From Lustre to Graphical Dataflow Programs

Lena Grimm

Master's Thesis
May 2019

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

Advised by
M.Sc. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

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

Kiel,

Abstract

Safety-critical applications often make use of a model-based approach for defining the behavior of components. This approach can be enhanced by the use of graphical languages. The Safety Critical Application Development Environment (SCADE) Suite is a tool that adopts the model-based approach and offers a graphical language to the user. Visual components can be placed, moved and connected to other components. However, the compiler of SCADE uses a textual language internally that is similar to the academic, synchronous language Lustre. Additionally, it is verified with different standards which makes SCADE well-suited for industry.

A different modeling approach is used in the KIELER project. The textual language is preserved for modeling, but a visual representation is generated from the textual one automatically. This approach combines the advantages of graphical modeling and textual modeling.

This thesis aims at preserving the graphical semantic known from SCADE but offering a textual entry to it using the synchronous language Lustre. The Sequentially Constructive Charts (SCCharts) dataflow language is compared to the SCADE language to prove that their visual representation is sufficiently similar and to investigate their semantical equivalence.

SCADE offers an entry to modeling at the visual dataflow level and compiles with Lustre which is based on a textual dataflow. This thesis aims at applying the modeling approach chosen in KIELER to Lustre. The textual language can be used for editing and the visual representation is generated automatically. Therefore, a transformation from Lustre programs to SCCharts dataflow programs is implemented. Nevertheless, Lustre offers features such as clocking that are not natively supported in SCCharts. An appropriate representation for these features is extracted in the process of the transformation.

Lastly, the implemented transformation is evaluated. The behavior of the original Lustre program and the transformed SCCharts program is compared in order to show that they behave the same. Furthermore, the underlying Lustre program of a SCADE example is extracted and transformed. The resulting visual dataflow is then compared to the SCADE dataflow.

Acknowledgements

First, I would like to thank my advisor Alexander Schulz-Rosengarten and my professor Prof. Dr. Reinhard von Hanxleden for the inspiring thoughts and talks during process of this thesis. SCCharts dataflow raise interesting questions that this thesis cannot cover alone and this topic gave me the opportunity to take part in the process of its development.

Moreover, I want to thank Steven Smyth for his work on the SCCharts dataflow synthesis. The dataflow visualizations are used frequently and thanks to his work and ideas they are improved and prettified.

Contents

List of Listings

List of Figures

List of Tables

Chapter 1

Introduction

Safety-critical applications are inevitable in the avionics and automotive industry. A non-functioning altitude control unit may cause the plane to crash and hundreds of lives would be in danger. Hence, these systems need to be developed with more caution than ordinary software. However, combining these safety requirements with standard programming paradigms such as concurrency, this becomes a tedious task. Systems usually require information from several sensors in order to react properly. Race conditions may occur and those can lead to unexpected behavior and nondeterministic errors.

This raises the need for a way to specify deterministic concurrency. In 1991 the Proceedings of the IEEE dedicated a special section to the three synchronous languages Esterel [BD91], Signal [LGL+91] and Lustre [HCR+91; BB91]. Their goal was to overcome this exact problem. Time is divided into discrete ticks and within each tick the possible operations are restricted in a way that each value of a variable can be determined uniquely without speculations. This usually requires a change in the programming perspective but offers a possibility to overcome concurrency problems. Moreover, properties of the systems can be proven due to the deterministic behavior. The possibility for verification is especially interesting for the safety-critical domain but also in other domains a deterministic concurrency facilitates the designing process.

Until today synchronous languages have emerged greatly. They are the language of choice when it comes to specifying, designing and verifying complex critical systems [BCE+03]. However, the tools supporting these semantics often use a graphical entry to the user. The Safety Critical Application Development Environment (SCADE) is an example for a modeling tool that is used in avionics. It is verified against several standards and its backbone is made from Lustre, a textual dataflow language.

Combining the benefits of textual editing and graphical reviewing for Lustre and SCADE respectively is the goal of this thesis. However, the Sequentially Constructive Charts (SCCharts) language is used as the visualization for Lustre. It has similar semantics and offers the possibility to define dataflow regions. Moreover, it is embedded in KIELER, an open-source tool for modeling. The automatic generation of a visualization from the textual version needs automatic layout generation and this is also a part of KIELER.

1.1 Synchronous Languages

Synchronous languages were developed to overcome the problems originating from concurrency and grounding deterministic semantics to reliably design safety-critical applications on a high abstraction level. Behind these promising properties lays a Sound computation model that defines how these languages work.

The main difference to languages like C or Java is determined by the *Synchrony Hypothesis*. It states that all computations are atomic and take no time, inputs arrive at the same time as outputs are produced. This property can only be achieved on a conceptual level because computations do not actually take no time. Thus, time is divided into *ticks*. Each tick represents a time unit that receives inputs and produces outputs. In Figure 1.1 this cyclic and discrete execution is visualized. One tick

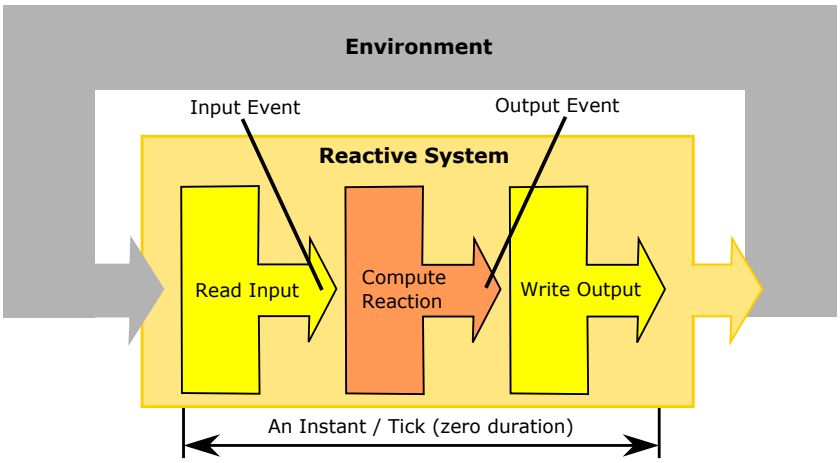


Figure 1.1. Discrete execution of a tick within the context of its environment [MHH13]

includes the reading of inputs, the computation of the reaction and the writing of the output. The outputs are given to the environment and in return the next tick receives new inputs.

Variables in the programs are usually referred to as *signals* and within a tick, each signal can either be present or absent. It follows that sequential access such as `x = 1; x = 0;` does not work in classical synchronous languages if the assignments are all executed in the same tick. The value of the signal `x` must be unambiguously defined and a program that offers a fixed point value for each variable is considered to be *constructive*.

The situation introduced above already introduces some of the disadvantages of synchronous languages. The work flow differs from what programmers are usually used to. Even for non-concurrent regions sequential variables access is restricted. Von Hanxleden et al. relaxed these requirements of the Synchronous Model of Computation (MoC) [HDM+14]. The Sequentially Constructive Model of Computation (SC MoC) is introduced and it accepts a strictly larger class of programs. Moreover, it offers a more intuitive approach to programming whereas the situation stated above would be a valid sequence of operations. Therefore, we do not refer to variables as signals. They may change their value during execution, thus the term of a *variable* remains. The semantics of the language SCCharts are founded on this SC MoC [HDM+13] and the modeling environment KIELER uses SCCharts as main programming language.

1.2 Lustre and SCADE

One of the first established synchronous programming languages is Lustre [HCR+91]. It was designed for programming reactive systems and describing hardware. It offers a dataflow syntax and provides concurrency with a deterministic semantics through the traditional synchronous MoC. Additionally, the language itself is well-suited for expressing specification properties of a program and thus, they can be directly integrated in the program for verification. This offers the great possibility to combine programming and verification in one model.

A Lustre program consists of *nodes*. Each node defines its inputs, outputs and optionally it may specify internal variables. In addition, a node has *equations* that are executed each tick. These equations are interpreted in a mathematical sense and may not be taken as assignments.

```
1 node counter(a:bool) returns (y:int);
2 var c: int;
3 let
4   c = 0 -> if a then pre(c) + 1 else pre(c);
5   y = c;
6 tel
```

Listing 1.1. A counter in Lustre

```
1 node counter(a:bool) returns (y:int);
2 var c: int;
3 let
4   c = 0 -> if a then c + 1 else c;
5   y = c;
6 tel
```

Listing 1.2. A counter in Lustre that contains a causal cycle

In Figure 1.1 a simple Lustre program is shown. This program takes a boolean `a` as input and returns an integer `y`. The program has one internal variable `c` of type integer. The lines 4 to 5 in between the `let` and `tel` keywords define the behavior of the program, thus the equations. In this example the internal variable is initialized by 0 in the first tick. In all subsequent ticks it is incremented by one if the input is true. Otherwise, the internal variable is assigned its value from the previous tick. In line 5 the output is set to the value of the internal variable. Altogether, this program is a counter that only reacts when the input is true.

In contrast, Figure 1.2 shows the same program without the `pre`. Since `pre` refers to the value of the variable in the previous tick, this breaks potential causal cycles for the value of the variable `c`. Not including the `pre` results in the value of `c` for the current tick to depend on the value of `c` in the current tick. This obviously yields a cycle.

SCADE is a modeling environment for safety-critical embedded software and its backbone language is founded on Lustre. [1] SCADE was started in the 1990's by the research laboratory VERIMAG and the software editor VERILOG. Since 2000 it is developed by ANSYS/ESTEREL-TECHNOLOGIES [Dor08]. Up until today, its code generation is qualified with the highest standards for safety-critical applications.

[1] http://www.esterel-technologies.com/products/scade-suite/

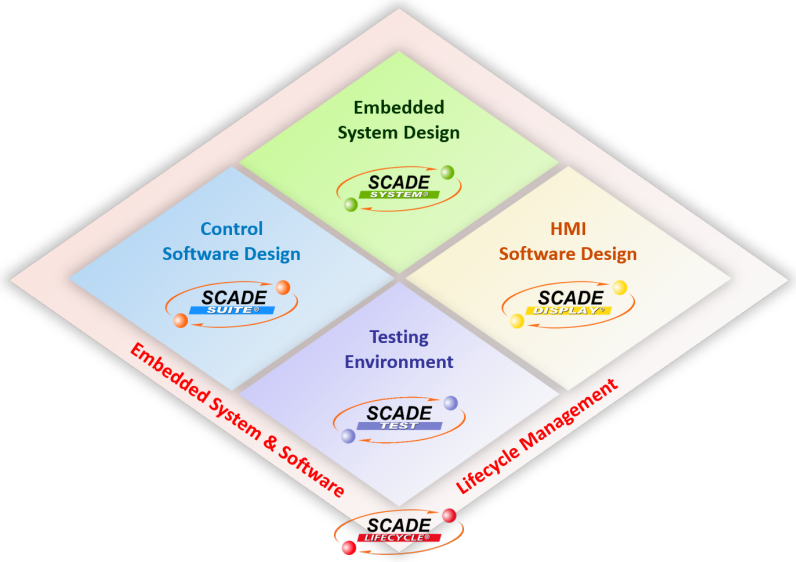


Figure 1.2. The SCADE product family[Est16]

Thus, requirements of various domains like avionic, automotive or transportation can be met using this tool.

The SCADE product family includes different environments for system design that serve specific purposes. In Figure 1.2 the different systems are visualized. Not only does it offer tools for the design of control systems, embedded systems and Human Machine Interface (HMI) systems, also a testing suite and a life cycle management is included. For the scope of this thesis we only look at the SCADE Suite, so the tool for designing control software.

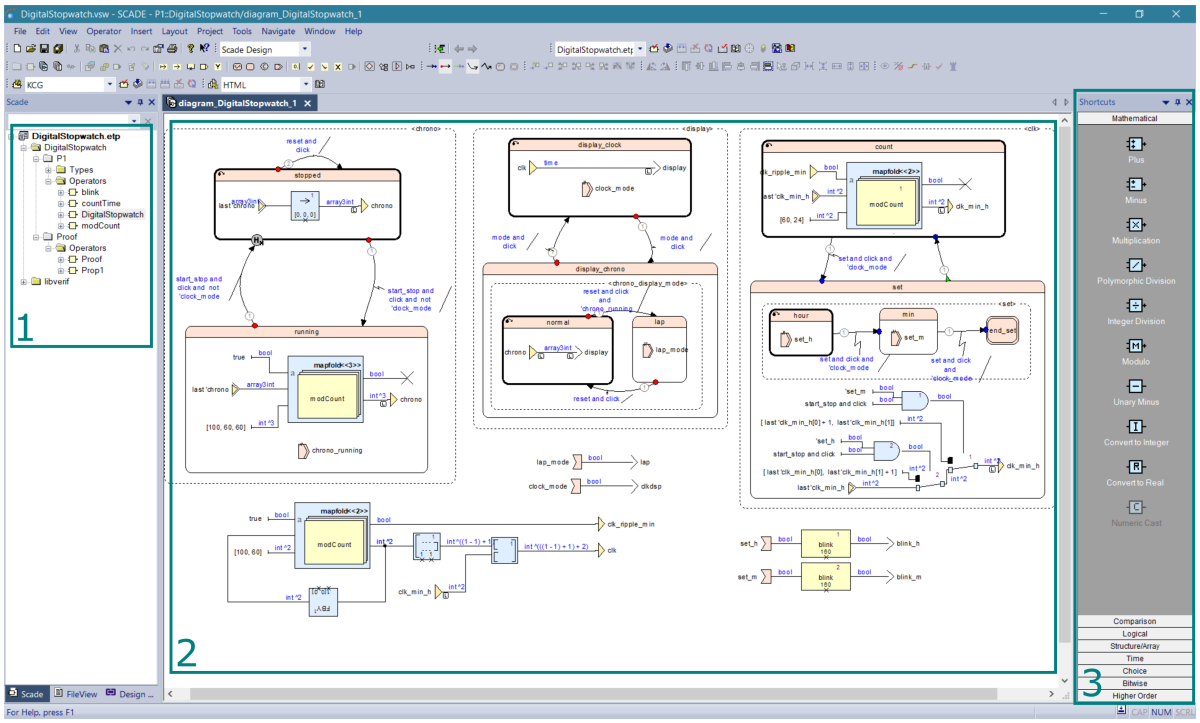


Figure 1.3. The SCADE Suite User Interface

SCADE Suite itself is a visual modeling environment. The user may specify the interface of components and then model the behavior via drag-and-drop. In Figure 1.3 the user interface is shown. It is divided into three main parts. On the left side, as Area 1, the project structure is shown. Self-defined operators and the package structures are listed here.

The Area 2 is preserved for modeling. One specific operator from the project structure may be selected and shown here. The behavior is modeled using components and connectors, whereas the user may choose from a dataflow or a controlflow approach. In the figure, both approaches are shown. The dotted lines titled `chrono`, `display` and `clk` include control-flow mechanisms. The user may define states and transitions. These states may then again choose between containing control-flow or data-flow. Below these dotted areas there are dataflow constructs.

The Area 3 lists the possible basic operators provided by SCADE Suite. They make up the predefined set of operations that may be performed. Moreover, these operators have an interface, too. After positioning an operator it is the user that needs to connect the ports of the operator's interface to the correct wires.

As already mentioned, the backbone of SCADE Suite is built from the Lustre language. SCADE is often referred to as a visual representation for Lustre. The semantics of both are still the same except for SCADE adding some more operators as syntactic sugar to improve the user experience. In fact, SCADE offers the possibility to convert an operator with a graphical diagram to a textual version. This textual version corresponds to the equations included in Lustre.

1.3 SCCharts and KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is a research project that was started by the Real-Time and Embedded Systems group at Kiel University. It was designed to enhance the graphical model-based design of complex systems. Moreover, the idea is to automatically generate visual representations from a program as a transient view [FH10]. This allows the user to take advantage from the visual model which may provide an overview of the program structure and give a faster understanding of the interconnection of components. Additionally, the benefits from a textual editor are still preserved. Fast copy-pasting, commenting program parts, putting them back and the fact that typing in general offers the user the possibility to build models faster [GKR+14] are just some of the advantages that this approach allows to keep.

Besides implementing the concept of transient views, the KIELER project also introduced a new synchronous language. The SCCharts are a visual and textual synchronous language. It was designed for specifying safety-critical systems with a deterministic semantic. Its notion is similar to Harel Statecharts [Har87] but, similar to SCADE, the user may also define data-flow regions. However, these dataflow regions are transformed to controlflow in order to generate code. This is different to SCADE, which uses a language that naturally supports dataflow.

However, the restrictions implied by synchronous languages are relaxed. As already mentioned, the semantics of SCCharts are based on the SC MoC. These sequentially constructive semantics are conservative in the traditional sense, so constructive programs are still accepted and behave the same. However, programs that would be traditionally rejected as not constructive might yield acceptance under the SC MoC. This conservative extension of the synchronous MoC allows to express programs in languages like Lustre in the SCCharts language.

1.4 Problem Statement

In traditional modeling tools the designing is usually done on the graphical level using a pallette. Components can be added, moved and connected to other components. This provides a fast overview and helps understanding new models or new components and their relation to each other quickly. Therefore, this modeling strategy has been the common case in established tools such as SCADE. However, textual modeling is al lot quicker and offers useful features such as refactoring, tracing and copy-pasting.

In Section 1.3 KIELER is introduced as a tool that combines the advantages of textual and graphical modeling. SCADE however, is based on graphical modeling with the backbone made of the Lustre language.

The goal of this thesis is the preservation of the language Lustre as the entry point to modeling combined with the automatic transformation to a visual representation. As visual representation the language SCCharts is used. It provides a conservative extension to the synchronous MoC, thus the expressiveness needed for designing Lustre programs is given. Additionally, the SCCharts have a dataflow extension so its visual representation is already close to the one provided in SCADE.

Hence, the first step of this thesis is the comparison of SCADE Suite and SCCharts in KIELER. Their semantics and appearances are analysed with the goal for Lustre programs, transformed to SCCharts programs, to look familiar to those that worked with SCADE.

The second step is the transformation from Lustre to SCCharts itself. Lustre is a dataflow language and there are certain properties that are not transformed easily into a controlflow language like SCCharts. There have been approaches transforming Lustre to SCChart using only its controlflow features but with the extension of dataflow the generated models are closer related to the original model. This facilitates for the user to trace the written model within the generated SCCharts model.

1.5 Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 introduces related work to the topic presented here. In the context of KIELER there exist quite some transformations for automatic layout generation of different languages. Also imperatives languages like C can benefit from a visual representation.

In Chapter 3 the prerequisites for this thesis are given. The used technologies are introduced and more specific properties of SCCharts and Lustre are given.

Chapter 4 introduces the conceptual results of this thesis. It starts with the comparison of SCADE and KIELER and the drawn consequences for the scope of this thesis. The second step is the transformation from Lustre to SCCharts itself.

Moreover, in Chapter 5 more implementation-dependent details are given. The embedding of the concepts into KIELER is explained and the interaction of the newly introduced components with the overall program is described.

Chapter 6 evaluates the results. The behavior of different Lustre programs is compared to the behavior of the transformed SCCharts programs.

Lastly, in Chapter 7 the results of this thesis are summarized and thoughts about potential future work are given.

Chapter 2

Related Work

The design process of model-based programs can be handled differently. In tools well-known in industry like SCADE[1] or Simulink [2] a graphical design process is chosen. However, a textual design process is possible as well and for the tool KIELER this approach is chosen.

In the first section of this chapter, synchronous dataflow languages are introduced. Lustre itself is a synchronous dataflow language and thus other languages and their characteristics are further explained.

The rationale for a textual approach and ideas on enhancing this process are presented in the second section. As already mentioned, the common design flow in model-based applications is realized through the user moving around components. These visual components are still helpful combined with a textual modeling approach. These visualizations do not need to be created by hand. However, this requires automatic layout.

In the third section some examples for graphical dataflow are given. These visualizations are useful in order to determine a sound basis for a visual dataflow. Transforming Lustre to SCCharts dataflow should provide a user experience that feels familiar to those that already know visual modeling tools.

Finally, the fourth section presents work on model-to-model transformations. This includes transformations with Lustre in general and model-to-model transformations with different languages. These transformations are not all reasoned on the need to generate an actual visual representation from text. The Lustre V3 compiler uses a controlflow transformation for efficient compilation.

2.1 Dataflow Languages

Lustre is one of the first synchronous languages. Unlike Esterel, it uses a dataflow approach for programming. The language Signal was also developed around the same time [GGB+91]. It is a

[1] http://www.esterel-technologies.com/products/scade-suite/
[2] https://de.mathworks.com/products/simulink.html

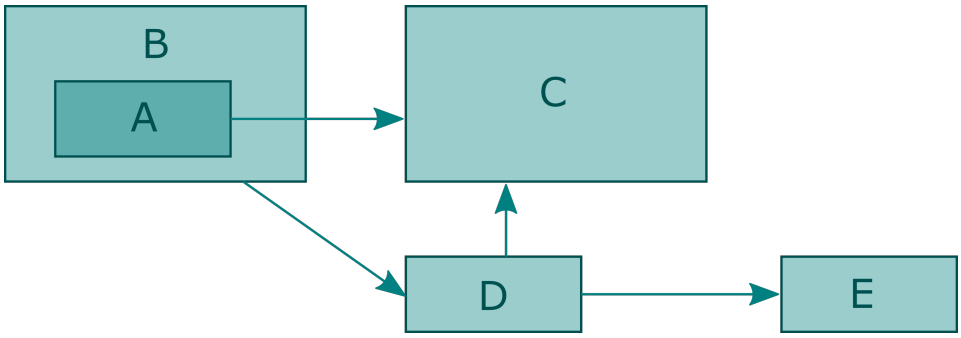


Figure 2.1. Example for a block diagram using hierarchical and flat transitions

```
1 node edge (c : bool) returns (e : bool);
2 let
3   e = c and not (false fby c);
4 tel
```

(a) Lustre

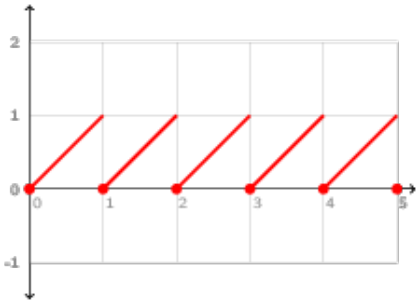
```
1 let node edge c = c & not (false fby c)
```

(b) Lucid Synchrone

Figure 2.2. Programs implementing edge detection in Lustre and Lucid Synchrone

```
1 let hybrid sawtooth() = x where
2   rec der x = 1.0 init 0.0
3   reset up(last x -. 1.0) -> 0.0
```

(a) Zélus program



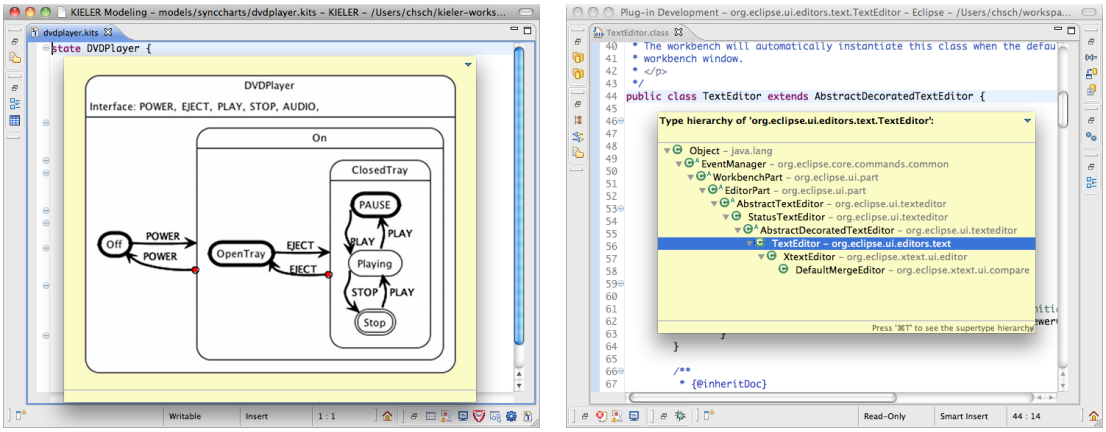
(b) The output in a coordinate system

Figure 2.3. Zélus example program with a sawtooth like output

dataflow language but it was specifically designed for systems. Other than programs, systems may be modified by adding, deleting or changing components at any time without the need to change the rest of the specification. They are specified as block diagrams. In Figure 2.1 an example for a block diagram is given. Components `A` a to `E` are connected and partially organized in a hierarchical manner. In Signal each block has its own clock so the environment and the system can be specified to run in different time instances.

Resulting from the three basic synchronous languages Lustre, Estrel and Signal, other dialects and variants have emerged. The language Lucid Synchrone is a functional extension of Lustre, also providing some features of ML languages [CP99]. This functional extension reduces the overall amount of code needed due to its functional structure. In Figure 2.2 a program detecting a rising edge is implemented in Lustre and in Lucid Synchrone. Lustre needs an explicid interface whereas Lucid Synchrone derives the input and outputs and their associated type through the defined function. The output variable does not need an explicit name in this case and therefore is simply defined through the right side of the equation.

The language Zélus is another example of a Lustre variant [BP13]. It is a rather new language, being developed in the last decade. The idea was to mix discrete logical time with a continuous time behavior. This is achieved by mixing the features of Lustre with Ordinary Differential Equationss (ODEs). In Figure 2.3a an example program in Zélus is presented. It takes no input, indicated by the `()` in line 1 after `sawtooth`. The output has the value of x, whereas x is defined by the derivative 1 with the initial value 0, indicated by `der x = 1 init 0` in line 2. So up to this point the value of `x` is defined by a linear function starting at 0 with the slope 1. However, the value is reset each time that the last value of `x` minus 1 crosses zero from negative to positive. This is the meaning of the statements `reset up(last x -. 1.0) -> 0.0`. Moreover, the value of `x` is defined through continuous time but also discrete time equations can be defined just like Lustre. Altogether, the program defines `x` to hold a sawtooth-like value. In Figure 2.3b the value of x is presented over time.



