement instanceof CASTWhileStatement) {
12         actualState = statement.transformWhile(actualState)
13         checkForNestedReturn(actualState)
14     }
15     else if (statement instanceof CASTForStatement) {
16         actualState = statement.transformFor(actualState)
17         checkForNestedReturn(actualState)
18     }
19
20     . . .
21
22     actualState
23 }
```

Listing 5.2. Code excerpt from the transformStatement function of the CDTProcessor

The next step focuses on the transformation functions of control structures. Since all of these functions share a similar structure, it suffices to review only one of the implementations at this point. Listing 5.3 and Listing 5.4 present the transformWhile function. As mentioned before, each control structure is nested inside its own superstate. Therefore, a whileState is created and added to the parent region (lines 7-10). Furthermore, a whileStateRegion is added to the whileState (lines 13-16). This region contains the inner body of the while loop. Thereafter, the function createConnectingTransition is called in order to connect the newly created whileState with the previously created state. Additionally, this function also determines the necessary transition type and checks for eventually needed transition triggers and actions. If the given state is a superstate, a connecting termination transition is needed. In case the while loop is nested inside a control structure and is located right after the condition check of its condState, the appropriate trigger condition must be added to the connection transition.

Next, the inner behavior of the loop is created. The entry point of the while construct is the condState (lines 22-26). Starting from here, the condition of the loop is checked. If the condition holds true, the body of the loop is executed. Thereafter, the control flow returns to the condState in order to recheck the condition. The body is represented as a

[1]http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html

CASTCompoundStatement. Hence, the function transformCompound is called (line 30). The condState is passed as an argument to serve as a point of resumption for the model creation process. The last created state of the function call is stored inside the value bodyState. Since the AST is traversed in depth-first order, the bodyState is the last state of the body of the loop. After the body is executed, the condition needs to be checked again for the second iteration of the loop. Therefore, a backTransition is created between the bodyState and the condState (lines 32-40). In order to avoid instantaneous loops for the code generation process, the backTransition is marked as *"notImmediate"* (line 34). Later on, this tag is used by the *Immediate Transitions* M2M transformation step of the KiCo compilation chain to identify to be delayed transitions. This transformation is detailed in Section 5.2. Line 35 checks for the eventuality of the body state being a superstate. If this is true, the backTransition must be of type TERMINATION.

Finally, the final state of the loop falseState (lines 43-47) and its connection transition falseTrans from the condState (lines 50-54) are created. The falseTrans transition activates in case the condition of the while loop is not satisfied anymore. Consequently, the final state falseState is entered and the loop state whileState terminates.

```
1  def State transformWhile(CASTWhileStatement statement, State state) {
2      val f = statement
3      // condition
4      val exp = f.condition
5      val kExp = exp.createKExpression
6      // While loop state.
7      val whileState = scc.createState => [ s |
8          s.id = "_S" + trC + WHILEID
9          s.label = WHILELABEL
10         state.parentRegion.states += s
11     ]
12     // Region of whileState. It contains all states of while loop.
13     val whileStateRegion = scc.createControlflowRegion => [ region |
14             region.id = whileState.id + REGIONLABEL
15             region.label = ""
16             whileState.regions += region
17     ]
18     // Create transition which connects the parent state "state" to the whileState.
19     createConnectingTransition(state, whileState, statement)
20     // The condState serves as starting point to the loop.
21     val condState = scc.createState => [ s |
22         s.id = trC.toString + CONDID
23         s.label = ""
24         s.initial = true
25         whileStateRegion.states += s
26     ]
27     // continue to transform body of while loop
28     val body = f.body as CASTCompoundStatement
29     val bodyState = body.transformCompound(condState, condState)
30     . . .
```

Listing 5.3. Code of the transformWhile function of the CDTProcessor

```
1  ...
2      // transition returns to condition-check
3      val backTransition = scc.createTransition => [
4          targetState = condState
5          annotations.add(createStringAnnotation(NOTIMMEDIATELABEL,""))
6          if (bodyState.hasInnerStatesOrControlflowRegions) {
7              type = TransitionType::TERMINATION
8          } else {
9              immediate = true
10         }
11         bodyState.outgoingTransitions += it
12     ]
13     // final state of loop. It is reached when loop condition is not met anymore
14     val falseState = scc.createState => [s |
15         s.id = trC + ENDID
16         s.label = ""
17         s.final = true
18         whileStateRegion.states += s
19     ]
20     // transition to falseState
21     val falseTrans = scc.createTransition => [
22         targetState = falseState
23         immediate = true
24         condState.outgoingTransitions += it
25     ]
26     whileState
27 }
```

Listing 5.4. Continuing Listing 5.3

After every statement node of the extracted AST has been handled and a representing SCChart has been created, the function transform, shown in Listing 5.1, continues its execution. Listing 5.5 continues where Listing 5.1 left off. The final step of the CDTProcessor is the creation of references for function calls. If a function call expression is found during the traversing of the AST, a pair containing the referencing state and the corresponding expression is added to the list functionCallRefs. First, the *valued objects* of the caller are stored in the emptied VOSet (lines 10-11). The entries of the VOSet are used later on to create the parameters for the argument list of the callee. After the referenced state is found in a list of root states (lines 13-15), the reference is created by setting the attribute referencedScope of the referencing state to the referenced state (line 17). As a result, the referencing state is transformed into a reference state. Next, the arguments of the function call must be added to the parameter list of the reference state (lines 21-27). The function createKExpression creates a parameter expression with the help of the VOSet of the function (line 26). Furthermore, the variable to which the return value of the function call is written to, is fetched and stored in the variable assignedVar (line 34). If the called function is of type *void*, the called method getAssignedVariableForReturnVal returns null and the following code block is not executed. The creation of the reference of the return variable (lines 36-39) proceeds as the previous creation of references of the parameters.

The only difference is the setting of a callByRederence flag to true in line 37. This flag identifies a return variable reference. Thereafter, the creation of the SCChart concludes.

