Augmenting State Models with Data Flow

Nis Wechselberg (✉), Alexander Schulz-Rosengarten,
Steven Smyth, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University
Olshausenstr. 40, 24118 Kiel, Germany
{nbw|als|ssm|rvh}@informatik.uni-kiel.de

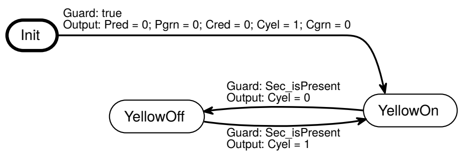
Abstract. Numerous modeling languages have adapted a graphical syntax that emphasizes control flow or state rather than data flow. We here refer to these as *state diagrams*, which include classic control flow diagrams as well as for example Statecharts. State diagrams are usually considered to be fairly easy to comprehend and to facilitate the understanding of the general system behavior. However, finding data dependencies between concurrent activities can be difficult as these dependencies must be deduced by matching textual variable references.
We here investigate how to extract data flow information from state diagrams and how to make that information more accessible to the modeler. A key enabler is automatic layout, which allows to automatically create dynamic, customized views from a given model. To set the stage, we propose a taxonomy of state and data-flow based modeling and viewing approaches. We then compare traditional, static view approaches with dynamic views. We present implementation results based on the open-source Ptolemy and KIELER frameworks and the Eclipse Layout Kernel.

1 Introduction

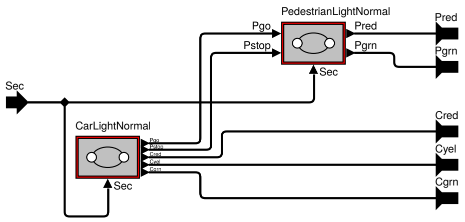
In model-driven engineering (MDE), instead of directly programming a certain behavior, the developer creates a model, specifying the behavior of the system in a more abstract form. The model is then usually used to generate specific code for the target system or to simulate the behavior beforehand. One feature often found in modeling languages is a graphical *diagram* of the model, be it as the primary input like in Scade[1], Simulink[2], LabView[3], or Ptolemy[4] [20] or generated from a textual model like in SCCharts[5] [12].

The graphical diagrams can be grouped in two major styles, *control flow diagrams*, which include *state diagrams*, and *data flow diagrams*. Both of theses styles have their own benefits and drawbacks in practical application.

[1] http://www.esterel-technologies.com/products/scade-suite
[2] https://de.mathworks.com/products/simulink
[3] https://www.ni.com/labview
[4] https://ptolemy.eecs.berkeley.edu/ptolemyII
[5] https://rtsys.informatik.uni-kiel.de/kieler



(a) Sample state diagram: A blinking yellow traffic light



(b) Sample data flow diagram: A traffic light controller

Fig. 1: State and data flow diagrams

State diagrams are usually composed of *states* and *transitions*. States are places the execution of the program rests in, while transitions control the change from one state to another. A basic example of a state diagram can be seen in Fig. 1a. The initial state Init is highlighted by a bold outline. The outgoing transition from Init is guarded by the trigger true, meaning it is always taken. As soon as the transition is taken the Output action comes into effect, in this case setting the Cyel output to 1 and other outputs to 0. The control flow then reaches the state YellowOn. From this point on, the control alternates between the states YellowOn and YellowOff each time the input Sec is present. Every time the state is changed, the output Cyel is turned to either 0 or 1.

Most languages expand this basic form of state diagrams by adding more features such as, for example, concurrent control flows or hierarchical composition of state diagrams. One benefit of state diagrams is that the model is usually close to the natural description of the behavior and to the developer's mental model. However, as noted before [2], some model aspects are rather difficult to infer from state diagrams. For example, if the model employs concurrency and shared data, the exact nature of data sharing is not graphically visualized, but requires the modeler to scan the textual transition labels and look for common variable names.

More insight in the usage of shared data is presented in data flow diagrams. The nodes in a data flow diagram represent *actors* that are all executed concurrently [17]. Two actors are connected if they share data. A simple example is shown in Fig. 1b. The input signal Sec is passed to the CarLightNormal and PedestrianLightNormal actors, which communicate through Pgo and Pstop and produce the outputs Pred, Pgrn and so forth.

Motivation One way to overcome the shortcomings of a specific diagram style (state or data flow) is the combination of the different aspects. This approach is known as *multimodeling*. Here multiple different kinds of diagrams can be combined to form the complete model. This approach has been used in Ptolemy II [20] by embedding *modal models*, which contain some form of extended state machine, into normal data flow models and allowing *refinements* of states in state machines to contain data flow models. Multimodelling is now well understood from a semantic level. However, as we argue here, there is still room for improvement concerning the *pragmatics* of multimodelling [8], that is, how to support the user in applying multimodeling in a productive manner. Specifically, the traditional modeling approach of having the modeler produce the one and only static view of a model, which we here refer to as a *static view*, limits productivity and hampers human model analysis and understanding.

Contributions / Outline We first propose a taxonomy of modeling and viewing alternatives, see Sec. 2. In Sec. 3, we survey traditional approaches based on static views in more detail. The main contribution follows in Sec. 4, where we present how to automatically derive hybrid state/data *dynamic views* from state models. We also investigate how dynamic views can help model analysis, in particular concerning schedulability and synthesizability. Related work is discussed in Sec. 5. We conclude in Sec. 6.

2 State and Data—A Taxonomy of Models and Views

In the domain of software engineering, the distinction of models, views and controllers is common place, not the least because of the MVC pattern [21], the perhaps most widely employed software design pattern [9]. However, in the MDE community this distinction seems less common, even though it can (and as we argue should) be employed there as well.

2.1 Models vs. Views

State and data diagrams adhere to some concrete, visual syntax, which for example entails that edges (representing for example state transitions) must have some source and sink nodes (representing a states). We say that such a diagram is a *view* of some underlying *model*, which comes with a certain semantics. The work flow of today's modeling tools typically prescribes that the model developer directly works on such graphical views, as shown in Fig. 1, using some WYSIWYG graphical editor. This is so common that modelers typically regard the view they draw as "the model," even though the modeling tool first has to translate that view into a model. However, as argued elsewhere, this unification of view and model has some drawbacks [8]. To just name a few, modelers often spend an inordinate amount of time with drawing activities [16]; comparison and version management of visual models is difficult; there is just one and only view for a model. In particular this last issue is central to the work presented in this paper, as we wish to argue that especially when working on applications that have both a state and a data aspect, one would like to have flexible, dynamic views available.

As a general remark, in our experience this unfortunate unification of models and views in MDE is mainly due to two factors: (1) the automatic synthesis of a view from a model requires automatic graph drawing capabilities, which most tools lack, (2) users want to keep control of the views and are sceptical that an automatic drawing algorithm can do a good job, just as the first high-level language compilers at the day were not necessarily welcomed by experienced assembler programmers. However, our experience also indicates that when (1) users get to work with a modeling tool that makes high-quality, state-of-the-art automatic graph drawing a priority, and (2) users employ automatic view synthesis from the beginning instead of first drawing model views manually, they do appreciate the ability of focussing on a model and getting automatically created, well-readable, customizable views for free. We thus in the following build on the premise that models and views can and should be treated separately.