(a) Transient view generation as a state machine

(b) Transient view generation showing the class hierarchy

Figure 2.4. Examples for transient views supporting workflow used by Schneider et al. [SSH12]

2.2 Transient Views

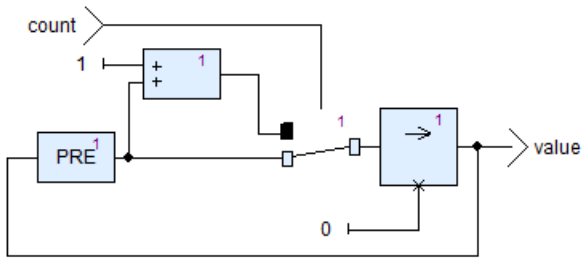
Designing a new system using visual models has the potential to consume a lot of time. Nodes and edges must be connected manually and the need for new components firstly creates the need to manually make room for new components.

Fuhrmann and von Hanxleden addressed the problems arising from graphical models [FH10]. Creating, maintaining or browsing visual models becomes time consuming and tedious. Additionally, Schneider et al. introduced the concept of a transient view [SSH12]. The idea is to synthesize a graphical view from an existing model automatically or on demand. This avoids the time consuming side effects of graphical modeling and combines them with the advantages. However, in order to provide transient views, automatic layout is needed.

The concept of transient views can be applied for various problem domains. An example that is applicable not only for synchronous languages is mentioned in Schneider et al. [SSH12]. During the process of programming the user might be given the option to show a visualization of the class hierarchy. Schneider et al. proposed the transient views in Figure 2.4 as a possible approach. This way the programming process is enhanced with the visual information about the class hierarchies, thus the user does not need to draw it themselves.

The concept of transient views was implemented in the context of KIELER. The language SCCharts has a visual and a textual semantics [HDM+14]. Designing a model allows textual editing and the visual representation of the model is generated automatically. This way the user is always offered both approaches. This thesis uses this concept to combine Lustre with the visual semantics of SCCharts dataflow.

For the scope of the thesis, thus the focus is on the language Lustre. Is defines the basis for SCADE and it is possible to express all Lustre feature with SCCharts.



(a) Visual

```
1 L1 = count;
2 value = L7;
3 L2 = 1;
4 L3 = pre L7;
5 L4 = L2 + L3;
6 L5 = if L1 then (L4) else (L3);
7 L6 = 0;
8 L7 = (L6) -> (L5);
```

(b) Textual

Figure 2.5. A counter in SCADE

2.3 Graphical Dataflow

The synchronous language Lustre is a dataflow language is based on a textual syntax. In order for Lustre to be visualized, a graphical representation of the dataflow is needed. However, there are several tools that use a graphical dataflow language. This section introduces theses tools and their approaches on how to visualize dataflow.

2.3.1 SCADE Suite

The SCADE Suite is a modeling environment using graphical components to express behavior [CPP17]. It has a compiler that is verified with several standards for avionics and automotive needs and holds sound semantics. Particularly, these semantics are based on the dataflow language Lustre. This visual representation defines a good outline for the visual representation that the transformation should achieve. It is frequently used and has a synchronous basis.

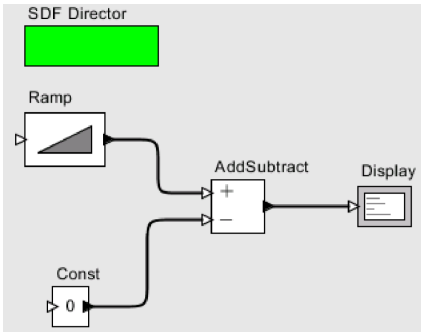
Each wire connecting components in SCADE is expressed as one equation in Lustre. Additionally, a new variable is introduced that holds the value of the wire. Each component receiving the wire as input uses the newly introduces variable in the equation. In Figure 2.5 a counter modeled in SCADE is shown. There is the visual model on the left side and the right side shows the conversion to the textual format with the resulting equations.

2.3.2 Ptolemy II, Simulink and LabView

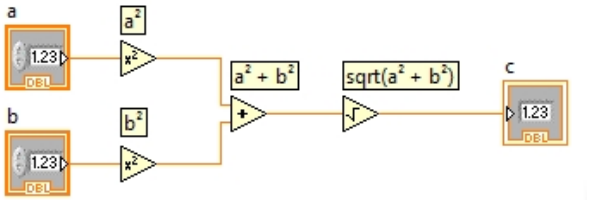
Besides SCADE there are several other tools using visual dataflow for modeling. Ptolemy II[3], Simulink[4] and LabView[5] are some examples we consider here, and they are all actor-oriented just like SCADE. All component entities are defined as actors and those can be predefined operators or self-defined operators that can be placed and connected. Nevertheless, they all work with different MoCs. Ptolemy II uses directors that can be added to the model. In Figure 2.6b the director is defined by the *SDF Director* component. This director determines the used MoC, for example dataflow, continuous time or discrete events. In the figure it is synchronous dataflow. In Simulink graphical blocks are used to visualize a component and a predefined set of basic blocks is available that are either discrete or continuous. In Figure 2.6a an example for a continuously designed function is given. Lastly, in LabView the MoC is based on availability. A component or actor executes when all its inputs are available. Multiple

[3]http://ptolemy.berkeley.edu/ptolemyII/
[4]https://de.mathworks.com/products/simulink.html
[5]http://www.ni.com/getting-started/labview-basics/

(a) Simulink: Calculating distance of a car from start point with gas pedal indicator



(b) Ptolemy II: model using the synchronous dataflow director



(c) LabView: model calculating the Pythagorean theorem

Figure 2.6. Examples for dataflow in Simulink, LabView and Ptolemy II

nodes might get their inputs available at the same time and this allows actors to execute inherently in parallel. In Figure 2.6c the addition operator for example only executes when both square operations yielded a result.

2.4 Diagram and Code Synthesis from Models

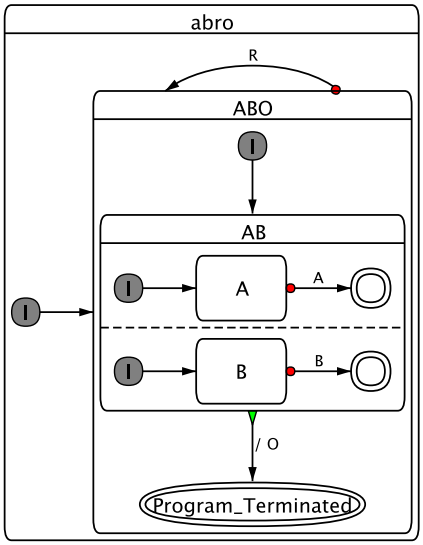
Visualizing Lustre requires a synthesis from the Lustre code to a model that is actually visualized. This type of synthesis from textual to visual is rather uncommon because it requires an automatic layout for the visualization. Tools like SCADE, Simulink, Ptolemy or LabView use the graphical model to extract or synthesize a textual form. The direction textual to textual is rather common for transformations within one model language. The Esterel language uses model-to-model transformation to take care of extended features [Ber02]. A set of kernel features is defined and other more complex features are just syntactic sugar and are simplified through transformations into kernel statements.

2.4.1 Visual Paradigm

The Unified Modeling Language (UML) tool Visual Paradigm[6] was developed by Visual Paradigm International. It can be used to model software from different perspectives. Detailed class diagrams, component diagrams or sequence diagrams are just some examples. The idea is to start the design process on an abstract level and think about the connection of different components. Similar to Simulink, LabView or SCADE the user uses a palette combined with drag-and-drop to build the model. There is also an automatic layout generation button for the model, but the connections of components must be done manually before that.

Moreover, this tool also includes model synthesizing options. The user has the option to generate code from the previously designed diagram. This code builds the stubs for the program fitting the

[6]https://www.visual-paradigm.com/



(a) Safe State Machine

```
1 module ABRO:
2 input A, B, R;
3 output O;
4 loop
5   [ await A || await B ];
6   emit O;
7 each R
8 end module
```

(b) Esterel

Figure 2.7. The ABRO program as Safe State Machine and in Esterel

designed software. This code generation process also works the other way around. The user may open their code and chose a diagram to be generated. Note that these techniques do not work with every diagram since not every diagram can be generated statically.

2.4.2 Safe State Machines to Esterel

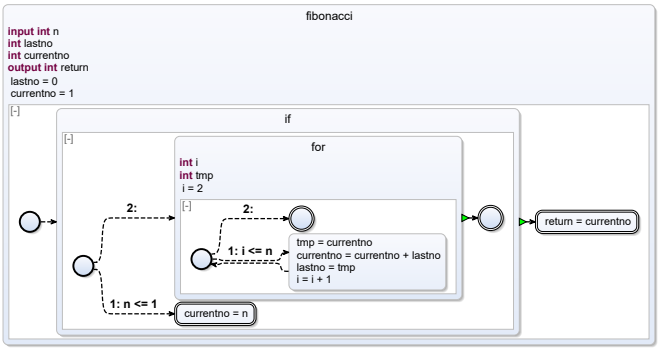
The language Esterel was one of the first three synchronous languages [BC84]. It is an imperative language and there is a graphical language, SyncCharts, that is semantically equivalent [And96]. Therefore, each Esterel program can be transformed into a SyncChart and every SyncChart can be translated into an Esterel program and they all have the same behavior. SyncCharts are the academic version of Safe State Machines (SSMs) but instead the the term SSM is often used.

Prochnow et al. presented their ideas on an automatic Esterel to SSM transformation [PTH06]. The motivation is similar to transforming Lustre to SCCharts, the benefits of both modeling approaches are combined. They introduce two steps to the transformation. First, the Esterel program is translated to an equivalent SSM. Second, this SSM is further optimized in order to create a more readable model.

In Figure 2.7 the ABRO program is shown in Esterel and as a SSM. The model has three inputs `A`, `B` and `R` and one output `O`. The model waits for `A` and `B` to be set, and then the output is emitted, too. The input `R` resets the model so it starts waiting for `A` and `B`. In Figure 2.7a the state machine is shown. The grey `I` states indicate the initial state. A state with a double-circled outline is a final state. Within the `AB` state, the inputs `A` and `B` need to be emitted for it to terminate. The strong abort on `ABO` with `R` causes the model reset. In Figure 2.7b the equivalent Esterel program is shown. The parallel statement of the two `await` corresponds to the `AB` state and the `loop ... each R` corresponds to the strong abort with the `R` as a trigger.

```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



(b) Visualization gained through transformation

Figure 2.8. Program computing the fibonacci sequence in C and as visualization [SLH16]

2.4.3 Other Syntheses and Transformations

Transforming models is not only relevant for the creation of diagrams. Also compiler can take advantage of transformations and user stories can be facilitated. In this subsection some other transformations and their rationales are introduced.

The language Lustre was already supported prior to this thesis. There was a transformation taking a Lustre program and transforming it to SCChart control flow developed by Clement Pascutto [Pas17]. Each expression was realized as two states with a transition connecting them. However, sometimes this approach does not give a good insight because this controlflow perspective does not always improve readability. Especially with highly nested expressions the level of hierarchy grows and thus the complexity. Figure 2.9 shows an example for the result of this transformation. The expression is evaluated hierarchically and then rebuilt using the associated variables. This SCChart represents the Lustre code but the abstraction from the dataflow to the controlflow comes at the cost more complex model in comparison to the original model.

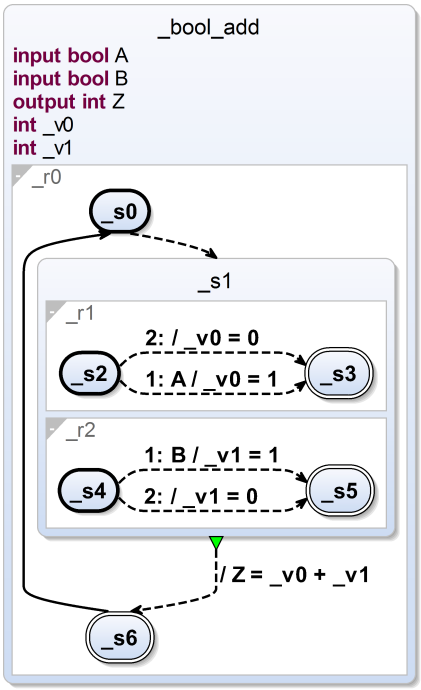
The rationale for the Lustre to SCChart dataflow transformation introduced is not only applicable for the language Lustre. There has been work on a visualization from C code [SLH16]. The greater goal is to get a visual insight on C code projects so the review of legacy code is facilitated. Also the maintaining can be enhanced by this feature because interconnection of components and the complexity can be extracted without further steps needed. In Figure 2.8 an example for this transformation is shown. The fibonacci sequence is implemented in C code in Figure 2.8a and in Figure 2.8b the corresponding visualization is shown. The conditional execution is shown by two transitions, one with a trigger, and the for-loop is hierarchically bundled.

The classical Lustre compilation approach constructs a finite state automaton from the dataflow code. This automaton uses the clocks in Lustre to determine the flow of control. Berry and Sethi introduced an efficient algorithm that is used to extract this automaton from the dataflow [BS86]. However, the new Lustre compiler [BBD+17] now uses a clock-directed approach [BCH+08]. This approach causes the code to be less efficient, however it produces a lot less code. In practice, the controlflow approach tends to cause the generated code to explode.

KIELER has a compiler using model-to-model transformations, and implementing new transformations is greatly facilitated through the modular transformation processor architecture [SSH18]. A

```
node bool_add(A: bool; B:bool) returns (Z
     : int);
let
Z = (if A then 1 else 0)
+
(if B then 1 else 0);
tel
```

(a) Lustre



(b) SCChart result

Figure 2.9. Example for a transformation to a controlflow SCChart from Lustre

transformation can simply be defined as a processor taking the one model as input and producing a model of the target type. Thus, many transformations are already implemented for different purposes.

Chapter 3

Preliminaries

The comparison of Lustre and SCCharts requires some basic knowledge about the languages themselves as well as theoretical aspects of synchronous languages are needed. Additionally, some tools and compilers are used and reused that are relevant for implementation and evaluation. This chapter gives a more detailed insight into the two languages Lustre and SCCharts and furthermore it introduces the technologies used.

3.1 Lustre

Besides basic operations, Lustre offers many extended features like array iterators, parametric nodes and user-defined types. However, for the scope of this thesis we limit the supported features of the Lustre language to a basic set. The relevant language feature are introduced in this section.

In Section 1.2 a Lustre program was already outlined. It was mentioned that it is made from nodes that consist of an interface, local variables and equations. In Figure 3.1 these different program parts are highlighted. Additionally, constants may be defined outside of nodes. An example for a constant declaration and initialization is also given in the Figure 3.1.

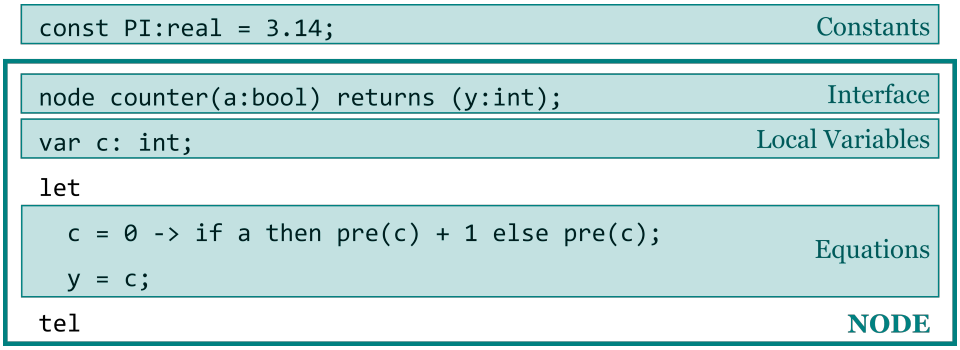


Figure 3.1. Structure of a Lustre program

With Lustre being a synchronous language, it abstracts time to be divided into discrete ticks. Variables in Lustre are called *streams*. They are infinite sequences of values with each value defining the variable for a tick. A variable $x = (x_1, x_2, x_3, \ldots)$ has the value x_1 in the first tick, x_2 in the second tick and so on. Each value of the variable is defined through an equation, thus the value of a variable in tick n is the value of the respective equation in tick n.

Table 3.1. Numerical and boolean data operators in Lustre

Syntax	Semantics
`- x`	Sign
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x div y`	Integer Division
`x mod y`	Modulo
`x > y`	Greater
`x < y`	Less
`x >= y`	Greater or Equal
`x <= y`	Less or Equal
`x <> y`	Unequal
`x = y`	Equal
`if a then x else y`	Conditional

Syntax	Semantics
`not a`	Not
`a <> b`	Unequal
`a = b`	Equal
`a or b`	Or
`a and b`	And
`a xor b`	Exclusive Or
`a => b`	Implies
`if c then a else b`	Conditional
`#(a, b, ...)`	At most one
`nor(a, b, ...)`	None of

3.1.1 Operators

The operators available in Lustre can be divided into two categories, the *data operators* and the *sequence operators*. The data operators can be unary, binary or n-ary operators. They work on each tick instance of a variable, so for two variables $x = (x_1, x_2, x_3, \ldots)$ and $y = (y_1, y_2, y_3, \ldots)$, an arbitrary binary data operation $\circ$ is defined through

$$x \circ y = (x_1 \circ y_1, x_2 \circ y_2, x_3 \circ y_2, \ldots).$$

In Figure 3.1 the supported data operators are listed. They are divided into numerical operators working with integer or real numbers and boolean operators. The respective syntax of the operations and their definitions are listed as well. The separating lines in between the table structure the operators into groups of the same arity.

Besides the data operators, there are four sequence operators. They can be used to work on and manipulate streams, thus the sequence of a variable's values.

The first sequence operator is *pre*. It can be used to refer to the previous value of a variable, usually the value in the previous tick. In the first tick the value of *pre* is not defined, indicated by *nil*. Altogether, the *pre* operator for a variable $x = (x_1, x_2, x_3, \ldots)$ is defined through

$$\texttt{pre}(x) = (\texttt{nil}, x_1, x_2, x_3, \ldots).$$

As already mentioned, this introduces an undefined value at the first tick. For this cause the *initialization* operator is introduced. It takes the value of the left side in the first tick and the value of the right side in all following ticks. So for two variables $x = (x_1, x_2, x_3, \ldots)$ and $y = (y_1, y_2, y_3, \ldots)$ the *initialization* is defined through

$$x \texttt{ -> } y = (x_1, y_2, y_3, \ldots).$$

The *pre* and the *initialization* are often used together. Therefore, in Lustre there is a convenient language addition that facilitates the combination of both operators. It is called the *followed by* operator, used with `fby`. For two variables $x = (x_1, x_2, x_3, \ldots)$ and $y = (y_1, y_2, y_3, \ldots)$ the *followed by* is defined through

$$x \texttt{ fby } y = x \texttt{ -> pre}(y) = (x_1, y_1, y_2, \ldots).$$

Table 3.2. Example for clock sampling and projecting using *when* and *current*

Name	Stream of values								
clk	true	false	true	true	false	true	false	false	true
b	true	false	false	true	true	false	false	true	true
b1 = b when clk	true		false	true		false			true
x	0	1	2	3	4	5	6	7	8
x1 = x when b1	0			3					8
x2 = current x1	0		0	3		3			8
x3 = current x2	0	0	0	3	3	3	3	3	8

The last two sequence operators are for sampling and projecting of streams. Each stream has an associated clock. For simple streams this is the base clock, so the clock with the highest frequency. However, other frequencies can be specified that cause the value of the variable to only be computed in certain time instances given through the clock. The *when* operator can be used to sample a stream. The expression x `when` b with b being a boolean variable only holds a value in those ticks where b is true. The value is defined by the value of x in that tick. Table 3.2 illustrates an example for the sampling. In the third row the expression b is sampled to the clock clk. Note that the expression b `when` clk does not define a value in the ticks where b is $false$. Likewise, row five samples the expression x to the clock $b1$.

The opposed operation, the projection, is defined by the *current* operator. For an expression b `when` clk, the operation `current`$(b$ `when` $clk)$ is defined to run on the clock of clk and project the value of b `when` clk to all tick instances that need to be filled. In Table 3.2 there are some examples on how the current operator works. In line 6 the expression $x1 = x$ `when` $b1$ is projected. Because $b1$ runs on the clock clk the operation `current`$(x$ `when` $b1)$ is defined to run on the clock of $b1$, thus clk. In line 7 this result is again projected. Because clk is on the base clock, this causes the values to be projected into all missing tick instances.

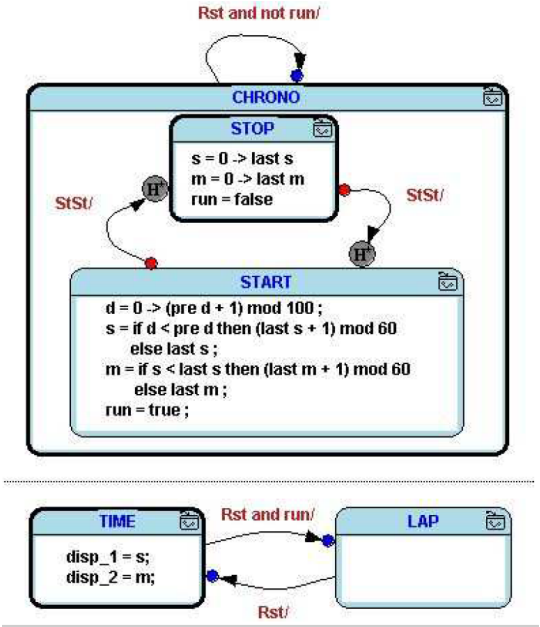
```
1  node AND (a:bool; b:bool) returns
2    (o:bool);
3  let
4    o = a and b;
5  tel
6
7  node ORXOR (a:bool; b:bool) returns
8    (resultOr:bool; resultXor:bool);
9  let
10   resultOr = a or b;
11   resultXor = a xor b;
12 tel
```

(a) Nodes of the Lustre program that are referenced

```
10 node ANDOR (a:bool; b:bool) returns
11   (resultAnd:bool; resultOr:bool; resultXor:
         bool);
12 let
13   (resultOr, resultXor) = ORXOR(a, b);
14   resultAnd = AND(a, b);
15 tel
```

(b) Lustre node referencing other nodes

Listing 3.3. Example for a Lustre program using the reference mechanism



(a) Visualization as state machine

```
node chrono (StSt:bool; Rst:bool)
 returns (disp_1:int; disp_2:int);
let
 automaton
  CHRONO ->
   automaton
    STOP ->
     s = 0 -> last s
     m = 0 -> last m
     run = false
    unless StSt continue START
    | START ->
     d = 0 -> (pre d + 1) mod 100
     s = if d < pre d
          then (last s + 1) mod 60
          else last s
     m = if s < last s
          then (last m + 1) mod 60
          else last m
     run = true
    unless StSt continue STOP
   end
 until Rst and not run then CHRONO;
 automaton
  TIME ->
   disp_1 = s
   disp_2 = m
   until Rst and run then LAP
  | LAP ->
   until Rst then TIME;
tel
```

(b) Lustre Program

Figure 3.3. A Lustre program using the state machine extension introduced by Colaço et al. [CPP05]

3.1.2 Node References

Besides the operators introduced above, Lustre also allows to call already defined nodes. In Figure 3.1 the general program structure is outlined but besides the one defined node, multiple nodes may be defined within one file. These nodes can then call the others assuming their call tree is acyclic. This allows to modularize the functionality and behavior into nodes. The inputs are provided to the called node through the braces.

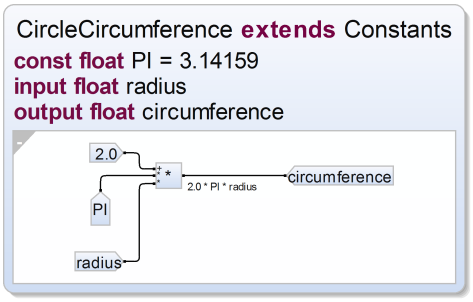
In Figure 3.3 an example for the usage of a reference call is shown. The node `AND` only has one output, so it can be simply referenced on the right side of the equation. For the node `ORXOR` there needs to be a tuple on the left side of the equation because it has two outputs.

3.1.3 State Extension

In 2005 Colaço et al. introduced a conservative extension to the classical Lustre/ Lucid Synchrone. They added state machines as an alternative way to express the program behavior besides dataflow