```
1  // Create reference of function call.
2  if (!functionCallRefs.empty) {
3      functionCallRefs.forEach [ entry |
4          /* Look up the to be referenced function inside CASTFunctionCallExpression (key)
5           * and connect it to the referencing state (value). */
6          val funcCallExp = entry.key
7          val referencingState = entry.value
8          val rootState = referencingState.getRootState
9
10         VOSet.clear
11         VOSet += functions.filter[ key == rootState ].head.value
12
13         val funcID = funcCallExp.functionNameExpression.children.head.toString
14         // Search the list functions for the to be referenced function state
15         val State referencedState = lookForFunctions(funcID)
16         if (referencedState != null) {
17             referencingState.referencedScope = referencedState
18         }
19
20         // Handle arguments of function call.
21         val argumentList = getFunctionCallArguments(funcCallExp)
22         if(!argumentList.empty){
23             // Create a parameter for each given argument.
24             for (arg : argumentList) {
25                 kex.createParameter => [ p |
26                     p.expression = arg.createKExpression
27                     referencingState.parameters.add(p)
28                 ]
29             }
30         }
31
32         /* If necessary, add reference to the variable which is assigned to the retun
33          * value of the function call */
34         val CASTIdExpression assignedVar = getAssignedVariableForReturnVal(funcCallExp)
35         if (assignedVar != null) {
36             kex.createParameter => [ p |
37                 p.callByReference = true
38                 p.expression = assignedVar.createKExpression
39                 referencingState.parameters.add(p)
40             ]
41         }
42     ]
43 }
```

Listing 5.5. Continuing the Code excerpt from Listing 5.1 of the transform function of the CDTProcessor

5.2 Immediate Transitions Transformation

This final section of this chapter presents a code excerpt from the implementation of the new addition to the KiCo compilation chain, the *Immediate Transitions* M2M transformation step. Its task is to delay every marked immediate transition in order to avoid instantaneous loops. This step is necessary due to the limitations of the data-flow low-level compilation phase. For more information see Section 4.1.2.

ImmTransTransformation.xtend defines the functionality of this transformation step. Its implementation of the transform function can be examined in Listing 5.6 and Listing 5.7. In order to understand the concept of the transformation, the essential code lines are explained step-by-step. Basically, every outgoing transition of every state inside the SCChart is checked for an annotation *"notImmediate"*. Every marked transition is delayed. Therefore, every state of each root state is added to a Todo list nextStates (line 10). While iterating over every state inside this list, each outgoing transition is checked for the annotation *"notImmediate"*. This annotation is added by the CDTProcessor while creating the SCChart model in order to alert of possible instantaneous loops. If a marked transition is found, it is delayed (lines 18-21). Thereafter, the state is checked for possible internal states. In case of a superstate, every state inside needs to be checked as well. Therefore, they are added to a second list tmpList (lines 27-33). After every state inside the Todo list nextStates is checked for immediate outgoing transitions and internal states, nextStates gets cleared. Now, the tmpList contains every state of the already checked superstates. As a next step, the content of tmpList is copied to the Todo list (line 41) and cleared (line 15). Hence, the state space can be seen as a tree structure. A superstate represents a parent node while its internal states are the children nodes. The transformation process traverses the state tree in a breadth-first order. Afterwards, the procedure is repeated as long as there are states inside the Todo list at the end of an iteration of the while loop (line 14). Otherwise, the transform function terminates and returns the adjusted SCChart.

Now that the M2M transformation is defined and implemented, it needs to be assigned to a *Feature*. This Feature is then used to inform KiCo of this transformation step. Therefore, *CbasedSCChartFeature* defines an ID CbasedSCChart which is given to the "Immediate Transitions" transformation. This ID is transferred to KiCo.

