On the Pragmatics of Model-Based Design

Hauke Fuhrmann and Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
{haf,rvh}@informatik.uni-kiel.de
www.informatik.uni-kiel.de/rtsys/

Abstract. The pragmatics of model-based design refers to the practical aspects of handling graphical system models. This encompasses a range of activities, such as editing, browsing or simulating models. We believe that the pragmatics of modeling deserves more attention than is has received so far. We also believe that there is the potential for significant productivity enhancements, using technology that is largely already available. A key enabler here is the capability to automatically and quickly compute the layout of a graphical model, which frees the designer from the burden of manual drawing. This capability also allows to compute customized view of a model on the fly, which offers new possibilities for interactive browsing and for simulation.

1 Introduction

Linguists distinguish the syntax, semantics and pragmatics of languages. Together these three categories are referred to as *semiotics*—the study of how meaning is constructed and understood. All three categories can be applied to programming languages as well as natural languages. In the context of programming languages, *syntax* is determined by formal rules saying how to construct expressions of the language, *semantics* determines the meaning of syntactic constructs, and the *pragmatics* of a language refers to practical aspects of how constructs and features of a language may be used to achieve various objectives [1]. In this paper, we argue that the pragmatics of modeling languages deserves more attention than it has received so far. Specifically, it appears that the practical issues of how to create, maintain, browse and visualize effective graphical models have been neglected in the past. This largely offsets the inherent advantages of visual languages, makes it difficult to design complex systems, and unduly limits designers' productivity. Petre [2] quotes a professional developer as follows: "I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it."

Traditionally, "pragmatics" refers to how elements of a language should be used, e. g., for what purposes an assignment statement should be used, or under what circumstances a level of hierarchy should be introduced in a model. It is usually not considered how the practical design activities themselves (editing,

browsing, etc.) are performed—simply because this is usually not much of an issue when textual languages are concerned. There may be differences in convenience of use in different text editors, and integrated design environments (IDEs) can provide various levels of support in building and maintaining large software artifacts. However, the basic mechanics of writing or changing a line of code is rather standard and efficient. In comparison, the mechanics of editing a graphical model are much more involved, and it appears that there is much to be gained in this area. Hence by "pragmatics of modeling languages" we here slightly extend the traditional interpretation of "pragmatics" to encompass all practical aspects of handling a model in a model-based design flow, including the traditional aspect of how a model should be constructed to effectively communicate its meaning.

There are several established fields that can provide valuable input here, such as the area of human computer interaction, cognitive psychology, and the graphical layout community. For example, there are fundamental practical differences in using textual or graphical languages [1], and freeing the modeler from the burden of manually drawing a graphical model opens the door to a number of productivity-enhancing techniques that allow to combine the best of both worlds [3]. Furthermore, there are already a number of paradigms well established in software engineering that could be put to use for model-based design processes, including the design of the modeling infrastructure itself. For example, the state of the practice in creating a graphical model, say, a dataflow diagram or a Statechart, is to directly construct its visual representation with a drag-and-drop (DND) What-You-See-Is-What-You-Get (WYSIWYG) editor, and henceforth rely on this one representation. We here propose instead to apply the Model-View-Controller (MVC) paradigm [4] to separate a model from its representation (view). Together with a modeling environment (the controller/editor) capable of automatic model layout, one can thus provide flexible representations. These views can be adapted according to specific design activities, balancing useful information with cognitive complexity [5].

In this paper, we survey the different aspects of the pragmatics of graphical modeling languages. This covers a broad range of existing work, as well as a number of observations and proposals that to our knowledge have not been reported on before. As space is limited, we do not attempt to investigate any of these aspects in much detail here, but rather try to cover as much ground as possible. A non-trivial question at the onset was how to organize the subject matter. There exist extensive surveys in the area of model-based design, see for example Estefan's overview of model-based systems engineering methodologies [6], or the overview of hybrid system design given by Carloni et al. [7]. An annotated bibliography by Prochnow et al. [8] inspects the visualization of complex reactive systems. There exist numerous surveys on automatic graph drawing, which we consider an essential enabler for efficient modeling [9,10]. However, we are not aware of an existing taxonomy that focuses on the aspect of pragmatics. We here opt for the aforementioned MVC concept as a guiding principle. For a first overview, see Fig. 1. In some cases, it may be arguable how a certain aspect

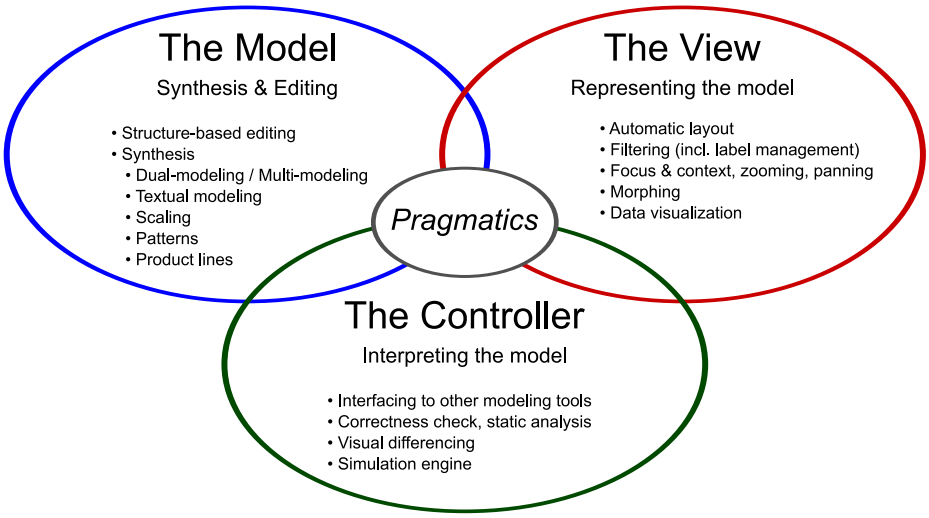


Fig. 1. The MVC paradigm applied to the pragmatics of graphical model-based system design.

should be classified; e. g., we here consider editing to be part of the model, but it could also be classified as part of the controller. However, we still find the MVC classification helpful.

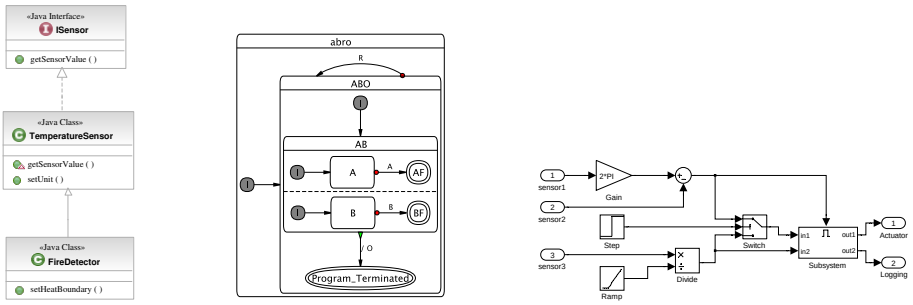
This structure is also reflected in the organization of this paper, except that we start with the view (Sec. 2), followed by the model (Sec. 3) and the controller (Sec. 4). We conclude in Sec. 5.

Example figures in the following sections are mainly taken from different graphical modeling tools like Mathwork's Matlab/Simulink, Esterel Technology's E-Studio and SCADE and graphical editors basing on the Eclipse platform. None of the tools handles pragmatics very well so far, so the images are mainly for illustration of the concepts but not for showing the state-of-the-art in implementation. An implementation of the concepts presented in this paper is ongoing work in the project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1].