```
scchart CircleCircumference extends Constants
    {
  input float radius
  output float circumference

  dataflow {
    circumference = 2.0 * PI * radius
  }
}

scchart Constants {
  const float PI = 3.14159
}
```

(a) Textual



(b) Visual

Figure 3.4. An SCChart calculating the circumference of a cycle with the usage of an imported constant from another model

[CPP05]. This extension is conservative in the way that all programs in the basic language still work and behave the same. The basis for this extension is the usage of clocks. In dataflow they offer a way to express conditional execution and are often seen as the counter part to controlflow. Allowing the specification of state machines combines both possible modeling approaches. In SCADE this extension is also used, allowing the user to chose between the modeling perspectives.

Figure 3.3 shows an example for a Lustre program using the state machine extension. It is taken from Colaço et al. except for the program being in Lustre and not in Lucid Synchrone. It simulates a stopwatch with a start and a reset button. The outputs `disp_1` and `disp_2` show the seconds and minutes since the start.

In general the automaton extension can be used at the same place equations qould be defined. Instead of specifying an equations in between the `let` and `tel` keywords, one can also define an automaton with the keyword `automaton`. Following the keyword, a state can be defined and optionally equations follow that shall be executed within the state. The transitions have two different variations for defining their kind. The first option defines the stop behavior. An `unless` introduces a strong abort, thus when the condition is met, the transition is taken immediately. However, the `until` defines a weak abort. The second option defines how the target state is entered. It is differentiated between `then` and `continue`. The continue starts the target state in the state it was before leaving, it defines a history transition. The `then` keyword does not save the state of the target state.

3.2 SCCharts

SCCharts are a synchronous language extending the classical synchronous MoC with sequential constructiveness [HDM+14]. They use a StateChart notation based on the Harel StateCharts [Har87] and additionally allow the specification of dataflow. The syntax is given on a textual and a visual level.

The entry to a program is defined through a root SCChart, a state that contains all the modeled behaviors. It is possible to specify another SCChart and import it through the `extends` feature. This feature is oriented on the inheritance in object-oriented languages and provides a possibility for importing declarations, regions or actions. For this thesis we only need this features for importing declarations. In Figure 3.4 an example for the usage of the `extends` feature is given. The constant `PI`

in the model `Constants` can be imported in this calculation of the circumference but it would also be possible to additionally import it in other models. This renders the constant variable `PI` visible in the scope of the `CircleCircumference`.

Within a root state of an SCChart different regions can be defined and they all run in parallel. These regions can be defined as either a controlflow region or a dataflow region. In the following we take a look into both features because the controlflow regions are interesting for the state extension of Lustre. The dataflow regions however, offer a good representation for the classical Lustre.

Concurrent regions using the same variables may induce data dependencies. In classical synchronous languages this problem is faced by allowing one value for each signal or stream value for each tick. However, the SCCharts use sequential constructiveness. Instead of signals or streams we use variables. Those may hold different values during one tick and this value persists beyond tick boundaries. This requires a regulation for variables being modified concurrently.

Within the scope of SCCharts this problem is solved by introducing the *initialize - update - read* protocol [HDM+14]. This protocol defines the ordering of variable access in concurrent regions. The first operation performed is therefore the initialization of a variable. All assignments that are not classified as update are considered to be an initialization. Following initialization, updates on the variable are performed. These are defined through a commutative function f in an assignment of the form $x = f(x, e)$ whereas e does not reference x. Lastly, the reads for a variable are scheduled. This ensures that the variable has reached its final value, without considering sequential modification after the read.

3.2.1 Controlflow

The controlflow regions of SCCharts use state machines for defining the program. The states can be specified as initial, final or regular states. Initial states are used as starting point for the execution and final states mark the termination of execution for that region.

In Figure 3.5 the ABRO example introduced in Figure 2.7 is implemented in SCCharts using controlflow. It takes three boolean inputs `A`, `B` and `R`. It waits for `A` and `B` to be set to true and as soon as this happens, the boolean output `O` is also set to true. In case that the input `R` is set, the behavior is reset. The states `wA`, `wB`, `WaitAB` and `ABO` define initial states, and `dA` and `dB` are final states. The remaining state `done` is a regular state.

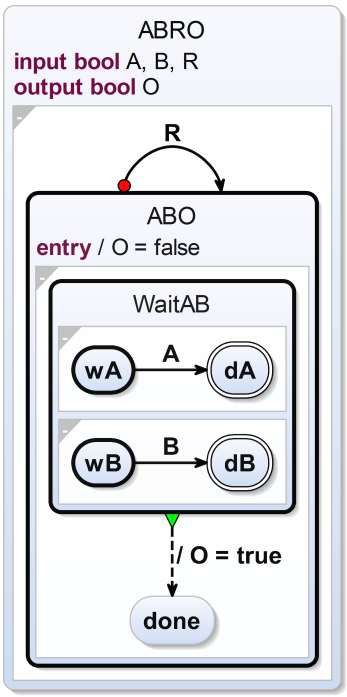
Transitions also have different types. They can be immediate or delayed, where immediate transitions react without implying a tick boundary. Additionally, they can specify a weak abort, a strong abort or a termination transition. Weak and strong abort define whether the source state of the transition is allowed to perform an action when this transition triggers. The weak transition is for example used as transition from `wA` to `dA`. The strong abort transition is used at the self transition on `ABO` that triggers with `R`. In contrast to a weak abort, the strong aborts prevents the source state to execute its inner behavior. Lastly, the transition from `WaitAB` to `done` is a termination transition. It ensures that the source state has terminated, so all its inner behavior has reached a final state.

3.2.2 Dataflow

The Lustre language is a dataflow language, so the possibility to specify dataflow in SCCharts is especially interesting. As already mentioned, one can define a region as dataflow and then equations are provided instead of states and transitions. In Figure 3.6 an example for an SCCharts using dataflow is illustrated. The inputs and outputs are specified just like in the controlflow example. However, the

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

  initial state ABO {
    entry do O = false

    initial state WaitAB {

      region {
        initial state wA
        if A go to dA

        final state dA
      }

      region {
      initial state wB
      go to dB if B

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

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

(a) Textual

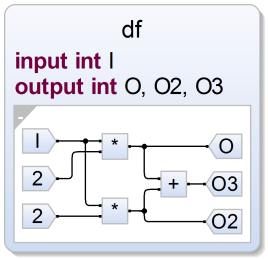


(b) Visual

Figure 3.5. An SCCharts modeling the ABRO exampling using controlflow regions

```
scchart df {

  input int I
  output int O, O2, O3

  dataflow {
    O = I * 2
    O2 = I * 2
    O3 = O + O2
  }
}
```

(a) Textual



(b) Visual

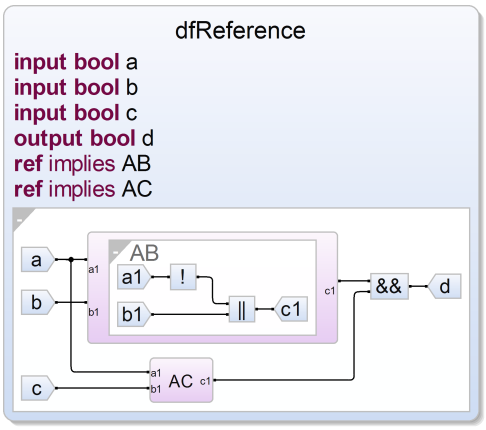
Figure 3.6. An SCCharts with a dataflow region

```
1  scchart dfReference {
2    input bool a
3    input bool b
4    input bool c
5    output bool d
6    ref implies AB
7    ref implies AC
8
9    dataflow {
10     AB.a1 = a
11     AB.b1 = b
12     AC.a1 = a
13     AC.b1 = c
14     d = AB.c1 && AC.c1
15   }
16 }
```

(a) Root SCChart

```
1 scchart implies {
2   input bool a1
3   input bool b1
4   output bool c1
5
6   dataflow {
7     c1 = !a1 || b1
8   }
9 }
```

(b) Referenced SCChart



(c) Visual

Figure 3.7. An SCCharts with a dataflow region and a reference implementing the boolean function *implies*

`dataflow` keyword introduces a new region using dataflow. Equations can now be defined and in the visualization in Figure 3.6b these equations are synthesized into a network of components and wires.

Besides predefined operators, also already defined SCCharts can be used as operator. Figure 3.7 illustrates an example program that references another SCChart. The boolean function *implies* is realized in a separate SCChart using the equivalence rule $A \Rightarrow B \Leftrightarrow \neg A \vee B$. This function is referenced in the main program two times. The inputs and outputs of the referenced SCCharts need to be connected to values or variables through equations. In lines 10 to and 13 the proper values are bound to the operator as input. The output is forwarded to the output `d` of the main model as the conjunction of both outputs. In the visual representation the referenced node can be expanded as shown in the Figure 3.7c for the first reference, `AB`, or it can be collapsed to hide the implementation details as shown for the second reference, `AC`.

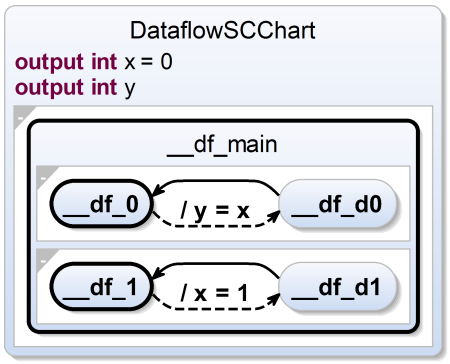
3.2.3 Semantics of SCCharts Dataflow

The implemented transformation focuses on the usage of dataflow SCCharts and thus their semantics are important in order to properly understand the generated models. Dataflow SCCharts can be defined through a dataflow-typed region that includes an arbitrary number of equations. Those equations are treated as *concurrent assignments* that are executed each tick. Each assignment is transformed to its own region that only handles the execution of the designated assignment. Figure 3.8a shows a dataflow SCChart with two concurrent assignments `y = x` and `x = 1`. In Figure 3.8b this SCChart is illustrated with a concurrent regions for each assignment. The assignments `y = x` and `x = 1` are included in the dataflow region, thus there are two concurrent regions in the SCChart that perform the corresponding assignment once per tick.

This concurrent realization of the assignments in a dataflow region in SCCharts differs from the controlflow approach. Figure 3.8 show examples of dataflow and controlflow SCCharts and their visualization using controlflow constructs. Additionally, in both approaches the Sequentially Constructive Language (SCL) is shown that is generated from these SCCharts. This SCL is a textual imperative language that is used during the compilation process as a minimal basis to express concurrency. Assignments

```
scchart DataflowSCChart {
  output int x = 0
  output int y

  dataflow {
    y = x
    x = 1
  }
}
```

(a) A dataflow SCChart with concurrent assignments



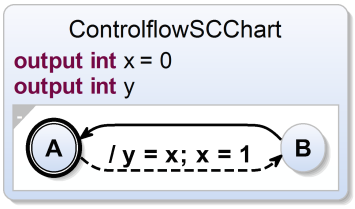
(b) Resulting concurrent regions from the dataflow SCChart

```
module P2
int x, y;
{
  x = 0;
  fork
    one:
    x = 1;
    pause;
    goto one
  par
    two:
    y = x;
    pause;
    goto two
  join;
}
```

(c) The SCL version of the dataflow program

```
scchart ControlflowSCChart {
  output int x = 0
  output int y

  region {
    initial final state A
    immediate do y = x; x = 1 go to B

    state B
    go to A
    }
}
```

(d) A controlflow SCChart with sequentially ordered assignments



(e) A visual controlflow SCChart with two sequentially ordered assignments

```
module P1
int x, y;
{
  x = 0;
  tick:
  y = x;
  x = 1;
  pause;
  goto tick
}
```

(f) The SCL version of the controlflow program

Figure 3.8. A dataflow and a controlflow SCCharts in textual form, in the SCChart visualization using only controlflow and the resulting SCL

Table 3.3. Language scope of SCCharts in comparison to Lustre

	Lustre	SCCharts Dataflow	Lustre Equivalent
`x = pre(x)+ 1`	ok	ok	

`x = x + 1`	invalid	ok	`x = pre(x)+ 1`
			OR
			`x = M(x)+ 1`

`x = 1` `x = x + 1`	invalid	ok	`x0 = 1` `x = x0 + 1`

`x = 1`
`x = x + 1`
`y = x`
	invalid	ok	`x0 = 1`
			`x = x0 + 1`
			`y = x`

on transitions or actions result in sequentially ordered assignments in the SCL. The syntax of those sequential assignments in SCL looks similar to those assignments that can be included in the dataflow region of SCCharts.

Comparing the Figures 3.8a and 3.8f shows that both variants include the two assignments `y = x` and `x = 1`. In the SCL these assignments are treated like assignments in imperative languages. First, `y` is set to `x` and then `x` is set to `1`. This results in `y` with the value `0` and `x` with the value `1` after the first tick.

However, the assignments in dataflow regions are treated differently. They are defined to execute concurrently, thus they do not imply sequential ordering but each assignment is calculated concurrently. Therefore, changing the order of the assignments does not result in a different behavior for the model. SCL realizes this concurrency through the `fork ...par ...join` construct as shown in Figure 3.8c.

Nevertheless, the two concurrent regions in the dataflow example in Figure 3.8b and 3.8c introduce data dependencies for the variable `x`. Concurrent variable access in SCCharts is ordered through the protocol *initialize-update-read*. Therefore, no matter the syntactic ordering, the initialization `x = 1` is done before the read of `x` in the assignment `y = x` is performed. Therefore, after the first tick, the variable `y` holds the value `1` and `x` holds the value `1`.

This protocol for concurrent variable access and the usage of variables instead of streams is the main difference of SCCharts dataflow and Lustre. In Table 3.3 some examples are shown that contrast the SCCharts dataflow and Lustre. The first line shows a regular *pre* operation. Those are supported in both languages and they also behave the same. In the second line an update is performed. Updates define the value of a variable relative to its current value. These assignments are not allowed in Lustre. Each assignment is an actual equation, thus the left and the right side of the equation are defined to be equal. Updates, however, define the value of a variable in relation to its current value. Equations like `x = x + 1` do not yield equivalence and are thus not supported.

However, SCCharts use variables and the read value for `x` is either `pre(x)` or the value of `x` that is created in the current tick, indicated through the `M(x)`. The third line executes two equations concurrently. In SCCharts this concurrent access is ordered through the *initialize-update-read* protocol. However, in Lustre this protocol does not exist, thus there are two writes to the same variable which results in an error. Using Static Single Assignment (SSA) with an extension for concurrent regions can help to create versions for variables so that their concurrent access conforms with the Lustre semantics again. These versions of the variables are created such that each variable is only assigned to once. The corresponding SSA version for the third line is shown in the column *Lustre Equivalent* of the

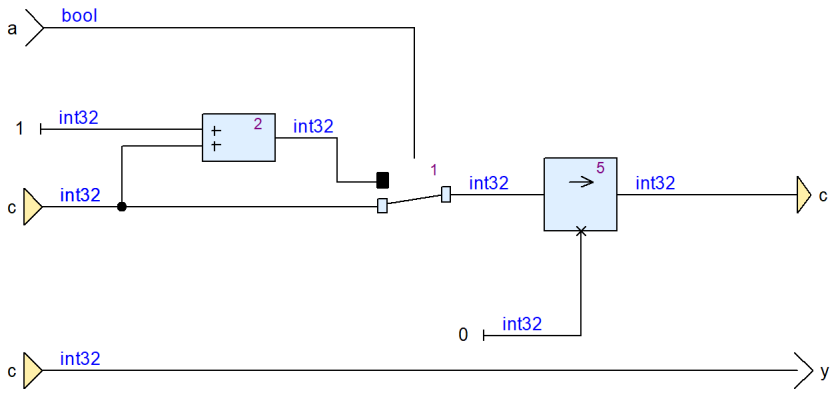


Figure 3.9. Counter modeled in SCADE with a causal cycle for the local variable c

1 error(s) detected - 0 warning(s) detected

Category	Code	Message
Causality Error	ERR_400	**Causality error at mocExample::Counter/ _L33=** the definition of flow _L33 depends on flow c ; (mocExample::Counter/c=) the definition of flow c depends on flow _L32 ; (mocExample::Counter/ _L32=) the definition of flow _L32 depends on flow _L34 ; (mocExample::Counter/ _L34=) the definition of flow _L34 depends on flow _L38 ; (mocExample::Counter/ _L38=) the definition of flow _L38 depends on flow _L33 ;

Figure 3.10. Results of the Check option for the counter with a causal cycle

corresponding line. The fourth line extends the usage of the *initialize-update-read* protocol by a read in the assignment of y. Just like the line above, these assignments are valid in SCCharts but not in Lustre. The Lustre equivalent now reads from the last variable version that is assigned. In this example it is variable x1.

In conclusion, SCCharts dataflow extends the Lustre semantics. Regular dataflow assignments just like those in Lustre can be expressed. However, also sequentially constructive features can be used in SCCharts dataflow assignments. Those could be transformed back to Lustre using SSA-like principles. Sequential variable access and concurrent access through the *initialize-update-read* protocol is included in the sequentially constructive feature scope. Lastly, the persistence of variable values beyond tick boundaries allows for implicit *pre* operations. Therefore, SCCharts dataflow is a combination of Lustre, the *initialize-update-read* protocol and an automatic *pre* achieved through variables persistence.

3.3 Used Technology

The first goal of this thesis is the comparison between the SCADE Suite and KIELER. Both are designed to build synchronous models using visual components. Moreover, KIELER is an Eclipse project, though there are also variants using web based technologies [Ren18; Dom18]. In order to compare KIELER and SCADE properly, they are first introduced on a more detailed level.

3.3.1 SCADE Suite

The Safety Critical Application Development Environment (SCADE) Suite is a tool that is designed to handle large projects using a model-based perspective. For the scope of this thesis we used the SCADE Suite 19.2 combined with the SCADE compiler KCG 6.6. The user may choose between capturing dataflow or controlflow and a basic set of predefined operators is provided. For further needs there are also optional SCADE Suite libraries. They need to be imported explicitly and offer more advanced features such as temporal operators needed for verification.

In Section 1.2 the SCADE user interface was already introduced. As already mentioned, the actual behavior is modeled in Area 2 of Figure 1.3. However, SCADE has a synchronous backbone. Therefore it is possible to design models that do not conform with the requirements. As an example we take the counter in Lustre from Listing 1.2 and model it in SCADE. In Figure 3.9 this counter is illustrated. There is a causal cycle within the definition of `c`. It is initialized with `0` and after the initialization it depends on the value of itself. SCADE offers the possibility to check operators for possible problems. This can be done by right-clicking on the operator and choosing the `Check` entry. For the given examples this yields the result shown in 3.10. A causality error was found regarding the flow definitions.

3.3.2 Lustre V6 Compiler

The Lustre language is the backbone of SCADE but it received adjustments and extensions for some functionalities so the SCADE compiler is specific to the SCADE/ Lustre dialect. Therefore, the compiler for the classical Lustre language is needed for testing and evaluating pure Lustre. We use the academic Lustre V6 compiler offered by the research center at Verimag[1]. The language scope of Lustre V6 is described in the Lustre reference manual [JRH16].

The compiler is named `lv6`. For the purpose of evaluating, comparing and simulating programs we want to compile Lustre to C code. This can be achieved using the options `-2c` and `-n` followed by the name of the node that is treated as main node. The `-n` can be omitted if the main node is name the same as the file name. So assuming we have a program file `program.lus` with a defined node `node`, we can compile it to C code using `lv6 program.lus -2c -n node`. The compiler now generates the following files:

```
▷ lustre_types.h
                          ▷ filename_node.h          ▷ filename_node_loop.c
▷ lustre_consts.h
                          ▷ filename_node.c          ▷ node.sh
▷ lustre_consts.c
```

The files `lustre_types.h`, `lustre_consts.h` and `lustre_consts.c` are generated for including user defined types and constants. They are always generated no matter the content of the user-defined file. The `filename_node.h` and `filename_node.c` contain the actual behavior of the node. In case of the node having an internal state, there is a `step` function and a `reset` function. Additionally a `struct` is needed as parameter that saves the memory of the node. In case the node does not have an internal state, the generated files also contain only one function with the same inputs and outputs the node/ function defines. Lastly the `filename_node_loop.c` defines a simple console simulation, initializing the structures if needed and parses system in and out. The `node.sh` is for convenient compiling of the `filename_node_loop.c` file.

[1] http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/

These files and their purposes are important for the embedding of the Lustre compiler into the context of KIELER. C files can be compiled and simulated but the inputs must be provided properly and the outputs need to be passed on.

3.3.3 Eclipse

Eclipse[2] is an open-source Integrated Development Environment (IDE). Initially it was developed for Java support but there is also support for many other languages such as C, C++ and Ada. It is made up from plug-ins that are based on Equinox[3], an implementation of the Open Service Gateway initiative (OSGi) core framework specification. The goal is to provide a modular, extensible and loosely coupled system. Functionality is encapsulated in plug-ins and they may provide or extend so called *extension points*. This opens up the possibility for a plug-in to contribute something to another plug-in. A simple example is an extension of the ui menu bar with additional buttons.

A minimal set of plug-ins needed for an application is bundled in the Eclipse Rich Client Platform (RCP). Combining this basic set with self-defined plug-ins offers the possibility to define a specialized domain-specific development environment. There exist many other Eclipse projects that further facilitate the creation of such an IDE. KIELER is built up from this RCP in combination with frameworks like the Eclipse Modeling Framework (EMF) and Xtext to offer model-based design features and language support for SCCharts.

3.3.4 EMF

The Eclipse Modeling Framework (EMF)[4] is an Eclipse project offering basic features for structured data models. The models are defined in XML and EMF provides tools and runtime support to generate a set of Java classes from these models. Additionally adapter classes are built that enable viewing the model, provide a basic editor and a command-based editing of the model.

In KIELER these EMF models are for example used as a meta model for the supported languages such as SCCharts and Esterel.

3.3.5 Xtext

Xtext[5] is an Eclipse project that offers a grammar languages that greatly facilitates the development of domain-specific languages. A parser, a linker for cross-references, a typechecker, and editing support like code-completion or syntax-highlighting go along with the definition of a language in Xtext. Model-based data structures are extracted from the grammar that specify the meta-model using the EMF framework.

All textual languages in KIELER use Xtext for their implementation. Also the textual SCCharts have a corresponding Xtext grammar.

3.3.6 KIELER

KIELER[6] is an open-source research project focusing on enhancing the graphical model-based design. It implements the RCP and used EMF and Xtext for the implementation of domain-specific languages such

[2] https://www.eclipse.org/
[3] https://www.eclipse.org/equinox/
[4] https://www.eclipse.org/modeling/emf/
[5] https://www.eclipse.org/Xtext/
[6] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Home

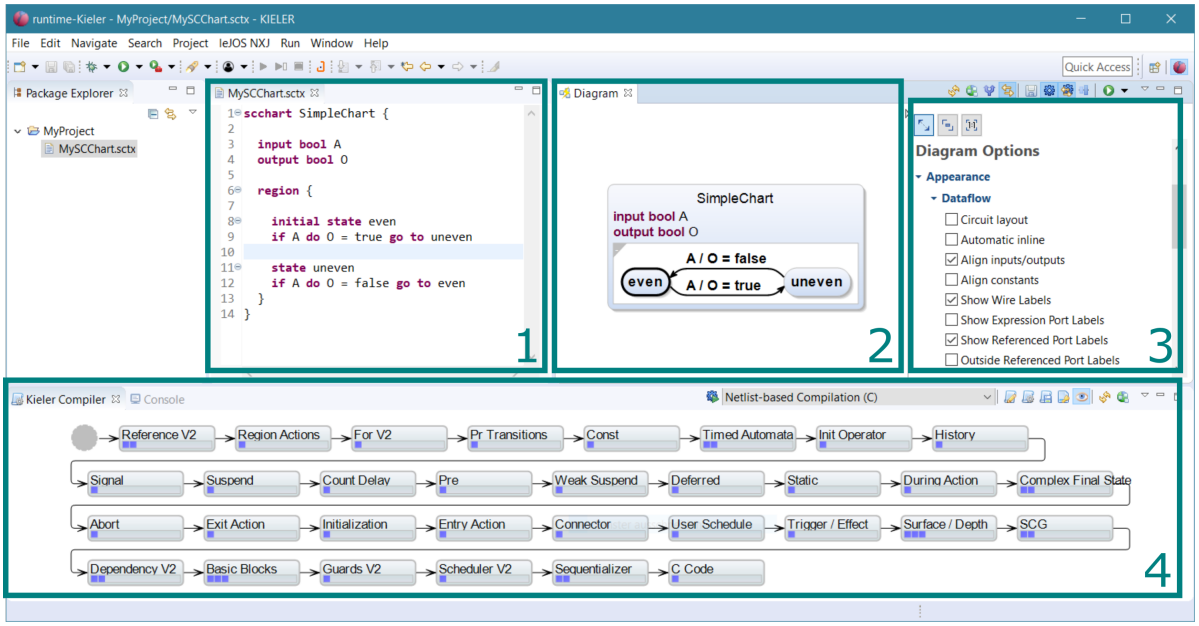


Figure 3.11. KIELER user interface in the modeling perspective

as SCCharts or Esterel. The idea of the project is to improve comprehensibility of diagrams, development time, maintenance time and the analysis of dynamic behavior.

In Figure 3.11 the user interface of KIELER is shown. An SCCharts model is opened in a textual editor in Area 1 and the visual representation is shown in the diagram view in Area 2. It is generated automatically every time the textual model is changed. The visual representation may be modified by the user using the options side bar in Area 3. Area 4 shows the compile chain.

KiCo

Besides the automatic layout generation, KIELER offers a special kind of compiler implementation, called KIELER Compiler (KiCo) [SSH18]. In Figure 3.11 in Area 4, the visual parts of the selected compiler are already shown. The general idea is for a model to be compiled in small steps and providing the intermediate result of each step to the user. The different components in the Area 4 represent one of these small steps, called *processors*.

A *processor* marks the smallest compilation unit in KiCo. They can perform different tasks such as transformations, optimizations or analysis and therefore offer a great range of functionality. A list of processors forms a *process system*. The connected components in Area 4 of Figure 3.11 show the selected process system. It is a compilation system that transforms step-by-step the SCChart model to C code. Each processor may include intermediate results, shown by clicking on the blue rectangle within the component. This offers the possibility to manually trace the effect of the different processors.

In SCCharts there are certain features that are called *extended features*. In Figure 3.12 the different features and their classification as extended or core feature is shown. The compilation chain from SCCharts to C code includes processors that expand these extended features to core features. This compilation chain is shown in Figure 3.13.

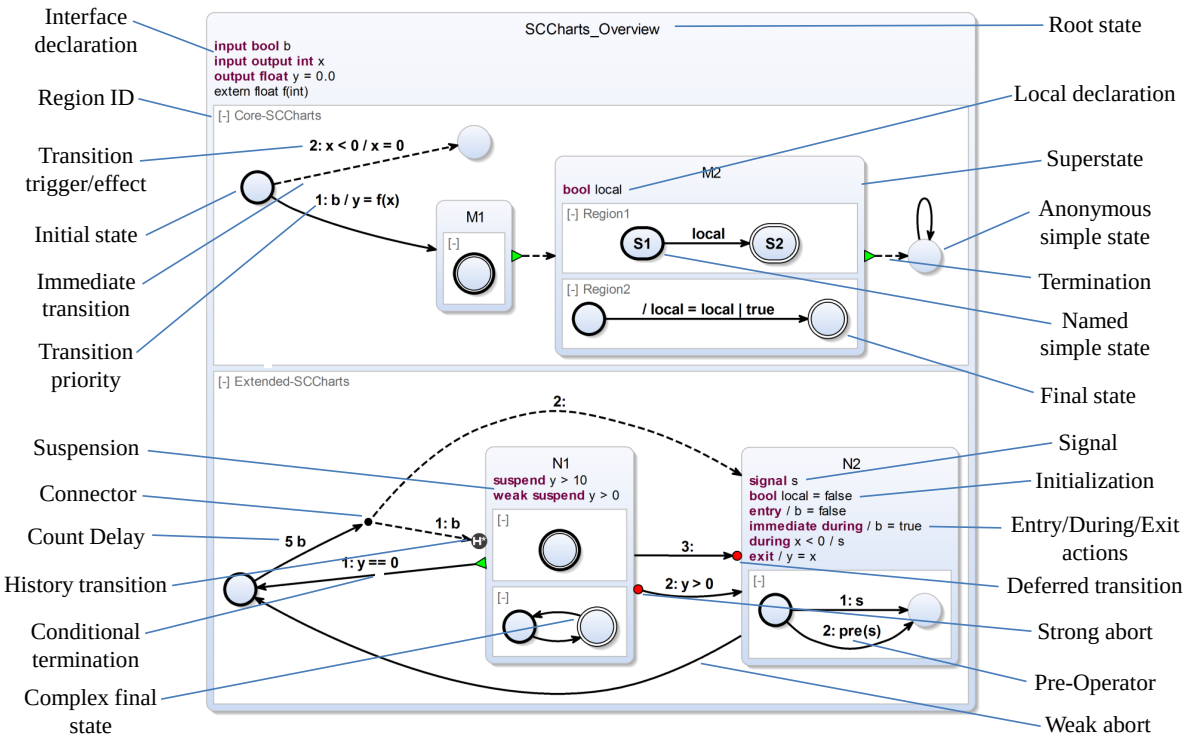


Figure 3.12. Overview of the core SCCharts feature in the upper region and the extended features in the lower region [HDM+14]

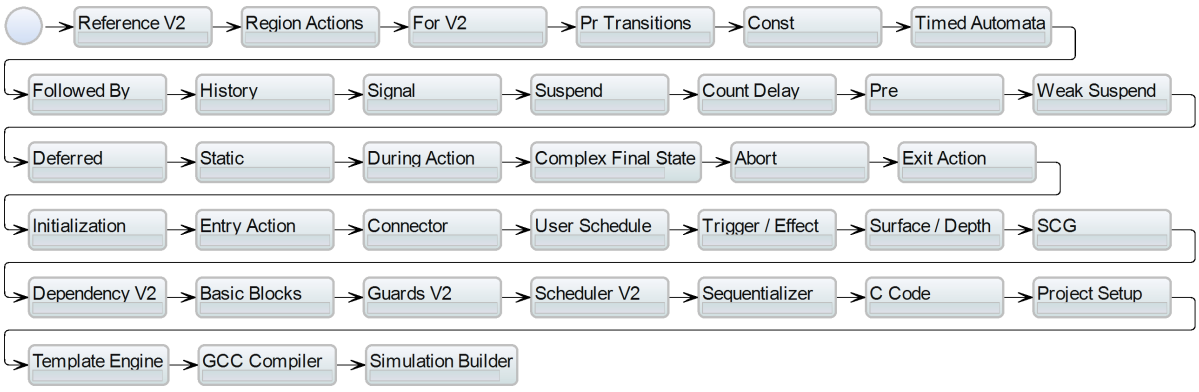


Figure 3.13. The compilation chain for SCCharts to C code

In KIELER this compiler framework is also used for model-to-model transformations. A lot of languages are supported in the KIELER project and often transformations to SCCharts are provided. These SCCharts can then use other existing systems to compile to for example C code.

Chapter 4

Concept

The languages Lustre and SCADE are equivalent in their expressiveness with only one-way conversion provided. Lustre is based on a textual syntax and SCADE defines a visual syntax. The goal of this thesis is the transformation from Lustre to SCCharts dataflow to enhance the modeling process. This requires the languages Lustre and SCCharts to be sufficiently equivalent.

This chapter introduces the conceptual ideas towards this transformation. First, the languages SCCharts and Lustre and their corresponding computation models are compared in order to define a formal foundation for the transformation. Moreover, the visual syntax of SCADE and SCCharts is contrasted to show that SCCharts serve as a good visual representation for Lustre.

Next, the concept of a transformation from Lustre to SCCharts is introduced. There are certain aspects in both languages such as streams and variables that differ and might cause the program to behave differently in the transformed language if not considered carefully. The differences and the approaches to overcome those differences are introduced.

The transformation sometimes allows to convert non-valid Lustre programs to SCCharts and the sequentially constructive MoC may cause the program to have a defined and valid behavior in SCCharts. However, this sometimes causes the dataflow visualization to contain cycles within the usage of a variable, which is not permitted in dataflow diagrams. The last section introduces this problem and ideas for possible visualization strategies.

4.1 SCADE vs. SCCharts

The SCADE visual language has a textual foundation that is a Lustre dialect. In this section the MoCs of the languages Lustre/SCADE and SCCharts are compared. The transformation from Lustre to SCCharts does not only create a visualization for the Lustre program. The original program behavior shall be preserved during the transformation to SCCharts.

The sequentially constructive semantics of SCCharts defines a conservative extension to the classical MoC of synchronous languages. Thus, languages using the classical MoC such as Lustre can be expressed using SCCharts. Additionally, some of the concepts in Lustre dataflow are the same for SCCharts dataflow.

The first similarity of SCCharts dataflow and Lustre is the definition of the program behavior through equations. These equations have an output or a local variable on the left side and the corresponding expression on the right. They define both sides to be equal in each tick. However, this is just partially true for SCCharts. It is possible that equations define both sides to be equal but also equations like `x = x + 1` are valid due to the sequentially constructive MoC. Thus, Lustre actually defines equations and in SCCharts they are actually assignments. Therefore, there is a potential equality of the left and the right side of the equation but it is no defining property for all dataflow SCCharts. Nevertheless, the description for the behavior of the program in Lustre already defines a structure that can be transferred to SCCharts.

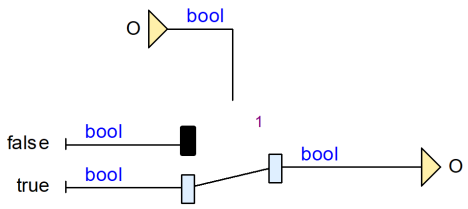
Next, we further look at the execution semantics of those equations. In Lustre they are defined to run in parallel. The sequential order of the equations in the program is not relevant for the behavior of

