Label Management: Keeping Complex Diagrams Usable

Christoph Daniel Schulze
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Kiel, Germany
Email: cds@informatik.uni-kiel.de

Yella Lasch
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Kiel, Germany
Email: ybl@informatik.uni-kiel.de

Reinhard von Hanxleden
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
Kiel, Germany
Email: rvh@informatik.uni-kiel.de

***Abstract*—Most visual languages are not purely graphical but include textual labels to complete the picture. However, in some languages labels tend to become rather long and thereby enlarge diagrams considerably. Since today's state-of-the-art development tools usually display diagrams in full detail, users must often scroll through the diagram or zoom out until the diagram fits inside the available drawing area, but then ceases to be legible.**

In this paper, we address this problem by examining ways to dynamically shorten the text of labels to keep the size of a diagram manageable. We introduce a number of label shortening strategies, explain ways to integrate them into diagram generation processes based on automatic layout algorithms, and explain their relation to the established *focus and context* approach which aims at solving a similar problem. We evaluate our strategies based on the SCChart visual language and an open-source, Eclipse-based modeling environment.

I. Introduction

Visual languages are in widespread use for a variety of applications. Languages such as the Unified Modeling Language (UML) provide different complementary views on the systems built by developers, for example by visualizing the different components they are composed of, or by showing how those components interact. In the automotive, avionics, and embedded systems industries, visual languages are largely used as programming languages to develop software following model-based development concepts. Languages such as ASCET (ETAS Group), LabVIEW (National Instruments), SCADE (Esterel Technologies), or Simulink (MathWorks) allow developers to define software systems using *node-link diagrams*: *nodes* are entities that can consume and produce data, which are then transmitted between them through *links* (or *edges*).

One reason for the existence of the UML is that it is hard to keep an overview of a software system while looking at code written in traditional, textual programming languages, giving rise to the need for additional visual representations of the system's high- to medium-level architecture. One might assume that this is a flaw inherent only to textual languages, but visual programming languages suffer from the same shortcoming: in practice, software systems can easily consist of hundreds of diagrams, with each diagram usually containing about 10 to 50 levels of hierarchy (we received reports of diagrams with hundreds of levels of hierarchy).

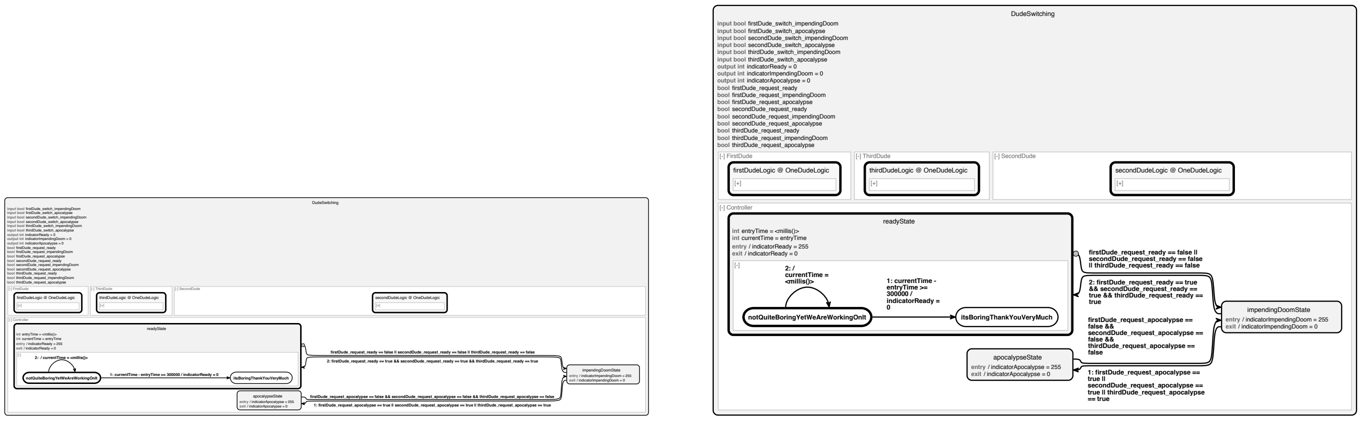
Developers may easily get lost in this kind of complexity, but even with only one level of hierarchy a diagram can quickly grow too large to be displayed on a single screen in its entirety. One reason for this—that we are particularly interested in in this paper—is that most visual languages cannot make do without labels that give meaning to the visual elements. Depending on the language, these labels can grow rather long and thus enlarge the diagram significantly, regardless of whether or not they are contributing any significant information at a particular moment. Developers are thus forced to either keep scrolling back and forth in the diagram, or to decrease the zoom level and causing legibility to suffer. In fact, we have witnessed developers feeling the need to work around these problems by introducing more levels of hierarchy, each kept as small as possible.

A. Model-View Separation

In keeping with the established separation between the model and its views, the solution just described attempts to solve a view problem by changing the model when in fact the model's development should be guided only by the problem it is supposed to solve, not by shortcomings of the software used to develop it. Today's commonly used development tools usually consider all information contained in a model to be of equal importance to the developer and thus provide them with views of the model with all of its details. Instead, it should be recognized that the importance of different pieces of information varies both depending on the task at hand and from element to element. Showing all the details should be replaced by a view filtering approach that displays only the relevant information. Incidentally, this is how views were defined in the first place when the model-view-controller paradigm was introduced by Reenskaug [1]: "[The view] would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*."

