From Lustre to Graphical Models and SCCharts

Lena Grimm, Steven Smyth,
Alexander Schulz-Rosengarten, Reinhard von Hanxleden
Department of Computer Science
Kiel University
Kiel, Germany
{lgr, ssm, als, rvh}@informatik.uni-kiel.de

Marc Pouzet
Département d'informatique
École normale supérieure
Paris, France
marc.pouzet@ens.fr

***Abstract*—We introduce a systematic approach for automatically creating a visual diagram, akin to a SCADE model, from a Lustre program. This not only saves tedious manual drawing effort but also allows the creation of different views for the same program. Furthermore, we present an extension of the SCCharts language with data-flow constructs that adhere to the Lustre semantics, which in turn permits a translation from Lustre to graphical SCCharts. This allows to use the SCCharts simulation and code synthesis machinery as an alternative to existing Lustre compilation techniques. Finally, we investigate how the sequentially constructive model of computation underlying SCCharts can be used to conservatively extend Lustre, thus providing a deterministic semantics to some Lustre programs that would be rejected under its original semantics. We have implemented and validated this work with the Eclipse-based open-source KIELER framework.**

***Index Terms*—Synchronous Languages, Modeling Pragmatics, SCCharts, Lustre, View-Synthesis**

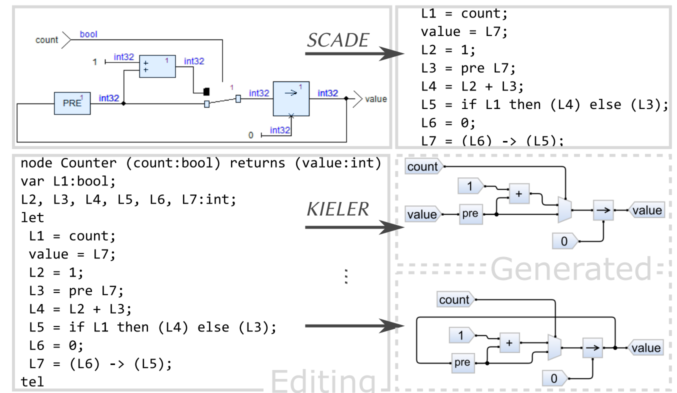


Fig. 1: Traditional modeling flow (top) vs. modeling pragmatics (this work, bottom)

I. Introduction

Traditionally, languages tend to fall into either the *textual* category, as is the case for most programming languages (C, Java, Python, ...), or the *graphical* category, as is the case for most modeling languages (UML, Simulink, Labview, ...). The synchronous data-flow language Lustre illustrates that a textual and a graphical syntax for a single language could both be reasonable; Lustre is originally defined with a textual syntax [1] but is also semantically closely related to the visual language employed by the Safety Critical Application Development Environment (SCADE) [2]. Similarly, the synchronous state-oriented language Sequentially Constructive Charts (SCCharts) [3] is primarily considered a graphical language, using a Statechart syntax, but since its beginning it has been accompanied by a textual syntax as well.

Thus there is not necessarily a clear line between programming and modeling, but this is a debate that we do not investigate further here [4]. Instead, we here explore, for the specific case of Lustre-like languages (elaborated further in Sec. II), how we can make the best use of both textual and graphical representations when designing (complex) systems. We do so irrespective of the question of whether we consider this activity as programming or modeling and we will use both terms in the following.

978-1-7281-8928-4/20/$31.00 ©2020 IEEE

Compared to the process of typing in a textual program, possibly using a modern IDE with code completion etc., the creation of graphical models can be a rather time-consuming affair. Typically, graphical elements are dragged from a palette to a canvas, and the user can spend significant time on positioning elements, possibly moving other elements around to make room first. As models evolve, readability tends to suffer, unless users keep investing significant time just to keep things readable [5]. As a case in point, some SCADE users do not use the graphical editor for modeling, but instead use the textual Scade language that is behind the graphical interface of SCADE. However, for such textually written models, no graphical representation is available; SCADE translates graphical models into Scade, but there is no translation the other way. For some users and use cases that might be irrelevant; for others, however, this might be a loss. In particular for communication and documentation purposes, graphical models are appealing and useful. Arguably, this is why graphical languages have arisen in the first place, after textual languages have already been around for a long time.

In this paper, we investigate how to go the other way, from textual synchronous data-flow to a graphical representation. In that we broadly follow the *modeling pragmatics* approach proposed by Fuhrmann et al. [6]. The main idea there is to separate a (textual) model from a (graphical) view of that model, and to automatically synthesize the view from

the model. In fact, once such an automatic generation of a graphical view is possible, it is just a small step to synthesize different graphical views for one and the same model. Compared to the traditional way of graphical modeling, the modeling pragmatics approach not only has the advantages of efficient textual editing, such as powerful IDE operations, revision control, etc., but one also is liberated from having to work with just one fixed, hand-crafted graphical representation. Fig. 1 illustrates this approach, synthesizing two views with different strategies for handling delay references, as detailed later.