```
scchart SeqReadWrite {
  bool O = false

  dataflow {
    O = O? false : true
  }
}
```

(a) SCCharts

```
node SeqReadWrite ()
       returns ();
var O:bool;

let
  O = if O
     then false
     else true;
tel
```

(b) Lustre



(c) SCADE

Figure 4.1. An SCChart performing read sequentially followed by a write of the same variable and the equivalent but invalid Lustre and SCADE programs

the node. This concept also holds for SCCharts equations. They are assumed to be executed concurrently. It follows that also the semantics of the equations in the program is defined the same. Thus, an equation in Lustre is generally equivalent to an equation in SCCharts.

The last similarity concerns the nature of those equations. In Lustre it is necessary that each variable or output must be defined by exactly one equation. Therefore, each value for a variable or output can be extracted from exactly one equation. This is an important difference to controlflow languages. They usually define different condition branches and in each of those branches arbitrary assignments can be executed. However, in Lustre all possible values for a variable are defined by a single equation and this equation is executed each tick. This is partially true in SCCharts, too. Variables and outputs that are defined by a single equation in Lustre can be defined by a single equation in SCCharts. Nevertheless, SCCharts allows for multiple equations for an output or a variable as long as the concurrent access can be scheduled sequentially or concurrently following the ordering *initialize-update-read*.

In conclusion, there are more models that result in a valid SCCharts than there are valid models for Lustre. For the transformation this implies that all valid Lustre programs can be transformed to an SCChart.

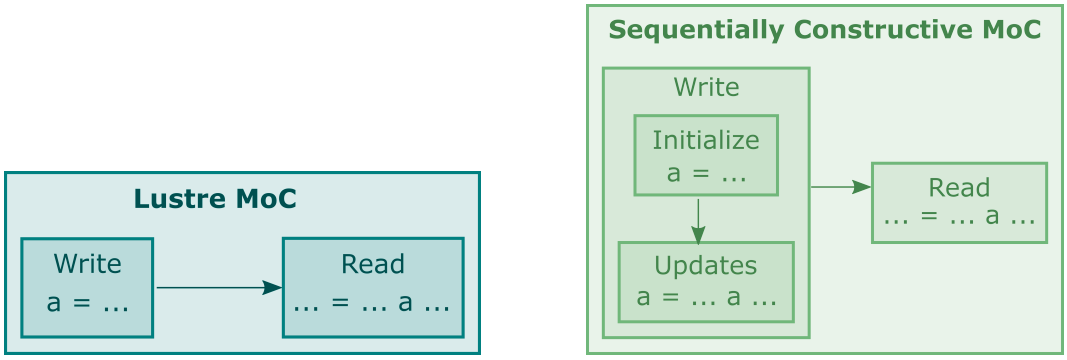
4.1.1 Sequentially Constructive Extension of Lustre

The SCCharts MoC extends the MoC used by Lustre. Therefore, all valid Lustre program can be transformed to SCCharts. Also the approach for defining the dataflow is similar in both languages. However, there are concepts in SCCharts that cannot be expressed in Lustre.

In SCCharts the classical MoC is extended by sequential reads and writes as well as a protocol for ordering concurrent variable access. The idea is to prevent race conditions without unnecessarily restricting the expressiveness. These concepts can be applied to Lustre, too. They allow to define sequentially constructive semantics for Lustre programs that would otherwise be considered invalid. In the following, we take a look at how these concepts are expressed in SCCharts and what the equivalent Lustre/ SCADE program would look like.

Sequential Read and Write

One extension the sequentially constructive MoC introduces are sequential reads and writes. The variable access is ordered in its equation but in the Lustre MoC this would be forbidden. The equations are defined to imply actual equality of the left and the right side and this is violated.



(a) Lustre MoC

(b) Sequentially Constructive MoC

Figure 4.2. The variable access sequence in the Lustre and the sequentially constructive MoC

In Figure 4.1a is an example for an SCCharts program that sequentially reads and then writes the variable `O`. The value of `O` alternates between `false` and `true`. The dataflow can be modeled in Lustre, too. In Figure 4.1b the same program structure is applied to Lustre. The Lustre compiler would reject this program because the value of the variable `O` depends on itself. However, in the sequentially constructive MoC it has a defined behavior. In order to illustrate the dataflow of this program, in Figure 4.1c the SCADE equivalent to this dataflow is given. This SCADE program is rejected due to the instantaneous loop implied by O.

Initialize-Update-Read Protocol

The next concept that is introduced in the sequentially constructive MoC is the ordering of concurrent variable access following the protocol *initialize-update-read*. In the Lustre MoC the variable access is only ordered by *write-before-read*. This is due to the fact that each variable has only one value during each tick defined through exactly one equation. Therefore, this one value is written to the variable and afterwards it can be read without the risk of it being modified again. In Figure 4.2a this variable access is visualized. The writing is done first and the reading of a variable follows. In the sequentially constructive MoC, however, the writing is split up into initialization and updates as introduced in Section 3.2. This is illustrated in Figure 4.2b. This allows to have more than one value for a variable within one tick. However, all updates need to be confluent in order for program to be schedulable.

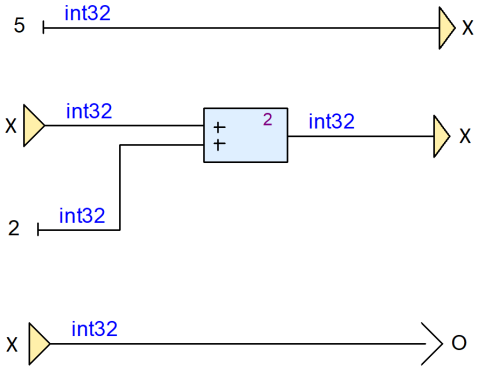
In conclusion, all programs that include an initialization and an update on a variable are considered not valid in the Lustre MoC. Moreover, an update on a variable is defined to set the value according to its current value. The equations could not be considered to illustrate equality but a sequential ordering can be concluded. Thus, all updates are considered invalid in the Lustre MoC. However, the sequentially constructive MoC accepts those programs.

The possibility for an initialization and multiple updates for a variable also removes the requirement to have only one defining equation for a variable. Initializations and updates can also be perfomed concurrently and thus multiple equations are possible in the dataflow.

Figure 4.3 shows an SCCharts program, a Lustre program and a SCADE program that perform an initialization, an update and a read on a variable. The Lustre and the SCADE program are rejected in the classical MoC. Nevertheless, the sequentially constructive MoC translates them to behave equivalent to the illustrated SCChart. Each tick the variable `OLocal` is initialized with `5` and then updated by adding

```
scchart IUR {
  output int O
  int X

  dataflow {
    X = 5
    X = X + 2
    O = X
  }
}
```

(a) SCCharts



(b) SCADE

```
node IUR () returns (O: int);
var X1,X2:int;

let
  X1 = 5;
  X2 = X1 + 2;
  O = X2;
tel
```

(c) SSA Lustre

```
node IUR () returns (O: int);
var X:int;

let
  X = 5;
  X = X + 2;
  O = X;
tel
```

(d) Lustre

Figure 4.3. An SCChart performing an initialization, an update and a read on a variable, the equivalent but invalid Lustre and SCADE programs, and an equivalent Lustre program with SSA

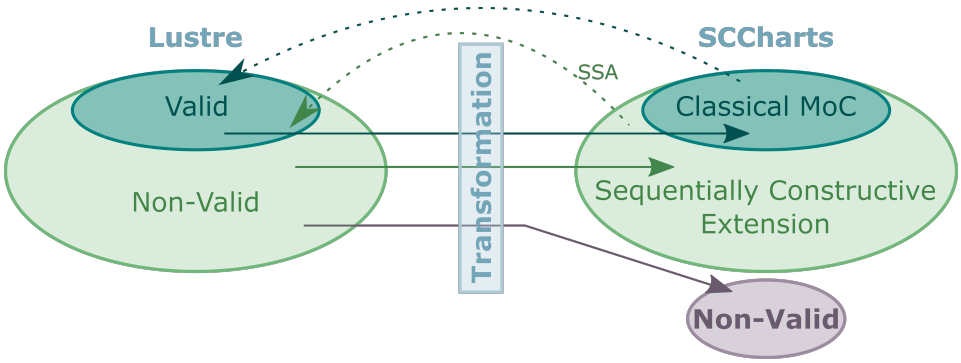


Figure 4.4. The mapping created through the transformation from valid and invalid Lustre programs to SCCharts

2. The value of this variable is then read and written to the output variable `O` creating the value `7` for it in each tick.

4.1.2 Transformation Objective

SCCharts and Lustre are both founded on a synchronous MoC. However, as proposed before, SCCharts offer an extension to the MoC of Lustre. The goal for the transformation is to create a bijective mapping. The idea of this mapping is visualized in Figure 4.4. Valid Lustre programs are transformed to SCCharts

that only use features of the classical MoC. However, also non-valid Lustre programs can be transformed and might either reside in the sequentially constructive set of programs or the transformation yields an error. This gives a first approach towards extending the language scope of Lustre with sequential constructiveness just like the MoC of Esterel was extended by sequential constructiveness in Sequentially Constructive Esterel (SCEst) [SMR+17]. The dotted lines give an idea for a transformation from SCCharts back to Lustre. Programs using sequentially constructive features need to use SSA in order to conform with the Lustre MoC. However, this transformation is not considered in the scope of this thesis.

This transformation approach ensures that the MoC is not extended if it is not needed. However, programs in the sequentially constructive MoC could be transformed back to the Lustre MoC using methods like SSA [Sch16]. It creates new variables for each variable value in a tick in order to reestablish the clearly defined value for each variable. In Figure 4.3c is an example for a Lustre program that is created from the Lustre program in Figure 4.3d using **ssc!** (**ssc!**). Nevertheless, preserving the MoC during the transformation creates a more exact definition of the program. A valid Lustre program can be transformed to SCChart and back to Lustre and most of the program properties are preserved and not recreated artificially.

4.1.3 Visualization

The SCCharts dataflow and Lustre are both dataflow languages and SCCharts is chosen to act as the visual representation of Lustre code for this thesis. We already know that the semantics match sufficiently to transform Lustre programs to SCCharts programs. However, we now take a look at the graphical syntax of dataflow in SCADE and KIELER in order to ensure that SCCharts dataflow is able to conveniently represent the Lustre programs.

The main aspect of this comparison is the visualization of expressions. In Section 3.1 we introduced the data operators for Lustre. In Table 4.1, 4.2 and 4.3 these data operators are contrasted to the SCADE and the SCCharts visualization. In general they look very similar. They all use boxes to express operators and connect inputs and outputs with operators using wires. However, SCADE prefers the hardware style for the logic operators whereas in SCCharts dataflow the operator syntax is mostly used as description for the operator itself. The condition operator in Lustre is expressed using a hardware alike switch. The numerical operators are all almost identical. One difference is the division operator. In SCCharts the integer division is performed when the types result in an integer. Therefore, a non-integer division is performed when the type of the operation remains at a real level. Therefore, in SCCharts there is no explicit operator for integer division because there is no explicit type system.

Both tools offer options for *filtering*. Filtering describes the interactive process of dynamically choosing the level of detail for the visualization. The type of the wire can be omitted in SCADE and other information can be displayed such as the name of the wire. In the SCCharts column the description of the wire after the operator can be omitted, too.

As a result of the comparison, the dataflow of SCCharts seems visually suited to express Lustre because it is very similar to SCADE. However, there are some Lustre operators that do not have a designated operator in either SCADE or SCCharts yet. The implies operator `=>` is not implemented but can easily be reworked using logical equivalences. Additionally, the n-ary operators *at most one of* used as `#(a, b, ...)` and *none of* used as `nor(a, b, ...)` cannot be expressed with a visual component either. In SCADE these can easily be remodeled using `map` and `fold` constructs but we do not focus on implementing those in SCCharts so we also use logical equivalence to express these operators.

The next category of operators to look at are the sequence operators. In SCCharts not all of the sequence operators are necessary and thus they are not available. The *pre* and *init* are supported by a dedicated operator. The *followed by* operator is not available as a single operator, but it can be

Table 4.1. Boolean and condition operators in comparison for Lustre, SCADE and SCCharts

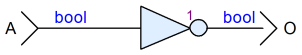
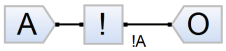
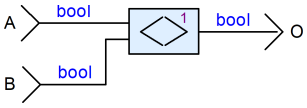
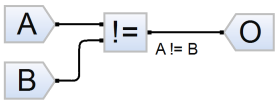
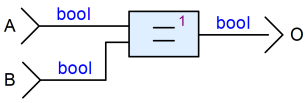
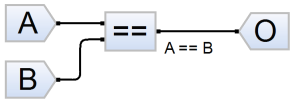
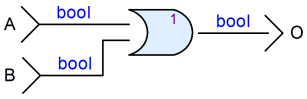
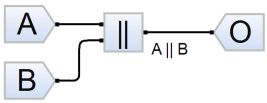
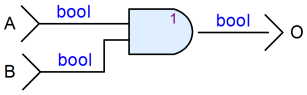
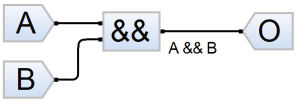
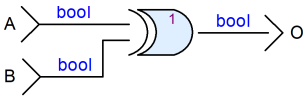
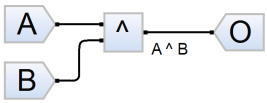
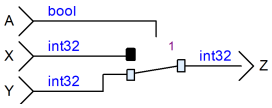
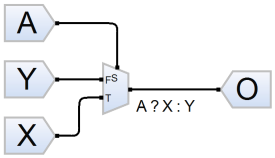
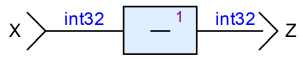
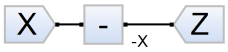
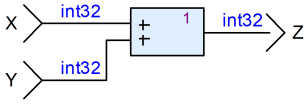
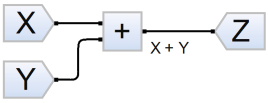
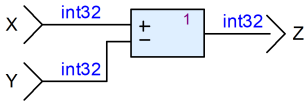
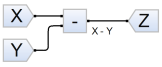
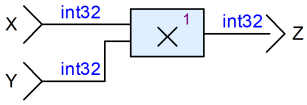
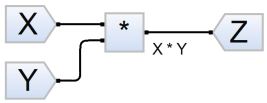
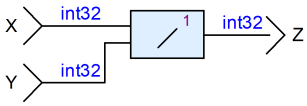
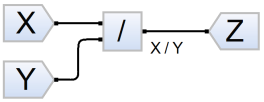
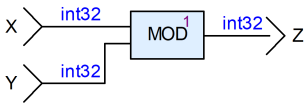
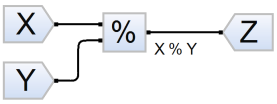
Lustre	SCADE	SCCharts
`not A`		
`A <> B`		
`A = B`		
`A or B`		
`A and B`		
`A xor B`		
`if A then X` `else Y`		

Table 4.2. Numerical operators in comparison for Lustre, SCADE and SCCharts

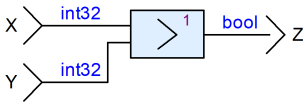
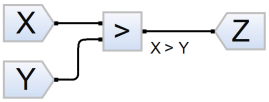
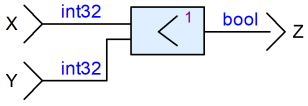
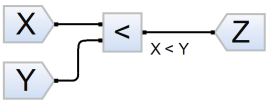
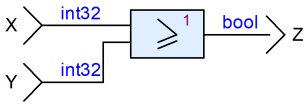
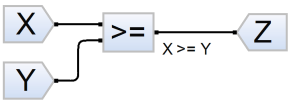
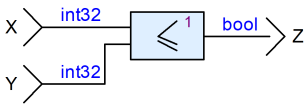
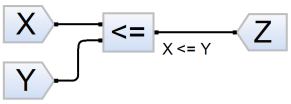
Lustre	SCADE	SCCharts
`- X`		
`X + Y`		
`X - Y`		
`X * Y`		
`X / Y`		
`X mod Y`		

easily remodeled using *pre* and *init*. In Figure 4.4 the particular statement and the SCADE operator in comparison to the SCCharts operator are listed. In SCADE the *followed by* operator allows to specify an integer number for how many ticks the first value shall be used before the previous value of the second argument is used. We do not further consider this aspect because in SCCharts there is no corresponding *followed by* operator and Lustre does not support this either. In SCCharts a reference could be added as a macro operator that realized the combination of *pre* and *init* in order to express this behavior.

However, the *when* and the *current* operators are not available in SCCharts because the concept of clocks is rather simple as just the base clock is assumed apart from extensions such as dynamic ticks [HBG17]. The visualization in SCADE for the when operator is simple. One wire defines the boolean stream that defines whether the second stream shall be forwarded or not. Nevertheless, this introduces some complexity because the result wire does not hold a value every tick. The entire wire is clocked at a different clock than the base clock and this is not distinguishable at first sight.

The *current* operator is not available in SCADE either. Instead the `merge` operator is used. In Table 4.5 is an example of how the merge operator works. It takes a clock `A`, so a boolean stream, and merges the two incoming streams `X` and `Y`. On a clock value `true` the value of the first stream is used and otherwise the value of the second stream. This requires the incoming streams to have complementary clocks. However, this is not part of the selected set of Lustre features for this thesis. Moreover, in

Table 4.3. Compare operators in comparison for Lustre, SCADE and SCCharts

Lustre	SCADE	SCCharts
X > Y		
X < Y		
X >= Y		
X <= Y		

SCCharts we chose a different way for visualizing clocks. The idea behind this and the reasons for this approach are introduced in the following section.

4.2 Transformation

In this section the actual behavior of the transformation is introduced. In Figure 4.5 the different transformation steps are outlined. In the following, each of these steps is explained in more detail. Moreover, it is explained why they are designed this way. Note that the transformation structure is also reasoned on the grammar implemented for the Lustre language. The definition of constants and node declarations for example are introduced in the same grammar rule and thus they are transformed sequentially. Moreover, constants might be used in nodes so they should be transformed first. Nevertheless, other conceptual implementation-independent thoughts are introduced, too.

4.2.1 Constants

A Lustre program may contain global constants outside of the declaration of a node as described in Section 3.1. These are translated first. The idea is for those to be available in all nodes within this program file, so they all need to be known in all SCCharts representing a specific node. In Section 3.2 we also introduced SCCharts and the concept of the `extends` keyword. This functionality is useful for the use case introduced here. We introduce a new SCChart containing only the constants defined in the program. All SCCharts representing a node within this program file then extend this constants SCChart and therefore have access to the values of those. This offers an abstraction of the constants themselves and they do not need to be included manually in all nodes within that file. Note that inheritance

Table 4.4. Sequences operators in comparison for Lustre, SCADE and SCCharts

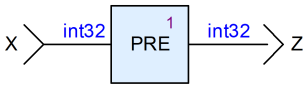
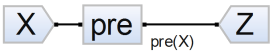
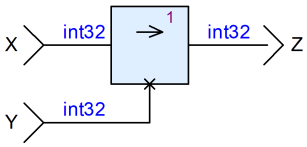
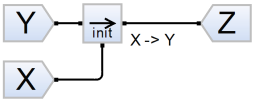
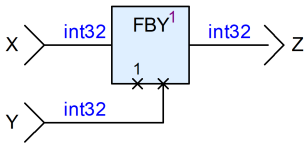
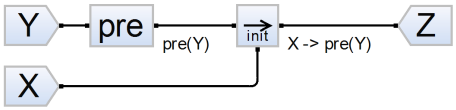
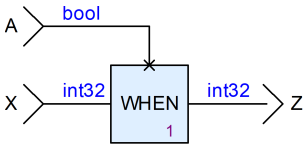
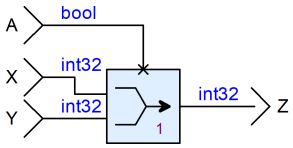
Lustre	SCADE	SCCharts
`pre(X)`		
`X -> Y`		
`X fby Y`		
`X when A`		none
`if A then X else Y`		none

Table 4.5. The merge operator

Name	Stream of values								
`clk`	true	false	true	true	false	true	false	false	true
`x`	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
`y`	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
`a = x when clk`	x_1		x_3	x_4		x_6			x_9
`b = y when not clk`		y_2			y_5		y_7	y_8	
`merge clk a b`	x_1	y_2	x_3	x_4	y_5	x_6	y_7	y_8	x_9

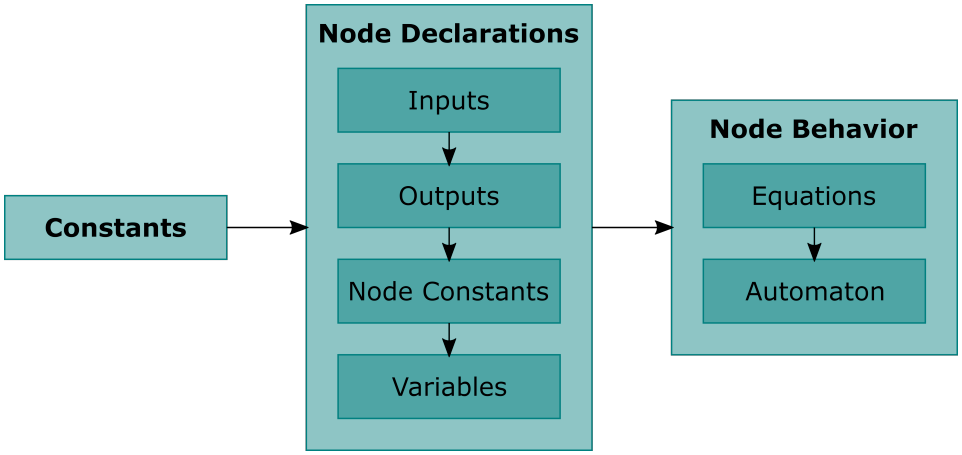


Figure 4.5. Overview of the transformation order going from Lustre to SCCharts

is not designed to be used for this kind of scenario. The new SCChart is not a constants SCChart, the inheritance just offers a convenient way to handle this.

4.2.2 Node Declarations

The next step is the parsing of the nodes interfaces and their constants and variables. This step needs to be separated from the transformation of the behavior because of possible node references. Inputs and outputs as well as the transformed name of the SCChart are available after this step and can be properly bound.

In general, for each node a new SCChart is created. The inputs and outputs of the node are added to the SCChart as inputs and outputs. The variables and local constants are also added this way.

4.2.3 Node Behavior

For the node behavior the equations, automata and assertions within the `let` and `tel` statements are transformed. Each of these three types are handled differently whereas the equations make up the main part of the behavior of the node. The automata result from the state extension for Lustre introduced in 3.1.3 and the assertions are just a proof of concept that use the currently worked on model checking for SCCharts [Sta19].

Equations

The equations in a Lustre program define the actual behavior of a node. In Section 4.1.3 it is already shown that most operations have an equivalent operation in SCCharts.

The data operations are mostly supported through SCCharts constructs. However, some of the sequence operators are for clock manipulation whereas the concept of clocks is not available in SCCharts. The operators *pre*, *initialization* and *followed by* are easily transformed using the equivalent SCCharts operator or simple extensions. However, for *when* and *current* simple SCCharts language properties are not sufficient. In Section 4.2.4 problems that may result from clock usage are illustrated and the approach for the transformation of *when* and *current* is outlined.

Nevertheless, some operations like the implies operator are currently not supported through a dedicated operator in SCCharts. It can be translated using the following equivalence rule

$$A \Rightarrow B \Leftrightarrow \neg A \vee B.$$

The n-ary operation *nor* does not have a corresponding operator in SCCharts. It describes the negation of the disjunction. Therefore, the following equivalence rule is sufficient for the transformation

$$nor(A_1, A_2, \ldots) = \neg(A_1 \vee A_2 \vee \ldots).$$

Lastly, the nary operation *at most one of* needs a more complex transformation. It checks for an arbitrary number of parameters that either all of them are false or at most one of the parameter is true. For an operation with n parameters, we need $n + 1$ clauses to cover all possible cases. As an example for the expression $\#(A_1, A_2, A_3)$ there are four cases that need to be checked:

1. None of the three is true
2. A_1 is true and the other two are false
3. A_2 is true and the other two are false
4. A_3 is true and the other two are false

So in conclusion we can construct the logically equivalent formula

$$\begin{aligned}\#(A_1, A_2, A_3) \Leftrightarrow &\neg(A_1 \vee A_2 \vee A_3) \\ &\vee (A_1 \wedge \neg(A_2 \vee A_3) \\ &\vee (A_2 \wedge \neg(A_1 \vee A_3) \\ &\vee (A_3 \wedge \neg(A_1 \vee A_2).\end{aligned}$$

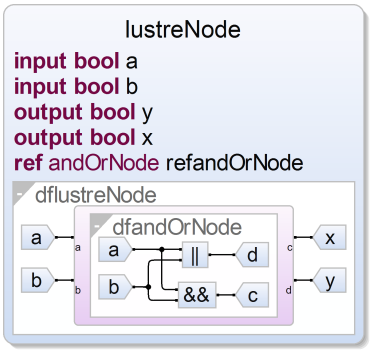
This idea can be applied to an arbitrary number of parameters and is used as the transformation from Lustre to SCCharts dataflow.

The last feature that may be used in Lustre equations instead of the operators introduced above are node references. In 3.1.2 it is explained how they may be used and how they work. The SCCharts dataflow also supports references in a similar way. However, references may return more than one output value and tuples at the left side of equations are not possible yet. Therefore, for each output and each input there is an equation that links the outputs and inputs correctly. In Figure 4.6b a Lustre program using a reference of a node with two outputs is illustrated. Beside in Figure 4.6c and 4.6a the resulting transformed SCChart is shown. The reference to the other node is achieved through a reference variable `refandOrNode`. With an object-oriented-alike notation the inputs or outputs of the node can be accessed and linked.

Automata

The syntax and semantics of the automata in Lustre are already introduced in Section 3.1.3. For each of these constructs, there is an equivalent element in SCCharts. Therefore, the transformation for automata to Lustre can be applied one-to-one on the model elements and the semantics are preserved. The weak and strong transitions are available as such and history and normal transitions are possible, too. States may contain hierarchy but do not need to. Moreover, they are also allowed to contain dataflow regions again so all needed elements in the Lustre state extension are available in SCCharts.

In Figure 4.7 is an example for an automata in Lustre that is transformed to SCChart. It uses all possible transformation types and has two concurrent regions. The output `0` is initialized with zero



(a) SCCharts visual