2 The View—Representing the Model

We believe that a key enabler for efficient model handling is the capability to automatically compute the layout of a graphical model. If one frees the user from the burden of manually setting the coordinates of nodes and bendpoints, sizes of boxes and positions of connection anchor points, this can open up enormous potentials. The following section explores this further.

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



(a) UML Class Diagram (b) Statecharts (here SyncCharts) (c) Dataflow diagrams (here Matlab/Simulink)

Fig. 2. Different graphical syntaxes with different properties for their layout.

2.1 Automatic Layout

The correct use of pragmatic features, such as layout in graph-based notations, is a significant contributory factor to the effectiveness of these representations [2]. Automatic layout has to be appealing to the user such that he or she is willing to replace optimized manual layout with an automatically created one. Additionally this layout capability would have to be deeply integrated into the modeling tool and optimized for the respective graphical language syntaxes.

One must recognize that at this point, the automatic layouting capabilities offered by modeling tools, if they do offer any capabilities at all, tend to be not very satisfying. A major obstacle is the complexity and unclarity of this task. What are adequate aesthetic criteria for "appealing" diagrams [11,12,13]? Are there optimal solution algorithms or heuristics with acceptable results that adhere to the desired aesthetic criteria? An important aspect is the usage of *secondary notation*, which is specific to the modeling language used [14]. Used properly, an automatic layout does not only provide aesthetically pleasing diagrams, but can also give the viewer valuable cues on the structure of a model. For example, a standardized way of placing transition labels (e. g., "to the left in direction of flow") can solve the often difficult label/transition matching problem. Similarly a standardized direction of flow (e. g., "clock wise") can give a quick overview of the flow of information, without having to trace the direction of individual connections.

Fig. 2 shows three examples of different graphical formalisms that pose different layout challenges. Unified Modeling Language (UML) Class Diagrams look quite close to the standard graph layout problem, although sometimes hierarchy might be added by displaying packages in the diagram. While usual relations can be regarded as any graph edges, inheritance relations as shown in Fig. 2(a) have a special role. They are typically drawn from top to bottom, which is a strong constraint for the layout algorithm. So even here one needs a specialized layout algorithm for this diagram type [15].

Statemachines fit pretty well to graph layout, but introducing hierarchy requires special handling. In a diagram with hierarchy and without any inter-level connections crossing hierarchy boundaries, the layout algorithm for a flat layout can be called recursively. This was employed for Statecharts as shown in Fig. 2(b) using the layered based Sugiyama layouter of the GraphViz library in the KIEL project [3]. Small enhancements of the graphical syntax might have severe consequences for the layout. Inter-level transitions, which are possible in some Statechart dialects as UML State Machine Diagrams or Stateflow of Matlab/Simulink, cannot be layouted with this approach and would require a special handling again.

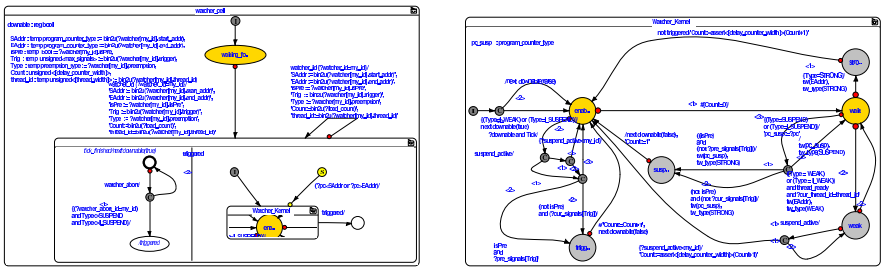
Another special class are actor oriented dataflow languages [16]. The notion *dataflow* sometimes is used in different contexts resulting in different diagram syntaxes. We here consider languages usually used in the control engineering domain such as Ptolemy, the Safety Critical Application Development Environment (SCADE), or Matlab/Simulink (Fig. 2(c)). The connections denote flows of data and two distinct connections will likely carry different data and possibly different data types. Data are consumed by operators, and to distinguish the different incoming and outgoing data sources and sinks, an operator has special input and output *ports*. For many operators it is very important to specify explicitly which data flow is connected to which port because an alternation would also alternate the semantics. The example shows subtraction, division and switch operators which are not commutative and hence need their incoming flows exactly at the right input ports. The graphical representation also reflects this issue by presenting specific anchor points for the connections at the border of the operators. For the mentioned languages these ports have fixed positions relative to the operator, usually showing the data flow from left to right by positioning inputs left and outputs right. However, some special purpose ports may also be positioned on top or bottom of the operator, in general at pre-defined and static locations. These *port constraints* induce a great complexity to the problem and require special care such as by the approaches of Eiglsperger et al. [17] or a modified Sugiyama layout as implemented in the KIELER project[2].

Summing this up, we cannot hope for one ultimate layout algorithm that is applicable for all languages and applications. Instead, we need a set of different layouters to cover a wide range of language syntaxes and layout styles.

2.2 Filtering

Card et al. [18] define approaches for reducing information in a diagram: *Filtering*, *Selective Aggregation*, *Highlighting* and *Distortion*. A filter simply hides a set of objects in the diagram, to reduce the complexity of a diagram. For technical scalability issues it is often not feasible to construct and inspect models with many objects—hundreds or thousands of nodes—but consistently working with filters it can be. Only a small set of objects should be visible while all others are hidden and do not consume graphical system resources. By navigating through

[2] `http://www.informatik.uni-kiel.de/rtsys/kieler/`



(a) Top-level state (b) Inner state

Fig. 3. Filtering in E-Studio, showing a part from a processor design [19]: The composite state in the lower right corner of (a) displays only a very small part of its inner life (b).

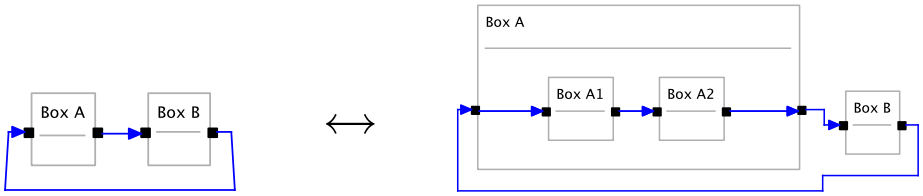
the model, a user reveals some parts and hides others. In order to properly work, we need strategies to apply this automatically to free the user of the burden to manually selecting the items to show and to hide. (This also leads to the focus and context paradigm, see Sec. 2.4.)

Simple filters can already be found in some tools that hide objects on the canvas while the canvas size resp. the bounding box stays the same size. Hence only the number of elements is reduced but not the size and therefore the same zoom level or paper size is required to display the model and there is hardly any chance to see more of the surrounding context as before. A rather unusual way of filtering can be used in Esterel Studio, see Fig. 3. The hierarchy mechanism in E-Studio allows to create the relatively clearly arranged top-level diagram Fig. 3(a). However, the macrostate Watcher Kernel in the lower right reveals only a very small part of its contents, the rest is hidden. One has to manually open the Watcher Kernel state in a new canvas in order to see its whole extend shown in Fig. 3(b) where a complex inner life is revealed compared to what small part of it is shown in the parent. This feature becomes more useful in combination with automatic layout that uses the free space gained from the filters.