This realization and subsequent improvements to the usability of visual languages are closely related to *modeling pragmatics* [2], which aims to increase developer productivity by making working with visual languages easier. One approach to increase the usability of visual languages is *view management*: adjusting the level of detail of each element to be appropriate for the current task. This is not limited to the main view,

978-1-5090-0252-8/16/$31.00 ©2016 IEEE


DudeSwitching
input bool firstDude_switch_impendingDoom
input bool firstDude_switch_apocalypse
input bool secondDude_switch_impendingDoom
input bool secondDude_switch_apocalypse
input bool thirdDude_switch_impendingDoom
input bool thirdDude_switch_apocalypse
output int indicatorReady = 0
output int indicatorImpendingDoom = 0
output int indicatorApocalypse = 0
bool firstDude_request_ready
bool firstDude_request_impendingDoom
bool firstDude_request_apocalypse
bool secondDude_request_ready
bool secondDude_request_impendingDoom
bool secondDude_request_apocalypse
bool thirdDude_request_ready
bool thirdDude_request_impendingDoom
bool thirdDude_request_apocalypse
[-] FirstDude
firstDudeLogic @ OneDudeLogic
[-] ThirdDude
thirdDudeLogic @ OneDudeLogic
[-] SecondDude
secondDudeLogic @ OneDudeLogic
[-] Controller
readyState
int entryTime = <millis()>
int currentTime = entryTime
entry / indicatorReady = 255
exit / indicatorReady = 0
2: / currentTime = <millis()>
1: currentTime - entryTime >= 300000 / indicatorReady = 0
notQuiteBoringYetWeAreWorkingOnIt
itsBoringThankYouVeryMuch
firstDude_request_ready == false || secondDude_request_ready == false || thirdDude_request_ready == false
2: firstDude_request_ready == true && secondDude_request_ready == true && thirdDude_request_ready == true
firstDude_request_apocalypse == false && secondDude_request_apocalypse == false && thirdDude_request_apocalypse == false
1: firstDude_request_apocalypse == true || secondDude_request_apocalypse == true || thirdDude_request_apocalypse == true
impendingDoomState
entry / indicatorImpendingDoom = 255
exit / indicatorImpendingDoom = 0
apocalypseState
entry / indicatorApocalypse = 255
exit / indicatorApocalypse = 0

(a) Without label management.

(b) With label management.

Fig. 1. The same diagram (a) with all labels displayed without modification, and (b) with label size managed by wrapping the label's text appropriately. The area available to draw the SCChart is the same in both figures, but the diagram in (b) can be drawn with a larger zoom factor. In this particular case, the label size adjustment is performed without reducing the amount of information in the diagram.

but can include additional views generated on the fly with automatic layout algorithms [3] to show a filtered and possibly highly specialized view of the model. *Label management* is one of the building blocks of view management, focussing on how labels are displayed. As an example, Fig. 1 shows the same model with different label management strategies applied to the edge labels. If the area available to draw a diagram stays constant, as is typical when viewing and editing diagrams, reducing label detail allows for a larger zoom factor and increased legibility by only sacrificing information that is not of immediate importance.

B. Contributions

In this paper, we explore ways to integrate label management into view management. Based on a case study, we present different label management strategies based on modifying a label's text, as well as different ways of integrating them into the view generation process. We show how the role of automatic layout algorithms can be expanded to include decisions not only regarding where elements are placed, but also what information these elements show. We thus do not focus only on shortening labels, but also on the impact this has on the whole process of generating and displaying diagrams. While our case study focusses on *viewing scenarios*, the introduced concepts should be applicable to *editing scenarios* as well. For the scope of this paper, we limit our discussions to edge labels only.

C. Related Work

This paper builds upon work done by Hauke Fuhrmann as part of his dissertation [4]. Therein, he defines label management as one important part of filtering views presented to the user. He introduces the concept of a *label manager* and the need for it to be customizable to specific languages and introduces several approaches to label management, which we extend by new approaches. We build on this foundation by investigating and evaluating concrete label managers and proposing ways for combining them and for integrating them into view generation and user interaction.

Over the years, different methods have been proposed to solve the problem of too much data and too little screen space. Opticially distorting fisheye views [5] show the areas of interest to the user with increased magnification, with results that mimic the classic fisheye lens effect. Applied to our use cases, the distortion would negatively impact legibility. Graphical fisheye views [6] remove the distortion by magnifying the nodes themselves and applying automatic layout to keep nodes, edges, and labels straight. In this paper, we focus on changing the text of labels instead of their magnification. Both techniques however may well be combined.

Been et al. apply label filtering concepts to dynamic map exploration [7]. As the user zooms in and out, labels are shown and hidden depending on the space available for them. Especially if combined with a notion of a label's importance, this approach seems to work very well for map exploration, as demonstrated for example by Google Maps. However, in this approach a label is either completely visible or completely hidden and does not influence its surroundings much. In this paper, we are interested in changing a label's actual content, and with it its size, to allow more diagram elements to fit on the screen. This can include, but is not limited to, hiding a label altogether.