In the remainder of this section we present an overview classification of different modeling and view options. The subsequent sections will explore these in more detail.

2.2 Modeling Options

We first consider the different options of what is modeled explicitly by a human developer. This is not always clear cut, but we identify broadly the following categories:

State Model (M1) This is the traditional state-based modeling approach, using implicit data flow through signal scopes and name matching. This modeling style is supported by various Statecharts tools.

Data-Flow Model (M2) This uses only data flow diagrams. This is typically used for models that do not really have a notion of state. This is provided, for example, by Simulink (without Stateflow).

Multimodel (M3) This uses state diagrams as well as data flow diagrams, explicitly modeled by the user. This is supported, e. g., by Ptolemy, SCADE, or Simulink with Stateflow.

Again, this classification concerns *what* is modeled, not *how* it is modeled. Concerning the latter, this could be either done the traditional way, using some WYSIWYG graphical editor, or it could also be done for example by providing a textual description of the model.

2.3 Viewing options

As mentioned in the introduction, we distinguish between *static views*, which are directly created by a human modeler (with a varying degree of layout support from the modeling tool) and from which a model is derived, and *dynamic views*, which are synthesized automatically from a model. Orthogonal to the static/dynamic distinction, we here propose the following classification:

State View (V1) This is the traditional view for state-based modeling languages, showing only the state diagram without visual indication of shared data or data flow. Statechart tools traditionally offer static state views, an example is shown in Sec. 3.1.

Data-Flow View (V2) Analogously, this consists of data flow diagrams only, as in a typical Simulink diagram.

Multimodel View (V3) This shows data flow as well as state diagrams, using separate diagrams for each purpose. This is what is provided by Ptolemy (see Sec. 3.3), SCADE, or Simulink with Stateflow.

Hybrid View (V4) This uses a single diagram for data flow as well as state, combining the different layers of hierarchy. Static V4 is offered by SCADE (see Sec. 3.4), dynamic V4 is provided by the Ptolemy Browser (Sec. 4.1) and the KIELER SCCharts tool (Sec. 4.4).

Data Overlay View (V5) This is an enriched version of V1, with added indication for access to shared data. This is also offered by the KIELER SCCharts tool (Sec. 4.3).

Naturally, both V1 and V2 can be seen as special cases of V3, V4 and V5. Thus tools that support V3–V5 also support V1 and V2.

3 Static Views

As explained before, the traditional modeling approach entails that the modeler creates one static view of the model. In this section we review the different options that have emerged so far, following the model/view taxonomy presented in Sec. 2. We also introduce a canonical example, a simple traffic light controller, that we will use throughout the paper.

3.1 State Modeling (M1) and Viewing (V1)

Fig. 2 shows the traffic light controller modeled with a state diagram. The example has been presented in previous work on multimodeling [2] as a SyncCharts model [1] and has subsequently adapted to different modeling languages. The diagram in Fig. 2 is SCChart [12], which can be viewed as a conservative extension of SyncCharts, which in turn can be viewed as a synchronous version of Harel's Statecharts [13]. However, for the purpose of this paper, the specifics of SCCharts are not relevant. We can thus see SCCharts as a generic place holder for a state-oriented modeling language. Furthermore, the diagram in Fig. 2 happens to be automatically synthesized from a textual model that the modeler has written in the SCCharts Textual (SCT) language[6], and is thus is, technically, a dynamic view. However, the same type of diagram is also used for static, user-created diagrams in traditional Statechart tools, hence we show it in this section that focusses on existing modeling approaches.

The idea of the traffic light controller is that there is a street with a pedestrian crossing controlled by one traffic light each for the pedestrians and the cars (and any other street traffic). There are three lights for cars, Cgrn, Cyel and Cred, as well as two lights for the pedestrians, Pgrn and Pred. In normal operation, the traffic light should alternate between cars and pedestrians passing, with the

[6] `http://www.sccharts.com`

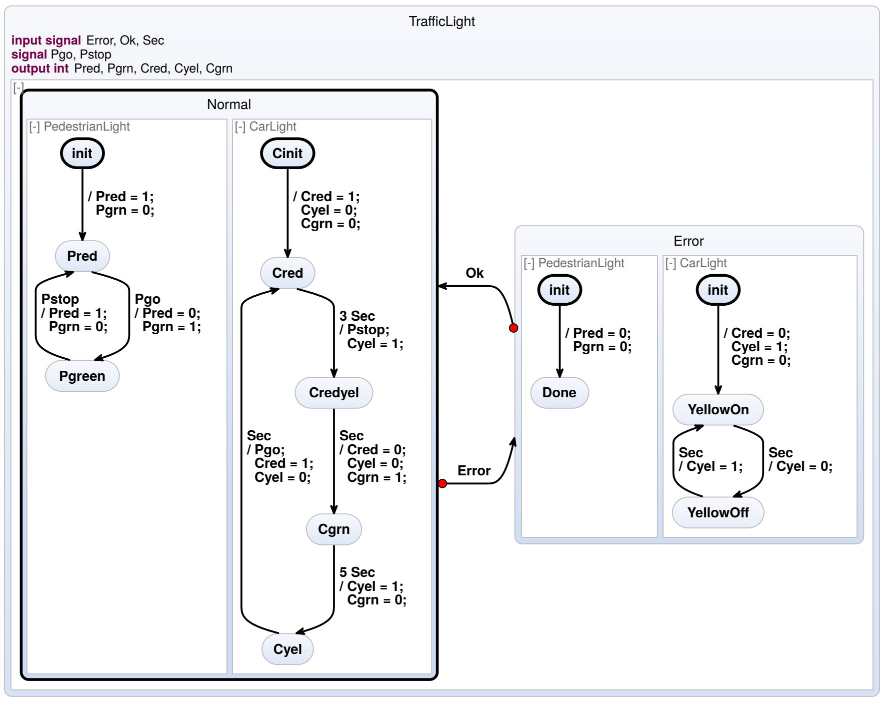

TrafficLight
input signal Error, Ok, Sec
signal Pgo, Pstop
output int Pred, Pgrn, Cred, Cyel, Cgrn
Normal
[-] PedestrianLight
init
/ Pred = 1; Pgrn = 0;
Pred
Pstop / Pred = 1; Pgrn = 0;
Pgo / Pred = 0; Pgrn = 1;
Pgreen
[-] CarLight
Cinit
/ Cred = 1; Cyel = 0; Cgrn = 0;
Cred
3 Sec / Pstop; Cyel = 1;
Credyel
Sec / Pgo; Cred = 1; Cyel = 0;
Sec / Cred = 0; Cyel = 0; Cgrn = 1;
Cgrn
5 Sec / Cyel = 1; Cgrn = 0;
Cyel
Ok
Error
Error
[-] PedestrianLight
init
/ Pred = 0; Pgrn = 0;
Done
[-] CarLight
init
/ Cred = 0; Cyel = 1; Cgrn = 0;
YellowOn
Sec / Cyel = 1;
Sec / Cyel = 0;
YellowOff

Fig. 2: Simple traffic light controller created as state model (M1), shown in state view (V1)

green lights active for a few seconds in each cycle. In case of an error, the lights for pedestrians should be turned off completely and the cars should be alerted by a blinking yellow light.