Dynamic Visible Hierarchy Dynamic hierarchy is a special case of a filter where all children of some parent object are filtered. For filters one might select to hide items regardless of the hierarchy level to reduce the complexity, see Fig. 4 for a simple example. This corresponds to the *folding features* of text or XML editors [20].

2.3 Label Management

Working with real-world applications quickly leads to the question of how to handle long labels. Label placement is a big issue in graph drawing [21] and geography with map feature labeling [22]. The problem is computationally intensive,



(a) Composite Box A folded

(b) Composite Box A unfolded

Fig. 4. Example for dynamic visible hierarchy, here for an actor oriented data-flow language implemented in KIELER. This utilizes collapsible compartments and a layer-based automatic layout algorithm supporting port constraints.

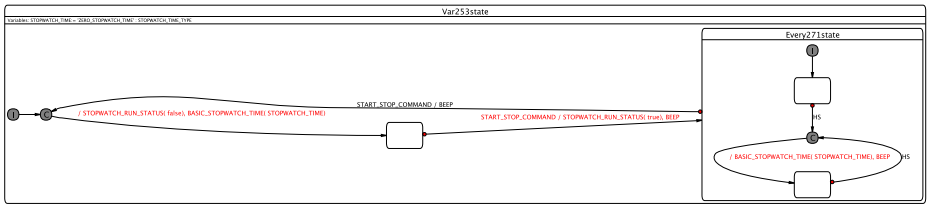


Fig. 5. Long labels prevent good layout. Here a small part of Harel's wristwatch example [25], converted from Esterel to Statecharts [26].

for bended edges it is NP-hard [23]. In map labeling labels are rather short—city, street or river names—but in arbitrary Domain Specific Languages (DSLs) they do not need to be, as Fig. 3(a) shows. We assume to have an automatic layout algorithm that takes care of the label positioning, taking the label as is and not changing it. There are innovative approaches changing the diagram syntax, e. g. to replace an edge by the label itself [24] by optical scaled down distortion. However, we do not consider such invasive changes as universal option.

Instead, we try to dynamically reduce the complexity of the label to give the layouter better chances to find appealing layouts and to avoid difficulties as illustrated in Fig. 5. A *label filter* might use different strategies, see also Table 1. *Wrapping* aims to compact the label by wrapping the text while *abbreviation* hides part of the text to actually shorten the length of the string. Syntactically arbitrary labels might be handled with possibly suboptimal results. Even soft wrapping respecting identifiers will wrap compound labels inappropriately by not keeping related identifiers on one line. Semantical abbreviation could hide specific token types while showing only more important ones, like operators vs. variable/signal references. With a *label manager* in charge, the labels can be dynamically displayed with different levels of detail.

Table 1. Ways to reduce label complexity temporarily. Here for a Statechart transition label.

Original	`(not SignalA) and (not SignalB) / SignalC(counter)`		
Wrapped	syntactical	hard	`(not SignalA) and (n ot SignalB) / Signal C(counter)`
		soft	`(not SignalA) and ( not SignalB) / SignalC( counter)`
	semantical		`(not SignalA) and (not SignalB) / SignalC(counter)`
Abbreviated	syntactical		`(not SignalA) and (not Si...`
	semantical		`SignalA, SignalB / SignalC`

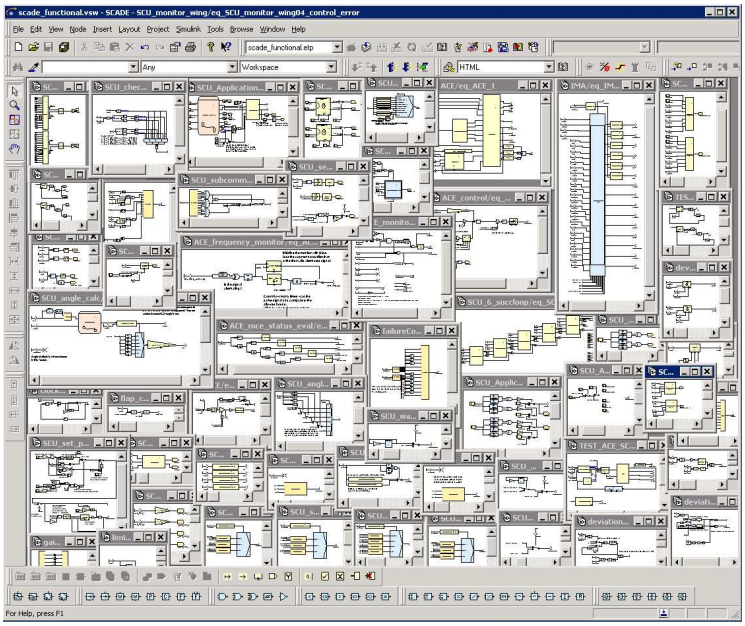
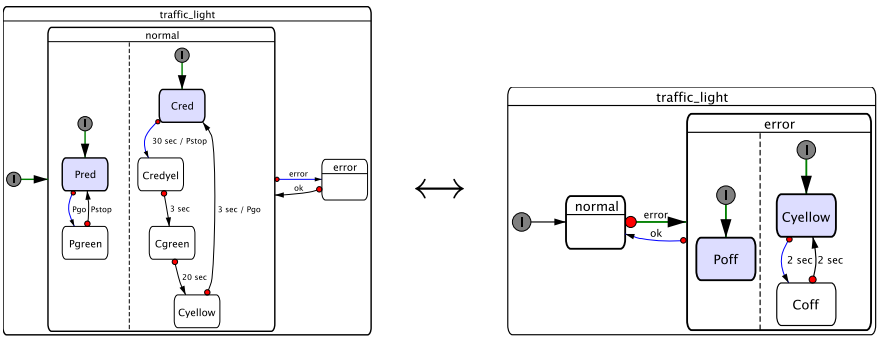


Fig. 6. Illustration for the lack of proper view management: showing the whole system entails loosing details, windows get to small to be usable.

2.4 Focus and Context

In classical modeling environments, the user typically has the alternatives of either seeing the whole model without any detail, or seeing just selected parts of the model. Fig. 6, from an avionics application, shows what may happen if one does try to see the whole system. To find a way out of this dilemma, we



(a) View with focus on state normal (b) View with focus on state error

Fig. 7. Semantical graphical focus and context in KIEL: The two large composite states normal (a) and error (b) are only displayed in full detail when the respective state is active. The inactive state is filtered by dynamic hierarchy and forms the context.

note that when working with a model, it is common that there are parts of the model of particular interest for the current operation or analysis, which we refer to as the *focus*. Other objects next to it comprise the *context*, which might be important information to understand the focus objects but may be displayed with less details. This leads to a *focus and context* approach where filters are employed to hide irrelevant objects [27].

Focus and Context in KIEL The Kiel Integrated Environment for Layout (KIEL) project [3] uses a semantical graphical focus and context technique to hide details in the context while highlighting the focus. It is *semantical*, because the decision of objects to be filtered is made automatically from semantic background information from the model [27]. The concept is used during simulation of a Statechart diagram where the focus seems to be quite natural to state machines: the currently active states. Hence all active simple states are displayed together with their whole hierarchy, i. e. all ancestor states (which actually are also active). Dynamic visible hierarchy, as presented in Sec. 2.2, is used to show all sibling states but hide their contents. Hence, tidy diagrams are presented always with reduced complexity as shown in Fig. 7.

During simulation the user never sees the whole diagram but only either one of the focused views. Smooth animated morphs between the views guide the mind of the user from one view to the other so he or she can keep the *mental map* of the whole application [28].