Musial and Jacobs follow a similar idea and apply filtering concepts to UML diagrams [8]. They gradually reduce the amount of details classes are visualized with as their graph-theoretical distance increases to classes focussed on by the user. While this does change the diagram's layout to show more classes to the user, labels are again either completely shown or hidden, but do not have their level of detail changed. Also, Musial and Jacobs concentrate on modifying node labels based purely on focus and context information. In this paper, we focus on edge labels and investigate ways to take layout information into account when making label management decisions.

Regarding the decision of how much detail an element should be shown with, *focus and context* is an important concept [9]. Herein, the diagram elements are divided into the set of elements the user is currently focussing on and surrounding elements that provide context for the focussed elements. We show how label management decisions can be made based on this method.

D. Outline

We start by introducing the visual language used as a case study for label management in Sec. II. Sec. III introduces strategies for reducing the detail of different kinds of labels, which answers the question of *how* to shorten labels. We will look at how to integrate these strategies into the view generation process in Sec. IV, which answers the question of *when* and *how much* to shorten labels. We evaluate the presented ideas in Sec. V with a survey and a metrics-based evaluation and conclude the paper with future work in Sec. VI.

II. Problem Setting

The concepts we introduce should be generally applicable to all visual languages that employ textual elements and suffer from the problems described above. However, to make things concrete our main motivation for the purposes of this paper are our experiences working with a visual language called SCCharts [10], a synchronous language inspired by David Harel's statecharts [11].

At their most basic, SCCharts consist of *states*, displayed as nodes, and *transitions* that transfer control between states, displayed as edges (see Fig. 2 for an example). States can contain further SCCharts to define what happens when they are active, allowing their behavior to be arbitrarily complex. Transitions can have a *trigger* condition that must be true for a transition to become eligible. Trigger conditions are expressions that usually depend on the presence, absence, or value of signals (which can be thought of as variables). Since more than one transition leaving an active state can be eligible at a given time, transitions can be assigned *priorities*: the eligibility of transitions is evaluated in order of their priority, and the first eligible transition is taken. When that happens, it can cause an *action* to be executed, which can change a signal or execute host code calls that invoke externally defined functions. The signals used in triggers and actions can be taken from or made available to the state's environment by declaring them as input or output signals in the state's *interface*.

State names, state interfaces, priorities, triggers, and actions are all displayed as textual labels inside or next to the respective graphical elements. As already hinted at in the introduction, an SCChart thus derives much of its meaning from its textual content.

SCCharts are actually not edited graphically, but through a textual editor—again, see Fig. 2—complemented by a graphical view of the SCChart as currently specified. Both are part of an open-source Eclipse-based modeling environment.[1] The graphical view is generated and kept up to date on the fly, with all of its elements placed and routed by automatic layout algorithms. The idea of this kind of editing environment is to give developers the efficiency of textual editing they are used to as well as the possibility to spot problems easily through the graphical view. The latter can also be used to navigate through the code by clicking on graphical elements, which causes the editor to jump to the corresponding piece of code.

The graphical view's usefulness as a support to the developer obviously hinges on whether what it displays is readable and of any significance to the current editing task. This is where view management—and with it label management—become relevant. The trigger and action of a transition can become quite long even in simple SCCharts, causing the graphical view to quickly become unusable since the zoom level is lowered to a point where details cease to be recognizable. The methods introduced in this paper aim to dynamically reduce the size of labels and thereby of the whole model to keep it legible.

III. Label Management Strategies

In this section, we examine ways for *how* to shorten labels before we turn to the *when* and *how much* in the next section. We start with basic label management strategies that should be applicable to all visual languages and then concentrate on more specialized strategies for SCCharts. Finally, we explore ways to combine different strategies.

All of these strategies modify the text of labels. Of course it would also be possible to for example change font sizes. While that would work well towards reducing the area of a diagram, it may even hamper legibility and not serve to reduce visual clutter by hiding unnecessary details, which our strategies are capable of.