```
node lustreNode(a:bool; b:bool)
     returns (y:bool; x:bool);

let
  (x,y) = andOrNode(a, b);
tel.

node andOrNode(a:bool; b:bool)
    returns (c:bool; d:bool);
let
  c = a and b;
  d = a or b;
tel.
```

(b) Lustre

```
scchart lustreNode {
  input bool a
  input bool b
  output bool y
  output bool x
  ref andOrNode refandOrNode

  dataflow dflustreNode {
    refandOrNode.a = a
    refandOrNode.b = b
    x = refandOrNode.c
    y = refandOrNode.d
  }
}
scchart andOrNode {
  input bool a
  input bool b
  output bool c
  output bool d

  dataflow dfandOrNode {
    c = a && b
    d = a || b
  }
}
```

(c) SCCharts textual

Figure 4.6. A Lustre program using hierarchical clocks and the resulting transformed SCChart

and incremented if the input `A` is set and decremented otherwise. Setting the input `R` cause the output to be reset to zero.

4.2.4 Handling Clocks

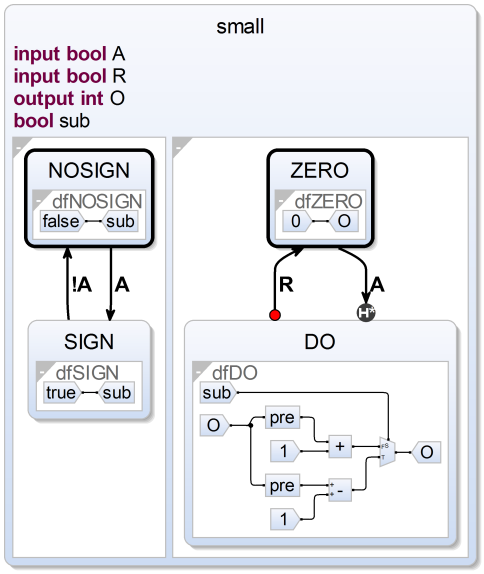
The operators *when* and *current* are not available for SCCharts because the concept of clocks is not needed. However, for the transformation from Lustre to SCCharts we need to find a representation of clocks in SCCharts dataflow that behaves the same. In the following we elaborate how clocks can be expressed using variables.

The assignment $X = E$ when C only evaluates E in those cases that hold true for C. Moreover, X holds a value only in those cases. So clocking introduces two main ideas: absence of stream values and conditioned evaluation of expressions. The concept of clocks establish the idea of *control* in dataflow. However, the absence of the values is not the relevant aspect because there is no check on the presence or absence of a value in Lustre, just its designated clock. This absence results from the dataflow interpretation of control.

In SCCharts variables are used instead of streams or signals and they hold the value from the previous tick if not further modified. We already stated that the actual absence of values is not the main aspect about clocks but the conditioned evaluation. Therefore, variables can be fitted to be used

```
1  node small (A:bool; R:bool) returns (O:int);
2  var sub:bool;
3  let
4   automaton
5    NOSIGN ->
6     sub = false;
7    until A then SIGN;
8    |
9    SIGN ->
10    sub = true;
11   until not A then NOSIGN;
12  end;
13  automaton
14   ZERO ->
15    O = 0;
16   until A continue DO;
17   |
18   DO ->
19    O = if sub then pre(O)-1 else pre(O)+1;
20   unless R then ZERO;
21  end;
22 tel
```

(a) Lustre



(b) SCCharts

Figure 4.7. A Lustre program and the resulting SCChart using the state extension for Lustre

Table 4.6. Hierarchical clocks in Lustre and gray values illustrating variable values

Name	Stream of values								
clk	true	false	true	false	true	true	false	false	true
x	true	false	false	true	true	false	false	false	true
y	true	false	false	true	false	false	true	true	false
xClk = x when clk	true	true	false	false	true	false	false	false	true
yXClk = y when xClk	true	false	false	false	false	false	false	false	false

for expressing streams. They hold the value of the previous tick so they implicitly apply *current* up until the base clock is reached. Additionally, this idea makes a transformation for *current* obsolete.

A conditioned updating of a variable with an expression can be created using an `if...then...else...` statement with the `else` branch containing the value of the previous tick. Both branches are evaluated every tick but only the one corresponding to the clock value is forwarded to the variable.

In Figure 4.8 is an example for the transformation of the *when* and the *current* operation. The SCCharts components illustrating the *when* operation are the condition and the pre. As already mentioned, the *current* operation does not require a transformation. The value of the clocked variable is simply read and written to the target of the current operation.

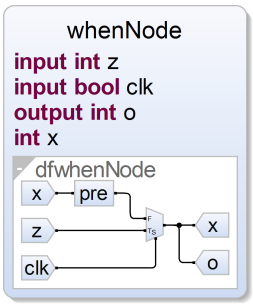
This approach works for flat clocks. However, clocks may be hierarchical so a clocked boolean stream can be used as a clock for another stream. In Table 4.6 is an example for hierarchical clocks. The stream `yClk` is sampled to the clock `xClk` which has the clock `clk`. The black values show the stream and when they actually hold a value. The gray values show the behavior if instead of streams,

```
node whenNode(z:int; clk:bool) returns (o:int);
var x : int when clk;

let
  x = z when clk;
  o = current x;
tel.
```

(a) Lustre

```
scchart whenNode {
  input int z
  input bool clk
  output int o
  int x

  dataflow dfwhenNode {
    x = clk ? z : pre(x)
    o = x
  }
}
```

(b) SCChart textual

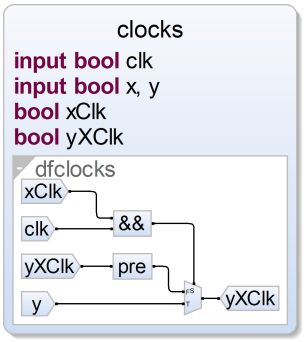


(c) SCChart

Figure 4.8. A Lustre program and the resulting SCChart using the operations *when* and *current*

```
node clocks(clk:bool; x,y:bool) returns ();

var xClk : bool when clk;
   yXClk : bool when xClk;
let
  yXClk = y when xClk;
tel.
```

(a) Lustre



(b) SCCharts

Figure 4.9. A Lustre program using hierarchical clocks and the resulting transformed SCChart

variables are used with the above defined behavior. The problem occurring is in the second column of the last line. The value of `y` would be set to `yXClk` because the clock `xClk` still holds the value true. This is because it is a variable and it contains the value of the previous tick. This illustrates that the transformation introduced above is not sufficient to take care of clocks.

In order to solve the problem, we need to make sure that all clocks hierarchically above the current clocks are also true. This can be done by creating a conjunction of all hierarchical clocks and using this conjunction as the trigger for the conditional `if` statement. Looking at Table 4.6 we see that the problem is solved because the value of `yXClk` would not be updated because of `clk` not being set in the

Table 4.7. Streams with a pre operation as it is supposed to work in Lustre in the last two lines and the equivalent using variables

	Name	Stream of values								
	`x`	1	2	3	4	5	6	7	8	9
	`clk`	**true**	**false**	**false**	**true**	**true**	**false**	**true**	**false**	**true**
	`x1 = x when clk`	1	1	1	4	5	5	7	7	9
Lustre	**pre**(x1)	**nil**			1	4		5		7
	pre(**pre**(x1))	**nil**			**nil**	1		4		5
SCCharts (old)	**pre**(x1)	**nil**	1	1	1	4	5	5	7	7
	pre(**pre**(x1))	**nil**	nil	1	1	1	4	5	5	7
SCCharts (new)	**pre**(x1)	**nil**	nil	nil	1	4	4	5	5	7
	pre(**pre**(x1))	**nil**	nil	nil	**nil**	1	1	4	4	5

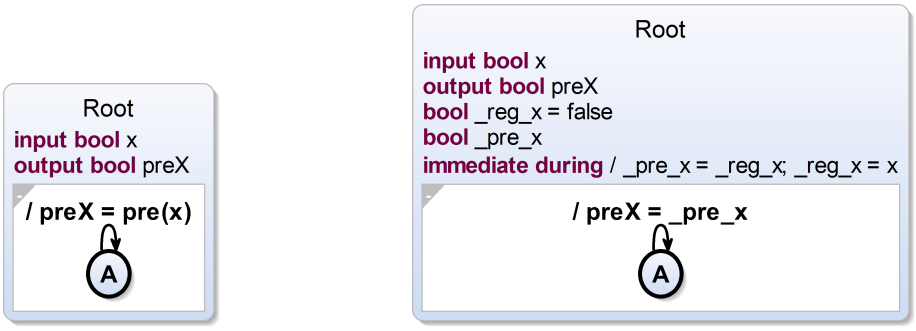
second tick. In Figure 4.9 the Lustre code and the corresponding SCChart that illustrates the described scenario with the solution using a conjunction are shown.

Pre and Clocks

Some of the sequence operators in Lustre are available as such in KIELER. The *pre* operator for example is part of the operators in KIELER, too. However, in KIELER this operator is part of the extended features for SCCharts. It is sufficient to make use of the persistent values of variables beyond ticks. In Figure 4.10 the behavior of the *pre* transformation in KIELER is shown. Two new variables are introduced. The variable `_reg_x` holds the value of the variable at the end of the current tick and `_pre_x` holds the value of the variable at the end of the previous tick. The variable `_pre_x` can thus be used for accessing the previous value for the variable. Due to the during action, the updating of those variables occurs each tick, so each tick the variable containing the previous value is updated.

In contrast, the Lustre *pre* operation depends on the clock of the variable it is applied to. In Table 4.7 a stream `x1` is created that is the sampled version of the stream `x` to the clock `clk`. In line 4 of this table the Lustre version of the previous value of the clocked stream `x1` is calculated. This definition of the *pre* operation is different to the SCCharts realization when considering clocked streams. The clocked version of the *pre* references the value of the variable in the tick when the corresponding clock of the variable was last set. In the Lustre world the stream `x1` does not hold a value but in those tick when the clock is true, thus this is the only accessible last value for the Lustre approach. In SCCharts we have variables. Therefore, in those tick instances without the clock being set the variable `x1` just holds its value. For a single *pre* operation however, the SCCharts and the Lustre version of the *pre* still produce the same values as the table shows in line 4 and 6.

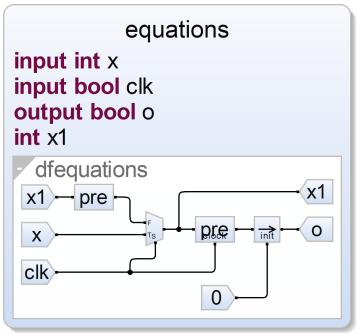
In line 5 of the table, the *pre* operation is applied twice following the Lustre definition. This yields the value of the second to last tick that the corresponding clock of the variable was set. However, for the SCCharts transformation the streams are converted to variables and thus the stream is implicitly sampled up to the base clock. In lines 6 and 7 of the table, the *pre* operations on variables are shown. The tick instances that the clock is not set are filled to hold the last known value. These *pre* operations define the previous value on the base clock. Therefore, the previous value is updated every tick and for example in tick 4, the second to last value is `1` and not `nil` as it is defined in Lustre. This problem also occurs in tick 7 and 9.



(a) Original SCChart **(b)** Transformed SCChart

Figure 4.10. SCChart using the pre operation and the result after the compilation of the pre processor

```
1 node equations(x:int; clk:bool) returns (o:
      bool);
2 var x1:int when clk;
3 let
4   x1 = x when clk;
5   o = current(0 -> pre(x1));
6 tel.
```



(a) Lustre **(b)** SCCharts

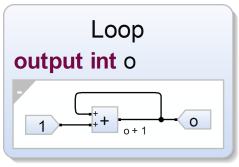
Figure 4.11. A Lustre program using pre combined with clocks and the resulting transformed SCChart

In order to overcome this problem, the transformation for the *pre* operation must be modified. The behavior of the transformation was explained above. The during action that is introduced for realizing the *pre* operation updates the variables each tick. In order to properly reflect the *pre* operation as defined in Lustre, this update needs to be executed only if the clock is set. Therefore, we extend the *pre* processor in KIELER to optionally accept a second argument. This second argument acts as the trigger for the during action. We can now extract the clock from the variable the *pre* operator is applied to and add it as second argument to the *pre* operation. This causes the previous value to only get updated if the corresponding clock is set, too. In Figure 4.11 is an example for a Lustre program that applies the *pre* operation on a clocked stream. The stream `x1` is clocked by `clk` and for the value of `o` the *pre* of `x1` is calculated. The transformation now adds the clock of `x1` to the *pre* expression and the SCChart shown in Figure 4.11b adds this clock to the bottom of the *pre* operator.

In Table 4.7 the lines 8 and 9 illustrate the behavior following this new transformation. The previous variables are only updated if the clock is set, thus the correct values are produced.

```
scchart Loop {
  output int O

  dataflow {
    O = O + 1
  }
}
```

(a) Textual

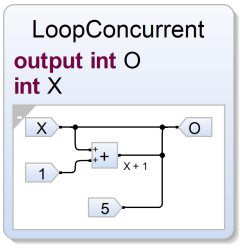


(b) Visual

Figure 4.12. A dataflow SCChart containing a cycle due to a read and a sequential write of a variable

```
scchart LoopConcurrent {
  output int O
  int X

  dataflow {
    X = 5
    X = X + 1
    O = X
  }
}
```

(a) Textual



(b) Visual

Figure 4.13. A dataflow SCChart containing a cycle due to a variable with an initialization, an update and a read

4.3 Sequentially Constructive Dataflow Synthesis

Lustre and SCADE share the same semantics, where SCADE illustrates a visual syntax for Lustre. Both are dataflow languages and also SCCharts offer a dataflow modeling approach. However, the MoC of SCCharts extends the classical one by allowing multiple values for a variable if the behavior is still deterministic. This MoC can be used to accept Lustre programs that are not valid in the classical world but in the sequentially constructive MoC. However, especially the concurrent modification of a variable using initialization and updates is not trivial to be illustrated in the dataflow.

In Figure 4.12 is an example for a dataflow SCChart that performs a sequential read followed by a write. The output wire of the addition is again served as input into the addition. This visualization is created due to the fact that this wire represents the variable O. This variable is used as input and as output for the addition and thus this wire is used in all cases. Without deeper knowledge about the MoC in SCCharts this visualization does hardly give enough insight into the actual flow of the data for it to be a helpful representation. Moreover, the synchronous world divides time into discrete ticks and this model does not imply a tick boundary on the visual level, thus it could be considered to contain an instantaneous cycle.

This effect can also be visualized using an example that makes use of the *initialize-update-read* protocol. In Figure 4.13 is a dataflow SCChart that concurrently initializes, updates and reads the variable X. All those operations are performed on the same variable and thus the visualization creates one wire for it. Nevertheless, the wire creates a cycle from the output of the addition to its input. Just like the example above, this visualization does not properly reflect the ordering of the variable reads and writes. Moreover, this example connects two wires that both write onto the wire. The

(a) Sequential read and write

(b) Initialization, update and read

Figure 4.14. SCChart example from Figure 4.12 and 4.13 using the memory operator to break the visual cycle

addition creates an output value for the wire and the initialization connects the value 5 to the wire. This situation cannot be solved without further knowing the ordering of the variable access.

During the scope of this thesis different conceptual approaches have been considered in order to improve this visualization. In the following, two of the approaches are introduced.

4.3.1 Memory Operator

The first idea for improving the visualization is the introduction of an operator that internally performs the *initialize-update-read* protocol and sequentializes variable access. The goal is for it to act like a memory that is accessed. The writing memory access is an input to the operator and reading access is the output.

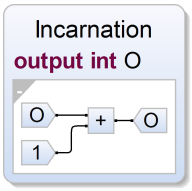
In Figure 4.14 is an example for how this operator could look like. In Figure 4.14a the sequential read followed by the write is ordered by the internals of the memory operator. The implicit initialization is indicated through the first output of the memory operator and subsequently feed back into the addition update. The final read is received through the second output of the operator. Moreover, in Figure 4.14b the initialization and the update are included as input to the memory operator. The update depends on a read output of the operator because it updates the currently held value after initialization. Lastly final output returns the value that is considered the read for this variable consecutively.

4.3.2 Incarnation for Variable Values

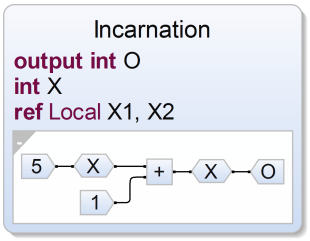
This approach tries to visualize the actual variable modification more accurately. Sequential reads and writes such as the *initialize-update-read* protocol define an ordering of the variable access and this ordering is used to create different incarnations of that variable. This idea follows the concept of Static Single Assignment (SSA). Each modified variable value creates a new incarnation for that variable that illustrates a new wire. This approach allows to visualize the actual flow of the data that is defined through the *iur* protocol. Especially for users that are used to the synchronous modeling approach, this stategy gives a more intuitive design of the program flow.

In Figure 4.15 is an example for how this incarnation could look like. The actual ordering of the variable access is determined and the dataflow is created corresponding to this ordering. The *iur* protocol is reflected directly by first initializing the variable, then updating it and lastly reading it.

A combination of this approach with multiple updates using the same operator could also be simplified into one operator. This would not require SSA to create incarnation for the variable after each update but only before and after the updates. In Figure 4.16a is an example SCChart that illustrates the idea. The two updates through an addition on the variable X can be simplified using only one addition operator with three ports. In Figure 4.16b the idea for the visualization is shown.



(a) Sequential read and write

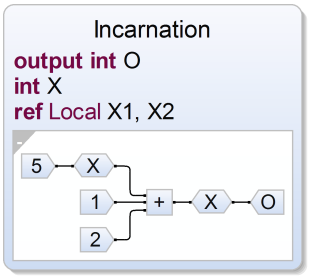


(b) Initialization, update and read

Figure 4.15. SCChart example from Figure 4.12 and 4.13 using the incarnation strategy to break the visual cycle