Alternatives for Focus and Context Experience showed that KIEL's specific interpretation of what objects comprise the focus and which the context is not always optimal. Sometimes it is difficult to follow the reasons of the view change,

i. e. the switch from state normal to error. Signals emitted in the collapsed state can cause this change but are immediately hidden and hence the user cannot follow the causal event chain. This calls for a more general approach for applying focus and context techniques. Even for this specific DSL one can come up with various other schemes to select the focus objects.

- One could show an intermediate step between the transition where the former active and the new active states are focused both.
- One might decide the focus by active transitions instead of states—this could also filter parallel regions that do not change configuration.
- The context does not necessarily must go up to the top level, but might be limited to some number of hierarchy levels.
- *Meta focus:* one could specify a more abstract focus, e. g. "focus on signal S," which would set the concrete focus on transitions/states that reference S

2.5 View Management

Considering diagram types other than Statecharts, it is not that obvious how to select the focus of the diagram, because there might be no such thing as an active state—e. g. in dataflow diagrams sometimes all operators are active in every step—or there is no visible step-wise simulation at all—e. g. structural diagrams such as UML class diagrams. To broaden applicability, it appears natural to upgrade layout information and directives to "first-class-citizens." By this we mean that the view of a model becomes part of the state of a model, which can be controlled by the user, the modeling tool, or the model itself. An engine for *view management* could for example categorize graphical entities in focus and context, maybe even multiple levels of context by setting different *levels of detail* as denoted by Musial for UML [29]. These and other aspects of view management are depicted in Fig. 8.

The view manager needs to listen to *triggers*, or *events*, at which it might change between the dynamic views, showing the user some objects in the focus and others in the context. These triggers might be *user triggers*, induced manually by the user, e. g. manually clicking on fold/unfold buttons at parent nodes or manually changing the focus by selecting a different node. They could also be *system triggers*, produced by the machine by some automatic analysis, semantical information, progress of time (real or logical), etc. *Memorized triggers* can for example be trigger annotations stored persistently with a model.

Obviously, this view manager can hardly be one monolithic application that carries all information and is applicable for all types of DSLs and application environments. We need a way to efficiently specify both the *triggers* to listen to and the *effects* that shall be performed. This *view management scheme* (VMS) needs to be provided by the developer, either by the application developer for application specific schemes or by the tool creator for more general schemes applicable for a whole DSL. For a practical user interface this VMS should be expressed by a simple syntax, maybe close to some general purpose scripting language. It would require expressions to

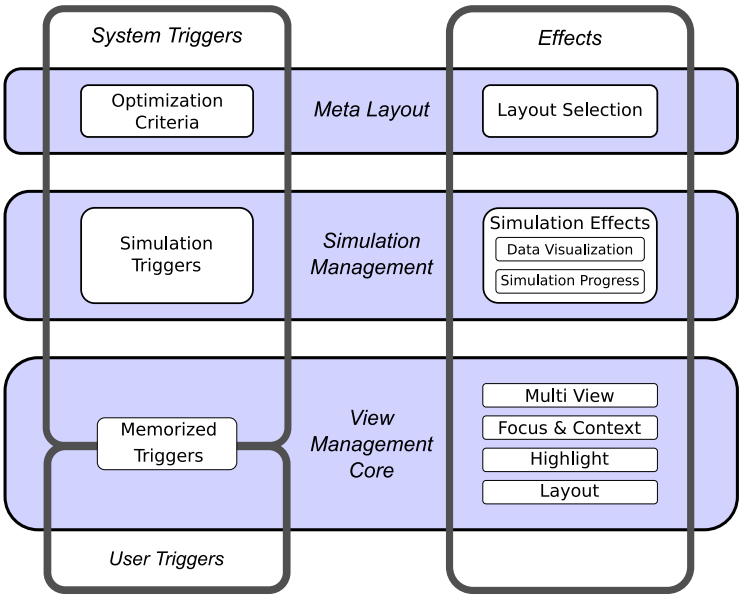


Fig. 8. Aspects of view management.

1. address different user triggers (mouse clicking, keyboard events),
2. specify custom system triggers,
3. address different system triggers,
4. address different visualization effects (folding, unfolding, filtering, layout triggering, choose layout algorithms), and
5. address graphical diagram objects or their properties, either specific objects (e. g. "State A") or classes of objects (e. g. "a node of type *state*") or specific *patterns* of such objects.

Some of the items can be implemented using standard techniques, such as addressing model elements. A set of predefined user triggers and visualization effects could be provided. It is not that obvious how to specify custom system triggers. Most of them will be very semantic-specific for a certain DSL. For example the trigger "a state has become active" in a Statechart would require interaction with the simulation engine and hence cannot be implemented only with the knowledge about the certain DSL meta-model and the modeling and visualization framework. Therefore an interface to the "outside" is required, the respective lower level programing environment of the modeling tool.

Such a view management engine could be employed to handle the ideas of semantical focus and context in a general way. It should also allow, via user triggers, to quickly navigate manually through a model, using for example *semantic zooming and panning* where one considers the structure of a model to navigate through it. For example, one would not just change the zoom on a linear percent-

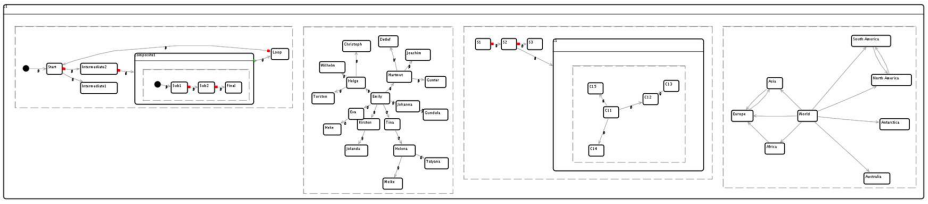


Fig. 9. Meta Layout: Multiple different automatic layout algorithms applied in one diagram, here from left to right GraphViz layered based Sugiyama Layout, the Zest Spring Embedder, a layered layout with radial layouter in child and GraphViz Circo [31].

age scale as is commonly the case, but could also change the zoom by hierarchy level.

2.6 Meta Layout

For a given graphical DSL there might be different layouts for the graphical representation conceivable. There may be different automatic layout schemes available, either the same algorithm but with different parametrization options, or completely different layout algorithms. Each layout algorithm results in a different layout style. We denote the process of selecting and combining different already existent layout algorithms as *meta layout*. This should be integrated into the view management.

Note that this is somewhat contradictory with the concept of having a *normal form* [30], where models with the same domain model will have the same graphical representation. The motivation for normalization is to limit ambiguity and subjectiveness when creating or analyzing diagrams. However, it may be hard to find one layout algorithm that provides optimal layout results for all possible applications—even within one DSL. So we may soften the idea of normalization by varying degrees. One could apply different layouters (1) to different models, (2) within one model, in different hierarchy levels and (3) within one model, in different regions of the same hierarchy level (see Fig. 9).

Layouter Choosing Strategies Having multiple layouters and different regions in the diagram, a question arises: When to apply where what layout? This is answered by *layout choosing strategies*.

The simplest strategy could be to let the user decide. The user manually annotates each part of the model with the specification of which layouter should be used. This way the user would be able to select the best layouters according to his or her personal subjective aesthetic criteria. Additionally, the user could consider application and system specific properties when choosing the layouters.