Thus, put simply, the original question that motivated the work presented here was how to get from textual synchronous data-/control-flow ("Lustre") to graphical representations ("SCADE"). While the basic concept to do so appears fairly straightforward, there are various options to do so, as illustrated in this paper. However, in the course of answering this original, pragmatics-oriented question, building on the concepts and diagramming infrastructure for the SCCharts language, it turned out that there were interesting semantic questions as well, which this paper will also will report on. Put briefly, **the first semantic question is how to map Lustre to valid, semantically equivalent SCChart models**. I. e., we aimed to not just synthesize possibly somewhat abstract/informal diagrams, such as could be drawn up by a programmer for documentation purposes, to be used along the original code, but we wish to synthesize models that fully capture the textual source. Apart from the practical consideration that a modeling/compilation framework (Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)) that supports SCCharts is openly available, SCCharts are an interesting target in that they originally come from a control/state oriented modeling world, as opposed to Lustre, which has its origins in data-flow. Thus, when exploring a mapping from Lustre to SCCharts, we also touch on the broader question of how data-flow maps into control-flow. We do not aim to provide a general answer here, but, as illustrated in this paper, at least in our setting such a transformation seems possible without significant costs, which we see as an indication that these worlds are not as separate as they sometimes are perceived.

Finally, the choice of SCCharts as a target leads to another question, raised by the underlying models of computation. SCCharts share with Lustre the basic synchronous principle that divides computation into discrete reactions/ticks and has a deterministic, concurrent semantics by abstracting from reaction time. The classic synchronous model of computation, also employed by Lustre, assumes that shared data have a unique value per tick. This simplifies semantic reasoning, and is a prerequisite for example for hardware synthesis. However, especially for programmers used to imperative languages, this requirement may be overly restrictive. This has motivated the *sequentially constructive* model of computation (SC MoC), which is also employed by SCCharts. That MoC takes sequential scheduling information into account and gives a—still deterministic—semantics to programs that would be rejected as "not causal" under the classic synchronous MoC. Thus, when constructing a mapping from Lustre to SCCharts, **the second semantic question is how the SC MoC may extend the class of valid Lustre programs**.

Contributions & Outline

- We contribute a synthesis strategy to generate graphical representations for Lustre-like programs using a block diagram notation as in SCADE (Sec. II).
- We extended SCCharts with a data-flow component that allows seamless modeling of hybrid control-flow and data-flow models. We present a transformation from Lustre to SCCharts that preserves the semantics of a Lustre program (Sec. III).
- We investigate how the Sequentially Constructive (SC) Model of Computation (MoC) conservatively extends the semantics of Lustre and allows to accept a greater class of deterministic programs (Sec. IV).

We evaluate our transformation approach in Sec. V and briefly discuss related work in Sec. VI. In Sec. VII, we conclude and give an outlook on future work.

II. From KIELER Lustre to Graphical Models

Since the original Lustre proposal, a number of language variants have been developed, which sometimes makes it difficult to discern what language one is specifically referring to. In this paper, we study a synchronous language, henceforth referred to as KIELER Lustre (KLustre), named after the modeling environment in which it is now realized, that mixes Lustre-like data-flow and a rich set of control-flow structures like hierarchical automata, and in that is fairly close to Heptagon [7][1]. Specifically, KLustre includes

- the core of Lustre V6 such as pre, ->, fby, general point wise applications, and clock operations such as when and current [8];
- control-flow constructs with automata, borrowed from Scade 6 [2], where transitions can be either weak or strong aborts and they can restart or reset the behavior of the target state;
- in the context of states, the operations pre and last, where pre refers to the previous value of a variable within the environment of the state it is used.

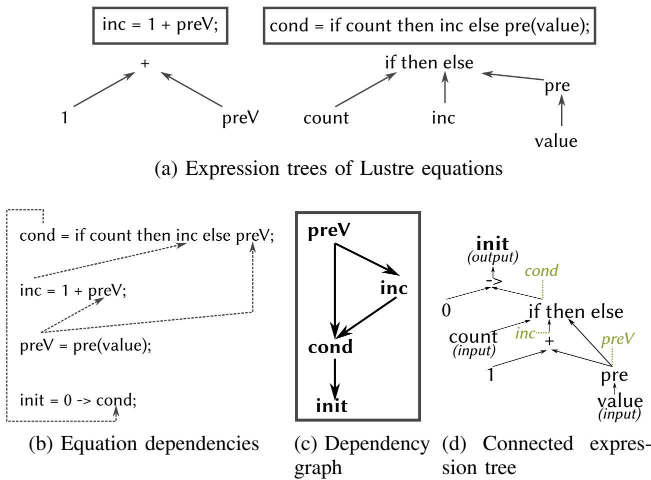
Type declarations, external nodes, static parameters, arrays and higher-order functions like map or red are not supported, but should be relatively straightforward to add, building on existing work. Also, there is no clock calculus required (but it could be added), as detailed later in Sec. III-B.

While we performed the specific work presented in this paper with the KLustre language, we argue that the principles apply to Lustre-like languages in general. Thus, when the distinction is not important, we may still refer to just "Lustre."

A. From Lustre to a Graph Structure

In order to retrieve the graphical model there are three steps that need to be performed.