```
1  scchart Incarnation {
2    output int O
3    int X
4
5    dataflow {
6      X = 5
7      X = X + 1
8      X = X + 2
9      O = X
10   }
11 }
```

(a) Textual



(b) Visualization

Figure 4.16. A dataflow SCChart using two updates with the same operator and an idea for its visualization

4.3.3 Conclusion

Both visualizations both from Section 4.3.1 and 4.3.2 have the advantage that they break the visual cycle introduced by the sequential constructiveness. However, they both have advantages and disadvantages.

In the first approach, using the memory operator, the complexity of the SCCharts internals is abstracted into the memory operator. However, no information about the actual ordering of the initialize and the update is given on the visual level. Moreover, the update reads from the output line of the memory operator and writes back into it as input. This is still a cycle that is only defined to be solved through the memory operator.

The second approach helps to identify the ordering of the variable access. The different variable variants can be ordered by initialization first, followed by the updates and last the read occurs. Note that the ordering of the update is not important because in order for them to be valid, they need to be confluent. This approach also has the advantage that the cycle is not only defined to be broken, but it is actual sequentialized.

However, both approaches have disadvantages, too. The newly defined visualization approaches should be purely visual. The user should not need to specifically define inputs and outputs for visualized wires or new operators. The code should remain the same with the visualization synthesized from the code. In order to properly reflect the dependency edges within the dataflow region of the program, an actual dependency analysis would be needed. Moreover, the strategy for incarnating the variable values requires SSA in order to determine the different needed incarnations. Both requirements need compilation-related computations and might overload the synthesis that is supposed to create a

visualization without noticeable delay. A compiler does not imply those timing constraints, the time needed to compile a model may be larger than the time needed to synthesize a visualization. Therefore, approximations to these strategies could be a viable option in order to realize an implementation for KIELER.

Chapter 5

Implementation

The goal for this thesis is the generation of visual dataflow from a Lustre program. We use the KIELER tool as the framework providing this feature because it supports the SCCharts language, which appears to offer a good representation for Lustre. Moreover, it implements the concept of transient views that built the foundation for the rationale of this thesis. Support for other languages can be added easily and the KiCo compiler architecture allows for a simple integration of transformations for different languages.

5.1 Lustre Grammar

The first step towards the Lustre language support in KIELER is the implementation of the Lustre grammar. Prior to this thesis, there already was a grammar for Lustre programs. However, this grammar implemented expressions itself and was not suited to support the syntactical usage of features like constants, or packages. Therefore, it was renewed in order to reuse the expressions defined through `KExpressions`. This grammar is the basis for most languages in KIELER. Also SCCharts expressions are based on this grammar. This redirection to `KExpressions` later on facilitates the transformation process because at some point both models work with the same type of objects. Moreover, the grammar should syntactically support a broad range of Lustre features. We use the Lustre V6 Reference Manual as the basis for this grammar [JRH16]. However, not all parts of this grammar are supported semantically or through the transformation. For example include statements or structuring of multiple Lustre files in packages or modules is only supported on a syntactical level. The parser can recognize those programs but the resulting behavior is not further considered. The grammar described here and the following transformations all resides in the plug-in `de.cau.cs.kieler.lustre`.

In general the main part of the Lustre program is in the `PackBody`. It may contain an arbitrary amount of constant declarations, type declarations, external node declarations and regular node declarations. However, type declarations cannot be transformed to SCCharts because it does not support arbitrary type definitions. Therefore, those are not further considered for the transformation. The external nodes can be used to specify external functions that do not contain a body. This feature is also not considered for the transformation.

Figure 5.1 shows a simplified version of the grammar for a Lustre node implemented in KIELER. The bold blue labels indicate that this is a terminal and the rectangles redirect to another rule. The first line in the graph defines the interface of the node. It is specified whether it is a node with an internal state or a function. Additionally inputs and outputs are specified. In the second line constants or variables may be defined. Lastly, the body of the node starts. Either equations, assertions or automata are defined. At some point all of them need expressions. Equations contain an expression on its right side, assertions are expressions with the keyword `assert` before it and automata may contain equations again and those need expressions.

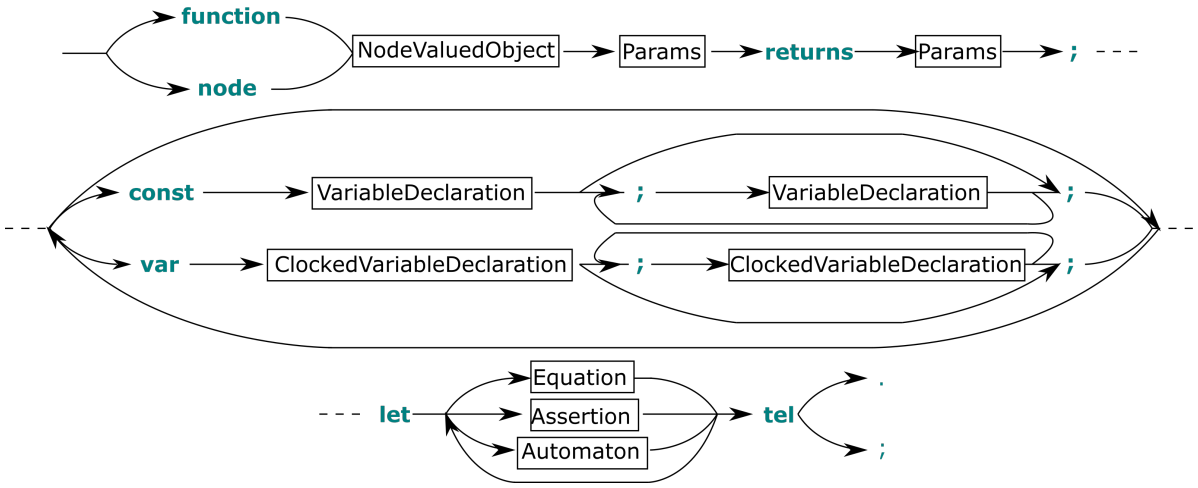


Figure 5.1. Simplified grammar rules for a Lustre node in KIELER

```
BoolExpression
 +InitExpression
  +TernaryOperation
   +ImpliesExpression
    +LogicalXorExpression
     LogicalOrExpression
      LogicalAndExpression
       -BitwiseOrExpression
        -BitwiseXorExpression
         -BitwiseAndExpression
          +CompareOperation
           +NotOrValuedExpression
            +ValuedExpression
             +NotExpression
              -BitwiseNotExpression
               AtomicExpression
                BoolValue
                ValuedObjectTestExpression
                 ValuedObjectReference
                ReferenceCall
                -FunctionCall
                -RandomCall
                -RandomizeCall
                -TextExpression
                +NorAtMostOneExpression
                BoolExpression
```

(a) Boolean expressions as rules in the Lustre grammar

```
ValuedExpression
 -ShiftExpression
  SumExpression
   ProductExpression
    +IntDiv
     NegExpression
      -TernaryOperation
       FBYExpression
        +WhenExpression
         +CurrentExpression
          +PreExpression
           AtomicValuedExpression
            IntValue
            FloatValue
            StringValue
            VectorValue
            AtomicExpression
            ValuedExpression
```

(b) Valued expressions as rules in the Lustre grammar

Figure 5.2. Lustre grammar rules for expressions

However, the `Kexpressions` also define the precedence and there are some operations in Lustre that are not included. Therefore, we need to override some rules to create a fitting structure. In Figure 5.2 the resulting structure of the expressions rules for Lustre is shown. It is based on the structure of the `KExpressions` with some adjustments. A bold line starting with a + indicates that this rule was added or moved and a green line starting with - shows that this rule is removed. The precedence can be extracted by reading this tree bottom up, so the `PreExpression` has a higher precedence than the `NegExpression`.

This grammar is implemented using Xtext. It allows for a notation that is similar to the extended Backus–Naur form and provides many features for auto generation of code. The model for the Lustre program for example is generated automatically from the grammar. Moreover, other useful classes are generated that can be further extended to provide more specialized features. We introduce the `Validator` and the `ScopeProvider` in the following.

5.1.1 Validator

The `LustreValidator` is a class that extends the automatically generated class `AbstractLustreValidator` and allows the specification of custom validation rules. Some methods within this class are annotated with `@Check` indicating that they shall be invoked when validation takes place. The parameter define on what type of objects this check is performed on.

For the Lustre validation there are two types of validation. The first is an error indicating that the used feature is not supported. This error is shown for all of the following features:

- ModelDeclaration
- PackageDeclaration
- PackageEquation
- ExternalNodeDeclaration
- TypeDeclaration
- StaticParam
- NodeReference

Furthermore, there are checks that enhance the design process. This includes information about duplicate variables or nodes or undefined outputs. In Table 5.1 the different methods of the validation class are listed and their behavior is outlined.

5.1.2 ScopeProvider

The content assist can help to show possible input alternatives at the cursor position. For specialized language constructs this content assist needs to know the scope for variables and nodes for the current position in the program. In Xtext this can be manually refined using the `ScopeProvider`.

For the `LustreScopeProvider` there are two cases that need to be covered. First, nodes can be referenced in other nodes. Therefore, the `LustreScopeProvider` needs to extract all available nodes and provide them in the scope.

Second, all inputs, outputs, variables and constants within a node are within the scope of that node. They need to be extracted and passed on. Furthermore, the constants outside of nodes need to be included, too.

Table 5.1. Methods in the `LustreValidator` that check on the general problems with the model

Method Name	Description
`checkDuplicateVariable`	Global and local constants, input and output variables as well as regular variables must differ in their names. A warning is displayed if this does not hold.
`checkDuplicateNodeName`	Node names must differ. A warning is added if this is violated.
`checkWhenExpressionVariableClockDefinition`	The `when` operation can only be used for variables that are defined with the same clock in the variable definition. An error is displayed if this is violated.
`checkCallReferenceReturnCardinalities`	A node reference may return multiple values. The left side of an equations must contain as many variables as the node returns. An error is displayed if this is violated.
`checkOutputDefined`	Each output of a node must be defined through exactly one equation. A warning is displayed if either there is no equation or there are too many equations for the output.

5.2 Lustre to SCCharts Transformation

In KIELER the compilation system KiCo allows for a generic usage and extension of existing functionalities. The transformation from Lustre to SCChart is implemented using the processor infrastructure. It has a generic definition that allows to define the source and the target for the transformation. In our case the transformation takes a `LustreProgram` as source and an `SCCharts` as target and the entire system only consists of this one processor.

In contrast to previous approaches, the focus of our thesis is the implementation of a transformation that primarily uses dataflow in SCCharts. There are already approaches that transform the dataflow to controlflow and therefore perform their transformation differently. However, both approaches need to transform the basic structure of a Lustre program to a basic structure of SCCharts and transform expressions fitting to the provided language scopes. Therefore, an abstract class is introduced that handles those basic parts of the transformation.

In Figure 5.3 is a simplified class diagram that shows the relation of the transformations and the abstract class. This abstract class is called `CoreLustreToSCC`. It is a `Processor` with the specified source and target. Basic functionality like the transformation of the package body, the node interface and variable or constant declarations are handled here. The class `LustreToSCCharts` then handles the aspects that depend on the transformation strategy for transforming dataflow to dataflow and controlflow to controlflow. The `LustreToSCCControlFlowAproach` realizes the strategy for using SCCharts controlflow for the visualization of the Lustre code.

The abstract class already handles the creation of a new SCChart for each node in the Lustre program. Constants outside of nodes also create an SCChart containing only those constant declarations. These constants are imported in all SCCharts representing a node using the `extends` feature as described in Section 4.2.1. The inputs, outputs and variables of a node are added to the SCChart as such.

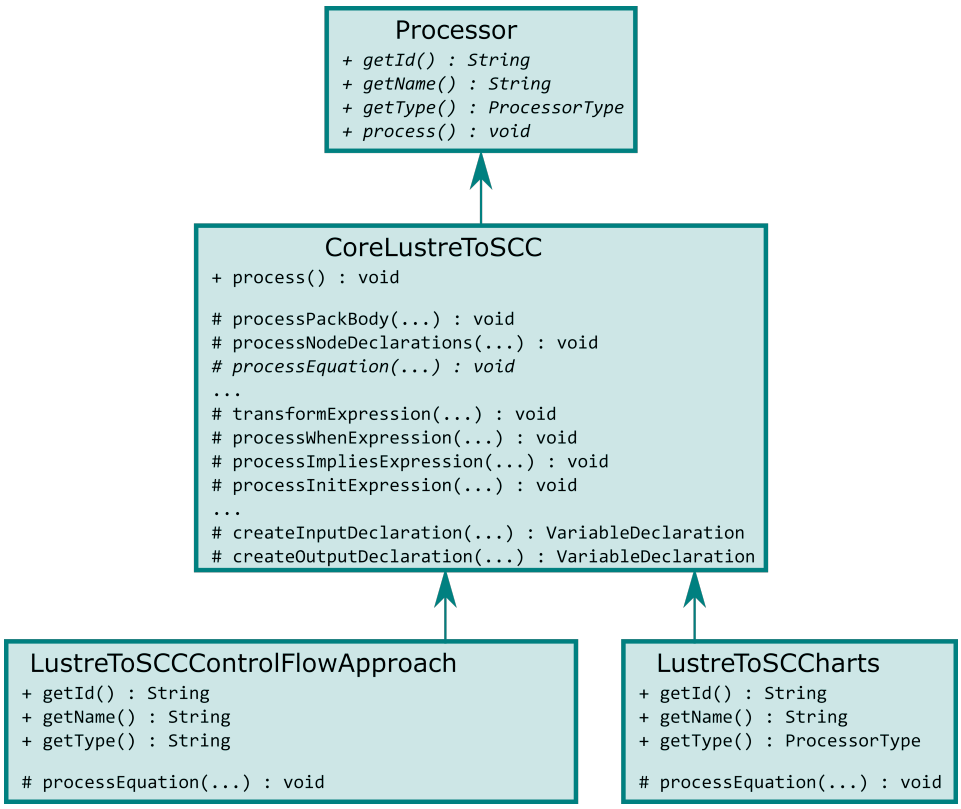


Figure 5.3. Part of the class diagram showing the inheritance of the new transformation processors and the abstract transformation class

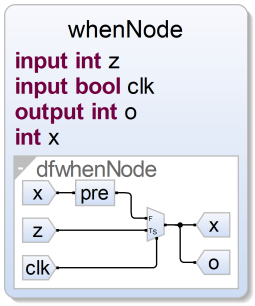
Next, the contents of the node body are transformed. The body may contain equations, automata or assertions. The transformation of equations is delegated to extending classes through the abstract method `processEquation`. Automata, however, are transformed to equivalent SCChart controlflow automata. The different transition types are all available in SCChart, therefore it is just a one-to-one mapping to the SCCharts side. The assertions are transformed to an annotation to the root SCChart. More details on this transformation and how those model checking properties are handled in SCCharts is explained in Section 5.2.1. If the implemented behavior in the abstract class does not describe the desired behavior, the extending classes may always override the corresponding method implementing a different handling of those elements.

The general transformation of expressions in Lustre to expressions in SCCharts is done through dedicated methods in the class `CoreLustreToSCC`. The idea is that different approaches only differ in the way they express the behavior, which is given through the equations. The actual transformation for each expression was already described in Section 4.2.

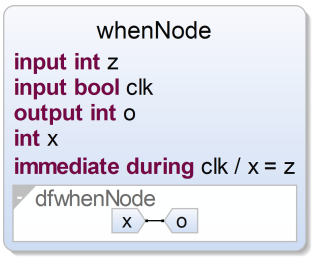
Moreover, the abstract processor has two boolean properties that can be used to influence the transformation behavior. The first is `de.cau.cs.kieler.lustre.processors.lustreToSCC.useDuringActionsForWhen`. If this property variable is set to true, the when transformation is not realized using the conditional but instead a during action is created with the corresponding clocks as trigger. Figure 5.4 shows the effect of this property. The Lustre program in Figure 5.4a is transformed to SCCharts with and without during actions. The clock `clk` is added as a trigger to the during action in case during

```
1 node whenNode(z:int; clk:bool)
      returns (o:int);
2 var x : int when clk;
3
4 let
5   x = z when clk;
6   o = current x;
7 tel.
```

(a) Original Lustre, copied from Figure 4.8a



(b) Transformation without during actions

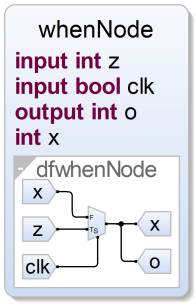


(c) Transformation with during actions

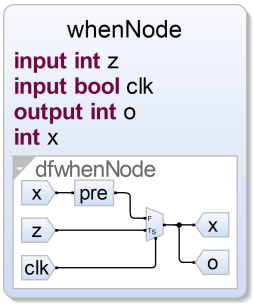
Figure 5.4. A Lustre program using clocks and the transformed SCCharts showing the usage of a conditional and a during action

```
1 node whenNode(z:int; clk:bool)
      returns (o:int);
2 var x : int when clk;
3
4 let
5   x = z when clk;
6   o = current x;
7 tel.
```

(a) Original Lustre, copied from Figure 4.8a



(b) Transformation without pre operator



(c) Transformation with pre operator

Figure 5.5. A Lustre program using clocks and the transformed SCCharts showing both variants to transform a when expression

actions are used. This approach has the advantage that we do not explicitly need to specify what happens if the clock is not set. However, we do not get to visualize the dataflow that results from the when statement. Therefore, this property is turned off by default.

The second property is `de.cau.cs.kieler.lustre.processors.lustreToSCC.noPreInWhenTransformation`. This might be interesting for optimizations. In SCCharts the sequential constructiveness is realized which allows to sequentially read and write a variable. For the transformation of the *when* statement using a conditional, this can be used. The *pre* expression for the else-branch is not needed since the variable can be read and then written to sequentially. Setting this property to true causes when statements to be transformed with no *pre* when using the conditional. This simplifies the transformation process because no variables and parallel regions need to be introduced in order to remove the *pre*. Note that this property causes Lustre programs to be transformed to SCCharts that are not constructive in the classical sense any more. Therefore, this property is turned off by default, too.

It was already mentioned that the class `LustreToSCCharts` implements the basic transformation using dataflow for dataflow and controlflow for controlflow. Basic methods needed by the `Processor` are implemented and additionally the method `processEquation` is defined to specifically fit this transformation for the dataflow approach. It creates a dataflow region containing an `Assignment` for

each equation in the program and parses the expression of the right side of the equation using the `CoreLustreToSCC` implementations.

5.2.1 Assertions in SCCharts

The transformation of assertions is already handled in the abstract class. They do not introduce model functionality but are relevant for model checking with the *lesar* model checker [Ray08]. This model checker *lesar* uses the principle of a synchronous observer for verification. A new node instantiates the to-be-checked node. It has one output that is true as long as the program behaves as expected. The expected behavior is defined through logical connections of the inputs and outputs of the to-be-checked node. The *lesar* model checker then assures that the output of the observer remains true for all possible paths.

The assertions in Lustre are used to specify that certain paths in the evaluation tree do not need to be checked. As an example it may be given that two inputs are never true at the same time. This can be specified using an assertion and during model checking the paths with both inputs set to true are removed.

Model checking for SCChart is currently in development. Stange works on integrating different model checkers into the KIELER environment [Sta19]. The properties that are checked can be specified above the SCChart using annotations but instead of properties, also invariants can be defined. These invariants are used to cut paths in the evaluation tree and thus are well suited for the transformation of assertions from Lustre to SCCharts. For each assertion an annotation starting with `@Assume` is added above the textual SCChart. Afterwards the Lustre assertion expression is translated to the SCCharts expression and concatenated to the annotation. This way we transform the meaning of the `assert` in Lustre to the SCCharts model.

As an example, the assertion `assert not(a and b)` in Lustre defines for boolean two inputs `a` and `b` that they never occur at the same time. On the SCCharts side this property is added above the corresponding textual SCChart as an annotation `@Assume "!(a && b)"`. During model checking this property can be loaded and checked for the behavior of the corresponding SCChart.

5.2.2 Revised: Lustre to SCCharts Controlflow

A transformation from Lustre to SCCharts using the controlflow feature of SCCharts was already implemented prior to this thesis. However, during the process of this thesis, the grammar for Lustre was restructured and founded on `KExpressions`. This facilitates writing new transformations, but the existing transformations need to be reworked.

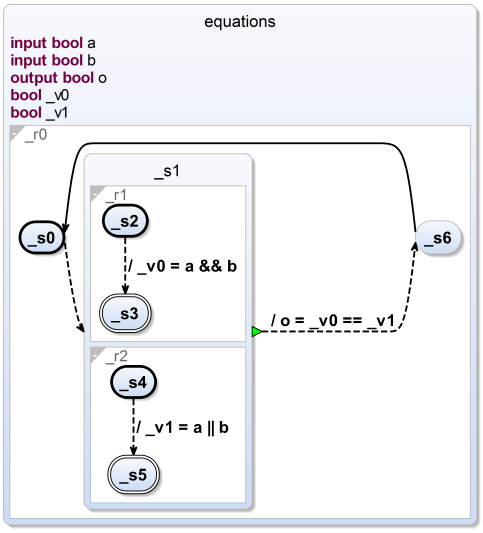
The abstract class `CoreLustreToSCC` helps to achieve this revision of the transformation. Basic functionality is already provided and only the specific behavior for an equations needs to be implemented.

In general the previous behavior is preserved. We added a boolean option to the transformation named `de.cau.cs.kieler.lustre.processors.lustreToSCC.controlFlow.useDuringActions`. If this property is set to true, the transformation uses during actions to express the right side of the equation if there are no complex operations included. The previous transformation by Pascutto defined *initialization* and *conditional* as complex operations [Pas17]. The during action does not have a trigger and executes the transformed expression each tick.

If the property `de.cau.cs.kieler.lustre.processors.lustreToSCC.controlFlow.useDuringActions` is not set, expressions are also illustrated in the controlflow. The idea is to create a hierarchically state representing the expression tree. The leaves of the expression tree are the most nested expressions and they are within the highest hierarchy level of the model. In Figure 5.6 this idea is illustrated.

```
node equations(a:bool; b:bool) returns (o:bool
    );

let
  o = (a and b) = (a or b);
tel.
```

(a) Lustre



(b) SCCharts

Figure 5.6. A Lustre program with nested expressions and the transformed SCChart using the controlflow approach

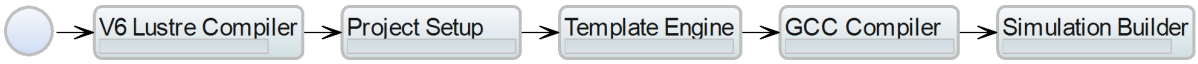


Figure 5.7. Lustre V6 compile and simulation chain

The leaves are the logical operations *and* and *or*. They are calculated in parallel and a new variable is introduced that holds the value. On a higher level, these newly created variables are reused for the *equals* operation.

This transformation approach is interesting because it uses only controlflow in the target model. Moreover, expressions can be evaluated in parallel and this might save time. However, the expression complexity is usually not as high as this would make an actual difference. Additionally, the complexity of the introduced hierarchy and the contrast of dataflow in the original Lustre model and controlflow in the target model might hinder it to be a visual representation for the Lustre program.

5.3 Lustre Simulation

In KIELER there is also the possibility to simulate a model. For SCCharts this is achieved through the transformation to C code and its compilation. Communication from the compiled model inputs and outputs to the user interface is handled through Json. However, also other compilers can be integrated. For example in Esterel the Inria Esterel Compiler is integrated in KIELER. This offers the possibility to use the defined simulation interface with the provided Json communication and already existing compilers.

5.3.1 Lustre V6 Simulation Compile Chain

In Section 3.3 the Lustre V6 compiler was introduced. It can be used to compile Lustre models to C code and execute this C code. This compiler is integrated into the KIELER infrastructure.

```
<#include "/templates/injection.ftl">

<@inject position="header" />
#include "${tickdata_name}.h"

<@inject position="global-decl" /><#nt>

<@inject position="body" />

int main(int argc, const char* argv[]) {
   <@inject position="local-decl" /><#nt>

   <@inject position="init" /><#nt>

   // Tick loop
   int run = 1;
   do {
      <@inject position="start-loop" /><#nt>

      // Read inputs
      <@inject position="input" /><#nt>

      <@inject position="pre-tick" /><#nt>

      // Reaction of model
      ${tickdata_name}_step(<@inject position="step-parameter"/><#nt>);

      <@inject position="post-tick" /><#nt>

      // Send outputs
      <@inject position="output" /><#nt>

      <@inject position="end-loop" /><#nt>
   } while(run);

   <@inject position="end-main" /><#nt>
}
```

Listing 5.7. Template file used during the setup for the simulation

In Figure 5.7 the new compile chain is shown. The first processor, the *V6 Lustre Compiler*, implements the compilation with the V6 compiler to C code. The processor itself is contained in the plug-in `de.cau.cs.kieler.lustre` and it uses the binaries located in `de.cau.cs.kieler.lustre.compiler` through the `LustreV6Compiler`. The binaries themselves are only available for Linux systems. After the compilation of this processor, the files listed for the Lustre V6 compiler in Section 3.3 are generated.

The next processor, the *Project Setup*, copies some files into the directory of the compiled files. One of those files is the `c-main.ftl`. Listing 5.7 shows this file. It defines the structure of the main loop for the simulation. It is a C code file except for the `<@inject .../><#nt>` statements. Those statements define injections points for later addition of code to this file. This allows for a dynamic extension of this file. Other than the injection points in the `c-main.ftl` there is a main method containing a while-loop for the execution of the model.