For a larger benefit, the modeling tool could assist in choosing the right layouter settings by trying to optimize the layout result. The optimization criteria should be provided by the user while the machine should be able to work with them. Possible criteria are *syntactic aesthetic criteria* such as link crossings, link lengths, diagram area, aspect ratio; *semantic aesthetic criteria* such as alignment, symmetry or zoning [32,33]; prescribed *development patterns*; or *model element types*, e. g. graph-based vs. port-based.

3 The Model—Synthesis and Editing

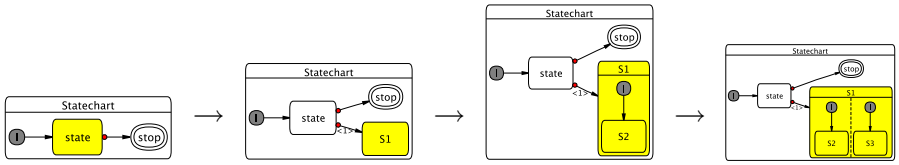
A graphical model is nice to look at, but can be effort-prone to create or change. Common editors have the paradigm of WYSIWYG DND interaction. In general it is desirable to immediately get visualized effects of editing steps in WYSIWYG. However, the way of interaction—DND—is the source of plenty of additional manual editing efforts. Strictly speaking, the term *drag-and-drop* (DND) denotes a specific sequence of steps including the dragging of elements. However, we will refer to DND for all DND *style editing* in current modeling tools. This includes all manual layout positioning of objects on the graphical canvas such as the placement of nodes and edge bendpoints, moving and resizing. Even moving an object by selecting it first and using the arrow keys on the keyboard falls into this category.

We advocate to try to avoid the tedium induced by DND editing as much as possible, to put back into the focus the system instead of its graphical representation. The basic enabler is the aforementioned capability for automatic layout (Sec. 2.1). One issue here again is the preservation of the mental map of the modeler. In the context of model editing, there exist different schools of thought. One direction argues that the appearance of a model after an edit should be changed only minimally, to preserve the mental map [34]. The other approach is to try to give models a uniform appearance, that "the same should look the same," proposing a *normal form* that is independent of the modeler and the history of the model (see also Sec. 2.6). There the issue of mental map preservation is addressed during the editing step by a morphing animation of the model.

3.1 Structure-Based Editing

The idea of structure-based editing comprises only structural decisions of the developer, which are (1) to select a position in the model topology and (2) to select an operation to apply to the model. This changes only the structure of the model, i. e. its topology, sometimes also referred to as the *domain* or *semantic* model.

The graphical representation also can be updated immediately. The automatic layout has to be applied to create a fresh *view* of the new structure of the model after the user operation. The complexity of the model and the performance of the layout algorithms determine whether it is feasible to apply the



(a) Simple state is selected (b) New successor state added (c) New state is upgraded (d) After another upgrade

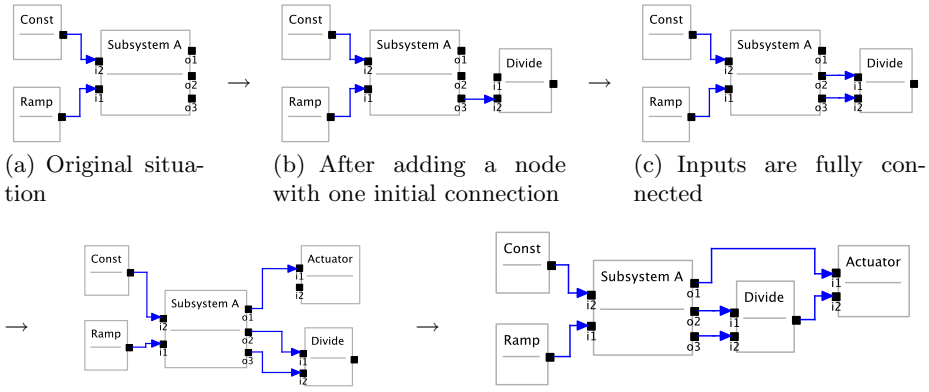
Fig. 10. Example for structure-based editing of a Statechart.

layouter after every small editing step in order to get immediate visual feedback. Therefore we eliminate the DND style editing but possibly keep the WYSIWYG nature of the editor. We believe that this immediate visual feedback is valuable enough to put a premium on fast layouting algorithms, even if this might give slightly sub-optimal results.

Structure-Based Editing for Graph-Based Models For DSLs that are based on graphs we gained some experience from the KIEL project, which applies this paradigm to Statecharts. Graphically they consist of states (nodes), transitions (edges), hierarchy and parallel regions. In this case only a small set of different structural operations are required to create or modify the charts. For a selected state these are only (1) *create a new following state* and (2) *upgrade the state*, as shown in Fig. 10. For transitions the operations are only (1) transition creation and (2) to reverse a transition. Some other "syntactic sugar" can be provided, but nevertheless the operation set is relatively small. Other changes to the model are done afterwards, e. g. changes of labels by filling out form fields.

This paradigm would also apply for other graph-based DSLs because the set of affected model elements in every step is small—up to two. For node operations one node needs to be selected, for edges there are two nodes, source and target.

Structure-Based Editing for Port-Based Models For *dataflow* models with *ports* (cf. Sec. 2.1) the case is a bit more complex. Especially adding new nodes requires more specification than a simple operation like "add a successor node" can provide. In a graph-based model this operation will generally transform one valid model to another valid model, because it can add a new state and simply connect old and new state with a transition. Port-based models have stricter connection requirements. In general there is an arbitrary set of different kinds of operator nodes; usually this node library is also extensible by the user. Each node has a certain *interface*, i. e. the set of input and output ports that specifies how the node must be connected to other nodes. Hence a new node in the model likely requires not only one but multiple connections which have to be specified not only between the nodes but between specific ports. There are different ways possible for the user interface in this case.



(a) Original situation

(b) After adding a node with one initial connection

(c) Inputs are fully connected

(d) After adding another node

(e) Connections complete

Fig. 11. Possible structure-based editing steps in a port-based language.

In the first approach the goal is to still provide the diagram itself as the user interface. To support incremental editing, the operation to be performed can be divided in small incremental steps where each does not necessarily lead to a valid dataflow model because it might be not sufficiently connected. After every step the view manager can update the layout and some meaningful graphical representation of the intermediate step is created. An example sequence of such operations is shown in Fig. 11.

In this scenario, the set of operations to connect ports determines the efficiency of creating or editing models. Shortcut operations to connect multiple ports can help to reduce the manual steps. For example the SCADE editor provides the operations *connect by rank* and *connect by name* which will interconnect all inputs of one with the outputs of another selected node either by name of the ports or successively by their rank. In SCADE this is not post-processed with the view management, but this can give a first inspiration for the type of connection operations that are helpful.

The operations can be hard-coded for each language or language class. Additionally, the paradigm can be used in conjunction with model transformation frameworks. Especially *in-place transformations* change the underlying domain model by pattern matching where source and target meta-model are the same. Hence the original model is only changed instead of transformed into another DSL. Therefore an in-place transformation framework such as from Taentzer et al. [35] can be used to specify the transformations while the view management with automatic layout adds the graphical feedback to get the full WYSIWYG experience.

3.2 Modification and Deletion

For all possibilities of model changes, the set of model operations must be augmented by operations for removing nodes and connections. Additionally a set of syntactic sugar operations should be provided to manipulate the models efficiently, e. g.

- replace a node by another node of another type,
- replace a connection by a different connection type,
- redirect a connection, or
- insert a new node into one or multiple connections if the port rank fits—i. e. break up the connection into two parts, insert the new node and connect the input and output to the connection endpoints.