The model has a main state TrafficLight which consists of two hierarchical states. The state on the left, Normal, manages the normal operation of the traffic light, while the state on the right is activated in the case of an error. The Normal state is marked as initial, shown by the thicker outline. There are signals, Error and Ok, generated by the environment, that guard the transitions between the two states. If Error is signaled, the normal operation is aborted and the control changes from Normal to Error. If OK is signaled, the control switches back, leaves the Error state and restarts normal operation.

Both states in the main module are hierarchical and employ two concurrent regions each. In both states the left region controls the lights for the pedestrians and the right region controls lights for the cars.

This state model/view nicely expresses the behavior of the application, however, the data handling is not obvious, in particular concerning the potential interactions between concurrent regions, as detailed next.

3.2 Data Flow Questions

There exist many semantic variations of data flow languages [14]. A prominent example are Kahn process networks (KPNs) [15], where concurrent actors communicate by producing and consuming *tokens*, and data flow edges represent unbounded FIFO queues. In the data-flow examples we present here, as far as it matters for the subject of this paper, we assume a *synchronous* setting with a clocking regime [3], and communication through shared variables.

One typical question, regarding the semantical validity of a model, is whether there is any *feedback*, that is, a mutual inter-dependence of concurrent regions (region A writes some x that concurrent region B reads, and B writes some y that A reads). Some modeling languages (including Ptolemy, or SCADE with "black-box scheduling") forbid such feedback outright, unless delays (registers) break the cycle, while others (such as SCCharts, or SCADE with "white-box scheduling") only allow it under certain circumstances [12].

Another typical question is whether there may be *conflicting writes* (concurrent regions A and B both write x). Again, some modeling languages always forbid it (Ptolemy, SCADE), others allow it under certain circumstances. For example, some synchronous languages, including SyncCharts and SCCharts, have the notion of *combination functions*, which can be used to combine concurrent writes in a deterministic manner, similar to resolution functions used in hardware design for signals that have multiple drivers. For example, a commutative, associative function such as addition is a valid combination function, and for a shared integer x, two concurrent writes x += 2 and x += 3, if executed atomically, do not impose a race condition; no matter how they are scheduled, their effect will be x += 5. More generally, a valid combination function f on x, y must fulfill that for all $x, y_1, y_2, f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$ holds. For example, "minus" is a valid combination function, even though it is not commutative.

In SCCharts, we say that assignments of the form $x = f(x, e)$ are *relative writes*, provided that f is a valid combination function and e is an expression that does not depend on x and whose evaluation does not have side effects. To clearly delineate relative writes for the compiler (and the modeler), we use the convention that relative writes must be written as *compound assignments*, such as x += 2 instead of x = x + 2. All relative writes of the same type are *confluent*, meaning they can be scheduled in any order. *Absolute writes* are those that are not relative, meaning they do not use a combination function, and absolute writes may also be confluent if they write the same value and do not have side effects. All told, the SCCharts scheduling regime permits concurrent writes to some variable x as long as, within a reaction (logical tick), all absolute writes to x are confluent and are scheduled before all relative writes to x, and all relative writes are of the same type [12]. Furthermore, writes must precede reads, which corresponds to the KPN scheduling constraint that tokens must be produced before they can be consumed.

Again, these questions are rather difficult to answer with the state view, more helpful here are Data-Flow modeling/viewing (M2/V2), or the multimodeling approach (M3/V3) described next.

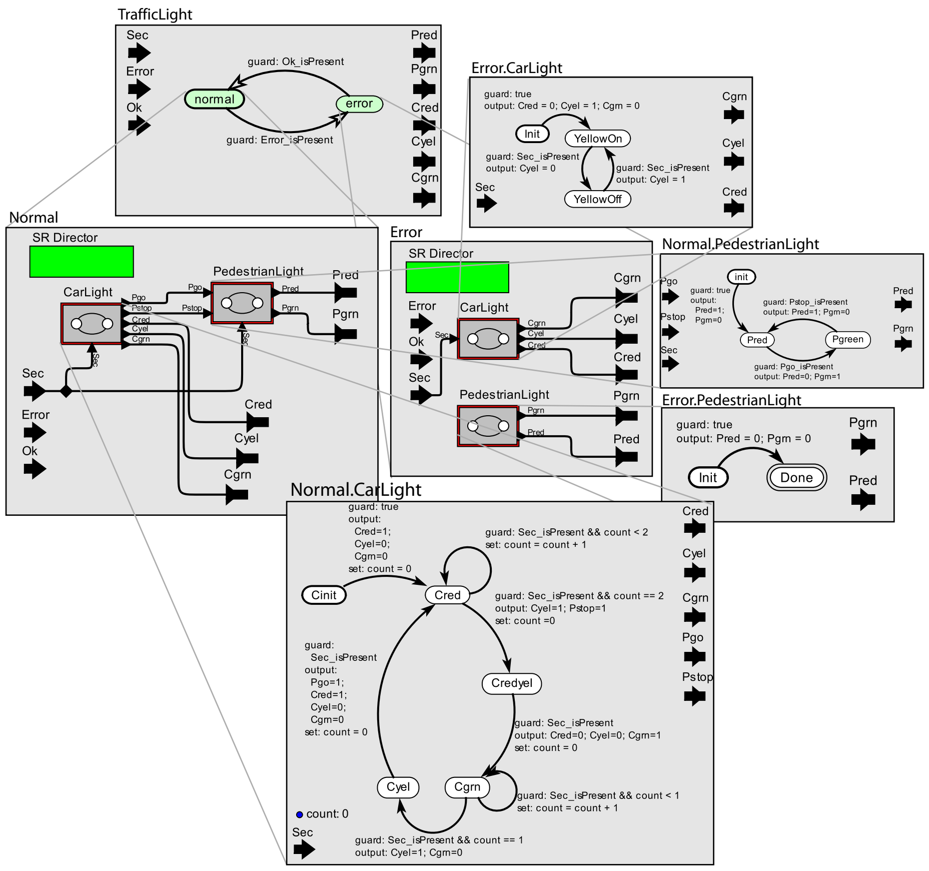


Fig. 3: Traffic light controller in Ptolemy II [2], illustrating multimodeling (M3) and multimodel view (V3)

3.3 Multimodeling (M3) and Multimodel View (V3)

In multimodeling tools, like Ptolemy II, the developer is not restricted to a single type of model. Instead the model can be composed of different types of actors. In Ptolemy II, main building blocks are *Modal Models*, which are used to define finite state machines [7], and *Composite Actors*, which can model different kinds of data flow models. Each of these types can be used in different levels of hierarchy.