[1]http://heptagon.gforge.inria.fr/



(a) Expression trees of Lustre equations

(b) Equation dependencies

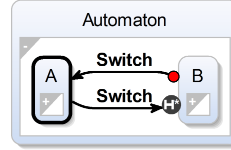
(c) Dependency graph

(d) Connected expression tree

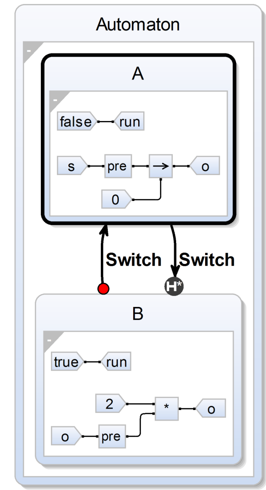
Fig. 2: Graph extraction steps

```
1  automaton Automaton
2  state A
3    let  o = 0 −> last s;
4       run = false;
5    tel
6  until if Switch resume B;
7  state B
8    let o = 2 * last o;
9       run = true;
10   tel
11 unless if Switch restart A;
12 returns ..;
```

(a) KLustre code



(b) Collapsed view



(c) Expanded view

Fig. 3: Example for an automaton in KLustre

1) Step 1, Create expression trees: The retrieval of the Abstract Syntax Tree (AST) of a Lustre program is the entry point for a graphical representation. Starting at this AST, the first step towards a graphical model is performed—the extraction of an expression tree for each equation. Each equation defines the value for one variable, hence an expression tree always corresponds to one variable. Fig. 2a shows examples of the expression trees for the variables cond and inc.

2) Step 2, Construct dependency graph: In Lustre, each variable needs to be written before it is read. An exception is made for variables read within a delayed operator such as pre or fby. They refer to the value of the previous tick and are thus read before they are written [9]. In a valid Lustre program there is always a topological sorting of the equations constrained by variable dependencies, as shown in Fig. 2b. The dotted arrows indicate that the variable at the source is required for the destination of the arrow. Following this dependency information, a general dependency graph for the variables on the left side of the equation can be established. Fig. 2c gives an example. The variable preV is read twice, so there are two dependency edges from the equation defining preV to the other equations. This dependency graph now defines the ordering in which the different expression trees are processed in the next step.

3) Step 3, Connect expression trees: The last step connects the independent expression trees according to the dependencies. The flow of the data in an expression tree is bottom to top, so an edge states the lower node as source and the top node as destination. If a variable occurs non-delayed in one of the leaves of an expression tree, this leaf is replaced by linking to the root of the expression tree this variable is referring to. Processing starts with the variable at the root of the dependency tree, Fig. 2d shows the resulting graph. The variable cond is used in the equation init = 0 -> cond. Therefore, the expression tree defining the value for cond is put in place of the actual reference to cond in the second argument of the -> operator. Inputs are an exception to this rule as they can be read without being written to explicitly. They remain as a leaf.

Since delayed operators do not introduce dependency edges in Step 2, it is possible that a variable is read before a corresponding expression tree for this variable is created. For local variables there can be two different strategies for visualization, with or without cycles, as illustrated in the two generated views in Fig. 1.

B. Visualizing control-flow/states

KLustre includes automata, which can be defined at every place that equations could be defined. Those automata can include states and each state may define equations that are executed when this state is active. Fig. 3 shows an example definition of an automaton and two possible views with collapsed and expanded inner behavior.

These states can also be processed into a graph model. The definition of an automaton already introduces a hierarchical node. The behavior of the automaton is then added within this node. This way the data-flow and the control-flow parts are also visually separated.

C. Automatic Layout

We now defined a graph that holds information about the data-flow and the control-flow of a given KLustre program. A non-trivial question, studied by a whole community of researchers on graph drawing, is how to turn such a graph into a visual representation. Even though the field of graph drawing goes back a long time and some libraries, such as GraphViz, are already widely used, the problem still appears hard enough such that very few visual modeling tools provide automatic layout functionality and instead leave that task entirely to the user. In fact, this may also be the reason why so far there are so few serious efforts to synthesize visual models from textual descriptions. However, as we also hope to illustrate here, today it is indeed practical to automatically construct visual models of the type we want here.

In addition to the usual requirements and *aesthetics criteria* employed in graph drawing——no overlapping nodes, short edges, few crossings, etc.—we typically have a preferred reading direction. In data-flow diagrams, we usually want inputs on the left and outputs on the right. In state diagrams, we prefer initial states on the left and final states on the right. One class of graph drawing algorithms that are guided by such a reading direction is the *layered approach*, pioneered by Sugiyama et al. [10] and provided for example by GraphViz dot. However, data-flow diagrams, unlike state diagrams, typically induce *port constraints* in that nodes (actors) often prescribe where they connect to which input/output data-flow edges. This imposes non-trivial further requirements on the layouting algorithm [11].

D. Implementation in KIELER

We have prototyped the diagram synthesis in the KIELER IDE. A main objective for KIELER was to synthesize views automatically from textual code, as illustrated in Fig. 4a. The graph model synthesized from KLustre is rendered with the Kieler Lightweight Diagrams (KLighD) framework [12]. As illustrated later in Fig. 8c to Fig. 8f, the styling/skinning of the graph can be defined separately. Thus, with KLighD, the resulting diagram may “look and feel” like SCADE, SCCharts, Ptolemy or some other visual language.