This can reduce manual steps especially by keeping attributes of the objects that were manually set after the object creation.

Error Handling We should learn from best practices in textual programming IDEs and try to adopt features to graphical modeling. For example the *Quick Fix* feature of Eclipse allows beginners to learn textual programming—e. g. Java—in an interactive tutorial-like way. Errors are displayed immediately with the help of incremental continuous compiling. Additionally the UI presents a list of possible solution operations which can be triggered by the user.

Features like this can be incorporated into graphical modeling by orchestration of different building blocks. There are generic modeling frameworks that support model validation such as the Eclipse Modeling Framework (EMF) with its Validation Framework[3]. Hence it is possible to consequently feedback the information about the model consistency to the user. For specific DSLs there should be a set of standard error cases provided together with a set of possible solution operations, again supported by automatic layout of the created solution model.

3.3 Synthesis

With an automatic layout capability, it is not only possible to change models interactively with the developer. One can also synthesize completely new graphical models, including the domain model and its graphical representation. There are multiple scenarios where this *model synthesis* can be of significant benefits and lead to innovative modeling environments.

Textual Modeling An alternative to the graphical representation of a model still is text. Having information in a textual representation can have many advantages [1,26]. There are already well accepted approaches for *textual modeling* available such as the Textual Concrete Syntax (TCS) [36] or Xtext [37], both

[3] `http://www.eclipse.org/modeling/emf/`

frameworks for Eclipse. The developer specifies the meta-model of the DSL and the textual syntax and the framework generates parsers and textual editors. The latter are equipped with convenient features like syntax highlighting, auto-completion, static analysis, structure outline view, source code navigation and folding. Textual models will be parsed into the actual domain model data structures so they can be processed like all other domain models.

The missing link is the one to a graphical model. Here, automatic layout and view management can be used to synthesize the graphical representations from the textual ones. This can be done in different levels of integration:

1. A graphical model is only once *initialized* from the text. Afterwards the graphical model is worked on. Usually there is no way back into the textual model; an exception here is Eclipse.
2. There is a transformation between textual model and graphical model in both directions. This is usually denoted as *round-trip engineering*. Some dedicated commercial tools support this for special DSLs, usually class diagrams, but this is still uncommon.
3. The tightest integration perfectly synchronizes textual and graphical representation. Hence the user sees two different views and every change in either of the views automatically updates the other view. So working in the views is interchangeable even for small steps. This paradigm has been explored in KIEL for Statecharts and is applicable for other DSLs as well.

To increase the integration further, text and graphics could be mixed in one view. If there is a textual representation for single graphical objects, there could be two different views of the graphical model. One view displays all graphical entities while the other exchanges one of the objects with a text box containing the textual representation of only this model part.

Scalable Models Model synthesis can be applied together with scripting techniques to create complex and large models according to predefined and parametrizable patterns. Scripts of different flavors could be applied just like *scripts*, *macros* or *templates* in textual languages. This leads to *scalable models*, as investigated in Ptolemy [38]. In this case the scripts that configured the model creation process are in the same graphical syntax as the models themselves. More sophisticated automatic layout techniques could enhance the graphical results. This approach could be applied more generally for arbitrary DSLs and combined with an appropriate user interface.

Pattern-Based Modeling Development patterns are a common technique in software engineering. When creating behavior diagrams such as Statecharts or dataflow models, one should model common tasks in a common way. This naturally leads to *patterns* for graphical modeling [39,40]. Examples are patterns for error handling, sequencing or loops—depending on the DSLs, many more can be identified. Graphical modeling environments could support the usage of pattern-based development in various ways.

– Design patterns can be highlighted in a model [41].
– A specific pattern can be chosen by the user and parametrized to be added to a graphical model.
– The view management should support user defined automatic layout schemes according to a given pattern. If in a state diagram a loop should be modeled, this could correspond to a pre-defined graphical positioning of the nodes, e. g. in a circle or in a sequence with one back transition.
– Analysis of the model could detect certain patterns for standard operations such as graph transformations [35,42]. Additionally it should be able to layout existing patterns to given pattern layout schemes.

A simple user interface is necessary so even beginners and intermediates can quickly start to employ patterns in their development.

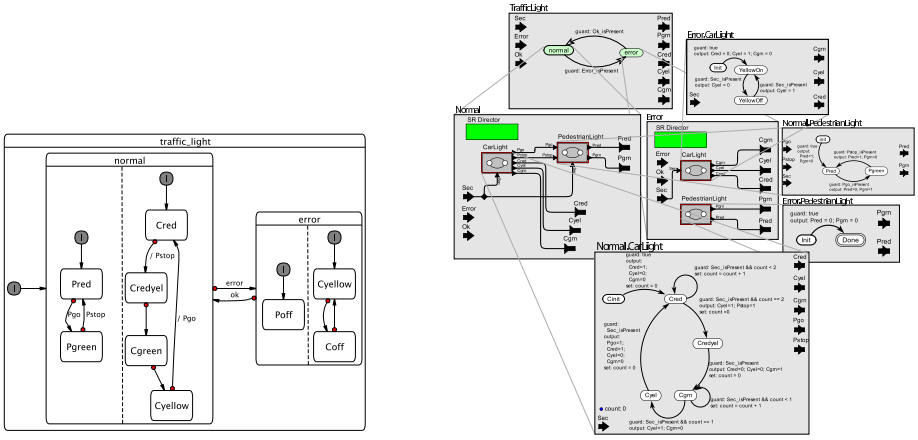
Product Lines Another use case for proper view management and/or model synthesis are *product lines*. Here, a set of closely related products is offered, where each product likely differs only by some specialization or configuration from the others. For textual programs, the source code comprises all features, whereas the build process configures different target products with different features deactivated. This could be analogous to the use of pre-processor macros in textual languages, where e. g. an #ifdef macro can hide parts of the program source.

A graphical model can also serve as a *master model* for a product line. To investigate one of the target products and further processing, the final *product model* should be accessible as any other model. To avoid the maintenance of multiple models, the product model should be synthesized from the master model and comprise only the elements necessary for the features of the product. This means omitting certain model objects of the master or configuration of scalable model parts. In certain cases this can be augmented by static analysis to identify the required model parts automatically, e. g., by deactivating superfluous outputs.

3.4 Multi-View Modeling

So far we were considering multiple views only within the same DSL in order to change the levels of detail in certain circumstances to get the best trade-off between overview and details. One can drive the idea of multi-view modeling further by defining completely different views instead of only manipulating the focus and context configuration.

The term *multimodeling* is referred to employing multiple modeling semantics in one single model [43]. For example mixing different semantics such as synchronous data flow with state machines and discrete events or others is a preeminent feature of the Ptolemy modeling framework. This still keeps only one view on the same model, although the model itself is of very heterogeneous character. However, one can for example establish semantical equivalence



(a) Statechart model (b) Ptolemy model

Fig. 12. From Statecharts to Ptolemy: both models implement the same behavior [43].

between Statecharts and mixed synchronous reactive and state machine models [43]. Hence for the same semantics, there exists a Statechart and a Ptolemy model that implements that behavior. This means for the same semantical behavior there exist multiple different graphical representations, each with their advantages and disadvantages. Considering the example in Fig. 12, one might argue that the Statechart model is more compact, but the Ptolemy model makes further information explicit, notably the information flow. We could exploit the equivalence by transforming a Statechart into a Ptolemy model or vice versa—at least for suitable Ptolemy subsets. The disadvantage would be that we still have two completely different models including two different domain models. Both models could be transformed only as whole in a global transformation of all model parts.