```
1  // It is required, that a state contains only one ControlflowRegion
2  def Scope transform(Scope rootSCChart, KielerCompilerContext context) {
3      if (rootSCChart instanceof SCCharts) {
4          for (rootState : rootSCChart.rootStates) {
5              val regions = rootState.regions.filter(ControlflowRegion)
6              // List of sstates of function
7              val states = regions.head.states
8
9              . . .
```

Listing 5.6. Implementation of ImmTransTransformation.xtend

```
1  ...
2                  // Todo-list
3                  for (s : states) {
4                      nextStates.add(s)
5                  }
6                  /* Delay all transitions, that are marked by the annotation
7                   * "notImmediate" of all states in this region. */
8                  while (!nextStates.empty) {
9                      tmpList.clear
10                     for (s : nextStates) {
11                         // Change all transitions of state s
12                         for (t : s.outgoingTransitions) {
13                             if (t.annotations.head != null) {
14                                 if (t.annotations.head.name.contains("notImmediate")) {
15                                     t.immediate = false
16                                 }
17                             }
18                         }
19                         /* If state s contains additional states that need to be
20                            checked for transitions, add them to Todo list. */
21                         var tmpRegions = s.regions.filter(ControlflowRegion)
22                         if (!tmpRegions.empty) {
23                             var tmpStates = tmpRegions.head.states
24                             // Save states in order to copy them to Todo list later on.
25                             if (!tmpStates.empty) {
26                                 for (state : tmpStates) {
27                                     tmpList.add(state)
28                                 }
29                             }
30                         }
31                     }
32                     // Update Todo list.
33                     nextStates.clear
34                     for (s : tmpList) {
35                         nextStates.add(s)
36                     }
37                 }
38             }
39         }
40         rootSCChart
41     }
```

Listing 5.7. Implementation of ImmTransTransformation.xtend

Chapter 6

Evaluation

This chapter evaluates the model-based compilation of legacy C programs in two steps. Since this thesis was developed on the basis of the prototype of Smyth and Olsson [Ols16], the results are evaluated by comparing them to the results of the prototype. Firstly, the overhauled model extraction is compared to the old extraction of the prototype in Section 6.1. By extracting an SCChart from the same source code with both versions of the mode-based compilation, differences and optimizations are highlighted. The second Section 6.2 compares the generated C code to the legacy source code. Furthermore, the limitations of the code generation process are discussed.

6.1 Evaluating the Code Visualization

In order to evaluate the code visualization, the prototype and the implementation of this thesis both extract an SCChart model from the same source code which is shown in Listing 6.1a. This C code implements the calculation of the n-th Fibonacci number. The extracted model of the prototype is shown in Figure 6.1b while the extracted model of this thesis is depicted in Figure 6.2. When comparing both models, one can argue that the SCChart of Figure 6.2 is more organized and the control flow through the model is easier to understand. This increase in comprehensibility is achieved by three steps of improvement.

Firstly, the SCChart is cleaned out of unnecessary information. Superfluous information obscure the gist of the model and overstrains the user. The goal of the model-based compilation of legacy C programs is the provision of semantically equivalent SCCharts to the legacy source code. Hence, the user rather works with the more maintainable model than with the legacy code which is harder to read. Consequently, using the line number of the originally transformed statement as the label of the state does not provide any important information to the user. Furthermore, the additional states of the if branches `T` and `F` are dropped in order to reduce the number of states. The true branch of an if statement can already be distinguished from the else branch by the higher priority and the trigger of its transition.

Secondly, as many statements as possible are composited to one state. Multiple variable initializations in a row are now represented by one state while the former visualization required one state for each initialization. This greatly affects the number of states of models for larger scale programs. Additionally, designated initializations are added to the declaration interface of the parent state of the associated control region. Consequently, the number of

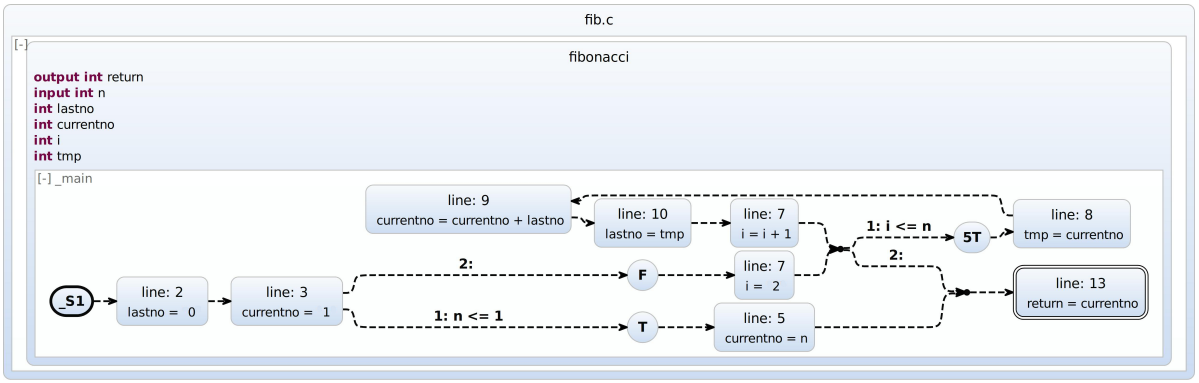
states is reduced even more. An example is the initialization of the integer variable lastno and currentno.

Lastly, control structures are aggregated into a separate superstate. This feature enables the identification of the boundaries of control structures at first sight. On the contrary, the nested control structures of Figure 6.1b blend into each other and their borders are only identified when taking a closer look. The introduction of a separate superstate however defines borders of control structures definitively. Additionally, this visualization also makes the definition and assignment of local variables possible such as the counter variable i of the for loop.

These three improvements provide a leaner SCChart model with fewer states and a clearer control flow. The SCChart of the prototype contains 17 states and 16 transitions while the SCChart of this thesis contains 11 states and 8 transitions. Therefore, the developed visualization of latter should be able to scale better than the prototype.