The automatic layout in KIELER is realized through the Eclipse Layout Kernel (ELK)[2], which provides a wide range of layout algorithms. All the SCCharts shown here are synthesized automatically within KIELER using ELK and KLighD. For data-flow and state diagrams, we use the ELK Layered algorithm, which also considers port constraints. For placing concurrent regions within a superstate, we use the LR-rectpacking algorithm.

III. From Lustre to SCCharts

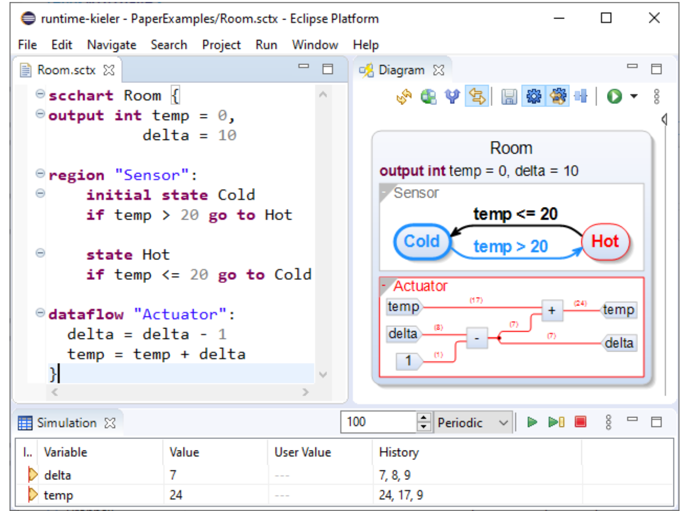
After focussing on the pragmatics of how to visualize Lustre programs in the previous section, we now consider the first semantic question raised in the introduction, namely, how to synthesize valid, semantically equivalent SCCharts from Lustre. As one motivation, such a translation allows not only to visualize a textual Lustre program, but it can also be compiled into code and be simulated, as shown in Fig. 4a.

A. Adding data-flow to SCCharts

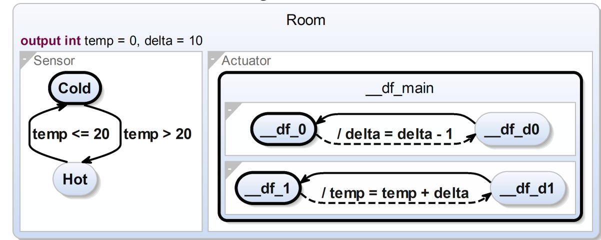
Data-flow in Lustre is defined in a declarative style as a set of concurrent equations that are evaluated each tick, each defining a *stream*. In principle, the original SCCharts [13] are already expressive enough to capture this. However, the SCCharts language was originally focused on control-flow. Thus, to close the conceptual gap to Lustre, we extended SCCharts with (unclocked) data-flow.

In the original SCCharts, concurrency is achieved through superstates that contain an arbitrary number of *regions* that are concurrently active until they reach a final state or the enclosing superstate is left. We incorporate data-flow into

[2]https://www.eclipse.org/elk



(a) Screenshot of KIELER, while simulating an SCChart that contains control-flow and data-flow regions. The user edits the text on the left, the graphics on the right is synthesized automatically whenever the text is saved. During simulation, entered states are highlighted in red, left states and taken transitions are shown in blue. Data-flow edges are annotated with their current values.



(b) The SCChart after transforming away the data-flow region.

Fig. 4: A temperature model with data-flow and states

SCCharts by providing *data-flow regions*, which can co-exist with the usual (control-flow) regions. Each data-flow region contains a set of data-flow equations, which are evaluated concurrently, just as in Lustre. Data-flow regions are an extended SCChart feature, meaning that they are transformed away through a model-to-model transformation that creates an SCChart without data-flow regions. The transformation generates one control-flow region per data-flow equation, as illustrated in Fig. 4b and actually as in the transformation of `during` actions. Each region has an initial state (bold outline), which is left immediately (indicated by the dashed transition) and unconditionally with a transition that, as its effect, evaluates the corresponding data-flow equation and that leads to another state where control rests until the end of the tick. In subsequent ticks, that other state is left through the non-immediate (solid) transition that leads back to the original state, from where immediately the next transition/equation evaluation occurs.

B. Streams vs. variables

Our translation from Lustre to SCCharts basically maps streams to variables. Note that such a mapping is also part of a traditional Lustre compiler that translates Lustre to, e. g.,

C code. We do this mapping at a somewhat higher level, where the result becomes visible at the SCChart modeling level.

Streams in Lustre are clocked, meaning that they carry a value in a tick iff a corresponding clock is true in that tick. Thus, clocks are also streams, of boolean type. Per default, streams are clocked by the base clock, which is always true. If we refer to stream x, we mean x as defined in the current tick, and the compiler must guarantee that x carries a value whenever it is referenced. More generally, Lustre programs must be clock-consistent, typically meaning that streams can be combined using data operators only if they operate on the same clock.