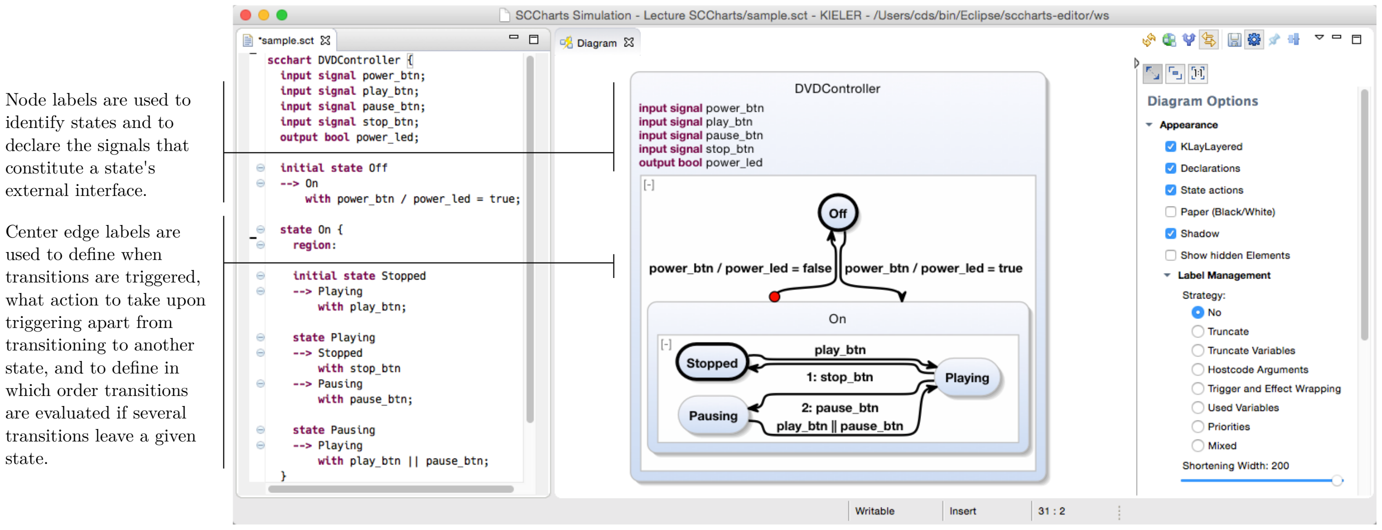
Before we start, note that many of the strategies we are about to introduce implicitly assume knowledge about a *target width*: how wide a given label should be in the diagram. For the remainder of this section, we assume this value to be magically known; as soon as we integrate the strategies into the view generation process in the next section, we will turn to the question of how to actually determine it.

A. Basic Label Management Strategies

In his PhD thesis, Fuhrmann suggested a number of basic label management strategies that Table I summarizes based on an SCChart transition label [4].

The first strategy, *syntactical abbreviation*, simply cuts the label's text off once the target width is reached and adds and ellipsis as a visual hint of the abbreviation. This is an easy to implement strategy which may work well for natural language, but does have shortcomings when it comes to formally structured text. In the example, the trigger contains references to three signals, `SignalA`, `SignalB`, and `SignalC`. Syntactical abbreviation removed the references to all but the first signal, which may confuse users. Nevertheless, depending on the visual language this may still be a viable strategy.

[1]http://rtsys.informatik.uni-kiel.de/kieler



Node labels are used to identify states and to declare the signals that constitute a state's external interface.

Center edge labels are used to define when transitions are triggered, what action to take upon triggering apart from transitioning to another state, and to define in which order transitions are evaluated if several transitions leave a given state.

SCCharts Simulation - Lecture SCCharts/sample.sct - KIELER - /Users/cds/bin/Eclipse/sccharts-editor/ws

*sample.sct

```
scchart DVDController {
  input signal power_btn;
  input signal play_btn;
  input signal pause_btn;
  input signal stop_btn;
  output bool power_led;

  initial state Off
  --> On
    with power_btn / power_led = true;

  state On {
    region:

    initial state Stopped
    --> Playing
      with play_btn;

    state Playing
    --> Stopped
      with stop_btn
    --> Pausing
      with pause_btn;

    state Pausing
    --> Playing
      with play_btn || pause_btn;
  }
```

Diagram Options

Appearance
- [✓] KLayLayered
- [✓] Declarations
- [✓] State actions
- [ ] Paper (Black/White)
- [✓] Shadow
- [ ] Show hidden Elements

Label Management

Strategy: [✓ No] Truncate / Truncate Variables / Hostcode Arguments / Trigger and Effect Wrapping / Used Variables / Priorities / Mixed

Shortening Width: 200

Writable | Insert | 31 : 2

Fig. 2. A typical small SCChart, currently being edited in our KIELER development environment. SCCharts are defined using a textual language. While working in the textual editor (left), a dynamically generated graphical view of the SCChart (right) is always updated to reflect the chart's current state (in this case, its aspect ratio is about 4 to 3). The graphical representation uses different kinds of labels, as described in the picture. Note how the view provides ways for the user to customize the visualization. Selecting an element in the graphical view selects its definition in the text editor.

TABLE I
BASIC LABEL SHORTENING STRATEGIES

Type		Example
Abbreviate	syntactical	`(not SignalA) xor (...`
	semantical	`SignalA, SignalB / SignalC`

Wrap	syntactical (hard)	`(not SignalA) xor (not`
		`SignalB) / SignalC(cou`
		`nter)`

	syntactical (soft)	`(not SignalA) xor (not`
		`SignalB) / SignalC(`
		`counter)`

	semantical	`(not SignalA) xor`
		`(not SignalB) /`
		`SignalC(counter)`

The second strategy, *semantical abbreviation*, was developed to solve that problem. The strategy uses semantic information about the label's content to reduce it to its most important parts. The aim is to give users enough information to find a particular label which they can then inspect in more detail. In the example, the trigger is abbreviated to the list of mentioned signals, which gives the user an idea of what contributes to the trigger while hiding details about the exact trigger expression. Note that this strategy may well result in labels that exceed the target width.

The abbreviation strategies result in a smaller label size solely by reducing a label's width. Depending on the visual language the width may indeed be the most critical part of a label's size; its height, however, may be much less of a problem, to the point where it could even be increased. This is the case for SCCharts, and can be dealt with not by shortening labels, but by inserting line wraps instead.