```c
int fibonacci(int n) {
    int lastno = 0;
    int currentno = 1;
    if(n <= 1) {
        currentno = n;
    } else {
        for(int i = 2; i <= n; i = i + 1) {
            int tmp = currentno;
            currentno = currentno + lastno;
            lastno = tmp;
        }
    }
    return currentno;
}
```

(a) C code example that calculates the n-th Fibonacci number



(b) Visualization of the C code by the previous prototype

Figure 6.1. Comparing the previous and the newly developed visualization of C code.

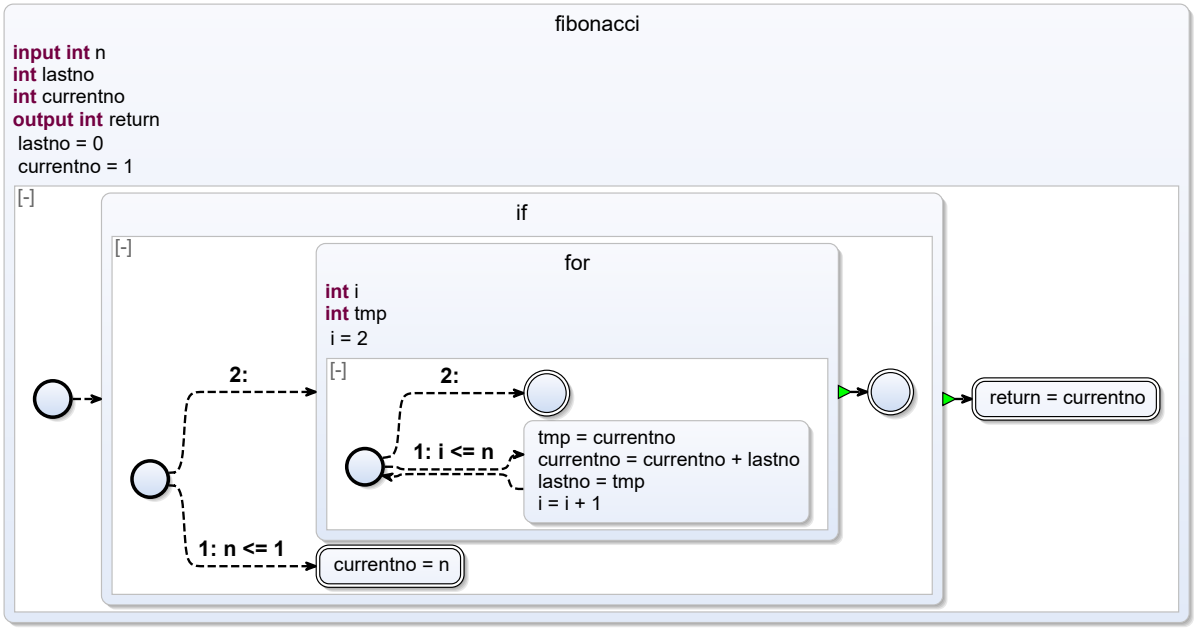


Figure 6.2. Newly developed visualization of the C code of Listing 6.1a

The next step of the evaluation is the comparison of function call visualizations. Figure 6.3b shows the extracted model of the old prototype, while Figure 6.3c presents the visualization of this thesis. The former visualization offers no possibility for showing the inner behavior of the called function. The function call is simply realized by a host-code statement `sum = 'add(a,b)'`. The new visualization utilizes reference states to integrate the inner behavior of the root state `add` into the root state of the caller `main`.

6.2 Evaluating the Code Generation

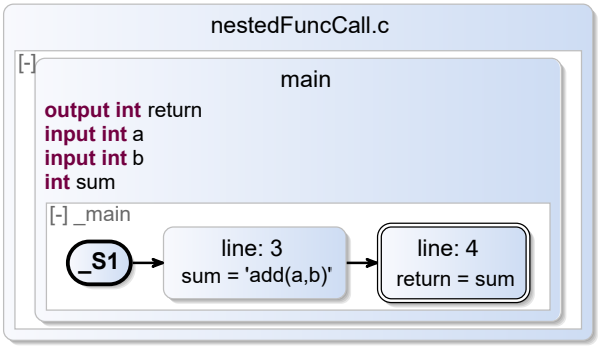
This final section of the evaluation takes a closer look at two examples of SCCharts and their generated code by the model-based compilation. The limitations of the code generation process are discussed and the legacy code regarding the number of used variables and lines is compared to the generated C code.

The first example is the previously shown SCChart of Figure 6.3c. Its generated source code is presented by the listings of Figure 6.4 which calculates the sum of two integer numbers. At first sight, it is obvious that the generated code is much longer and even harder to read for a human user. Furthermore, the introduction of guards for basic blocks and conditions greatly increase the number of used variables. However, means to reduce the number of used variables and therefore the necessary memory space is currently developed by the KIELER development team of the Real-Time and Embedded Systems Group. The enhancement of the readability of the generated code is of no high priority, since the generated source code is made for computers and not for humans. Providing an error-free executable code which

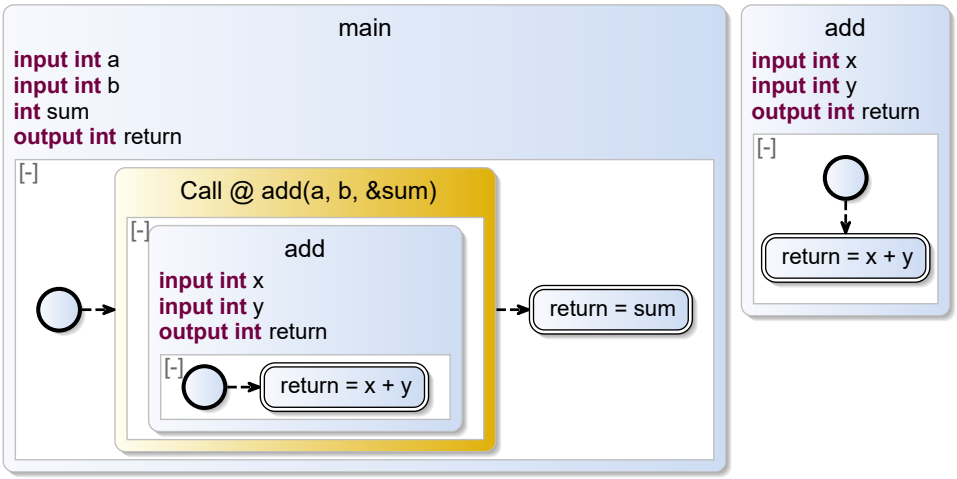
```c
int main(int a, int b) {
    int sum;
    sum = add(a,b);
    return sum;
}