Variables in SCCharts behave as in, e. g., C or Java, in that they basically refer to memory cells. This implies that they always carry a value, and that they persist from one tick to the next. Thus, if we refer to some variable x, we retrieve the value that was written to x last, which may be in the current tick or in some previous tick. Thus, unlike with streams, there is no clock consistency requirement when operating on variables.

C. Clock operators

As Lustre operates on clocked streams, it provides several functions that operate on clocks. We here discuss two of these operators, when and current.

1) Downsampling — when: The Lustre assignment x = e when c only evaluates e when c is true. Moreover, x is only defined in those ticks where c holds true. So clocking introduces two main ideas: absence of values and conditioned evaluation of expressions. This concept of clocks is also used to gain a mechanism for expressing control in data-flow. However, there is no presence or absence check of a value in Lustre, other than the clocking itself. During code generation of Lustre, those clocks are translated to conditionals [1], [9].

In SCCharts the clocking can be emulated within a conditioned evaluation for the corresponding expression. Only in those instances when the clock (implemented as a boolean variable) is true, the value for x needs to be computed. We express this with a simplified variant of the ternary operator, without an explicit else branch—the assignment x = e when c is translated to x = c? e, which in turn gets translated into if (c) x = e. This directly reflects the Lustre code generation [9].

2) Upsampling — current: If the clock of a stream x is false, current x stands for the last known value of x. The variables of SCCharts naturally support this through their persistence across ticks. Thus, each occurrence of current x in KLustre is simply replaced by x in SCCharts.

3) Clocks and Registers: Each operator individually now realizes the same behavior as in Lustre. Especially the clocking, however, has an effect on the behavior of other operations. Combining when with the pre operation behaves differently in SCCharts than originally in Lustre. To account for the clocked nature of streams, the plain SCCharts pre operator is not sufficient. Instead, a *clocked* pre *operator* is needed that we added as extended feature to SCCharts.

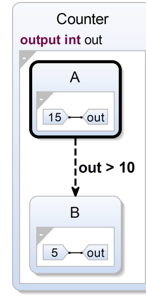


Fig. 5: Sequential execution of states

```
1  x += y;
2  x = 1;
3  x += (y > 2) ? 3;
4  z = x + 4;
5  y = 5;
```

Fig. 6: KLustre example with concurrent data-flow equations, including multiple writes to x

```
1  x = 1;
2  y = 5;
3  x += y;
4  if (y > 2) x += 3;
5  z = x + 4;
```

Fig. 7: Sequential C code resulting from Fig. 6, illustrating the IUR protocol

IV. Sequentially Constructive Lustre

As explained in the introduction, the translation of Lustre to SCCharts also raises the question of how the SC MoC that underlies SCCharts may open the door towards accepting more Lustre programs than under the standard assumption of just one value per variable (stream) per tick.

A. Sequentiality in data-flow

In original Lustre, every equation holds globally (w. r. t. clocks) and defines the current value of a variable. Consequently, recursive definitions that rely on the current value of variables to define its current value are rejected. With the notion of sequentiality in the SC MoC, this classical view of a globally consistent value is relaxed. For example, assignments that read the current value of the variable they assign are not rejected but simply overwrite the value in the sense of imperative programming. This is based on the argument that in an assignment x = e, there is a clear sequential ordering between evaluating expression e and assigning the value to x. There is no concurrency involved that may give rise to race conditions; thus determinism, which is the overall goal of the synchronous MoC, is not compromised. The following example of a simple counter illustrates this notion of sequentiality. The equation x = 0 -> x + 1 in SCCharts initializes x with zero. In subsequent ticks it reads the value of x that is currently held in memory, and sequentially afterwards, increments the value by one. This behavior can be realized without explicitly using the pre operator.

This convenience based on sequentiality also comes at a price. The substitution principle that applies for original Lustre does not apply in the SC universe. Splitting x = 0 -> x + 1 into x = 0 -> y and y = x + 1 would not be schedulable anymore because both equations depend on each other. The single-equation version inherently implies a read-write ordering, and for the two-equation version this information is lost. Thus, it may be a matter of taste whether one actually wants to program in a style that takes advantage of sequentiality or not.

B. Sequentiality in control-flow

In Heptagon/Scade 6, if there is a transition from some state A to some other state B, then A and B cannot be active in the same tick. If the transition is taken, then either A is still

executed (weak abort) and B is not entered yet (deferred entry, indicated with a blue circle), or A is not executed anymore (strong abort, indicated with a red circle) but B is entered (non-deferred entry). This design choice eliminates write-write conflicts in case A and B write to the same variable. However, under the SC MoC, we consider A to be sequentially ordered before B, thus giving a clear, deterministic semantics even if both A and B are executed in the same tick. Furthermore, the SC MoC considers strong/weak aborts to be sequentially ordered before/after the state they abort. Thus, for example, it is legal for the trigger of a strong abort to depend on a variable that is written (in case the abort does not take place) in the potentially aborted state. For example, in Fig. 5, out is first set to 15 in state A, then A is weakly aborted since out > 10 holds, and finally out is set to 5 in B.