An alternative could be to keep only one common domain model and on top of that create two different graphical representations, one for Statecharts and one for Ptolemy. This would be always applicable where one model part can be expressed in multiple ways. Then the model part could have multiple completely different views. The major benefit would be that the different graphical representations could be interchanged in any hierarchy level resulting in a mixed graphical model. The different views could be handled by the view management just as the other views proposed above.

4 The Controller—Interpreting the Model

Sophisticated static analyses can determine properties of a model, for example causality issues for dataflow models [44]. If such an analysis determines certain

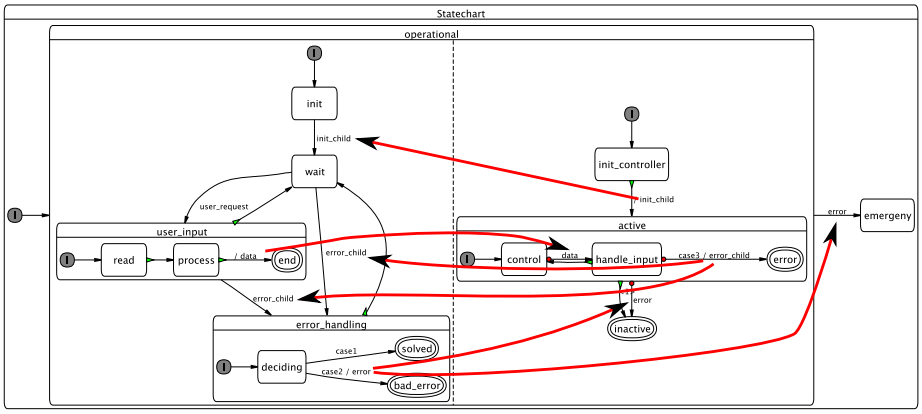


Fig. 13. Dual Model for Statecharts: Two parallel controllers communicate via broadcast. The dataflow is displayed as an overlay of the original control flow graphical representation.

properties of a set of model elements, it can be used as a trigger for the Meta Layouter in order to get a visual feedback of the analysis. Especially a categorization of model elements in two sets can be interpreted as a categorization into focus and context objects.

4.1 Dual Modeling

The graphical representation depicts the main model objects as nodes, where the containment relations can be reflected by hierarchy in the model. Explicit connections display some other relations between the model objects. However, there is typically a set of model attributes that is hidden in simple property dialogs or simply represented by a label in the graphical representation. Relations between those attributes are usually not visible.

We propose a dynamic extension of the graphical representation by its *dual model*, i. e. a graphical representation of the relations between referenced objects where this reference is not yet visualized. We again examine the example of Statecharts. The dual model of a Statechart is a graph where the transition labels are the nodes and the relations between guards and actions form the connections. The graph shows which transition produce triggers and which ones read those triggers. It makes explicit how the broadcast communication is used by showing the flow of data and signals in the model. By graphically overlaying the original graphical representation with its dual model, we reuse the same graphical view in order to keep the mind map within the user, as illustrated in Fig. 13.

The *dual model* methodology should not only be helpful for Statecharts, but applies to very different types models. References to other model parts are quite

common where an explicit graphical representation is omitted for the sake of clarity in the original model.

4.2 Dynamic Behavior Analysis

We usually distinguish the *structure* and the *behavior* of a model. To validate behavior, it is common practice to employ simulations prior to physical deployment. Therefore we employ DSLs with known specified semantics such that the models can be executed.

Simulation Management Employing the meta layouter during testing gives us the same benefits as for simple manual browsing, as interesting parts can be put into the focus while the context is still visible. Additionally, a simulation run gains a new dimension: time. Hence there might be times where nothing of relevance happens and other points in time with interesting events. The problem is to determine "interesting" parts and times during simulation.

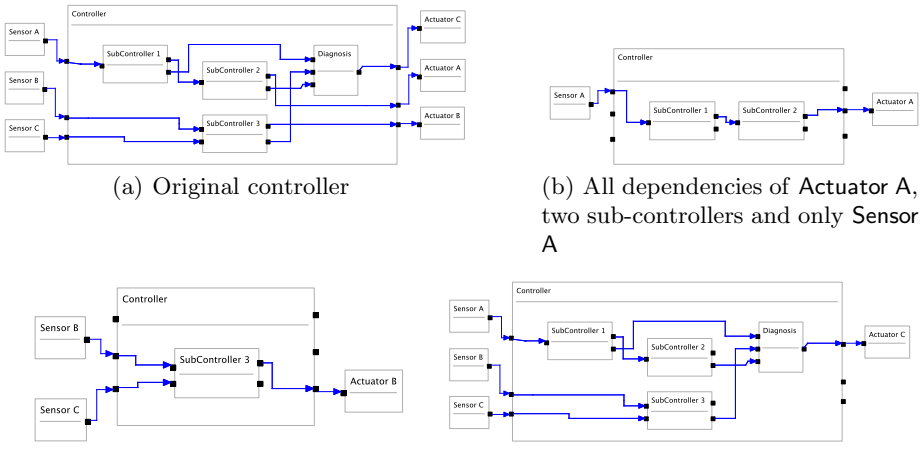
Therefore we propose to extend the meta layout view management by *simulation management*. It defines an additional set of system events for triggering view management effects and additional effects for manipulating simulation time.

Both simulation triggers and effects are highly dependent on the language semantics. Hence a simulation manager is usually only applicable for a small set of DSLs.

Visual Breakpoints Simulation *triggers* are customizable conditions over internal states and variables of the simulation. Hence both the specification and the interpretation of those triggers require access to the semantics of the model and the simulation engine. The triggers cause effects, on the one hand usual view changing effects, such as graphical focus change events, on the other hand *simulation effects* that alter the behavior of simulation time, such as simulation pause or stop.

A simulation manager should allow to specify *visual breakpoints*, the combination of a specific target view with the condition under which this view will be shown and possibly the suspension of the simulation to give time for analysis of the situation. A properly configured simulation manager knows what "interesting" items are, both in time and model objects. So during simulation a user always gets to see the right parts of interest without any manual user interaction; no manual navigation actions are required.

An example for dataflow diagrams is shown in Fig. 14. Here a focus is set to one actuator and all components in the dataflow towards that actuator. Other components are filtered. This results in tidy diagrams that illustrate specific aspects, e. g. for analyzing Actuator B. During a simulation run, the respective view could be shown, whenever some specific value is received by one of the actuators. The way of actually displaying the data is another issue but could be integrated into the diagram. The dynamic focus and context technique implemented in KIEL for Statecharts (cf. Sec. 2.4) could be implemented in a straight



(a) Original controller

(b) All dependencies of Actuator A, two sub-controllers and only Sensor A

(c) Dependencies of Actuator B

(d) Dependencies of Actuator C, which are all prior inputs

Fig. 14. View Management in a dataflow language for some embedded controller with three sensors and three actuators.

forward fashion by adding simulation events for every state change and setting the set of focus objects to the the active states.

Simulation Tracking and Control It is common practice to show (highlight) the current *state* of a system. In some areas, it is also common to show the current *change of state* (e. g., a transition in a Statechart). There are natural extensions that one could consider, such as showing the *recent past* (e. g., the last n states), or the *possible future* (states that might be reachable in the next n steps, this would require some kind of static/dynamic analysis).