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

(a) C code example that calculates the sum of two integers



(b) Visualization of function calls by the previous prototype



(c) Newly developed visualization of the function calls

Figure 6.3. Comparing the previous and the newly developed visualization of function calls

needs as little memory space as possible is of greater importance. This leads to the limitations of the code extraction process that still exist. The generated C code from the SCChart is not yet executable without manual additions. However, the code generation of function calls still is in the early stages of development and must therefore be seen as a prototype. When considering the generated source code of the listing of Figure 6.4a, the manual additions are marked by the comments above. The first addition is the inclusion of required header files. The second addition to the source code is the `main` function of the source code file in order to provide an entry point for its execution. In this given example, the `main` reads the given arguments of the user and stores it in the variables `a` and `b`. As a next step, the two inputs are copied to the corresponding fields of a newly created `TickData` variable `td`. This variable belongs to the root state `main` of the SCChart and contains its input variables, output variables and guard variables. Furthermore, a `TickData1` variable `td1` is declared which belongs to the root state `add`. In order to call the corresponding tick function `tick1` in line 60 of the listing of Figure 6.4b, the

variable `_call0` of the caller TickData needs to be linked to the TickData `td1` of the callee. This assignment is done in line 89 inside the main function. After resetting `td` to set the variable `_G0` in order to signal the beginning of the execution of the program, the `tick` function of the root state `main` is called. At the end, the computed result is printed out onto the console.