C. Concurrency

As explained in Sec. III-A, we map Lustre data-flow equations to concurrent SCChart regions that each handle one equation. Unlike Lustre, the SC MoC allows multiple concurrent definitions for the same variable, as long as they can be scheduled according to the Initialize-Update-Read Protocol (IURP). Briefly, there may be one *initialization* (*absolute write*), followed by an arbitrary number of confluent *updates* (*relative writes*), followed by an arbitrary number of reads. An update of a variable x is a write that depends on the current value of x via some *combination function*, e. g., addition in x += e, where the value of e must not depend on x. Updates are *confluent* if they use the same combination function. An initialization is a write that is not an update. This is akin to the combination functions of Esterel, except that we allow to combine them with initializations.

As example, consider variable x used in lines 1–4 in the KLustre code in Fig. 6. The IURP applied to x requires that the initialization of x in line 2 must be scheduled first, followed by the updates in lines 1 and 3 in any order, followed by the read in line 4. Similarly, the write of y (line 5) must be scheduled before the reads (lines 1 and 3). The resulting dependency graph is acyclic, thus the KLustre program is schedulable and valid. A possible resulting C code is shown in Fig. 7.

V. ASSESSMENT

The generation of code from Lustre-like languages is already a well-studied subject, with numerous compilers available including the qualified code generator of SCADE, and it was not a primary objective of this work to add yet another code synthesis approach. However, since we now have a path from KLustre to SCCharts, for which there are also several code generators available, it is a natural question to ask how the end results compare. To that end, we now examine the whole synthesis chain from a Lustre example through SCCharts to C code.

Fig. 8a shows the Lustre code of the counting node, originally presented by Colaco et al. [14]. The corresponding SCADE diagram is depicted in Fig. 8b. The KIELER IDE allows to edit the Lustre source code with common editing features, such as highlighting and code completion. While editing in the textual editor, a diagram as the one shown in Fig. 8c is (re-)synthesized automatically whenever the text file is saved. Similar to SCADE, both equations are depicted in the graphical representation. However, the diagram synthesis can be configured interactively to display several variations to fit the modelers needs. For example, the whole diagram can be *skinned* to get a SCADE look-and-feel, see Fig. 8d. The disconnected components can also be ordered sequentially, as in Fig. 8e. The sequential ordering is indicated by the red dashed hyperedge, because v, written in the first equation (the first line of the Lustre node), is read in the second one, even though their order is reversed in the textual representation of the node. Since v is not visible from outside the node, its graphical input/output representation can be omitted altogether, corresponding to the Lustre substitution principle, see Fig. 8f.

The KIELER Compiler (KiCo) included in KIELER is configured as depicted in Fig. 8i for the evaluation. First, the Lustre program is compiled into an SCCharts model. Possible graphical representation of the generated SCChart are shown in Fig. 8c to Fig. 8f. The data-flow representation of SCCharts is translated into its semantically equivalent control-flow oriented counter-part, shown in Fig. 8g. From there, we use the netlist-based compilation approach [13].

The compilation approach generates a netlist with guarded basic blocks from the control-flow graph of the program. The overhead, which might be introduced by the control-flow-based compilation approach, can be reduced in stateless models. After standard optimization techniques the graph is sequentialized. From here, guards that guard states that are evaluated in every tick form *persistent state* patterns, which always evaluate to true: The guard is true in the first tick (_GO) or if it was true in the previous tick. The final optimized version is depicted in Fig. 8h.

The C code of the counting node example generated by KiCo is listed in Fig. 8j. Code from the immediate environment, such as the reset function and the _GO signal which signals a program start, are omitted here. Hence, the generated logic function directly resembles the data-flow of the node. This example demonstrates, using one example, that the generated code can be still compact and readable even if SCCharts data-flow equations are translated into a statechart, then into a control-flow graph and finally into imperative code.

Note that saving the previous value of o is embedded in the code, because pre is an extended feature of SCCharts and not part of the underlying expression language. The register retrieval and save can be observed in Lines 3 and 8 of Fig. 8j. However, since sequential constructiveness inherently stores the immediate previous value, the modeler can omit the pre

[3]During this work, we noticed that the SCADE diagram from the original paper [14] was inconsistent with the Lustre code in that the addition operator was missing. Graphical inconsistencies such as these are another argument for synthesizing diagrams automatically. Additionally, in the original counting node [14] the type of o is bool, which is inconsistent with its assigned value. Here, we use a fixed version.

