Multi-View Modeling and Pragmatics in 2020*

Position Paper on Designing Complex Cyber-Physical Systems

Reinhard von Hanxleden[1], Edward A. Lee[2], Christian Motika[1], and Hauke Fuhrmann[3]

[1] Christian-Albrechts-Universität zu Kiel, Department of Computer Science
Olshausenstraße 40, 24118 Kiel, Germany
{rvh,cmot}@informatik.uni-kiel.de
[2] University of California at Berkeley, EECS Department
545Q Cory Hall, University of California, Berkeley CA 94720-1770
eal@eecs.berkeley.edu
[3] Funkwerk Information Technologies GmbH
Edisonstraße 3, 24145 Kiel, Germany
Hauke.Fuhrmann@funkwerk-it.com

Abstract. *Multi-view modeling* refers to a system designer constructing distinct and separate models of the same system to model different (semantic) aspects of a system. *Modeling pragmatics* also entails constructing different views of a system, but here the focus is on syntactic/pragmatic aspects, with an emphasis on designer productivity, and the views are constructed automatically by filtering and drawing algorithms.
In this paper, we argue that both approaches will have growing influence on model-based design, in particular for complex cyber-physical systems, and we identify a number of general developments that seem likely to contribute to this until 2020. This includes notably the trend towards domain-specific modeling and agile development, novel input devices, and the move to the cloud. We also report on preliminary practical results in this area with two modeling environments, Ptolemy and KIELER, and the lessons learned from their combined usage.

1 Introduction

A question prominently asked in computer science in model-based design is what kind of *model* (of computation) is particularly suitable for a given design problem. We here instead focus on the question of what *view* of a model might be best for a given task. When a designer creates two different models of the same system, e. g., one model for functional validation and another for deployment, this is referred to this as *multi-view modeling*. In this paper, we take a broader

* This work was funded in part by the Program for the Future Economy of Schleswig-Holstein and the European Regional Development Fund (ERDF).

look at multi-view modeling than that traditional interpretation, and try to extrapolate recent developments, including existing products, into the mid-term future. We target the year 2020 as a time frame when not only the basic technologies are in place (in fact, much of these technologies are in place already today, as this paper aims to illustrate), but also have found their way into mainstream modeling tools and practices. We do so with particular consideration of *modeling pragmatics*, which refers to the practical aspects of handling graphical system models of complex systems, encompassing a range of activities such as editing, browsing or simulating models [8].

Contributions and Outline. We advocate in this paper to expand multi-view modeling to constructing different model views even if they refer to the same semantic aspects. We will argue in the following that this approach meshes well with current trends towards agile, domain-adapted modeling, and propose to employ *usage-specific views* and *hybrid views*. These do not only consider the domain of an application, but also the current design activity a modeler is pursuing (Sec. 3). This approach has an immediate benefit for designer productivity, and thus supports "pragmatics-aware modeling." We also investigate what consequences the trend towards "post-PC devices" and their novel user interfaces might have on today's modeling activities, and propose *touch-based editing and browsing* to increase designer productivity (Sec. 4). Furthermore, in the context of the increasingly pervasive "move to the cloud," we propose an *actor-oriented, distributed tooling* approach (Sec. 5). This tooling approach should foster synergies and could also support agility as addressed in Sec. 3. We conclude in Sec. 6.

2 Background and Related Work

A *graphical model* is a model that can have a graphical representation, like a Unified Modeling Language (UML) class model. A *view* onto the model is a concrete drawing of the model, sometimes also *diagram* or *notation model*, e. g., a UML class diagram. The abstract structure of the model leaving all graphical information behind is the *semantical* or *domain model*, or just *model* in short. E. g., a class model can also be serialized as an XML tree. Hence, the *model* conforms to the abstract syntax, while the *view* conforms to the concrete syntax. Fig. 1 shows three different views of the same class model.

Model-Driven Engineering (MDE), or alternatively Model Driven Software Development (MDSD), denotes software development processes where models are central artifacts that represent software entities at a high abstraction level [5]. *Multimodeling* is the act of combining diverse models, to model, e. g., different parts of a software system or physical systems [6]. One form of multimodeling is multi-view modeling, as exemplified in Model-Integrated Computing (MIC) [21].