At this point of the development, no variable names can be reused between different functions. If, for example, two functions `f` and `g` both name their input variable `x`, the code generation will not work as intended. Therefore, one of the input variables needs to be renamed. Furthermore, the generated functions `reset`, `tick` and `tickLogic` are not properly named when an SCChart contains more than two functions. The generated functions belonging to the third defined function are called the same as the generated functions for the second defined function. Therefore, the names need to be manually adjusted by the user in order to avoid unwanted behavior.

```c
//Manually added header files.
#include <stdio.h>

typedef struct {
  char g5;
  char _GO;
  char ret;
  char x;
  char y;
} TickData1;

typedef struct {
  char g0;
  char _GO;
  char g4;
  char g3;
  char pg3;
  char g1;
  TickData1 _call0;
  char _cg1;
  char g2;
  char ret;
  char sum;
  char a;
  char b;
} TickData;

void reset1(TickData1 *data) {
  data->_GO = 1;
}
void tickLogic1(TickData1 *data) {
  data->g5 = data->_GO;
  if (data->g5) {
    data->ret = data->x + data->y;
  }
}
void tick1(TickData1 *data) {
  tickLogic1(data);

  data->_GO = 0;
}

void reset(TickData *data) {
  data->pg3 = 0;
  data->_GO = 1;

  reset1(&data->_call0);
}
...
```

(a) First part of the generated C code

```c
...

void tickLogic(TickData *data) {
  data->g0 = data->_GO;
  data->g4 = data->pg3;
  data->g1 = data->g0 || data->g4;

  if (data->g1) {
    data->_call0.x = data->a;
    data->_call0.y = data->b;
    tick1(&data->_call0);
    data->sum = data->_call0.ret;
  }
  data->_cg1 = 1;
  data->g3 = data->g1 && !data->_cg1;
  data->g2 = data->g1 && data->_cg1;

  if (data->g2) {
    data->ret = data->sum;
  }
}

void tick(TickData *data) {
  tickLogic(data);

  data->_GO = 0;
  data->pg3 = data->g3;

}

// Manually added main function.
int main(int argc, char* argv[]) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    int sum;
    TickData td;
    TickData1 td1;
    td.a = a;
    td.b = b;
    td._call0 = td1;

    reset(&td);
    tick(&td);
    sum = td.ret;
    printf("%d \n", sum);

    return 0;

}
```

(b) Continuing the listing of Figure 6.1

Figure 6.4. Generated C code plus manually added code from SCChart shown in Figure 6.3c. This code calculates the sum of two integers.

Chapter 7

Conclusion

This final chapter summarizes the results of this thesis. Furthermore, each development step as well as the evaluation of the model-based compilation of legacy C programs are recapped in Section 7.1. The thesis is concluded by presenting ideas for future work and by pointing out possible improvements of the model-based compilation of legacy C programs in Section 7.2.

7.1 Summary

This thesis continued the work of Smyth and Olsson [Ols16] by developing new features as well as improving existing features of the model-based compilation of legacy C programs. The first part of this thesis gave the reader an overview of the provided features of the visual programming language SCCharts. This overview was mandatory in order to fully understand the possibilities and the limitations of the SCCharts language. Furthermore, it presented first ideas of how to visualize C code.

After explaining the concept of Sequential Constructiveness, the interactive compilation process in KIELER was discussed. This section exemplified the high-level compilation approach as well as both low-level compilation approaches, while focusing on the data-flow approach. Therefore, the syntax of the SCG was introduced to the reader. This step was necessary to allude the problematic nature of instantaneous loops inside SCCharts. Next, the C code extraction process was explained which was realized with the help of the CDT plug-in. Furthermore, the structure of the created AST was depicted and the visualization procedure of how to create models from ASTs was described.

The subsequent section presents the concept of this thesis on how to visualize C code. Firstly, control structures were discussed. Thereafter, the graphical representation of function calls was discussed. As a final step, the structure and content of the generated source code from extracted SCCharts was explained.

The evaluation showed, that the visualization of source code improved by providing a leaner SCChart that is supposedly easier to understand. Furthermore, the limitations of the new prototype for C code generation including function calls, which is based on the data-flow low-level compilation approach, were stated and evaluated.

7.2 Future Work

Even though this thesis addressed several problems of the already existing prototype, the complex development of a model-based compilation of source code still provides yet unsolved problems and challenges. Therefore, this thesis is closed by presenting several ideas for future work.

Concurrency

One of the presumingly most important, and supposedly also one of the most challenging future tasks is the development of compiling concurrent C code. Concurrency is part of every real-time and safety-critical embedded system. Therefore, it is of great importance to expand the functionality of KIELER and support the model extraction of concurrent legacy C code.

Expanding the Set of Supported Code Concepts

This thesis implemented the visualization of important code concepts such as control structures and function calls. Nevertheless, legacy C code may contain many more constructs which need to be converted when extracting a representing SCChart. One of the more important concepts of the programming language C are pointers. Pointers are one of the specialties of C and provide a powerful tool to refer to memory addresses of other variables. Another important feature are recursive function calls. Currently, only non-recursive function calls are supported. However, recursiveness provides a powerful tool to the programmer.

Currently, only functions can be called whose SCChart does not contain any delayed transitions and therefore finishes its execution within one tick. In order to lift this limitation, the compilation process of the KiCo compilation chain needs to adjusted. Additionally, the conversion of unconditional goto statements, as well as global variables would improve the range of supported C code for the model-based compilation of legacy C code.