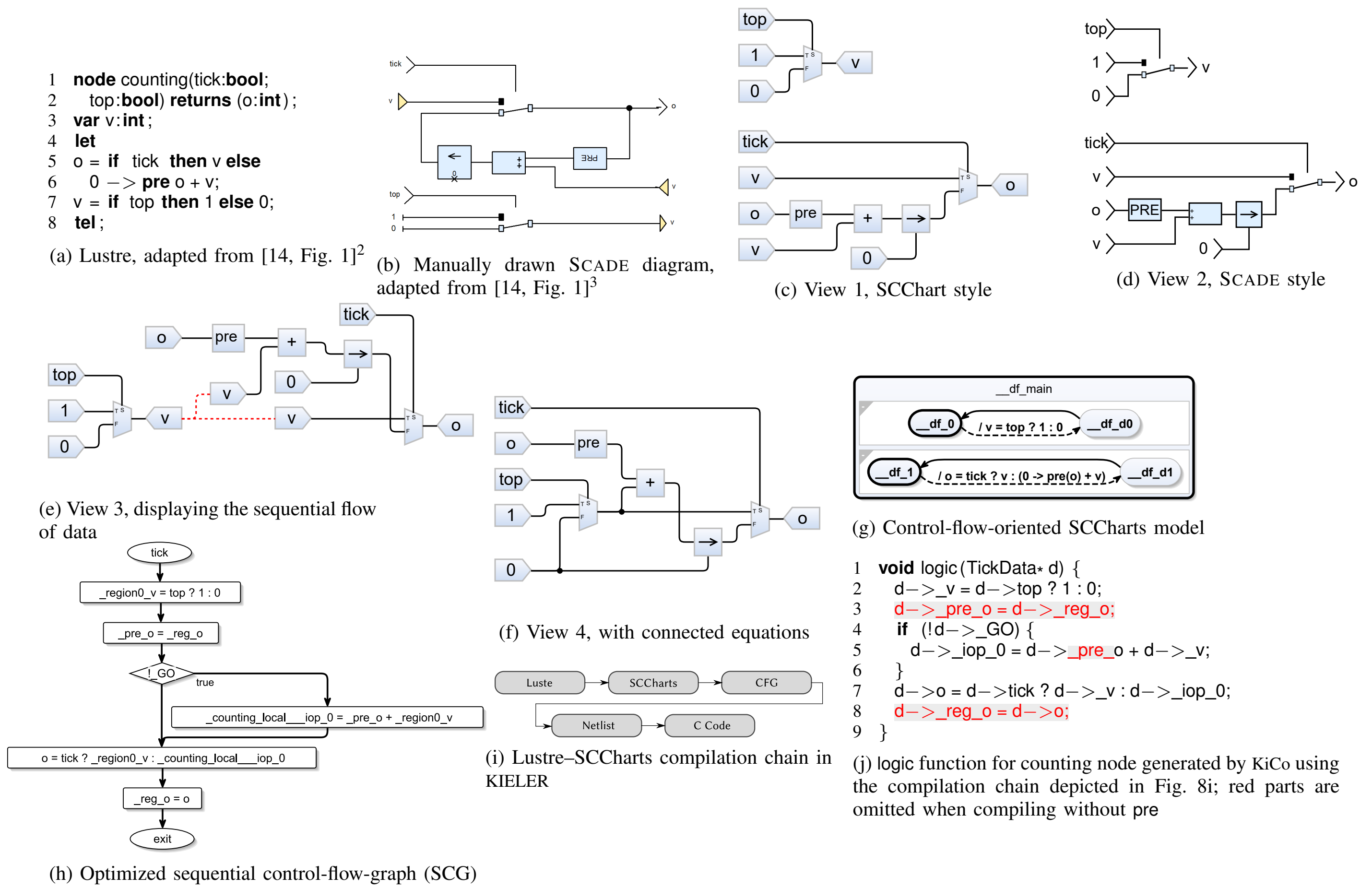


Fig. 8: From Lustre via SCCharts to C code, with several automatically generated views

operator altogether. In this case, the marked parts in the listing will not be generated.

Apart from the counting example, we included functional behavior tests for about 50 other programs. Moreover, we compared the computation times resulting from the SCCharts netlist compilation and the Lustre V6 compiler. They indicate that the translation from Lustre to SCCharts does not impose a significant performance penalty. However, in order to draw a more solid conclusion, we feel that more benchmarks with larger programs/models would be needed. In addition, the Lustre V6 compiler does not support the state extension. Therefore, compiler like the Heptagon[4] or KCG [2] should also be included.

VI. Related Work

This paper introduced an approach for enhancing the modeling experience of Lustre programs with visual diagrams. Graphical data-flow is used in various domains and tools. Simulink[5], Labview[6] and Ptolemy[7] also use a notion of block diagram for their diagrams. Besides their different underlying MoC, they all represent actor-oriented data-flow. Each uses a graphical editing approach, and the user can modify the diagram via drag and drop of the components. Ptolemy offers the possibility of an automatic layout, also harnessing ELK. However, the initial model requires manual graphical editing.

The approach of automatic diagram generation for programs with a textual syntax has been investigated for other languages before. Prochnow et al. introduced an automatic Esterel to Safe State Machine Transformation [15]. Moreover, they elaborated on the differences between visual and textual editing in practical use. Especially in terms of comprehensiveness and editing speed each has advantages and disadvantages. The motivation was similar to transforming Lustre to SCCharts—the benefits of both alternatives are combined.

The basic idea of automatically generated diagrams follows the principle of Model-driven Visualization (MDV) introduced by Bull et al. [16]. This automatic generation allows for multiple views on the same model. Schneider et al. elaborated more on this by introducing a meta-model and an infrastructure for automatic layout. The developed infrastructure is the KLighD framework that we use for the graphical view generation in the Lustre context.