Multimodeling is also closely related to the single vs. multiple model principle discussed by Paige and Ostroff [19]. The ISO/IEC/IEEE 42010:2011 stan-

Fig. 1. Different Representations of a Class Model: Diagram, Text and Tree View (created with yUML (`http://yuml.me`) and Eclipse)

dard [12], which is the latest edition of the original IEEE Std 1471:2000, *Recommended Practice for Architectural Description of Software-intensive Systems*, also defines *architecture views* (or simply, *views*) to address one or more of the concerns held by the system's stakeholders, as no single view adequately captures all stakeholder concerns. Multimodeling is also related to *aspect-oriented modeling* [22], which focusses on identifying cross-cutting concerns; a central concept here are *join points*, which represent a concern element, i.e., an identifiable element of the language used to capture a concern. Brooks et al. [2] have also advocated the usage of multimodeling to separate concerns during a model-based design flow, e. g., to separate functional aspects from deployment and verification. This is particularly relevant in the real of cyber-physical systems, which have to consider physical deployment domains as well as the embedded control, and whose growing complexity necessitates a clean separation of concerns. The designer should be able to specify different aspects of the same system independently, to allow a clean separation of concerns while keeping a model consistent. However, multi-view modeling can be applied at different levels and in very different ways. For example, it can refer to the animation of a model during a simulation, or to the alternation between graphical and textual representations, or indeed also to the alternation between a monolithic Statechart model and an explicitly hierarchical syntax, as discussed in this paper.

However, Brooks et al. concluded: *At this point, it is still largely up to the modeler to construct different views of the same system. How best to harness a modeling system to assist the user with this task still seems to be a largely open problem.* While this problem still is certainly not completely solved yet, we here argue that modeling tools in 2020 should have made significant progress towards that goal. In fact, already today there are significant steps in that direction. To

illustrate that point, we re-use in Sec. 3 the traffic light example from Brooks et al. [2], and present different views that are automatically synthesized.

3 Trend 1: Agile, Domain-Specific Development Processes

The processes in software development change from static monolithic one-way methods, which lead from an abstract specification to a concrete design, to more agile and iterative approaches. Agile development is accompanied by a move away from big, one-size-fits-all frameworks and languages or language families toward Domain-Specific Languages (DSLs). E. g., the UML has evolved into such a multitude of languages that by now, most designs and designers employ only a subset of the UML languages or variations tailored towards specific domains, and it is a challenge for tool providers to adequately support all languages. However, an iterative process requires not only to go from abstract to concrete. Developers jump arbitrarily between abstraction levels, and change either abstract specifications if they have to adapt the general system concept or details in the implementation if one iteration's prototype milestone needs to be finished. This *round-trip engineering* does not mesh very well with today's modeling tools.

3.1 2020 vision: Usage-Specific Views

Agile processes require agile and lean tool support and languages that are not only tailored towards particular domains, but also towards particular design activities. This meshes with the concept of DSLs, which are also called "task-specific" languages [16], even if this interpretation is less common than the "(application) domain-specific" interpretation. Note that this does not necessarily require the invention of a host of new languages, but rather expresses that we want to be able to switch model views according to different model usages, and that these different views may employ different (graphical or textual) languages. We refer to this concept as *usage-specific views*.

To illustrate, consider the traffic light control example presented in Fig. 2, adapted from Brooks et al. [2]. The example is shown in three variants, which at first sight look quite different and employ different visual languages. The first variant, shown in Fig. 2a, employs a SyncCharts [1] model, developed in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[4] modeling environment, to describe the behavior of the traffic light. As can be seen, there are two modes of operation, Normal and Error, and for each mode the behavior of the car light and the pedestrian light is specified. This *behavioral view* might be appropriate for a first specification of the traffic light. Fig. 2c now uses a very different language, or rather set of languages, namely a hierarchical combination of synchronous data flow with state machines, shown in the Ptolemy II[5] tool.

[4] http://www.informatik.uni-kiel.de/rtsys/kieler/
[5] http://ptolemy.eecs.berkeley.edu/

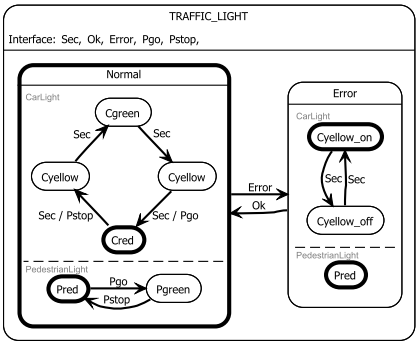
This *structural view* (or *deployment view*) emphasizes what components the traffic light consists of, namely the car light and the pedestrian light, and through which signals they interact with the environment and with each other. However, even though these two views use different languages that have different semantics and may be considered different models of a traffic light, they do express the same behavior, i. e., the semantics of these two models coincide. In fact, in this case the Ptolemy model that underlies the structural view has been synthesized automatically from the SyncChart model that underlies the behavioral view, with the original purpose of simulating the SyncChart model [17]. So, one may say that the model shown in Fig. 2c enhances the model from Fig. 2a in at least two ways, namely with a simulation capability and by illustrating to the user the structure of the traffic light.