The first line wrapping strategy, *syntactical wrapping*, inserts line wraps when a line of text is about to exceed the target width. This can be either at the exact position where this happens without regard to the text's structure (*hard syntactical wrapping*), or between the tokens the text is composed of (*soft syntactical wrapping*). In the example, hard syntactical wrapping does not make much sense since the text is composed of tokens small enough to insert line wraps between them.

In contrast to syntactical abbreviation, syntactical wrapping fares fairly well with formally structured text. A second strategy, however, offers an interesting alternative: *semantical wrapping* restricts the token pairs between which line breaks may be inserted in an attempt to visually preserve an expression's structure. In the example, line breaks are inserted after the two binary operators: `xor` and the division operator. Here, it is important not to restrict possible line break locations too much; otherwise, the target width may be exceeded too often.

B. Label Management Strategies Specific to SCCharts

The label management strategies just introduced are basic enough to be applied to any visual language, regardless of whether it employs natural-language labels or more formally structured text. Knowledge about the latter's semantics, however, may open up more possibilities for label management that go beyond the basic strategies. As examples, we introduce three such strategies we have developed for SCCharts, as summarized in Table II.

The first strategy concerns transition priorities, which define an ordering among all transitions leaving a given state and are thus only of interest if the user's focus is on that state or on one of the transitions leaving it. If this is not the case, they can usually be hidden altogether. While this will not reduce a diagram's overall size much, it will to a certain degree reduce the amount of detail and thus the amount of visual clutter.

TABLE II
SCCHART-SPECIFIC LABEL SHORTENING STRATEGIES

Type	Example
Transition priorities	`max(carCount, trainCount) > 1`
Host code calls	`2: max(...) > 1`
Signal abbreviation	`2: max(car..., tra...) > 1`

The inverse is also possible: reducing transition labels to their priority. We do feel, however, that this does not leave enough relevant information.

The second strategy concerns host code calls, which are invocations of externally defined functions. Host code calls use a notation similar to that of the C programming language, with the called function's name followed by a list of arguments surrounded by parentheses. When trying to reduce detail, the most important information contained in a host code call may not be the exact arguments, but the name of the called function and perhaps the mere presence of arguments. A call such as `areWobblersSynchronized(tick, wobbler1ID, wobbler2ID, 0.42)` can be shortened to `areWobblersSynchronized(...)`.

The third strategy is based on the observation that signal names can become quite long, especially in more complex SCCharts. This complicates matters for the basic strategies, which will have a hard time to achieve good results. A possible strategy to solve this problem is to shorten the names of the signals themselves, both in a state's interface declaration and in all transition labels the signals appear in. This is a more radical intrusion with the potential to confuse users and should thus probably be applied conservatively. An example of where this strategy may work is if multiple signals share the same prefix. Shortening the prefix properly could still keep the signal names recognizable while keeping them shorter at the same time, although this may make it harder for users to visually scan for signal names. A simpler variant that does not suffer from this problem is to simply apply syntactic abbreviation to signal names.

C. Combined Label Management Strategies

The strategies introduced so far prompt two observations. First, whether a label should have its details reduced at all may be subject to certain conditions, as in the case of the strategy that hides transition numbers if they are not of interest. Second, strategies may end up not meeting the target width, as in the case of semantic abbreviation. The following *combined strategies* are intended to address these observations.

The first combined strategy is the *filter strategy*, which will execute another strategy if a given condition is true and will otherwise leave the label's text untouched. Such a filter strategy can be used to encapsulate the logic required to check if a transition priority should be shortened or not, thereby keeping the *when* of label shortening separate from the *how* as implemented in the actual transition priority label management strategy.

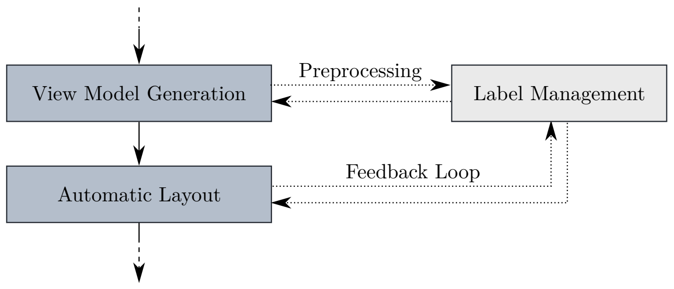


Fig. 3. The view generation process can be divided into two stages. Label management can be integrated into either stage or even into both stages, influencing the information available for making label management decisions.

The second combined strategy is the *list strategy*, which keeps an ordered list of shortening strategies and operates in one of two modes. The first mode simply executes all shortening strategies in order, with each strategy building on the result of its predecessor. This mode can be thought of as a processing pipeline. An example would be to first execute semantic abbreviation and then apply soft word wrapping to the result, thereby ensuring that the label does not exceed the target width. The second mode executes the strategies in order until one of them actually changes the label's text, at which point it stops. The idea is for this mode to be used with strategies wrapped in filters, to dynamically select from a number of possible strategies.