Fig. 3 shows the traffic light controller introduced in Sec. 3.1, modeled as a hierarchical model with separate data flow and state machines [2]. Each of the boxes shows one part of the hierarchy, gray lines show the relation between actors and the contained model.

The highest level of hierarchy is the state diagram TrafficLight in the top-left corner. It is equivalent to the first hierarchy level of the model described in

Sec. 3.1. The two states are marked green to indicate the presence of a *refinement*, a child hierarchy, inside the state. The refinement of the normal state is shown at the center left. This refinement is the data flow model already shown in Fig. 1b. Unlike in the state view shown in Fig. 2, the connection via Pgo and Pstop between the two controlling regions and the absence of feedback is immediately visible.

Note that to be precise, Fig. 3 is an enhanced version of what the modeler usually sees and works with. The different state/data flow diagrams are arranged carefully not to hide any information, and gray lines are added manually to show the inter-relationships. In practice, when working with a modeling tool, the modeler will have OS-managed windows for each data flow or state diagram. When exploring a complex model, this routinely requires re-organization of the windows on the screen. It may also pose a mental burden on the modeler to remember which part of the model is where on the screen, as some parts may become completely hidden behind other parts.

One approach that avoids the problem of overlapping parts of the view is presented next.

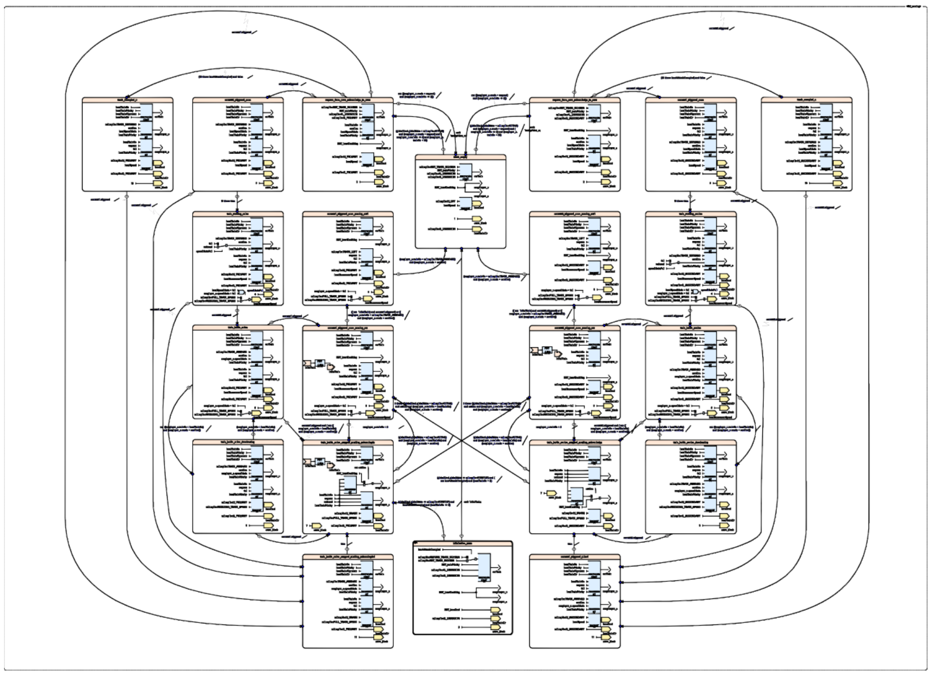


Fig. 4: Railway model in SCADE, illustrating multimodeling (M3) and static hybrid view (V4) and the fact that this can become rather unwieldy (from [24])

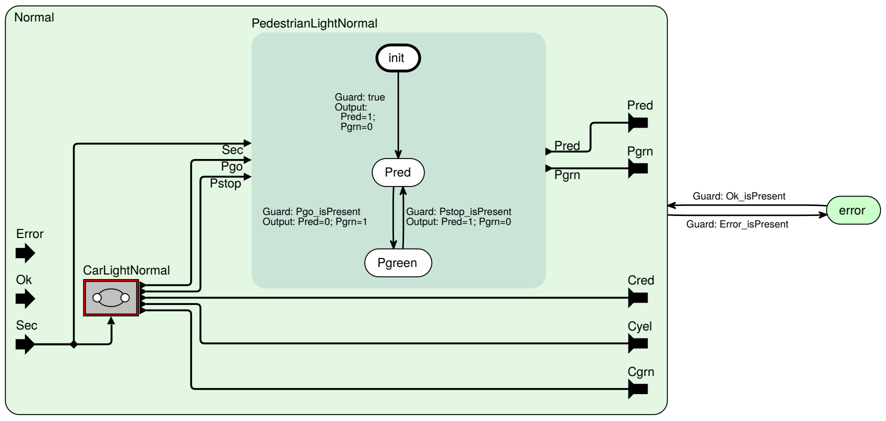


Fig. 5: Partial Ptolemy model, shown in KIELER Ptolemy Browser [25], illustrating multimodeling (M3) and dynamic hybrid view (V4)

3.4 Multimodeling (M3) and Static Hybrid View (V4)

Some modeling tools allow to mix data flow and state in the same diagram. Fig. 4 shows a railway controller modeled with SCADE. At the top level, there is a state machine, each state is refined with a data flow sub-model. This view exposes the whole model in one diagram, thus there are no overlap issues as in the multimodel view (V3). However, as this example illustrates, there are two issues with this: (1) a high manual effort in producing the drawing (this example has consumed 50+ hours of mere drawing time), (2) details are not legible when viewed as a whole, as is the case in this paper with Fig. 4, unless the viewing area is very large, e. g., if the paper happens to be printed on A0.

4 Dynamic Views

To avoid the burden of manually creating a view while modeling, dynamic views provide automatic representations of the model. Consequently, additional or derived information can be displayed to the user without interfering in the actual creation of the model.

4.1 Multimodeling (M3) and Dynamic Hybrid View (V4)

Fig. 5 shows a partial view of the same model as Fig. 3. The view has been generated, based on the original Ptolemy model, by the KIELER Ptolemy Browser, using automated view synthesis as well as automated layout algorithms, in this case provided by the Eclipse Layout Kernel (ELK)[7].

The view allows the user to expand or collapse any hierarchical actor as needed by clicking on it. In Fig. 5 the error state and the embedded modal model

[7] `https://www.eclipse.org/elk`

CarLightNormal are collapsed, while PedestrianLightNormal has been expanded. The hierarchy depth is visualized by a background gradient getting darker on deeper levels.

The Ptolemy Browser also supports different ways to filter diagram elements. In Fig. 5 port labels are only shown for PedestrianLightNormal. Additionally the directors, visible in Fig. 3, are currently hidden.

4.2 Inferring Data Flow

Before showing the enriched viewing options V5 and V4 for state models (M1), we briefly describe the way the data flow is extracted from the source model. This approach is not specific to a certain modeling language but should be adaptable to any state-based language using concurrent regions.