A desirable feature is to be able to not just run a simulation and to stop it at certain points, but also to step backwards again. This *tape recorder paradigm* has already been integrated into some modeling tools, e. g., Statemate [25].

5 Conclusions

We have presented an overview of different aspects of modeling pragmatics. A guiding principle has been the model view controller paradigm, which has been quite successful in software engineering and which we believe has much to offer in the world of model based design as well.

We consider automatic layout of the graphical representation to be one of the basic key enablers for good pragmatics. We build upon layouters by dynamic filters that reduce the complexity of diagrams and focus and context as a special case of such filters. A view management engine organizes different dynamic views synthesized with filters in order to assist the user in seeing the “interesting” parts

of the model. We extend the view management by meta layout, which plays with different layout styles even within different parts of one graphical model in order to get optimal layout results.

With these building-blocks we support a set of use-cases in the modeling process that will help us to cope with very large model instances. For creation and modification we propose structure-based editing to free the user of many manual effort prone tasks. Auto-layout enables graphical model synthesis and opens the door for perfectly synchronized textual and graphical representations, scalable models, pattern-based modeling and support for product lines.

This survey cannot hope to be complete in any way. What we do hope to achieve is to raise the level of awareness about the importance and possibilities of modeling pragmatics in general. In a way, this paper might thus be regarded as a (partial) road map for possible future developments in modeling pragmatics.

References

1. Gurr, C.A.: Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. Journal of Visual Languages and Computing **10**(4) (1999) 317–342
2. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM **38**(6) (June 1995) 33–44
3. Prochnow, S., von Hanxleden, R.: Statechart development beyond WYSIWYG. In: Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), Nashville, TN, USA (October 2007)
4. Reenskaug, T.: Models – Views – Controllers. Technical report, Xerox PARC technical note (December 1979)
5. Kopetz, H.: The complexity challenge in embedded system design. Research Report 55/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2007)
6. Estefan, J.: Survey of model-based systems engineering (MBSE) methodologies, Rev. B. Technical report, INCOSE MBSE Focus Group (May 2008)
7. Carloni, L., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Languages and tools for hybrid systems design. Foundations and Trends in Design Automation **1**(1) (2006) 1–204
8. Prochnow, S., von Hanxleden, R.: Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technical Report 0406, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany (June 2004)
9. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Algorithms for drawing graphs: An annotated bibliography. Computational Geometry: Theory and Applications **4** (June 1994) 235–282
10. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall (1999)
11. Coleman, M.K., Parker, D.S.: Aesthetics-based graph layout for human consumption. Software – Practice and Experience **26**(12) (December 1996) 1415–1438
12. Ware, C., Purchase, H., Colpoys, L., McGill, M.: Cognitive measurements of graph aesthetics. Information Visualization **1**(2) (2002) 103–110

13. Völcker, J.: A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (May 2008) http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf.
14. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: A 'cognitive dimensions' framework. J. Visual Languages and Computing **7**(2) (June 1996) 131–174
15. Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., Mutzel, P.: A new approach for visualizing UML class diagrams. In: SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization, New York, NY, USA, ACM (2003) 179–188
16. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers **12** (2003) 231–260
17. Eiglsperger, M., Fößmeier, U., Kaufmann, M.: Orthogonal graph drawing with constraints. In: SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM (2000) 3–11
18. Card, S.K., Mackinlay, J., Shneiderman, B.: Readings in Information Visualization: Using Vision to Think. Morgan Kaufmann (January 1999)
19. Tiedje, M., Traulsen, C.: Designing a reactive processor with Esterel v7. In: Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'08), Budapest, Hungary (April 2008)
20. Leung, Y., Wilson, G.: WinFold: a folding editor for collaborative writing. Communications, 1999. APCC/OECC '99. Fifth Asia-Pacific Conference on Communications and Fourth Optoelectronics and Communications Conference **2** (1999) 1073–1078 vol.2
21. Dogrusöz, U., Kakoulis, K.G., Madden, B., Tollis, I.G.: On labeling in graph visualization. Inf. Sci. **177**(12) (2007) 2459–2472
22. van Dijk, S., van Kreveld, M., Strijk, T., Wolff, A.: Towards an evaluation of quality for names placement methods. International Journal of Geographical Information Systems (2002)
23. Kakoulis, K.G., Tollis, I.G.: On the edge label placement problem. In: GD '96: Proceedings of the Symposium on Graph Drawing, London, UK, Springer-Verlag (1997) 241–256
24. Wong, P.C., Mackey, P., Perrine, K., Eagan, J., Foote, H., Thomas, J.: Dynamic visualization of graphs with extended labels. In: INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization, Washington, DC, USA, IEEE Computer Society (2005) 10
25. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: Statemate: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering **16**(4) (April 1990) 403–414
26. Prochnow, S., Traulsen, C., von Hanxleden, R.: Synthesizing Safe State Machines from Esterel. In: Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Canada (June 2006)
27. Köth, O., Minas, M.: Structure, Abstraction, and Direct Manipulation in Diagram Editors. In: DIAGRAMS '02: Proceedings of the Second International Conference on Diagrammatic Representation and Inference, London, UK, Springer-Verlag (2002) 290–304

28. Branke, J.: Dynamic graph drawing. In Kaufmann, M., Wagner, D., eds.: Drawing Graphs: Methods and Models. Volume 2025 of Lecture Notes in Computer Science. Springer Verlag (2001)
29. Musial, B., Jacobs, T.: 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) 75–80
30. Prochnow, S., von Hanxleden, R.: Comfortable modeling of complex reactive systems. In: Proceedings of Design, Automation and Test in Europe (DATE'06), Munich, Germany (March 2006)
31. Schipper, A.: Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel (December 2008) `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`.
32. Kosak, C., Marks, J., Shieber, S.: Automating the layout of network diagrams with specified visual organization. Transactions on Systems, Man and Cybernetics **24**(3) (March 1994) 440–454
33. Purchase, H.C.: Which aesthetic has the greatest effect on human understanding? In: Proceedings of Graph Drawing Symposium, Di Battista, G. (ed). Volume 1353 of Lecture Notes in Computer Science., Springer Verlag (1997)
34. Castelló, R., Mili, R., Tollis, I.G.: A framework for the static and interactive visualization for statecharts. Journal of Graph Algorithms and Applications **6**(3) (2002) 313–351
35. Biermann, E., Ehrig, K., Khler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science. Volume 4199/2006., Springer Berlin/Heidelberg (2006) 425–439
36. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, New York, NY, USA, ACM (2006) 249–254
37. Efftinge, S., Voelter, M.: oAW xText: A framework for textual DSLs. In: Eclipse Summit Europe, Esslingen, Germany (October 2006)
38. Feng, T.H., Lee, E.A.: Scalable models using model transformation. In: 1st International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES^MB). (September 2008)
39. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
40. Douglass, B.P.: Real-time Design Patterns: Robust Scalable Architecture for Real-time Systems. Addison-Wesley (2003)
41. Peters, A.K.: Musterbasiertes Layout von Statecharts. Masters thesis, Universität Hamburg, Falkultät für Mathematik, Informatik und Naturwissenschaften, Department Informatik (June 2008)
42. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science **9**(11) (2003) 1296–1321
43. Brooks, C., Cheng, C.H.P., Feng, T.H., Lee, E.A., von Hanxleden, R.: 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 (September 2008)
44. Zhou, Y., Lee, E.A.: Causality interfaces for actor networks. ACM Transactions on Embedded Computing Systems (TECS) (April 2008) 1–35