IV. INTEGRATING LABEL MANAGEMENT

For label management to be effective and helpful, the strategies introduced in the previous section have to be properly integrated into the development workflow. In this section, we will explore both the technical and the user experience aspects of this integration, thereby turning to the *when* and *how much* of label management.

A. Label Management and Automatic Layout

In our SCChart editing scenario, the generation of the graphical view can conceptually be divided into two stages (see Fig. 3):

1) View model generation. This stage generates representations of each element of the model to be displayed in terms of nodes and edges and determines how they will be drawn on the screen. For SCCharts, this basically means generating a node for each state and an edge for each transition as well as defining how they are rendered exactly.
2) Automatic layout. In this stage the automatic layout algorithm decides which coordinates each node will end up in and how the edges are routed between them. Traditionally, automatic layout does not change *what* is displayed, only *where* it is displayed.

It is not a hard requirement that view model generation should produce a representation for each of the model's elements. Instead, filtering techniques such as label management may well be integrated into this stage to reduce the detail of a model element's representation or leave it out of the view model completely. We call this the *preprocessing approach*: filtering decisions are made before automatic layout is invoked. The decisions made here can be based on a variety of conditions: what elements are currently selected, what mode the user has put the tool in, and what task the user is currently trying to accomplish are three obvious ones.

An example for a filtering method that fits the preprocessing approach perfectly is focus and context [9]. Focus and context is based on the assumptions that the user requires both, details and an overview of a model, and that both can be provided in a single view. This can be done by showing focussed elements with more details than their context or surroundings, which is a principle that label management fits well. Different label management strategies can be assigned to the elements in the focus and to the elements in the context. Similar to what Musial and Jacobs did [8], it is also possible to assign different label management strategies to different elements in the context to reduce their level of detail as their distance to focussed elements increases.

The decision of whether an element is in the focus or in the context can be made based on different criteria. Two obvious ones are which elements the user selected, or which elements are currently active in a simulation [2]. All of these criteria are based on information available at the view generation stage.

There are, however, other conditions that filtering decisions could be based on that are not available at this stage. For example, the size of a diagram may stay the same regardless of whether a given element has its detail level reduced or not because it is not that element which is the cause for the diagram's size. Information can thus end up being filtered out unnecessarily because information about the diagram's layout are not taken into account.

Traditionally, automatic layout algorithms have the responsibility of computing coordinates for nodes and of determining the routing of edges. But it is in this stage that information becomes available that may well be used for filtering decisions. We call this the *feedback loop* approach: information obtained during layout is used to adjust the previously generated view model. This changes the role automatic layout plays in the view generation process as well as the impact it has on the result.

As an example, let us take a look at how layouts are computed for SCCharts. We use a layout algorithm called KLay Layered [12] that implements the *layer-based method* introduced by Sugiyama et al. [13]: the set of nodes is partitioned into an ordered set of *layers*, with edges only running from lower to higher layers. The nodes in each layer are placed below one another, and the layers are placed in order from left to right. Each layer is conceptually as wide as its widest node, and since during layout we represent edge labels as dummy nodes, it is often the case that the widest node represents a label (see Fig. 4). Since we cannot shorten regular nodes, the widest regular node imposes a lower bound on a layer's width. It is this lower bound that is fed back to label management as the desired target width to shorten the label to. Depending on how wide regular nodes usually are, it may be necessary to impose a lower bound on the target width to keep labels from getting too short.

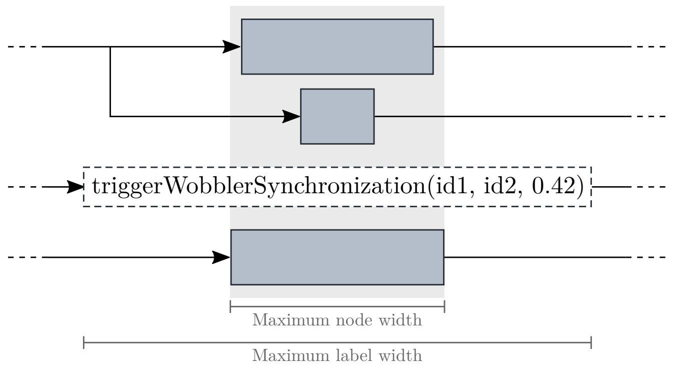


Fig. 4. Edge labels are represented as dummy nodes in our layout algorithm (white box). The maximum width of regular nodes they end up with in a layer (blue boxes) define a lower bound on the layer's width (gray background). It is this lower bound that the automatic layout algorithm uses as a target width to shorten the label's text to.

For SCCharts drawn in a left-to-right direction this approach works well. As Fig. 2 shows, however, parts of SCCharts can also be drawn in a top-down direction, either upon the user's request or based on the aspect ratio of the viewing area. The method based on layer widths then ceases to work since there are not vertical layers anymore to derive a target width from. In such cases, we currently resort to a default width to shorten labels to. However, for future research it seems worthwile to develop ways of deriving more meaningful target widths not based on layers, but on other aspects of the diagram.

It is worth noting that the preprocessing approach and the feedback loop approach are not mutually exclusive. The information obtained in both can rather be combined to reach filtering decisions. For example, preprocessing can determine how much an element's information can at most be filtered given the user's current task. Automatic layout can then decide whether and how much of the filtering is actually applied.