The synchronous language Esterel was also extended in

[4]http://heptagon.gforge.inria.fr/
[5]https://de.mathworks.com/products/simulink.html
[6]http://www.ni.com/getting-started/labview-basics/
[7]http://ptolemy.berkeley.edu/ptolemyII/

a conservative manner. A sequentially constructive approach was introduced by Rathlev et al. creating the language SCEst [17]. SCEst also overcomes some restrictions implied by the classical synchronous MoC without losing determinacy.

VII. Conclusion and Future Work

In this paper, we presented, to our knowledge for the first time, a generic approach for synthesizing graphical diagrams from a variant of textual Lustre. The original goal was driven by modeling pragmatics, based on the observation that graphical languages, including SCADE, have their merits for visualization and documentation, however, their practical usage can be rather tedious. Thus, by automating the process of generating customizable graphical views from textual Lustre, made possible by state-of-the-art layouting algorithms, we believe to have made a step towards having the best of both the textual and the graphical worlds.

Beyond that original goal, we also addressed two questions that concern semantics, irrespective of textual/graphical syntax. First, we made a link from Lustre to SCCharts, basically by extending SCCharts with data-flow and by mapping clocked streams to variables. Second, we investigated how the SC MoC that underlies SCCharts gives meaning to Lustre programs that classically would be rejected, by harnessing sequential scheduling information.

We have implemented these concepts in the open source KIELER tool, for a specific variant of Lustre, termed KLustre. With that, a programmer may now write Lustre code as usual, alongside an automatically synthesized visualization that is updated on the fly (and without significant delay) whenever the code is saved. That visualization is in fact a valid SCChart model, meaning that it can be simulated and compiled into software and hardware. Preliminary experiments indicate that the code synthesized from Lustre via SCCharts is of reasonable quality, but further evaluations of that would be needed.

Moreover, there are different programming languages that are variants of or inspired by Lustre. We chose to build our own subset of supported Lustre features, but there are also other Lustre-alike languages that could be used. Support for Scade, the internal language used in SCADE, would allow for direct diagram comparison and maybe automatic diagram generation in SCADE [14]. The language supported by the Heptagon compiler could also offer a nice entry for evaluation results. Other than the Lustre V6 compiler, it also supports automata. It was already considered to create efficient code [7], thus it is especially interesting for comparison with the SCCharts code generation.

Finally, we feel that the relationship between the classical synchronous MoC and the SC MoC in the context of Lustre-like languages would merit further investigation. This should include a formal treatment, building for example on the work by Aguado et al. [18].

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sept. 1991.

[2] J. Colaço, B. Pagano, and M. 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, Sep. 2017, pp. 1–11.

[3] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Edinburgh, UK: ACM, Jun. 2014, long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274.

[4] M. Broy, K. Havelund, R. Kumar, and B. Steffen, "Towards a unified view of modeling and programming (ISoLA 2018 track introduction)," in *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 3–21.

[5] M. Petre, "Why looking isn't always seeing: Readership skills and graphical programming," *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, Jun. 1995.

[6] H. Fuhrmann and R. von Hanxleden, "Taming graphical modeling," in *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*, ser. LNCS, vol. 6394. Springer, Oct. 2010, pp. 196–210.

[7] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages: Application to arrays in a Lustre compiler," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12, New York, NY, USA, 2012, p. 51–60.

[8] E. Jahier, P. Raymond, and N. Halbwachs, "The Lustre V6 reference manual," *Verimag, Grenoble, Dec*, 2016.

[9] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed Modular Code Generation of Synchronous Data-flow Languages," in *Proceedings of the ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*. Tucson, AZ, USA: ACM, Jun. 2008, pp. 121–130.

[10] K. Sugiyama and K. Misue, "Visualization of structural information: automatic drawing of compound digraphs," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 4, pp. 876–892, Jul/Aug 1991.

[11] C. D. Schulze, M. Spönemann, and R. von Hanxleden, "Drawing layered graphs with port constraints," *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, vol. 25, no. 2, pp. 89–106, 2014.

[12] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model! – Putting automatic synthesis of node-link-diagrams into practice," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA: IEEE, Sep. 2013, pp. 75–82.

[13] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, Jun. 2014, pp. 372–383.

[14] J.-L. Colaço, B. Pagano, and M. Pouzet, "A conservative extension of synchronous data-flow with State Machines," in *ACM International Conference on Embedded Software (EMSOFT'05)*. New York, NY, USA: ACM, Sep. 2005, pp. 173–182.

[15] S. Prochnow, C. Traulsen, and R. 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, Jun. 2006.

[16] R. I. Bull, M.-A. Storey, M. Litoiu, and J.-M. Favre, "An architecture to support model driven software visualization," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 100–106.

[17] S. Smyth, C. Motika, K. Rathlev, R. von Hanxleden, and M. Mendler, "SCEst: Sequentially Constructive Esterel," *ACM Transactions on Embedded Computing Systems (TECS)—Special Issue on MEMOCODE 2015*, vol. 17, no. 2, pp. 33:1–33:26, Dec. 2017.

[18] J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann, "Denotational fixed-point semantics for constructive scheduling of synchronous concurrency," *Acta Informatica, Special Issue on Combining Compositionality and Concurrency*, vol. 52, no. 4, pp. 393–442, 2015.