Further Improvement of the Code Generation

One goal of KIELER is to provide a T2M2T compilation for C code which provides automatically generated executable source code. Until new, the generated C code must still be manually adjusted by adding necessary header files and a main function in order to be executed. Future work needs to include these additions automatically. Furthermore, the compilation process must be adjusted in order to support function calls where the root state of the callee contains delayed transitions. Additionally, bugs like the correct naming of the generated functions have to be corrected.

Design Issues

The design of the extracted SCCharts can be further advanced by combining a variable declaration and initialization into a single statement inside the declaration interface of the respective

state. Furthermore, it can be assessed whether the relocation of variable declarations and initializations out of the declaration interface of the parent state into a separate state in its region provides a more readable model. Even though the number of states would be increased, the declaration interface and therefore the state itself would need fewer space overall.

Benchmark Testing of Generated Code

Until now, the generated source code is only analyzed without additional help like benchmark tools. In order to evaluate the quality of the generated source code further additional tests have to be done. Additionally, the savings of states and transitions of SCCharts created with the enhanced model-based compilation of this thesis compared to the SCCharts of the former prototype can be further analyzed. This would provide for a more detailed measurement for the actual improvements of the results of the model creation process.

Chapter 8

Appendix

A Generated AST

```
1  org.eclipse.cdt.internal.core.dom.parser.c.CASTFunctionDefinition@2b4ab7
2   org.eclipse.cdt.internal.core.dom.parser.c.CASTSimpleDeclSpecifier@58721f69
3   org.eclipse.cdt.internal.core.dom.parser.c.CASTFunctionDeclarator@5e91142c
4    main
5   org.eclipse.cdt.internal.core.dom.parser.c.CASTCompoundStatement@72a3b822
6    org.eclipse.cdt.internal.core.dom.parser.c.CASTDeclarationStatement@28872a33
7     org.eclipse.cdt.internal.core.dom.parser.c.CASTSimpleDeclaration@6046e11d
8      org.eclipse.cdt.internal.core.dom.parser.c.CASTSimpleDeclSpecifier@3620b94d
9      org.eclipse.cdt.internal.core.dom.parser.c.CASTDeclarator@59ad6503
10      a
11      org.eclipse.cdt.internal.core.dom.parser.c.CASTEqualsInitializer@48ea0ab7
12       1
13    org.eclipse.cdt.internal.core.dom.parser.c.CASTIfStatement@1f7354be
14     org.eclipse.cdt.internal.core.dom.parser.c.CASTBinaryExpression@2c2a6ccc
15      org.eclipse.cdt.internal.core.dom.parser.c.CASTIdExpression@1e9d8137
16       a
17      2
18     org.eclipse.cdt.internal.core.dom.parser.c.CASTCompoundStatement@6ba9fe1
19      org.eclipse.cdt.internal.core.dom.parser.c.CASTExpressionStatement@3e9ed2c
20       org.eclipse.cdt.internal.core.dom.parser.c.CASTBinaryExpression@7c5abd91
21        org.eclipse.cdt.internal.core.dom.parser.c.CASTIdExpression@ebc1f14
22         a
23        3
24    org.eclipse.cdt.internal.core.dom.parser.c.CASTReturnStatement@5c2bccb5
25     1
```

Listing A.1. AST of C code of Listing 4.11a

B Generated C code

```c
// Manually added header files.
#include <stdio.h>

typedef struct {
  int _GO;
  int g0;
  int _cg0;
  int a;
  int b;
  int g3;
  int _cg3;
  int g4;
  int g1;
  int g5;
  int g2;
  int ret;
} TickData;

void reset(TickData *data) {
  data->_GO = 1;
}

void tickLogic(TickData *data) {
  data->g0 = data->_GO;
  data->_cg0 = data->a < data->b;
  data->g3 = data->g0 && !data->_cg0;
  data->_cg3 = data->a > data->b;
  data->g4 = data->g3 && data->_cg3;
  if (data->g4) {
    data->a = data->b * 2;
  }
  data->g1 = data->g0 && data->_cg0;
  if (data->g1) {
    data->a = data->b;
  }

...
```

(a) First part of the generated C code

```c
...
  data->g5 = data->g3 && !data->_cg3;
  if (data->g5) {
    data->a = data->a + data->b;
  }
  data->g2 = data->g1 || data->g5 || data->g4;
  if (data->g2) {
    data->ret = data->a;
  }
}

void tick(TickData *data) {
  tickLogic(data);

  data->_GO = 0;
}

// <-- Manually added main function.
int main(int argc, char* argv[]) {
    int f = atoi(argv[1]);
    int g = atoi(argv[2]);
    int x;
    TickData td;
    td.a = f;
    td.b = g;
    reset(&td);
    tick(&td);
    x = td.ret;
    printf("%d \n", x);
    return 0;
}
```