The data flow analysis is performed in a postfix depth-first traversal order of the model hierarchy. This allows us to first analyze the usage of data in all child states of a region and to immediately use the data to visualize the data flow.

For each state, multiple sets of *valued objects*, meaning variables, signals or other kinds of data used in the model, are gathered. The objects are placed in different sets, to separate objects used locally in the state from objects used in a nested hierarchy and to separate read objects from written objects.

For each region, the sets corresponding to the child states are collected and combined with the objects used on the transitions inside the region. These aggregated data then contain all the valued objects, used in the region directly or in some nested hierarchy within the region.

To compute the resulting data flow, the sets of each region are compared to the concurrent regions in the state. Any valued object read in one region and written in a different region results in data flow. Additionally, reading valued objects that are marked as model inputs, or writing valued objects marked as outputs, should be regarded as data flow. These set intersections are used to create the inferred data flow visualizations presented in Sec. 4.3 and Sec. 4.4.

When the data flow between the regions is found, the sets of the regions are combined and propagated upwards to the parent state. This analysis can gather all data flow information in a single pass over the model.

4.3 State Modeling (M1) and Data Overlay Viewing (V5)

Using the information gathered in the data flow analysis, we can show the data flow as an overlay on the original state diagram as shown in Fig. 6. Every data flow between concurrent regions is shown as a direct connection from the writer to the reader of the data.

In the example we can see dependencies in the Normal state, from the write accesses to Pgo and Pstop in the CarLight region to the read accesses in the PedestrianLight region. These are the connections that have been manually modeled in the Ptolemy model. One benefit of this approach is the *stability* of the diagram. The original diagram of the model is not changed, but only enhanced with new

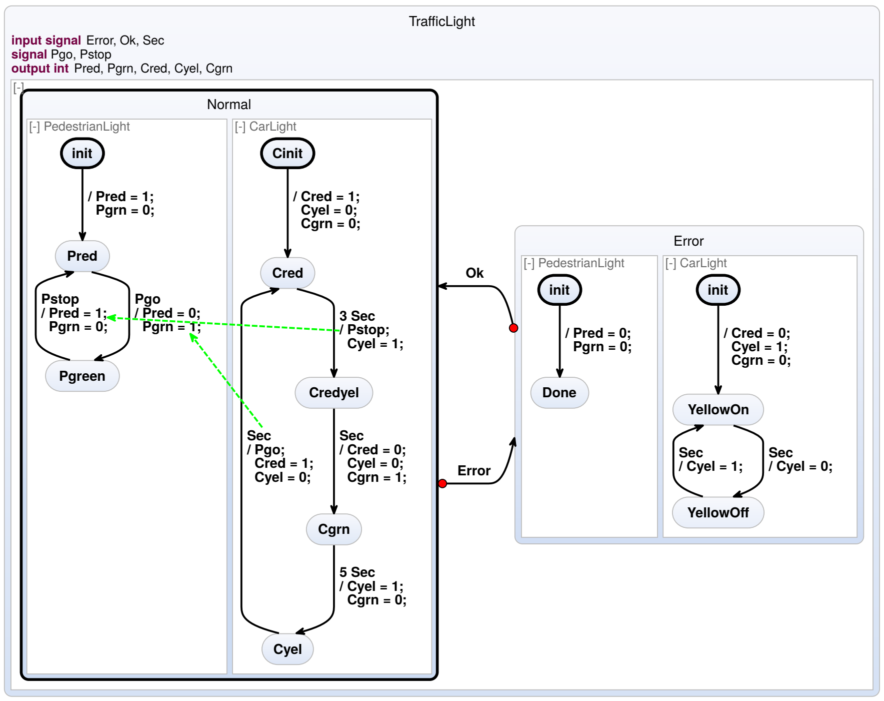

TrafficLight
input signal Error, Ok, Sec
signal Pgo, Pstop
output int Pred, Pgrn, Cred, Cyel, Cgrn
Normal
[-] PedestrianLight
init
/ Pred = 1; Pgrn = 0;
Pred
Pstop / Pred = 1; Pgrn = 0;
Pgo / Pred = 0; Pgrn = 1;
Pgreen
[-] CarLight
Cinit
/ Cred = 1; Cyel = 0; Cgrn = 0;
Cred
3 Sec / Pstop; Cyel = 1;
Credyel
Sec / Cred = 0; Cyel = 0; Cgrn = 1;
Cgrn
5 Sec / Cyel = 1; Cgrn = 0;
Cyel
Sec / Pgo; Cred = 1; Cyel = 0;
Ok
Error
Error
[-] PedestrianLight
init
/ Pred = 0; Pgrn = 0;
Done
[-] CarLight
init
/ Cred = 0; Cyel = 1; Cgrn = 0;
YellowOn
Sec / Cyel = 1;
Sec / Cyel = 0;
YellowOff

Fig. 6: SCCharts traffic light controller with dependency overlay (green dashed arrows), illustrating state modeling (M1) and data overlay viewing (V5)

information. The *mental map* of the developer is preserved [19]. However, compared to the Ptolemy model, this view lacks information about the inputs and outputs of the model regions. In particular, potential conflicting writes to the same output are not directly visible.

The overlay shows the connections between concurrent regions in as much detail as possible. Every relevant connection is individually drawn. In more complex diagrams, this may be more information than can reasonably be displayed visually, as illustrated in Fig. 7, which is part of another railway controller model. The data flow indications overlap each other and create a diagram that is rather unreadable. This problem can partially be addressed through proper filtering, i.e. only showing the dependencies for a selected element. Still, for models of this complexity, the data overlay viewing seems inappropriate to answer, e. g., the questions concerning possible feedback and write conflicts formulated before.

Looking at diagrams as the one in Fig. 7, one may wonder whether visual representations are appropriate for models of this complexity in the first place. In fact, past experience of working with complex models in a tool that offers textual modeling suggests that users are quick to dismiss the automatic graphical views altogether and just stick to the textual models. However, with dynamic, customizable views, graphics may become usable and valuable again. For exam-