```c
#include "lib/ticktime.h"
#include <stdio.h>
#include <unistd.h>
#include "lib/cJSON.h"
#include "equations_equations.h"

double _ticktime;
// Output variable declaration
int _SIM_VAR_0o;
// Input variable declaration
int _SIM_VAR_0a;
int _SIM_VAR_1x;
int _SIM_VAR_2y;

// METHODS FOR SIMULATION COMMUNICATION LEFT OUT HERE
// ...

int main(int argc, const char* argv[]) {
  // Init lustre
  equations_equations_ctx_type* ctx = equations_equations_ctx_new_ctx(NULL);
  notifyInterfaceVariables();

  // Tick loop
  int run = 1;
  do {

    // Read inputs
    receiveVariables();
    resetticktime();

    // Reaction of model
    equations_equations_step(_SIM_VAR_0a, _SIM_VAR_1x, _SIM_VAR_2y, &_SIM_VAR_0o, ctx);

    _ticktime = getticktime();

    // Send outputs
    sendVariables();

  } while(run);
}
```

Listing 5.9. Example main with completed injections

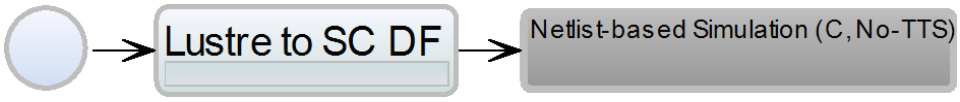


Figure 5.9. Lustre to SCChart compile and simulation chain

Listing 5.9 shows a main-file that was generated for an example program. All the injections needed for the simulation are added to the template file. In the first lines, the needed include statements for the simulation, general system in and out controlling and Lustre-specific files are included. In line 11 to 14 the inputs and outputs that the model works with are added. They are positioned at the injection `global-decl` in line 6 of the `main.ftl`. As already mentioned, the simulation communicates with the compiled files using Json. Therefore, methods are implemented that convert the system in and out to Json objects. Those methods are located after the interface declarations starting at line 16. They are called in the main routine in line 22, 29, and 38.

In the introduction to the Lustre V6 compiler in Section 3.3 it was mentioned that the generated step function of the compiled files for a model requires a `struct` in case that the node actually requires memory. During the execution of the V6 compiler, the output of the compiler is scanned and a variable is set that indicates whether the model has a state nor not. The example used for the creation of the main file in C format had an internal state. Therefore in line 21, the corresponding structure is created. In addition, the step function in line 33 passes this structure as an argument. If the compiled model is stateless, the structure is not generated and it also not added to the list of parameter for the tick function.

The next step is the execution of the GNU C compiler on the main file that is processed by the template engine by injecting all required code. A file named `simulation.exe` is generated and with the last processor, the *Simulation Builder*, the simulation in KIELER of the selected model can be started.

5.3.2 Lustre to SCCharts to C Simulation Compile Chain

In Section 5.2 we introduced the processor for the transformation of Lustre models to SCCharts models. For SCCharts models there exists a compilation chain that starts the simulation and in Figure 3.13 this compilation chain was shown.

The *Lustre to SC DF* processor now allows to define a compilation chain that reuses the SCCharts compilation chain. The Lustre model is translated to an SCCharts model and then this SCCharts model is compiled and executed. It basically consists of the processor chain in the Figure 3.13 except for the newly defined processor being chained in front of all the other processors. In Figure 5.9 this new compilation chain is shown. The component labeled *Netlist-based Simulation* is a container that includes the basic SCCharts compilation chain.

In contrast to the Lustre V6 simulation, this approach simulates the transformed SCChart. Therefore, the created behavior results from the transformation itself. This is useful for the comparison of the original model and the transformed model. They should both produce the same outputs. In Chapter 6 these two approaches are used to evaluate that the transformation works properly.

5.4 Automatic Tests

KIELER provides a continuous integration that creates a build job if changes are committed. This build job also executes a set of automatic tests. Figure 5.10 illustrates this test system. The idea is to have a repository containing various amounts of example files for all different languages and

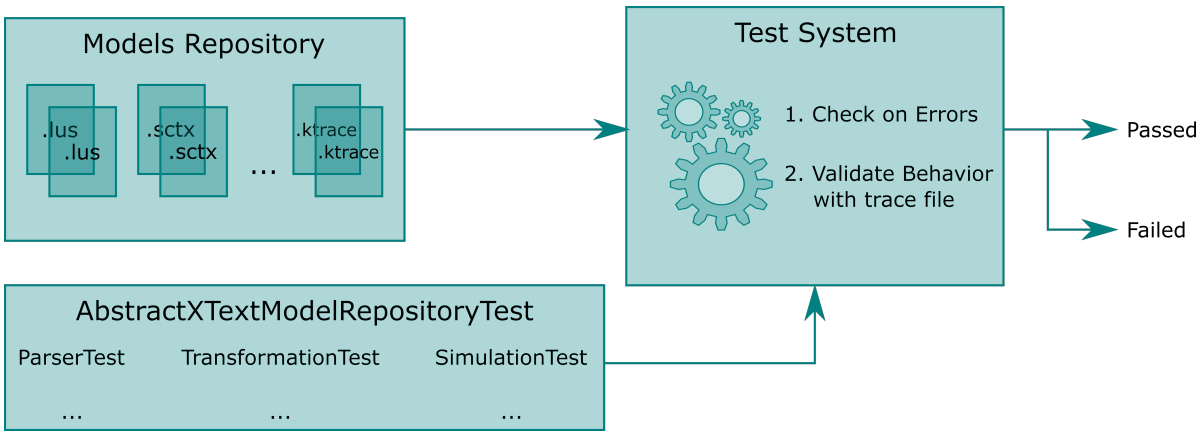


Figure 5.10. Overview of the test system

optionally files that contain execution traces to the specific model. Additionally, there are different tests extending `AbstractXTextModelRepositoryTest` that can fulfill different use cases. The abstract class `AbstractXTextModelRepositoryTest` already provides the needed functionality for loading the models from the repository, filtering them and running a compile chain on them.

For the Lustre scenario those automatic test classes are all located in the plug-in `de.cau.cs.kieler.lustre.test`. They define a subset of the programs in the models repository that they work with. For example the parsing of the files can be checked to ensure that there are only errors and warnings to a model if those were expected. Transformations can be checked, too. The transformation is performed and no errors should occur. Lastly, there is a special type of test. They extend the `AbstractSimulationTest` that itself extends `AbstractXTextModelRepositoryTest`. This abstract class `AbstractSimulationTest` adds features that are needed for comparing simulation behavior. These simulation tests look for an associated file containing traces for the model. The model is simulated with the inputs provided in the traces and the outputs are compared to the outputs in the traces. This allows to test if the models behave the way they used to or the way they should. These tests are later explained in more detail for evaluating the Lustre features in KIELER. In this section we focus on the parser and the transformation tests.

5.4.1 Parser Test

The first test performed on the Lustre models is a parser test. The `LustreParserTest` checks each model on possible errors or warnings. In general, the files in the models repository are divided into those programs that are valid and those that are expected to cause an error or a warning. There are 11 tests to cover inspection of the error messages, each test covering a specific use case. As an example, there should be a warning if an output is not defined or if a node is defined twice and there should be an error if not supported features are used or clocks are used inconsistently within the program.

Only those files that should create an error or warning pass the test if an error occurs, all other models would fail. Additionally, the grammar is checked to work with the provided models. All files provided should be accepted by the grammar and changes that affect the set of accepted programs lead to a failure in this test.

5.4.2 Transformation Test

The next test compiles the models with the dataflow transformation. This should not lead to an error or an invalid model. In case something does not work, the test would fail and return the model that it failed on.

Chapter 6

Evaluation

In the previous chapters we introduced a transformation from Lustre to SCCharts dataflow. This transformation is now reviewed together with the initial motivation for this topic. The transformation can convert a Lustre program to an SCCharts program. However, the behavior of those two models should remain the same. The first section of this chapter introduces automatic tests that are added for testing the behavior of the models. They ensure that the transformation preserves the input and output behavior of the original models.

In the second section the motivation for the topic of this thesis is examined. The idea was to create an SCCharts model from Lustre code just like SCADE represents Lustre in a visual way. Larger SCADE models are looked into, the Lustre code is extracted and transformed to SCChart and both visual models are compared.

6.1 Automatic Behavior Tests

Section 5.4 introduced the automatic test system in the KIELER continuous integration. For ensuring that the Lustre transformation works properly, simulation tests are added. They use a compile chain combined with the KIELER simulation and make sure that the behavior corresponds to the behavior defined in so called trace files. Those files save inputs to a model for each tick and the expected outputs. Each of these trace files belongs to a program within the models repository.

6.1.1 Models Repository

As already mentioned, the automatic tests system uses the models from the models repository as input. The amount and the complexity of the models contained in the repository determines how precise the different simulation tests can evaluate the transformation. Therefore, the first step towards implementing meaningful tests is the integration of various models into the models repository.

The model files that are valid and supported Lustre programs are partially created manually. Every operation supported in Lustre has an associated model file that contains only this operation. This includes simple data operations, sequence operations or references. Moreover, some of the operators work with multiple inputs or can be cascaded. Those special cases using three operands are also added manually. In order to add models with multiple equations, there are also files that combine all operator tests into one file. Also the corner cases for *when* and *pre* introduced in 4.2.4 are added as a test. The state extension is not supported in the Lustre V6 compiler but for the transformations those model files are interesting because they can be converted to SCCharts and then simulated. An example for an automata included in the repository is shown in Figure 3.3. Additionally, there is a simplified version of it and an even simpler automata that illustrates the basic concepts of automata in Lustre. Lastly, the repository[1] provided by Jahier is integrated into the models repository. It is a public repository that contains various Lustre models. However, not all of the models can be supported because many

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

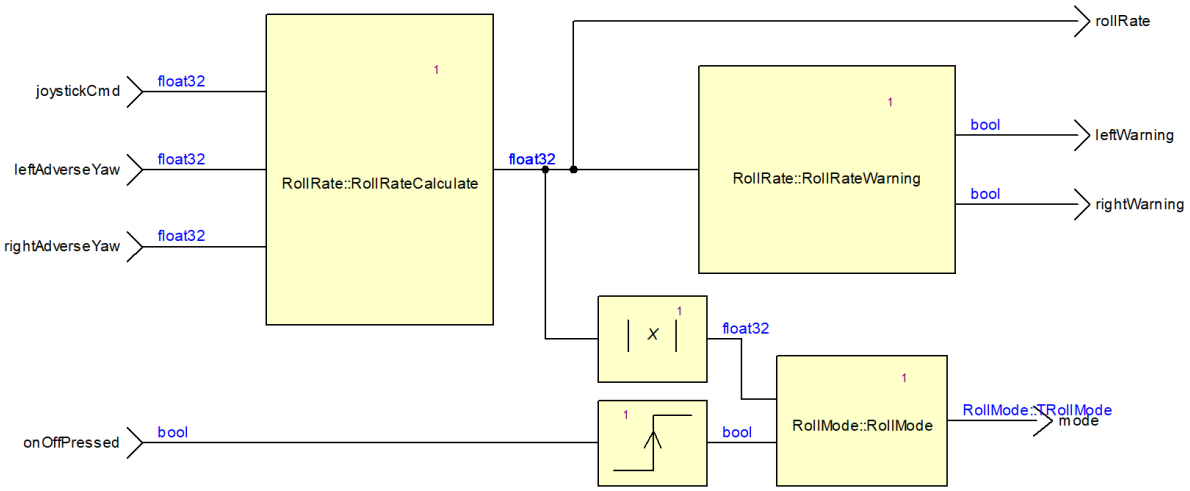


Figure 6.1. SCADE `RollControl` model

different features are used. Nevertheless, interesting models are included for the automatic tests because especially larger real-world models are included. The largest model is the heater control that contains 126 lines of Lustre code and within the entire models repository there are about 150 Lustre model files.

For most of these model files, a trace file is created and put into the repository that covers all possible behavior. These trace files are created using the Luste V6 Simulation in KIELER. This ensures that the actual behavior of the model is extracted on the Lustre side.

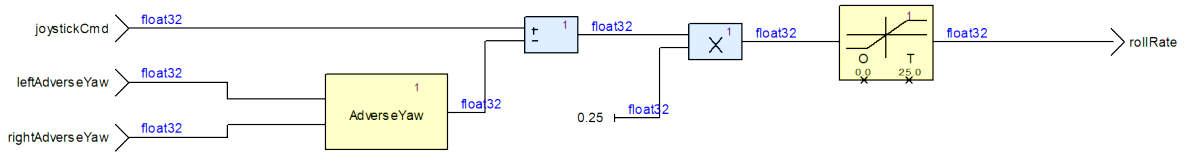
6.1.2 Simulation Tests

The `AbstractSimulationTest` was already mentioned in Section 5.4. Extending this class allows for basic functionality that is needed to create automatic tests for the simulation of the models behavior. This type of tests is especially interesting for the evaluation. Trace files can be provided and those are loaded during the simulation of the corresponding program.

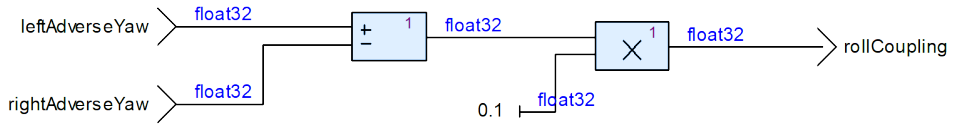
These files now offer a way to compare the behavior of different models to be equivalent. For this case we added three test classes: `LustreV6SimulationTest`, `LustreSccSimulationTest` and `LustreSccControlFlowApproachSimulationTest`. The first class, `LustreV6SimulationTest`, takes the Lustre V6 compilation chain and simulates the model with the trace files. This should always pass because the traces are created using this compilation chain.

The `LustreSccSimulationTest` first transforms the Lustre program to an SCCharts program. This is then compiled and simulated like a regular SCChart. The behavior of this model is then compared to the traces saved. There are certain features that are expressed differently through the transformation, especially clocks. However, as long as the outputs match, the behavior is considered to be the same. This test ensures that the dataflow transformation creates an SCChart that behaves like the original Lustre program.

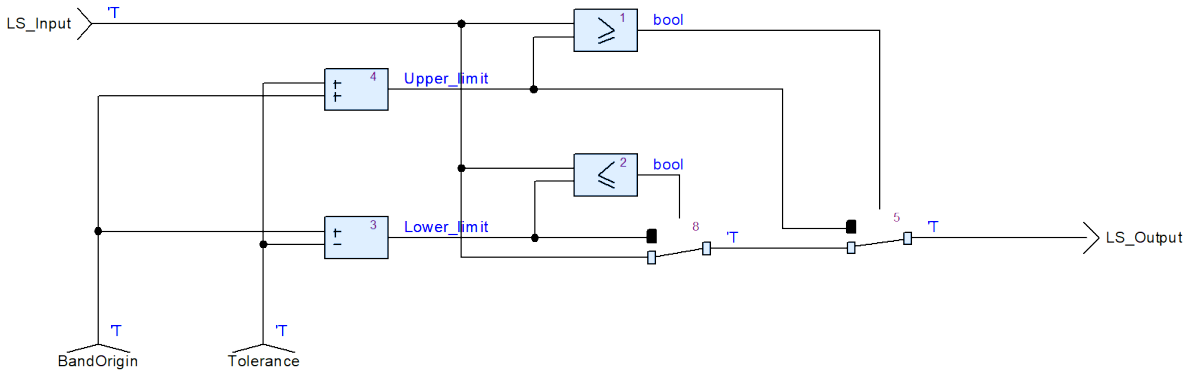
Lastly, the `LustreSccControlFlowApproachSimulationTest` checks the behavior of the program after the transformation from Lustre to SCCharts dataflow just like the `LustreSccSimulationTest` works for the dataflow to dataflow approach. The difference is in the used compilation system. In the `LustreSccControlFlowApproachSimulationTest` the controlflow approach is used for the transformation.



(a) SCADE `RollRateCalculate` model



(b) SCADE `AdverseYaw` model



(c) SCADE `LimiterSymmetrical` model

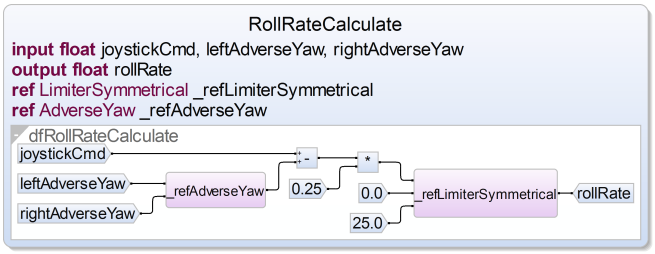
Figure 6.2. The `RollRateCalculate` example from SCADE and all referenced models

This might cause the model to look very different from the original model but this test preserves that the behavior is the same.

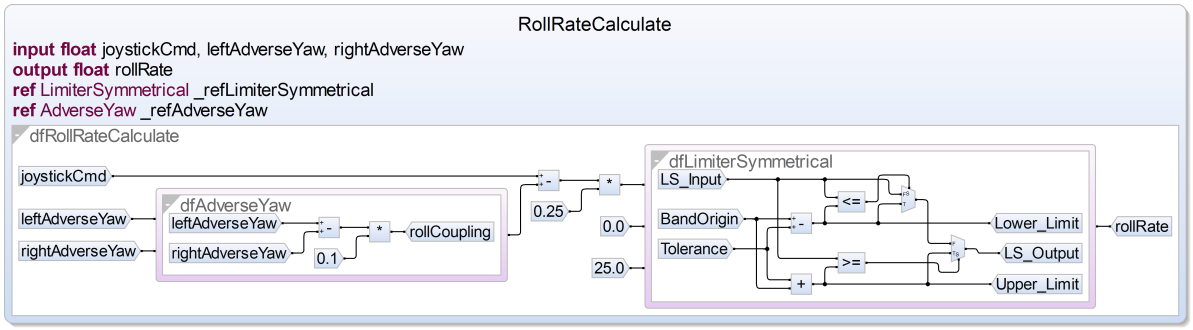
There are 56 tests that are executed through the `LustreSccSimulationTest`. All those tests pass with the result that the models have the same behavior in SCCharts as they had in the original model. The `LustreSccControlFlowApproachSimulationTest` can only handle fewer models because not the same scope of features is supported. References for example are not included in the current implementation for the controlflow transformation approach. However, all tests checking the behavior of the created controlflow SCChart also pass. In conclusion, the two transformation approaches appear to work properly and the creation of an SCChart from Lustre with the same behavior is accomplished.

6.2 SCADE Models

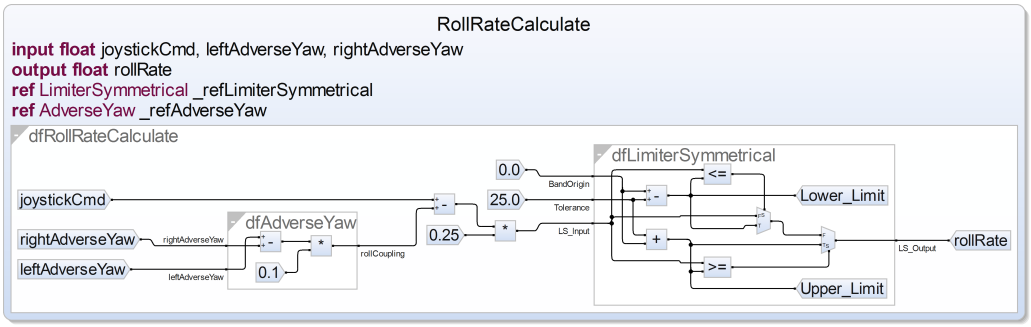
The motivation for the topic of this thesis is based on the equivalence of SCADE and Lustre. The idea is to improve the design process by allowing textual editing of Lustre models with a SCADE-like visualization that is generated automatically. In this section we evaluate SCADE models in comparison to the generated SCCharts dataflow model for this purpose.



(a) Collapsed Regions



(b) Expanded Regions



(c) Inlined Regions

Figure 6.3. Transformed `RollRateCalculate` example in SCCharts with expanded, collapsed and inlined referenced models

```
node RollRateCalculate(joystickCmd, leftAdverseYaw, rightAdverseYaw: real) returns (rollRate: real);
let
  rollRate = LimiterSymmetrical((joystickCmd - AdverseYaw(leftAdverseYaw, rightAdverseYaw)) * 0.25,
                        0.0,
                        25.0);
tel.

node AdverseYaw (leftAdverseYaw, rightAdverseYaw: real) returns (rollCoupling: real);
let
rollCoupling = (leftAdverseYaw - rightAdverseYaw) * 0.1;

tel.

node LimiterSymmetrical(LS_Input, BandOrigin, Tolerance: real) returns (LS_Output: real);
var Upper_Limit, Lower_Limit : real;
let
Upper_Limit = Tolerance + BandOrigin;
Lower_Limit = BandOrigin - Tolerance;
LS_Output = if (LS_Input >= Upper_Limit)
         then (Upper_Limit)
         else (if (LS_Input <= Lower_Limit)
              then Lower_Limit
              else (LS_Input));
tel.
```

Listing 6.2. Extracted Lustre code from the SCADE `RollRateCalculate` example

In SCADE there is an amount of example programs integrated into the tool that illustrate the usage of SCADE. Moreover, there are larger models for certain tasks such as a digital stopwatch. However, the introduced Lustre transformation cannot handle type definitions and self-defined types are a frequently used functionality in the provided SCADE examples. In order to compare a SCADE model to a transformed SCChart model, an example is used that does not use type definitions. Additionally, this model should contain some complexity so the effect on the visualization can be reviewed.

We selected the operator `RollRateCalculate` in the example `RollControl`. In Figure 6.1 is the diagram for this parent model. It calculates the roll rate depending on the joystick command and the left and right adverse yaw. If a button is pressed the roll mode toggled from on to off. During the activated roll mode, the mode failsoft is activated if the rate is extends a threshold. This resulting roll mode is also an output of the model. In addition, warnings are displayed if the roll rate is larger than an upper or lower limit.

Within this model, there are entities that are abstracted to calculate a specific part of this operation. One of these operators is the `RollRateCalculate` that calculates the roll rate depending on the adverse yaw of the left and the right side as well as the joystick command. The SCADE model for this operator is shown in Figure 6.2a. Inside of this model, there are two references to operators called `AdverseYaw` and `LimiterSymmetrical`. Those two models are also shown in Figure 6.2b and 6.2c.

SCADE offers the possibility to extract the Lustre equations for an operator. In order for the transformation to work, we need these Lustre equations combined with a manually added interface. The equations for the corresponding model parts extracted from SCADE are shown in Figure 6.4. Moreover, we optimized the equations manually so there is not an equation for each wire but for each output. An exception is the node `LimiterSymmetrical`. There are two wires that are names explicitly,

```
L8 = joystickCmd;
L7 = leftAdverseYaw;
L6 = rightAdverseYaw;
L5 = AdverseYaw(L7, L6);
L4 = L8 - L5;
L3 = L4 * L2;
L2 = 0.25;
rollRate = L1;
L1 = pwlinear::
     LimiterSymmetrical(L3,
      0.0, 25.0);
```

(a) `RollRateCalculate`

```
L1 = leftAdverseYaw;
L2 = rightAdverseYaw;
rollCoupling = L4;
L3 = L1 - L2;
L4 = L3 * L5;
L5 = 0.1;
```

(b) `AdverseYaw`