(b) Continuing listing of Figure B.1a

Figure B.1. Generated C code plus added header files and main function from SCChart shown in Figure 4.14.

```c
#include <stdio.h> // <-- Manually added header files.

typedef struct {
  char g0;
  char _GO;
  char lastno;
  char currentno;
  char _cg0;
  char n;
  char g3;
  char _null_fibonacci_int_local_i;
  char g6;
  char g5;
  char pg5;
  char g4;
  char _cg4;
  char g7;
  char g8;
  char pg7;
  char _cg8;
  char _null_fibonacci_int_local_tmp;
  char g1;
  char g2;
  char ret;
} TickData;

void reset(TickData *data) {
  data->pg5 = 0;
  data->pg7 = 0;
  data->_GO = 1;

}

void tickLogic(TickData *data) {
  data->g0 = data->_GO;
  if (data->g0) {
    data->lastno = 0;
    data->currentno = 1;
  }
  data->_cg0 = data->n <= 1;
  data->g3 = data->g0 && !data->_cg0;
  if (data->g3) {
    data->_null_fibonacci_int_local_i = 2;
  }

  . . .
```

Listing B.2. Generated C code plus added main function from SCChart shown in Figure 6.1

```
1  . . .
2
3    data->g6 = data->pg5;
4    data->g4 = data->g3 || data->g6;
5    data->_cg4 = data->_null_fibonacci_int_local_i <= data->n;
6    data->g8 = data->pg7;
7    data->_cg8 = data->_null_fibonacci_int_local_i <= data->n;
8    data->g5 = data->g4 && data->_cg4 || data->g8 && data->_cg8;
9    if (data->g5) {
10     data->_null_fibonacci_int_local_tmp = data->currentno;
11     data->currentno = data->currentno + data->lastno;
12     data->lastno = data->_null_fibonacci_int_local_tmp;
13     data->_null_fibonacci_int_local_i = data->_null_fibonacci_int_local_i + 1;
14   }
15   data->g1 = data->g0 && data->_cg0;
16   if (data->g1) {
17     data->currentno = data->n;
18   }
19   data->g2 = data->g1 || data->g8 && !data->_cg8;
20   if (data->g2) {
21     data->ret = data->currentno;
22   }
23 }
24
25 void tick(TickData *data) {
26   tickLogic(data);
27
28   data->_GO = 0;
29   data->pg7 = data->g7;
30   data->pg5 = data->g5;
31 }
32
33 int main(int argc, char* argv[]) { // <-- Manually added main function.
34     int n = atoi(argv[1]);
35     int x;
36     TickData td;
37     td.n = n;
38     td.ret = 0;
39     reset(&td);
40     while(td.ret == 0) {
41         tick(&td);
42     }
43     x = td.ret;
44     printf("%d \n", x);
45     return 0;
46 }
```

Listing B.3. Continuing Listing B.2

Bibliography

[And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.

[BHL+02] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. "Readings in hardware/software co-design". In: ed. by Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. Norwell, MA, USA: Kluwer Academic Publishers, 2002. Chap. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, pp. 527–543. ISBN: 1-55860-702-1. URL: http://dl.acm.org/citation.cfm?id=567003.567050.

[FWW+13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. "Live trace visualization for comprehending large software landscapes: The ExplorViz approach". In: *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT'13)*. Sept. 2013, pp. 1–4. DOI: 10.1109/VISSOFT.2013.6650536.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data-flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

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

[HDM+14a] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274. Edinburgh, UK: ACM, June 2014.

[HDM+14b] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, June 2014.

[HMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. *Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation*. Technical Report 1308. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013.

[IM14] Javier L. Cnovas Izquierdo and Jess G. Molina. "Extracting models from source code in software modernization". In: *Software and Systems Modeling* 13.2 (Sept. 2014), pp. 713–734. ISSN: 1619-1374. DOI: 10.1007/s10270-012-0270-z.

[Mot09] Christian Motika. "Semantics and execution of domain specific models—KlePto and an execution framework". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Dec. 2009.

[MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. "Compiling SCCharts—A case-study on interactive model-based compilation". In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[Ols16] Lars Olsson. "Modellextraktion aus C-Code". Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2016.

[RLR+13] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Diego Clerissi. "What are the used UML diagrams? A Preliminary Survey". In: *EESSMod 2013 — 3rd International Workshop on Experiences and Empirical Studies in Software Modeling*. Vol. 1078. CEUR Workshop Proceedings. Nov. 2013, pp. 3–12.

[Sch06] Douglas C. Schmidt. "Model-driven engineering". In: *Computer* 39.2 (Feb. 2006), pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58.

[SMH15] Steven Smyth, Christian Motika, and Reinhard 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, May 2015.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Boston, Massachusetts , USA: Addison Wesley, 2003. ISBN: 0321118847.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "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)*. San Jose, CA, USA, 15–19 09 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.