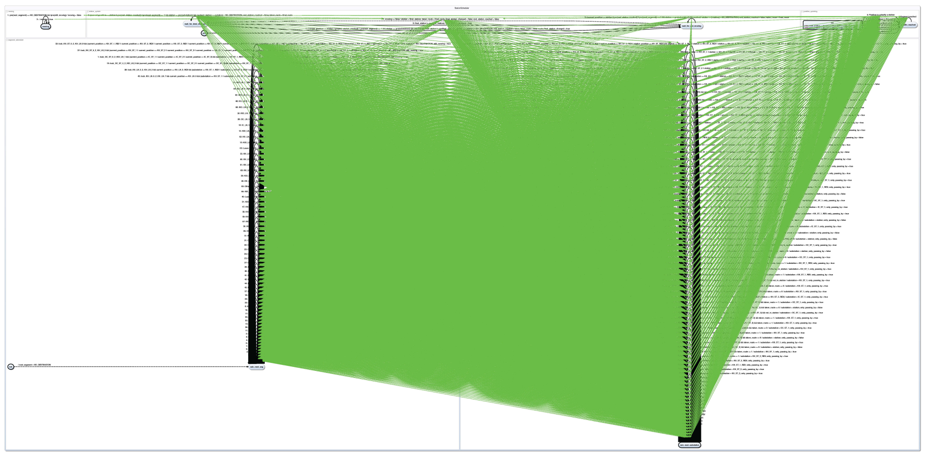
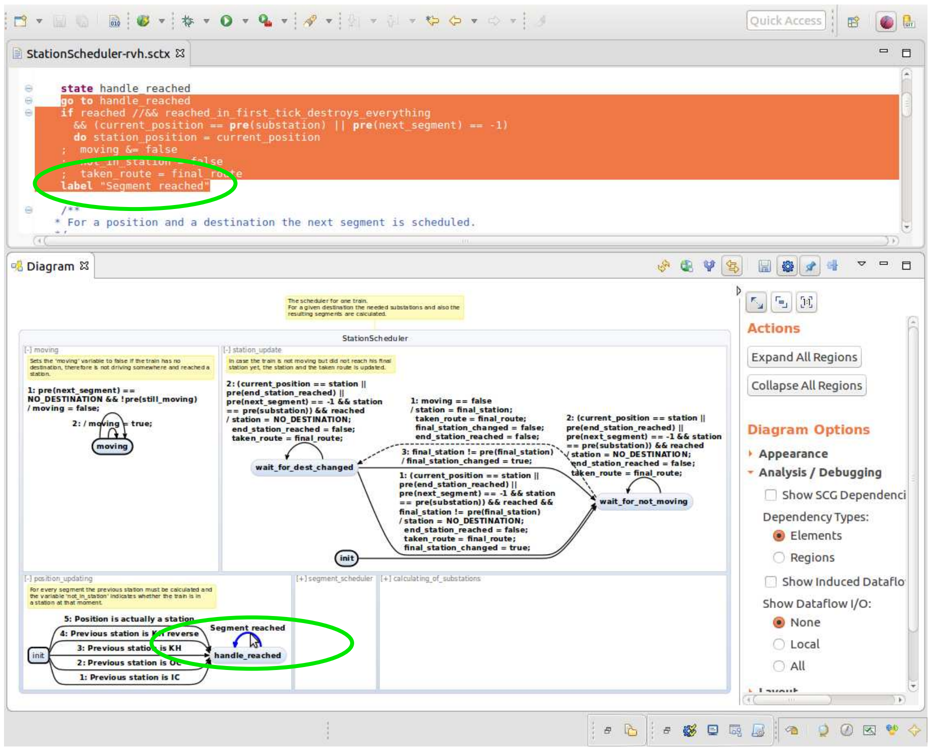


Fig. 7: Railway model, illustrating the limits of data overlay viewing (V5)



```
    state handle_reached
    go to handle_reached
    if reached //&& reached_in_first_tick_destroys_everything
      && (current_position == pre(substation) || pre(next_segment) == -1)
      do station_position = current_position
    ;  moving &= false
    ;  not_in_station = false
    ;  taken_route = final_route
    label "Segment reached"

    /**
   * For a position and a destination the next segment is scheduled.
```

The scheduler for one train.
For a given destination the needed substations and also the resulting segments are calculated.

StationScheduler

[-] moving

Sets the 'moving' variable to false if the train has no destination, therefore is not driving somewhere and reached a station.

```
1: pre(next_segment) ==
NO_DESTINATION && !pre(still_moving)
/ moving = false;
2: / moving = true;
```

[-] station_update

In case the train is not moving but did not reach his final station yet, the station and the taken route is updated.

```
2: (current_position == station ||
pre(end_station_reached) ||
pre(next_segment) == -1 && station
== pre(substation)) && reached
/ station = NO_DESTINATION;
  end_station_reached = false;
  taken_route = final_route;

1: moving == false
/ station = final_station;
  taken_route = final_route;
  final_station_changed = false;
  end_station_reached = false;

2: (current_position == station ||
pre(end_station_reached) ||
pre(next_segment) == -1 && station
== pre(substation)) && reached
/ station = NO_DESTINATION;
  end_station_reached = false;
  taken_route = final_route;

3: final_station != pre(final_station)
/ final_station_changed = true;

1: (current_position == station ||
pre(end_station_reached) ||
pre(next_segment) == -1 && station
== pre(substation)) && reached &&
final_station != pre(final_station)
/ station = NO_DESTINATION;
  end_station_reached = false;
  taken_route = final_route;
  final_station_changed = true;
```

[-] position_updating

For every segment the previous station must be calculated and the variable 'not_in_station' indicates whether the train is in a station at that moment.

[+] segment_scheduler

[+] calculating_of_substations

moving
wait_for_dest_changed
wait_for_not_moving
init
5: Position is actually a station
4: Previous station is KH reverse
3: Previous station is KH
2: Previous station is OC
1: Previous station is IC
Segment reached
handle_reached

☐ Show SCG Dependenci

Dependency Types:
[✓ Elements] Regions

☐ Show Induced Dataflo

Show Dataflow I/O:
[✓ None] Local All

Fig. 8: Screen shot of the railway model with selective region expansion and selected transition label in the KIELER SCCharts tool

ple, consider Fig. 8 for an the alternative view of the same railway model. The two regions that due to their size caused trouble in Fig. 7 are collapsed, and the remaining three regions are small enough to be viewed at once.

Fig. 8 also illustrates another feature of view management, namely the use of an abstract, summary graphical representation together with a detailed textual representation. The top part of the screen shot contains the textual SCChart description as written by the user. The lower part contains a state view of the model, where some transitions are labeled with a more compact summarizing text instead of the concrete trigger and action. One such transition is labelled Segment reached (see added green oval); when clicking on the transition, the part of the textual source that describes the details of the transition trigger/action is automatically scrolled to and highlighted. Finally, the screen shot also illustrates how comments (the "post-it" notes) can be shown in the dynamicl view, constructed from semantic comments in the textual model.

4.4 State Modeling (M1) and Dynamic Hybrid View (V4)

An alternative view, that aims to avoid the potential cluttering issues of the data overlay view (V5) that was illustrated in Fig. 7, is the dynamic hybrid view (V4) again. In Sec. 4.1, the dynamic V4 was synthesized from a multimodel, which already had explicitly modeled the data flow. We now synthesize dynamic V4 from a state model.

The main idea behind this visualization is to use the previously "unused" hierarchy layer between regions to show the data flow. In normal Statecharts there are no connecting edges between regions and the placement of the regions next to each other usually carries no semantic meaning except concurrency. To enrich the diagram, we leverage this hierarchy level and add the data flow between the regions.

The Traffic Light Example Fig. 9 shows the inferred data flow with *local* inputs and outputs, automatically created from the very same SCCharts model from which the state view of Fig. 2 was synthesized. Local inputs and outputs are the valued objects that are used on the same hierarchy level as the input or output. Alternatively, all inputs and outputs, including usages in nested hierarchy levels, could be shown, or all input and output nodes could be hidden, leaving only the concurrent data flow between neighboring regions. Inside every hierarchical state, each shared valued object is represented by one (hyper)edge. All writers of the valued objects are sources of the edge and all readers are sinks.