B. Presenting Label Management to the User

As already hinted at before, presenting changing views bears the risk of the user losing their mental map of the model. The concepts presented thus far therefore have to be carefully integrated into the user interface.

If a label was shortened or filtered out, a visual hint as to that fact may help the user recognize that there is more information available than is currently displayed. Visual hints may be as simple as adding an ellipsis to the end of an abbreviated label, or may be more complex changes to the visual appearance of the label or the element it is annotating. Whether a hint is necessary depends on the context: for SCCharts, it makes sense to indicate that a transition label was shortened because of the importance of its content to the overall model. It may be perfectly fine, however, to simply filter out transition priority labels since they are of lesser importance and it is obvious that they exist if more than one transition leaves a state.

Whenever we speak of filtering out information, this actually pertains only to what is immediately visible on screen. Filtered information can be made available through standard user interface techniques such as tool tips.

Since the amount of filtering should be adapted to the task the user is trying to accomplish, the environment has to determine what that task is. There are several ways for doing so:

1) Provide separate views, each designed to support a specific task. This can require the user to switch between views, but can also allows for having multiple views visible at once.
2) Allow the user to put the environment into modes tailored for specific tasks. The active mode then configures label management appropriately.
3) Allow the user to switch between different label management strategies to customize their view of the model. This can be either direct (the user is given the choice between, say, semantic abbreviation and semantic wrapping) or indirect (the user is given the choice between little, medium, and much shortening).

We adopted the second concept for SCCharts and provided users with rather direct choices between different label management strategies, but also supplied an option to directly control the target width of labels.

Approaches such as focus and context are more dynamic. Changing the selection in the model may also update the focus and the context and thus cause label management decisions to be revisited, resulting in view updates. For the user to keep their mental map of the model, it is necessary for these updates to keep the layout as stable as possible and to make it obvious what has changed. The former influences the choice of automatic layout algorithms; the latter can be done by animating changes over a short amount of time.

V. Evaluation

As a first evaluation of the ideas put forth in this paper, we have performed an informal survey among users of the SCCharts language and have analyzed the effectiveness of label management techniques in terms of objective aesthetics criteria.

A. Informal Survey

The survey was conducted among 35 students of a class on real-time systems. The students were asked to develop software for Lego Mindstorms-based robots using the SCCharts language with the development environment shown in Fig. 2. Label management was not explicitly advertised, but several label management strategies were readily available through the diagram options in the diagram view. At the end of the semester, the students participated in a survey on the SCCharts language and its usability that included two questions related to label management:

1) Did you have problems with long labels?
2) Did you use label management?

Regarding the first question, twelve students stated that they had problems with long labels. Several wrote that they have ignored the graphical view completely because they found long labels to make it unusable. Other students kept their models as small as possible to work around these problems. It is these kinds of statements that in our opinion indicate how important it is to find solutions to the problem of too detailed views.

Regarding the second question, six students stated to have used label management (five of which had said that they had problems with long labels in the first question). To our surprise, simple syntactical abbreviation was most popular. Most students stated that they either needed full details of a transition or a good overview of the whole model and thus mostly swiched between full details and heavy abbreviation.

B. Aesthetics Criteria

For a more objective, quantitative evaluation, we measured the effects different label management strategies had on diagrams in terms of different aesthetics criteria. We used two sets of models for this. The first consisted of models produced by the students the survey was conducted with. They submitted a total of 76 SCCharts that contained 2046 states (237 of which contained child states) and 3198 transition labels. The second set of models contained an additional 17 SCCharts that are part of a complex piece of software that controls about 10 trains on a model railway installation. The diagrams contain a total of 2058 states (120 of which contained child states) and 3032 transition labels.

We measured each diagram's width and height, the resulting aspect ratio (width divided by height), the width and height of each label, and the length of each edge. Since one of the ultimate goals of label management is to be able to increase the zoom level a diagram is displayed with, we calculated the zoom level required to fit each diagram into an 800 by 600 pixel area (note that the area's actual size is less important than its aspect ratio). The higher the zoom level, the more legible the labels (and other diagram features) are for the user.

We performed all of these measurements without label management as well as for a subset of the label management strategies summarized in Table I and in Table II: syntactical abbreviation, a form of semantical abbreviation that only lists the signals involved in a transition's trigger or action, a form of semantical wrapping where syntactical soft wrapping was applied to trigger and action independently, with a line break inserted between the two, and the removal of arguments to host code calls. For this evaluation, we concentrated on the feedback loop approach to label management integration and thus always had the layout computed in a left-to-right direction. It can be argued that this makes the unfiltered diagrams very wide compared to applying different layout directions to different parts of the diagrams, as is the case in Fig. 2. However, we think that this kind of evaluation does provide a solid indication as to how effective label management can be. We shortened every label that was wider than the nodes in its layer, which applied to 97% of all labels. However, we

TABLE III
DIAGRAM METRICS (AVERAGES)

Metric	Label Management				
	Inactive	Syntactical Abbreviation	Semantical Abbreviation	Semantical wrapping	Host code calls
Diagram width (pixel)	7322.4	3807.7	4970.5	3900.8	6932.8
Diagram height (pixel)	644.5	637.3	642.0	1481.8	645.0
Aspect ratio	12.8	6.2	8.5	3.2	12.1
Label width (pixel)	281.6	95.6	159.4	94.8	266.1
Label height (pixel)	13.0	13.0	13.0	53.6	13.0
Edge length (pixel)	653.9	291.4	432.6	419.1	627.8

limited labels to a minimum width of 100 pixels to keep them from becoming too short.