```
L1 = L6 >= Upper_limit;
L3 = L6 <= Lower_limit;
Lower_limit = L7 - L2;
Upper_limit = L2 + L7;
L4 = if L1
    then (Upper_limit)
    else (L5);
L5 = if L3
    then (Lower_limit)
    else (L6);
L6 = LS_Input;
LS_Output = L4;
L7 = BandOrigin;
L2 = Tolerance;
```

(c) `LimiterSymmetrical`

Figure 6.4. SCADE equations that are generated from the diagram

so those are handled as a variable and are not left out for optimization. So resulting from the SCADE model, we get the Lustre code presented in Listing 6.2. This Lustre code is then transformed to SCCharts.

Figure 6.3a shows the transformed SCChart resulting from this Lustre code. The referenced models are transformed, too. However, in KIELER it is possible to expand referenced models instead of opening them in a separate file. In Figure 6.3b is the transformed SCChart with expanded reference models. Moreover, those referenced models can be inlined, connecting inputs and outputs directly. Figure 6.3c shows this model with inlined references.

Comparing this SCCharts model to the SCADE models shows that they both have a similar appearance. Operators are illustrated using actors and inputs and outputs are connected to operators through wires. Moreover, the flow of the data is visualized going from left to right with the overall inputs and outputs of the model positioned at these ends.

An advantage of SCCharts is, however, that the interface of the node and local variables are included in the visualization. Additionally, the node `LimiterSymmetrical` has two hidden variables in the SCADE version. Their value can be set in the properties when clicking on that node reference, but there is no need to add a dedicated wire with a constant on it. In the diagram, those hidden variables are indicated by the `0` and the `T` at the bottom of the corresponding node in Figure 6.2a. Those are translated to parameters in the Lustre version and thus are also translated to inputs for the SCCharts diagram. Moreover, the SCADE version uses a specific designed reference operator for the `LimiterSymmetrical`. Those operator styles, however, can be created for SCCharts, too. This creation is not part of the automatic transformation since it uses Lustre code which does not provide any information about the operator style in SCADE but it can be modified after the transformation.

The selected flow of the data, however, is illustrated similarly in both models. The SCADE model needs manual layouting so this is taken as a template for how the user would want this layout to look like. The automatic layout in SCCharts also managed to visualize the flow in a very similar way.

The model `LimiterSymmetrical` has two variables. In SCADE those variables are visualized in the model by naming the wire. Therefore, the influence of variables on the visualization is small. In SCCharts however, the variables are visualized explicitly if they are added to the state. Variables can also be added to regions and for the dataflow regions this would cause them to not be visualized explicitly. However, this wire is currently not labeled with the variable name and so both approaches either suppress the variable entirely or create extra nodes for the variables.

In conclusion the goal for the transformation to create a visualization that looks like the one provided by SCADE is achieved. The benefits of using SCADE are transferred to the SCCharts visualization. The dataflow and the different actors connecting variable and values with operators is almost identical. However, in order to further evaluate the usability of the visualization in comparison to SCADE, type declarations should be supported. This would allow to compare larger models and the process of selecting a SCADE model for the comparison is more variable.

6.3 Limitations

The above evaluations show that the introduced transformation preserved the model behavior for the set of Lustre programs in the models repository. Moreover, the visualization achieves a SCADE like appearance except for minor differences. Nevertheless, there are also limitations for this transformation.

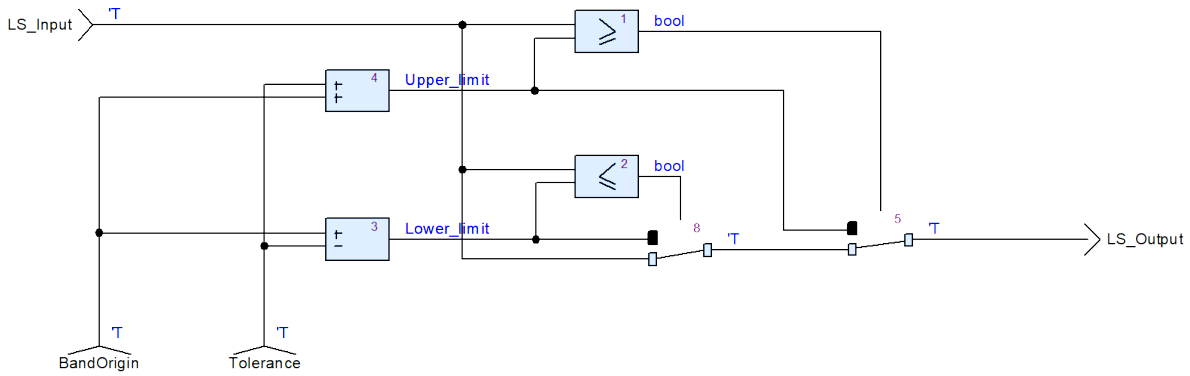
In SCADE the visualization is created manually. Therefore, it is an intuitive decision to reuse already calculated results for further calculations. In Figure 6.2c the result of the subtraction of `BandOrigin` and `Tolerance` is saved in the variable `Lower_limit` and the wire for this variable is reused in the comparison and the conditional without duplicating the subtraction operation.

For the transformation from Lustre to SCCharts, however, it is essential that calculations that shall be reused and not duplicated for each occurrence are defined as a variable. In Figure 6.5b is an example for the `LimiterSymmetrical` in SCCharts with the explicit definition of the variables `Upper_Limit` and `Lower_Limit`. Those variables are used for the comparison operators and the condition. This causes the dataflow to connect the input of those operators to the result of the addition and subtraction, thus the value of the variable.

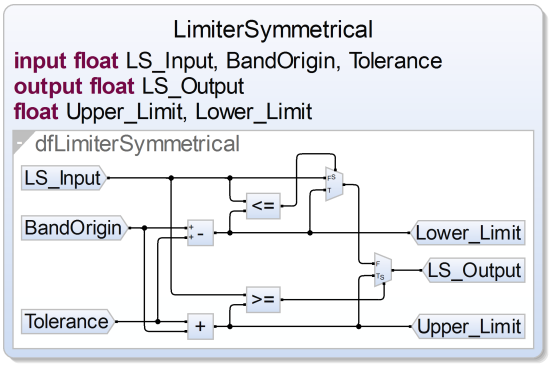
In contrast in Figure 6.5c the calculations are duplicated in the code instead of using a variable. This causes the visualization to also duplicate those operators. Instead of one subtraction and one addition, it includes two additions and two subtractions.

However, in the version using variables, the nodes for the variables are included in the diagram. In order to prevent these nodes, the variables can be added to the dataflow region instead of the state. In Figure 6.5d the version declaring the variable in the region is shown. This causes the explicit integration of the variable node in the diagram to be suppressed. This version is closest to the SCADE visualization and is used as the default strategy in the transformation.

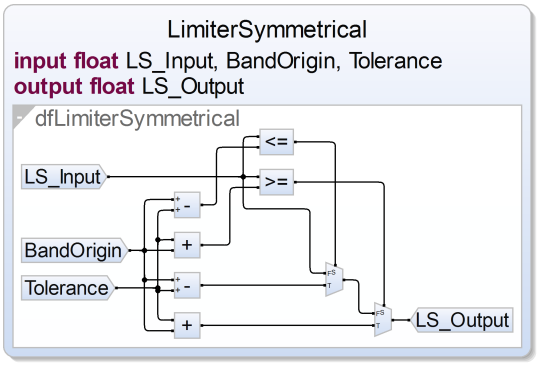
In conclusion, the usage and the position of variable declarations in SCCharts have an impact on the created model. In order to achieve a good visualization, computations that are made more than once should be declared for a variable. This process can already occur on the Lustre level. However, the label for the variable is not visualized by an explicit component in the SCCharts visualization. This preserves the similarity to SCADE that local variables are just named wires and no explicit components are used for the visualization.



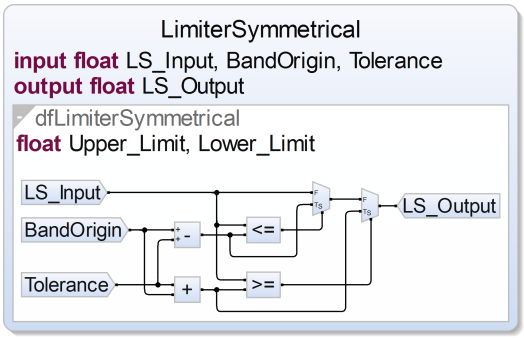
(a) SCADE `LimiterSymmetrical` model (copied from Figure 6.2c)



(b) With variables `Upper_Limit` and `Lower_Limit`



(c) Without variables `Upper_Limit` and `Lower_Limit`



(d) With variables `Upper_Limit` and `Lower_Limit` defined in region

Figure 6.5. The SCADE `LimiterSymmetrical` example in SCADE and transformed to SCCharts

Chapter 7

Conclusion

This chapter summarizes the achieved goals for the transformation from Lustre to SCCharts dataflow in the context of the initial motivation for this topic. Moreover, ideas for improvements and extensions are proposed.

7.1 Summary

The motivation for this thesis is the equivalence of Lustre and SCADE and the concept of transient views that is one of the leading principles for the tool KIELER. The design process of SCADE/ Lustre programs can be enhanced by allowing textual modification and implementation combined with an automatically generated visual component. This new design approach for Lustre is implemented in KIELER. The synchronous language SCCharts is used to illustrate the dataflow using a components and wires just like SCADE does.

In order to create this visual component for Lustre, the first step is the comparison of Lustre/ SCADE and SCCharts. Both languages use a synchronous MoC whereas the SCChart MoC is a conservative extension to the Lustre MoC. Moreover, the visual components of SCADE and SCChart are compared. Both use components and wires to express the dataflow and those operators are all designed similarly.

The next step is the implementation of a transformation from Lustre to SCCharts. In SCCharts the concept of clocks is not part of the language scope and instead of streams, variables are used. Therefore, the transformation takes care of these special languages features in order to transform them to plain SCCharts with the same behavior the Lustre program has. The simple data operations in Lustre are already equivalent to those in SCChart.

The last step is the evaluation of the introduced transformation. About 150 Lustre programs are used to express a broad range of the feature scope and program dimensions. Automatic tests are implemented that are executed on these programs in the models repository. They test the output of the program prior and after the transformation given the same inputs. This ensures that the transformation preserves the behavior for all test programs. Lastly, a SCADE program is transformed to Lustre and then to SCChart in order to compare them on the visual level. The result is that the dataflow uses similar components and wires like the original model. Except for details concerning visual nature of the components, the diagrams illustrate the same model. However, the position of the variable declarations in the SCCharts have a great impact on the design and this may result in slightly different variants of the diagrams. Also since clocking is not part of SCCharts those features are not visualized in a similar manner using a single component.

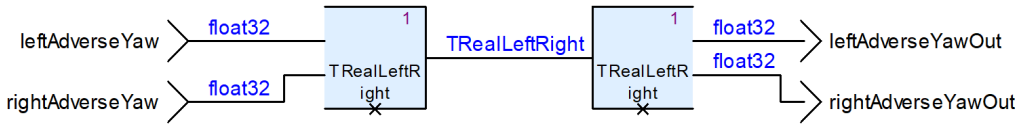
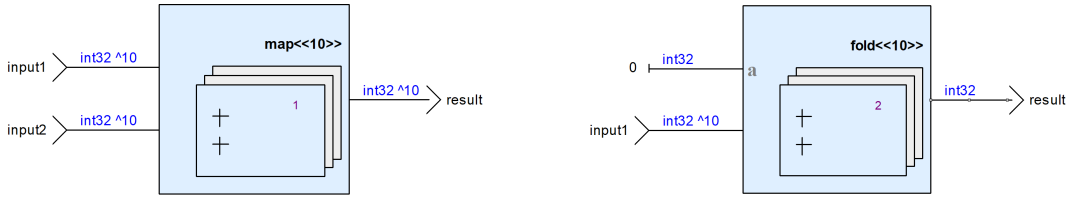


Figure 7.1. Composition and decomposition with structures in SCADE



(a) map operation **(b)** fold operation

Figure 7.2. The map and the fold operation in SCADE

7.2 Future Work

This thesis introduces a transformation that creates an SCChart for the dataflow in Lustre programs. However, there are also possible improvements for this transformation and other topics that originated during the development process.

7.2.1 Lustre Feature Extensions

In Section 3.1 a set of supported Lustre features is introduced. However, there are some language constructs that would increase the design comfort for Lustre in KIELER.

As already mentioned, type declarations are not supported yet. However, this features is used in many example programs in SCADE and Lustre. Supporting this feature would allow evaluating the results of the transformation for larger programs. Especially the visualization of those self-defined types is an interesting aspect for the comparison.

As an example in the `RollControl` model in SCADE different modes can be expressed by defining enumerations that represent the corresponding mode. The mode itself is a new type and a wire can have the type of modes. Moreover, values for the left and right yaw of an airplane are encapsulated in a structure representing the yaw. This structure definition introduces two new operators. The first operator takes the parts of the structure and creates a new structure with the output wire typed with the structure. The second operator takes a structure and decomposes the components from the structure. In Figure 7.1 those two operators are shown for the example of a structure containing left and right yaws. The label `TRealLeftRight` is the name of the created structure, thus the type of the wire after the first component is `TRealLeftRight`.

Extending the transformation by the concept of type declarations would greatly enhance the program design and allows for more possible comparisons with larger SCADE models.

In SCCharts references to other SCCharts can be skinned with an arbitrary `.kgt`-image. All operators and labels in KIELER are designed in this format internally but also users may define `.kgt`-files in order to define a custom skin for referenced SCCharts. The SCChart itself or the reference declaration may contain an annotation like `@figure "skinpath.kgt"` that specifies the design for this SCChart. This

custom operator design could be lifted to the Lustre level. References could be annotated and then the transformation annotates the SCChart in order to preserve the defined operator design.

Arrays are currently not supported during the transformation whereas SCCharts already has array support. Extending the allowed features for the transformation by arrays would also allow to think of a transformation that realizes the `map` and `fold` operations that are available in Lustre and SCADE. They facilitate array usage and operations with them. They can ease the visual complexity because operations on all array elements can be expressed with only one operator.

The map operation allows to perform an n-ary operation on n arrays of the same size, combining them to a single array of the same size. As an example the operation `map «+; 3»([1,0,2], [3,6,-1])` applies the $+$ operation index-wise on the two arrays in the parameters. It yields the result `[4, 6, 1]`. If the operation for the `map` has more than one output, the result is a list of all array outputs whereas in each array one output from the node is saved.

In SCADE this operation is visualized by a hierarchical component that contains the operation component within. In Figure 7.2a the visualization of the component is shown. The number of inputs and outputs for the map component depends on the number of inputs and outputs of the included operation.

The fold operation allows to reduce an array sequentially to a scalar using a node as operation. The prerequisite is that the node must have a single output, a first input of the same type and at least another input. It is called `fold` in SCADE and `red` in Lustre. As an example, the operation `red «+; 3»(0, [1,2,3])` adds the initial value `0` with the first value of the array and this result serves as the new first argument when adding the second array element until all elements are included. Thus, the result for this example is `6`.

In SCADE this operation is also visualized by a hierarchical component. In Figure 7.2b this component is shown. The number of inputs depends on the included operation but there is always one single output.

7.2.2 Transformation SCCharts to Lustre

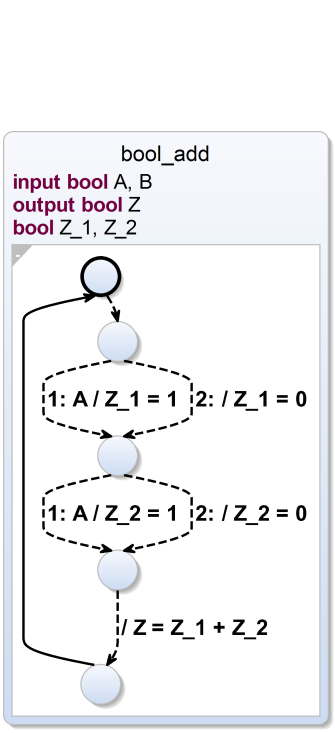
This thesis introduces a transformation from Lustre to SCCharts. However, also a transformation from SCChart to Lustre is reasonable. This would allow to compare the models initially and after a roundtrip to SCChart and back. Moreover, it might be possible to adapt sequentially constructive ideas to the Lustre language in order to design a sequentially constructive version of Lustre.

7.2.3 Improve Lustre to SCCharts Controlflow

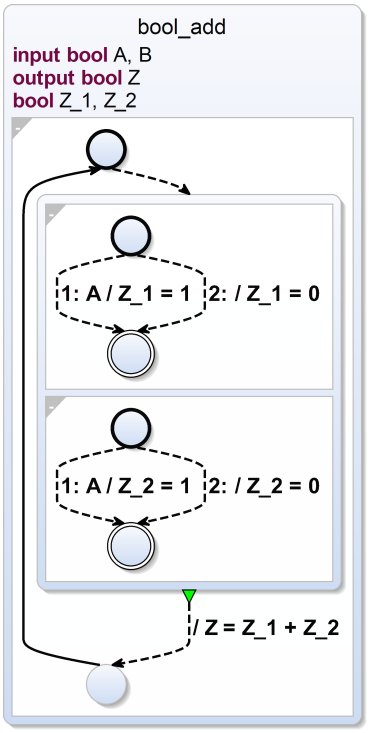
The transformation from Lustre to SCCharts using controlflow was re-implemented after the modifications of the Lustre grammar. The general concept is to hierarchically construct expressions in controlflow. However, also other approaches might be reasonable. The level of hierarchy could be given as option so also fully flat SCCharts or highly hierarchical SCCharts are possible.

Pascutto already proposed these three different visualization strategies. In Figure 7.3 all three approaches are shown for a program that includes only the equation `Z = (if A then 1 else 0) + (if B then 1 else 0)`. The current transformation implements the partially hierarchical strategy but all strategies could be implemented and chosen through options for the transformation in the future.

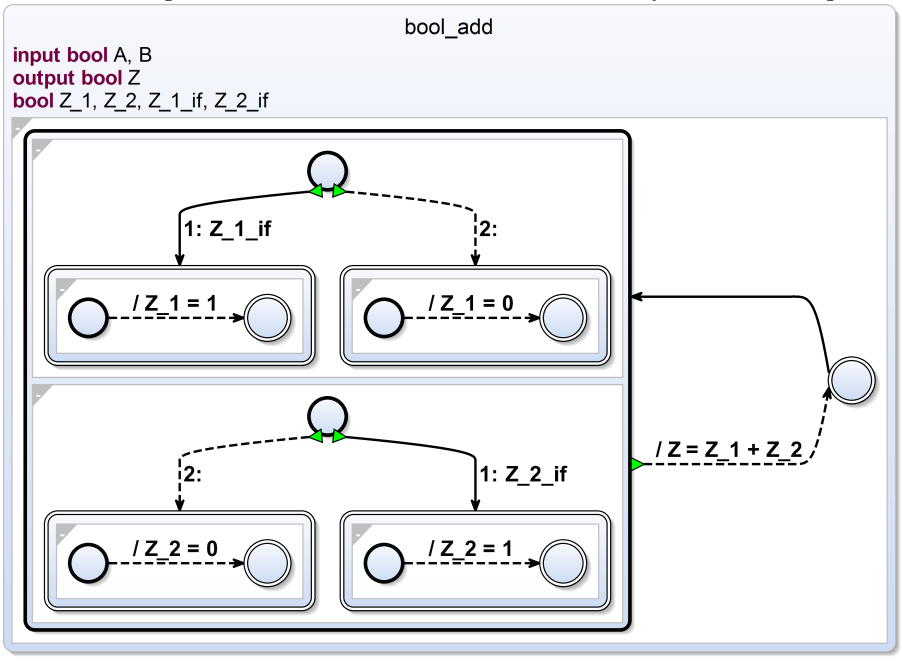
All the approaches have advantages and disadvantages. The flat or the partially hierarchical approach can be used as an alternative visualization of the Lustre code. The mixing of the programming paradigms dataflow and controlflow could lead to a different understanding of the code than the textual Lustre code. Moreover, ideas for improving this approach could also be relevant for improving



(a) Flat expressions

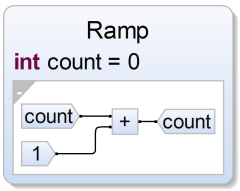


(b) Partially hierarchical expressions



(c) Highly hierarchical expressions

Figure 7.3. Different strategies to handle expressions for the Lustre to SCCharts controlflow transformation proposed by Pascutto



(a) Visual

```
1 scchart Ramp {
2   int count = 0
3
4   dataflow:
5   count = count + 1
6 }
```

(b) Textual

Figure 7.4. A counter in SCCharts using sequentially constructive properties and the new visualization proposed by Smyth

the SCCharts dataflow to controlflow transformation. This transformation is used during the compilation process of SCCharts programs so the controlflow compilation chain is reused instead of creating a new dataflow compilation chain. The general idea is to prove that dataflow and controlflow is essentially the same.

Moreover, the highly hierarchical strategy could be interesting for allowing a concurrent evaluation of expression. This could lead to a better execution time of certain programs. However, it may also introduce an overhead due to the concurrency so the potential benefit needs to be evaluated.

7.2.4 Sequential Variable Access Visualization

In Section 4.3 the challenges and ideas for the dataflow synthesis with sequentially constructive programs are outlined. There are also ideas for the visualization that might improve the dataflow in the diagrams. However, there is yet no clear answer for how this should visualization look like. Due to the concurrent modification of variables and possible sequential reads and writes, visualizing the modification order of variables is not trivial. A dependency analysis would provide the most options for visualizing the variable usage but it might overload the dataflow synthesis.

Smyth already works on an approach that creates different instances of the variables depending on their position in the operation. Equations containing a sequential read with a write such as `count = count + 1` create two labels for the `count` variable. One label is the one that is read for the addition and the other label is the target for the addition, thus it is written to. In Figure 7.4 is an example for how this instantiation looks like. Since the `count` variable is read and written to within one equation, there are two labels for this variable. However, currently this only works for sequential access. The visualization of concurrent modification of variables in dataflow is not yet covered.

These improvements for the visualization are interesting and important for the illustration of the sequentially constructive behavior of SCCharts. Variables can be modified multiple times within a tick and the classical visualization of dataflow in tools like SCADE or LabView imply only one value for a wire in a tick. It follows that the sequential constructiveness implies cycles on the visual level. Without knowledge about the sequential constructiveness, those cycles look like instantaneous loops. This could give the user a wrong intuition about the model and cause confusion.

7.2.5 Optimize Usage of Pre

In Section 4.2 the different transformation strategies for the different Lustre features are introduced. The *when* operation is transformed using a conditional whereas the else-branch holds the *pre* value of the output. This *pre* is introduced in order to preserve the program class during transformation.

However, the behavior would be the same if this *pre* is omitted because in SCCharts there are variables instead of streams. Those variables already keep their value beyond tick boundaries.

Situations like the one just described occur in various models. Often the usage of a pre is not necessary and due to the transformation a lot of overhead such as other parallel regions and new variables are introduced unnecessarily. The *pre* expressions could be optimized through a new processor. In situations when the *pre* is not relevant for breaking instantaneous cycles or for referencing the actual *pre* value of a variable they can be optimized and removed for improving the compilation process. Variables and concurrent regions introduced in the transformation of the *pre* are not needed and the transformed model remains simpler.

Chapter 8

Acronyms

EMF	Eclipse Modeling Framework
HMI	Human Machine Interface
IDE	Integrated Development Environment
KiCo	KIELER Compiler
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
MoC	Model of Computation
RCP	Rich Client Platform
ODE	Ordinary Differential Equations
OSGi	Open Service Gateway initiative
SC MoC	Sequentially Constructive Model of Computation
SSM	Safe State Machine
SCADE	Safety Critical Application Development Environment
SCChart	Sequentially Constructive Chart
SCEst	Sequentially Constructive Esterel
SCL	Sequentially Constructive Language
SSA	Static Single Assignment
UML	Unified Modeling Language

Bibliography

[And96] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Apr. 1996.

[BB91] Albert Benveniste and Gérard Berry. "The synchronous approach to reactive and real-time systems". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1270–1282.

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "A formally verified compiler for lustre". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 586–601. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062358. URL: http://doi.acm.org/10.1145/3062341.3062358.

[BC84] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. "The Synchronous Languages Twelve Years Later". In: *Proc. IEEE, Special Issue on Embedded Systems*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[BCH+08] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. "Clock-directed modular code generation for synchronous data-flow languages". In: *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '08. Tucson, AZ, USA: ACM, 2008, pp. 121–130. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375674. URL: http://doi.acm.org/10.1145/1375657.1375674.

[BD91] Frédéric Boussinot and Robert De Simone. "The Esterel language". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1293–1304.

[Ber02] Gérard Berry. *The constructive semantics of pure Esterel*. Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France: Draft Book, Version 3.0, Dec. 2002.

[BP13] Timothy Bourke and Marc Pouzet. "Zélus: a synchronous language with ODEs". In: *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*. HSCC '13. Philadelphia, Pennsylvania, USA: ACM, 2013, pp. 113–118. ISBN: 978-1-4503-1567-8. DOI: 10.1145/2461328.2461348. URL: http://doi.acm.org/10.1145/2461328.2461348.

[BS86] Gerard Berry and Ravi Sethi. "From regular expressions to deterministic automata". In: *Theoretical Computer Science* 48 (1986), pp. 117–126. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90088-5. URL: http://www.sciencedirect.com/science/article/pii/0304397586900885.

[CP99] Paul Caspi and Marc Pouzet. "Lucid Synchrone: une extension fonctionnelle de Lustre". In: *Journées Francophones des Langages Applicatifs (JFLA)*. Avoriaz, France: INRIA, Feb. 1999. URL: https://hal.archives-ouvertes.fr/hal-01574464.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "A conservative extension of synchronous data-flow with State Machines". In: *ACM International Conference on Embedded Software (EMSOFT'05)*. (Jersey City, NJ, USA). Jersey City, NJ, USA: ACM, Sept. 2005, pp. 173–182.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "SCADE 6: A formal language for embedded critical software development (invited paper)". In: *11th International Symposium on Theoretical Aspects of Software Engineering TASE*. Sophia Antipolis, France, Sept. 2017, pp. 1–11.

[Dom18] Sören Domrös. "Moving Model Driven Engineering from Eclipse to Web Technologies". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Nov. 2018.

[Dor08] Francois-Xavier Dormoy. "Scade 6: a model based solution for safety critical software development". In: *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*. 2008, pp. 1–9.

[Est16] Esterel Technologies, Inc. *SCADE Suite: Control and Logic Application Development*. http://www.esterel-technologies.com/products/scade-suite/. last visited 03/2016.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming graphical modeling". In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*. Vol. 6394. LNCS. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[GGB+91] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. "Programming real time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1321–1336.

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

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

[HBG17] Reinhard von Hanxleden, Timothy Bourke, and Alain Girault. "Real-time ticks for synchronous programming". In: *Proc. Forum on Specification and Design Languages (FDL '17)*. Verona, Italy, Sept. 2017.

[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*. Presentation at Synchronous Programming (SYNCHRON '13), Schloss Dagstuhl, Germany. Nov. 2013.

[HDM+14] 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.

[JRH16] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. "The Lustre V6 reference manual". In: *Verimag, Grenoble, Dec* (2016).

[LGL+91] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. "Programming real-time applications with Signal". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1321–1336.

[MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. "Programming deterministice reactive systems with Synchronous Java (invited paper)". In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Paderborn, Germany, 17/18 6 2013.

[Pas17] Clément Pascutto. *Mixing programming paradigms in synchronous languages – a compilation from lustre to sccharts*. Internship Report. July 2017.

[PTH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. "Synthesizing Safe State Machines from Esterel". In: *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '06)*. Ottawa, Canada, June 2006.

[Ray08] Pascal Raymond. "Synchronous program verification with Lustre/Lesar". In: *Modeling and Verification of Real-Time Systems* (2008), p. 7.

[Ren18] Niklas Rentz. "Moving Transient Views from Eclipse to Web Technologies". `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf`. Master thesis. Kiel University, Department of Computer Science, Nov. 2018.

[Sch16] Alexander Schulz-Rosengarten. "Strict sequential constructiveness". `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf`. Master thesis. Kiel University, Department of Computer Science, Sept. 2016.

[SLH16] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. "Model extraction for legacy C programs with SCCharts". In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '16), Doctoral Symposium*. Vol. 74. Electronic Communications of the EASST. With accompanying `poster`. Corfu, Greece, Oct. 2016.

[SMR+17] Steven Smyth, Christian Motika, Karsten Rathlev, Reinhard von Hanxleden, and Michael Mendler. "SCEst: Sequentially Constructive Esterel". In: *ACM Transactions on Embedded Computing Systems (TECS)—Special Issue on MEMOCODE 2015* 17.2 (Dec. 2017), 33:1–33:26. ISSN: 1539-9087.

[SSH12] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Transient view generation in Eclipse". In: *Proceedings of the First Workshop on Academics Modeling with Eclipse*. Kgs. Lyngby, Denmark, July 2012.

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

[Sta19] Andreas Stange. "Model checking SCCharts". Master thesis. Kiel University, Department of Computer Science, May 2019.