On the top hierarchy level, input nodes for Error and Ok are added because these signals govern the transition between the two top-level states. Inside the Normal state, the local data flow between CarLight and PedestrianLight, the reading of the Sec input, as well as the written outputs have been added. In terms of visualized information, the resulting diagram is similar to the corresponding Ptolemy diagram in Fig. 5. One deviation occurs in Normal.PedestrianLight, which in the manually specified data flow part of the Ptolemy model defines

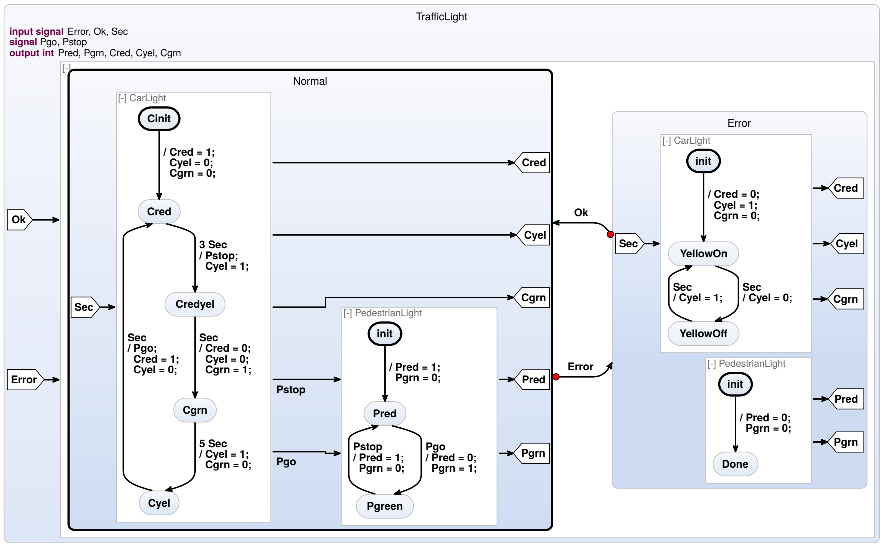


Fig. 9: SCCharts traffic light controller with inferred data flow, illustrating state modeling (M1) with a dynamic hybrid view (V4)

Sec as an input, even though Sec is actually not read in PedestrianLight. This "modeling glitch," manifested also in Normal, is probably not on purpose, as for example the Sec input is (correctly) omitted for Error.PedestrianLight. This glitch in such a small, well-studied example indicates that dynamically created data flow visualizations not only offer the convenience of not having to explicitly re-model information that is already present in the state part; dynamic, automatically inferred views also help to keep state and data flow consistent, in particular as models become more complex.

Another noteworthy detail is the order of the regions CarLight and PedestrianLight inside the Normal state. Compared to the original diagram in Fig. 2, these two regions switched their place in the diagram. This is done by the automatic layout algorithm, to always draw the data flow edges from left to right if possible. If the data flow edges create a cycle, one of the edges has to be reversed and will be drawn as a feedback edge from right to left, thus making potential feedback obvious to the modeler.

Compared to the overlay presented in Sec. 4.3, the precision of this approach is a bit reduced. The usages of the objects are not indicated directly, but only on a per-region granularity. On the other hand, this approach always produces a clean diagram without edges potentially overlapping relevant parts of the diagram. Additionally this approach can show the data flow between regions, even when the regions itself are collapsed and the child states are not currently visible.

The Railway Example Dynamic hybrid views also let us draw the railway example, shown before in the rather inaccessible Fig. 7, in a cleaner and more readable way. By collapsing all regions and configuring some label management we can create the diagram shown in Fig. 10.

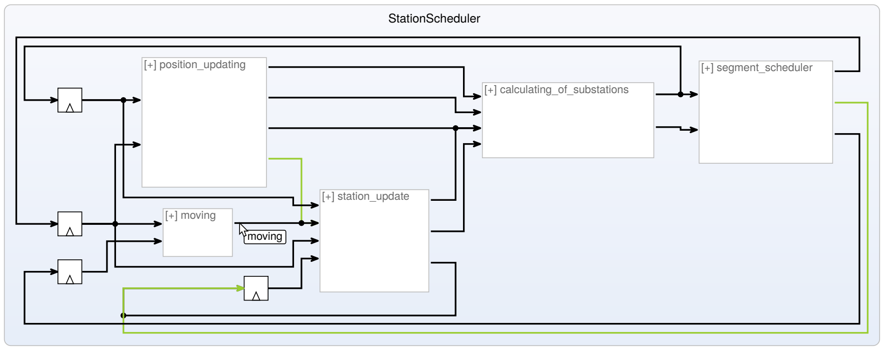


Fig. 10: Railway model with data flow, illustrating multimodeling (M3) and dynamic hybrid view (V4) for a complex model, with absolute/relative writes (black/green edges) and mouse-over annotation of data flow edges.

The view exhibits feedback, visible in the edges routed around the outside of the diagram. In this case, all feedback edges end in a register. These registers are automatically created, both in the code synthesis and in the inferred data flow view, if the value is not used directly but instead read in a pre operator. In SCCharts, as in other synchronous languages, the pre operator accesses the value a variable had at the end of the previous reaction. This is similar to a *delay* actor in Ptolemy.

As can be seen in Fig. 10, the automatic layout of the SCCharts editor tends to place registers at the left of the diagram. Specifically, when the layout algorithm encounters a cycle in the graph that makes it impossible to draw all edges in the same direction, it tries to break the cycle by reversing edges that enter a register. This is another convention that helps the modeler to quickly grasp whether feedback is broken by a register or not.

Another feature in Fig. 10 can be seen on the hyperedge from two writers, moving and position_updating, to the reader station_update. As explained in Sec. 3.2, multiple writers may indicate a problem in a model. However, the write performed by position_updating is a relative write (see again Sec. 3.2), indicated by a green edge segment. A closer inspection of Fig. 8, where the textual SCCharts description shows the relative write access moving &= false, confirms that this is a relative write of type conjunction. Thus it can be safely combined with the absolute write (black) from station_update. A similar situation is present in the model with an absolute write from station_update and the relative write from segment_scheduler. These two writes are combined and fed back to the corresponding register. There is another concurrent write, where position_updating and station_update perform absolute writes to the same value, but these happen to be confluent because they write the same value. As a possible extension to the current tooling, this fact could be fed back visually in the diagram as well, e. g., by showing some kind of error marker in case there are non-confluent absolute writes.

As a last detail, for compactness the dynamic view from Fig. 10 has been configured to not show any data flow labels. However, a mouse-over on an edge, such as the aforementioned hyperedge, shows the shared variable in question, in this case moving.