Table III shows the basic results for diagrams produced without label management, with syntactical abbreviation, with semantical abbreviation, and with wrapping. As we would expect, the width of the diagrams is decreased when switching label management on. Depending on whether the strategy can increase the height of labels, the height of the diagrams either increases or stays constant. Note that syntactical abbreviation defines a lower bound on how narrow diagrams can become since it always meets the target width. Wrapping also seems to be good at reaching the target width, but increases the average label height by a factor of four. Since SCCharts are usually very wide, this is not a problem and in fact makes them more suitable for common screen aspect ratios. Note that the average aspect ratios can only be seen as a rough guide since the actual aspect ratios vary considerably between the diagrams.

As Fig. 5 shows, the different label management strategies hat quite different effects on the achievable zoom level increase. Syntactical abbreviation had the largest effect, which makes sense in that it is the only strategy that will always achieve the desired target width without increasing the height of labels. However, the wrapping strategy came close even though it does increase label height and is the only strategy that does not reduce the amount of information in a label. Again, we attribute this to the fact that SCCharts tend to be rather wide in terms of their aspect ratio. Making them narrower but higher makes them more suitable for displayal on a computer screen. Replacing transition labels by the names of the involved signals has a noticeable effect as well. Only removing the arguments of host code calls is not very effective as the sole label management strategy and should thus probably be combined with other strategies.

VI. CONCLUSION

Building upon work by Fuhrmann [4], we have introduced label management, different strategies for its implementation, as well as ways to integrate it into the overall view generation process. We evaluated the performance of the different strategies both in an informal survey as well as with more objective aesthetics criteria. We found that label management indeed helps to increase the zoom level at which diagrams can be displayed, thereby increasing legibility if the full diagram is fit on the screen.

To us, the most surprising result is that while syntactical abbreviation was able to achieve the best results in terms of diagram size, the wrapping strategy came close and has the potential advantage of not hiding any information. However, lossy strategies remain valid as label management not only helps to reduce diagram size, but can also help to hide information not relevant to the user's current task.

Of course, there are several things left to be done. First, we have limited our discussions to edge labels, mainly due to the nature of the automatic layout algorithm we use. Of course, node labels and port labels could be shortened too and it would be interesting to investigate which shortening strategies best to apply to them. The same point can be made about comment boxes in diagrams, which can become quite large and may be prime candidates for shortening.

Second, more advanced shortening strategies should be evaluated. Interesting options here would be to investigate dictionary-based strategies that could be used for hyphenation or proper abbreviation of words, and to integrate more knowledge about the semantics of a label.

Third, while we have mentioned focus and context and graphical fisheye techniques and have described how they fit into label management (or vice versa), we have not investigated the full potential of their combination yet.

And finally, our main evaluation was based on objective layout aesthetics criteria as well as on an informal survey. A full user study is necessary to properly evaluate how label management helps users accomplish different tasks. This is especially true since the evaluation shortened all labels equally. It thus did not capture how diagram size changes if different label management strategies are applied to different labels, as would usually be the case in focus and context scenarios.

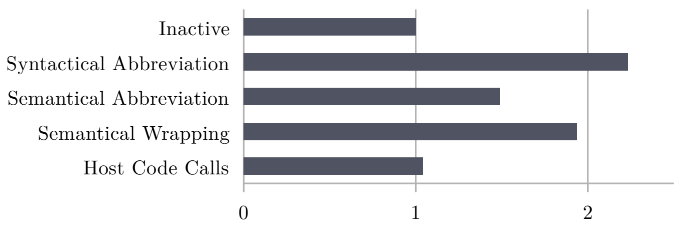


Fig. 5. Increase of the zoom factor we achieved for our set of evaluation models with different label management strategies as compared to without label management.

References

[1] T. Reenskaug, “Models – Views – Controllers,” Dec. 1979, xerox PARC technical note.

[2] 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.

[3] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[4] H. Fuhrmann, “On the pragmatics of graphical modeling,” Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

[5] Y. K. Leung and M. D. Apperley, “A review and taxonomy of distortion-oriented presentation techniques,” *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, pp. 126–160, Jun. 1994.

[6] M. Sarkar and M. H. Brown, “Graphical fisheye views of graphs,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1992, pp. 83–91.

[7] K. Been, E. Daiches, and C. Yap, “Dynamic map labeling,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 773–780, Sept 2006.

[8] B. Musial and T. Jacobs, “Application of focus + context to UML,” in *APVis ’03: Proceedings of the Asia-Pacific symposium on Information visualisation*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 75–80.

[9] S. K. Card, J. Mackinlay, and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, Jan. 1999.

[10] 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.

[11] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[12] 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.

[13] K. Sugiyama, S. Tagawa, and M. Toda, “Methods for visual understanding of hierarchical system structures,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, Feb. 1981.