5 Related Work

Beyond the work on multimodeling mentioned in the introduction [2], there are several proposals on combining different model types [10, 26, 4]. However, there appears to be little comparable work that focusses on modeling pragmatics the way we do here, compared to the large body of work on semantic and synthesis issues. There are some works, such as by Petre [18], which compare the utility of graphical and textual views.

Human-centered software engineering aims to improve usability in software development, but not with a focus on modeling [22]. Another related community focuses on software visualization [6]. Smyth et al. have presented an approach to extract SCCharts from legacy C code [23]. Together with the work presented here, this now allows a data flow view of C programs.

The synthesis of diagrams advocated here builds on automatic layout, for which the graph drawing community has developed a large variety of approaches, as for example surveyed by Di Battista et al. [5].

6 Conclusions and Outlook

We have illustrated that data flow views do not necessarily have to be created manually by a modeller, as is long-standing tradition, but can be inferred automatically, in our case from state models. More generally speaking, we made the argument that views can and should be separated from models. Designers should be able to concentrate on creating and maintaining models, in whatever formalism is most convenient, and a modeling tool should infer different, customizable views according to the task at hand. This not only saves valuable developer time, but can also help to avoid model inconsistencies.

Even though we did not question the difference between state models and data flow models here, the fact that a data flow view can be just synthesized from a state model begs the question of how fundamental that difference really is. For example, going in the other direction, ongoing work indicates that it is not too difficult to synthesize fairly concise SCCharts from Lustre [11], which is generally viewed as a data flow language.

There are numerous directions to go from here. For example, we have focussed on showing data flow relations between concurrent regions/actors, implying that this is what the modeler is most interested in. However, data flow languages are also used to express sequential computations, and one might extend the work here to infer data flow for sequential computations as well. For example, referring back to the existing work on the model extraction from legacy C code [23], it might be interesting to infer the data flow from a C program that performs some complex signal processing using various components. Conceptually, these components can be seen as actors and could be visualized as such, even if the actors may not be concurrent anymore but typically are already sequentialized in the C program.

Another, largely open question is how to give good visual feedback on more complex causality questions. As briefly alluded to, feedback may be permitted in some cases, such as when back-and-forth scheduling between concurrent actors is permitted within a reaction. This is a powerful language feature, but may lead to models that are hard to debug in case they are not schedulable.

Finally, as the concept of dynamic views hinges on the capability to automatically draw well-readable diagrams, the area of automatic graph drawing is called upon. We believe that auto-layout is already good enough to be usable in practice, with open source libraries that make state-of-the-art algorithms freely available and have stable interfaces. Thus auto-layout should become a standard feature in today's modeling tools, as is for example already the case in Ptolemy (which uses ELK). However, further improvements are still possible. One detail is the handling of hyper edges, which sometimes is still unsatisfactory, a broader issue is that of "interactive" layout where the modeler can influence the model drawing without having to fall back to manual layout.

Acknowledgements We thank the participants of the 2008 and 2017 editions of the model railway project conducted at Kiel University for the SCADE and SCCharts models they contributed. We also thank the contributors to the open source projects such as Ptolemy, as well as KIELER, ELK, Xtext and numerous other Eclipse projects, that made this work possible.

This work has been supported in part by the German Science Foundation, as part of the Precision-Timed Synchronous Reactive Processing (PRETSY2, DFG HA 4407/6-2) and Compact Graph Drawing with Port Constraints (ComDraPor, DFG HA 4407/8-1) projects.

References

[1] Charles André. "Computing SyncCharts Reactions". In: *Electronic Notes in Theoretical Computer Science* 88 (2004), pp. 3–19.

[2] Christopher Brooks, Chih-Hong Patrick Cheng, Thomas Huining Feng, Edward A. Lee, and Reinhard von Hanxleden. "Model Engineering using Multimodeling". In: *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management (MCCM '08), a workshop at MODELS '08*. Toulouse, Sept. 2008.

[3] Paul Caspi and Marc Pouzet. “Synchronous Kahn Networks”. In: *ICFP ’96: Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*. Philadelphia, PA, USA, 1996, pp. 226–238.

[4] Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. “Globalizing Modeling Languages”. In: *IEEE Computer* 47.6 (2014), pp. 68–71.

[5] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[6] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behavior and Evolution of Software*. Springer, 2007.

[7] Thomas Huining Feng, Edward A. Lee, Xiaojun Liu, Christian Motika, Reinhard von Hanxleden, and Haiyang Zheng. “Finite State Machines”. In: *System Design, Modeling, and Simulation using Ptolemy II*. Ed. by Claudius Ptolemaeus. Ptolemy.org, 2014.

[8] Hauke Fuhrmann and Reinhard von Hanxleden. “On the Pragmatics of Model-Based Design”. In: *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*. Vol. 6028. LNCS. 2010, pp. 116–140. DOI: 10.1007/978-3-642-12566-9.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] John C. Grundy, John Hosking, Jun Huh, and Karen Na-Liu Li. “Marama: An Eclipse meta-toolset for generating multi-view environments”. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*. ACM, 2008, pp. 819–822. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368210.

[11] 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.

[12] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014.

[13] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[14] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. “Advances in Dataflow Programming Languages”. In: *ACM Computing Surveys* 36.1 (Mar. 2004), pp. 1–34.

[15] Gilles Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Information Processing 74: Proceedings of the IFIP Congress*

74. Ed. by Jack L. Rosenfeld. IFIP. North-Holland Publishing Co., Aug. 1974, pp. 471–475.

[16] Lars Kristian Klauske and Christian Dziobek. "Improving Modeling Usability: Automated Layout Generation for Simulink". In: *Proceedings of the MathWorks Automotive Conference (MAC'10)*. 2010.

[17] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. "Actor-Oriented Design of Embedded Hardware and Software Systems". In: *Journal of Circuits, Systems, and Computers (JCSC)* 12.3 (2003), pp. 231–260. DOI: 10.1142/S0218126603000751.

[18] Marian Petre. "Why looking isn't always seeing: Readership skills and graphical programming". In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44.

[19] Jens von Pilgrim. "Mental Map and Model Driven Development". In: *Layout of (Software) Engineering Diagrams 2007*. Vol. 7. Electronic Communications of the EASST. Berlin, Germany, 2007.

[20] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[21] Trygve Reenskaug. *Models – Views – Controllers*. Xerox PARC technical note. Dec. 1979.

[22] Ahmed Seffah, Jan Gulliksen, and Michel C. Desmarais. "An Introduction to Human-Centered Software Engineering". In: *An Introduction to Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*. Vol. 8. Human-Computer Interaction Series. Springer Netherlands, 2005, pp. 3–14.

[23] Steven Smyth, Stephan Lenga, and Reinhard von Hanxleden. *Model Extraction for Legacy C Programs with SCCharts*. Poster presented at the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '16), Corfu, Greece. Oct. 2016.

[24] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SCCharts: The Railway Project Report*. Technical Report 1510. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.

[25] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. *Automatic Layout of Data Flow Diagrams in KIELER and Ptolemy II*. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.

[26] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan K. Jackson. "OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber-Physical Systems". In: *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*. 2014, pp. 235–248.