A common criticism of SyncCharts (and Statecharts in general) is that they, due to their signal broadcast semantics, have only implied, hidden signal communication links. One possible answer to this is the structural view just presented. However, we also want to propose another, third alternative, which we will refer to as *hybrid view*. To that end, we now examine another means to better understand the *references* in a graphical model. The graphical representation depicts the main model objects as nodes, where the containment relations can be reflected by hierarchy in the model and containment of graphical symbols like rectangles. Therefore, the diagram exhibits intrinsic properties, and these properties directly correspond to properties in the represented domain [10]. 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, such as the signal-based, name-bound broadcast communication in a Statechart.

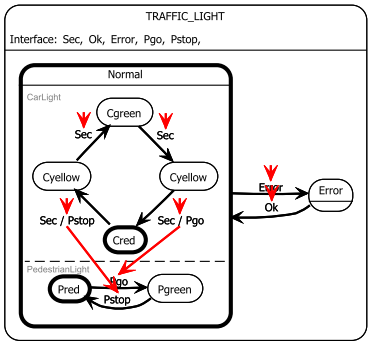
3.2 Dual Modeling

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. This dual model then results in a *hybrid view*, which emphasizes multiple semantic aspects of a model at once. The hybrid view in Fig. 2b reveals the rather simple communication of the traffic light example. The Error state has no inter-communication, hence focus&context [20] automatically collapses it. The structural view in Fig. 2c also shows this communication explicitly, however, the simplicity is more obvious in the hybrid view; this may also be due to the visible hierarchy there.

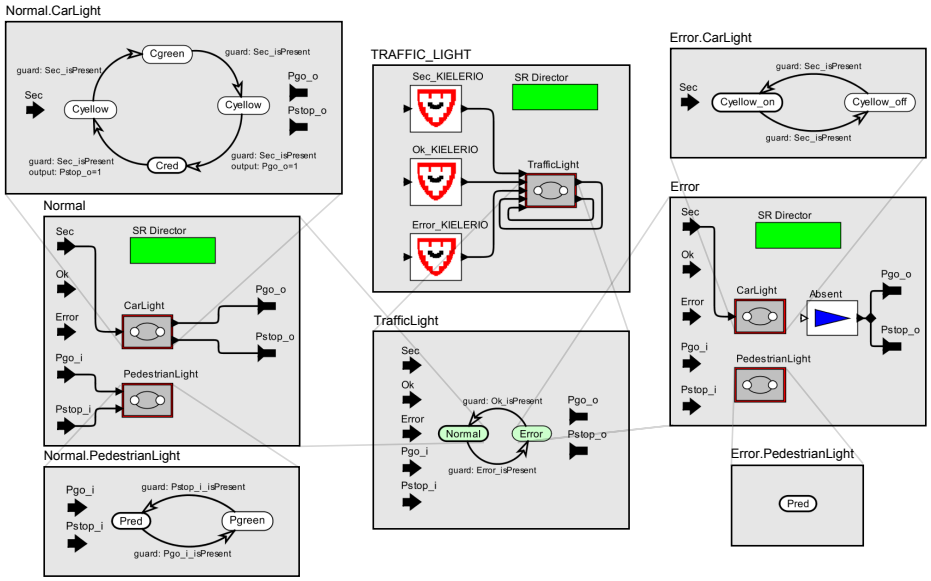
The *dual model* methodology should not only be helpful for Statecharts, but applies to very different types of models. References to other model parts are quite common where an explicit graphical representation is omitted for the sake of clarity in the original model. Two examples are:



(a) Behavioral view (SyncChart)



(b) Hybrid view, revealing the communication via signals (SyncChart with dual modeling and focus&context filtering).



(c) Structural view (hierarchical data-flow + automata, from Motika et al. [17])

Fig. 2. Traffic light example, usage-specific views.

Class diagrams The attributes of a class are presented more or less textually including the type of the field. However, the type may also reference another class or a data type definition node in the model. The dual model of a class diagram would reveal the data type usages of the classes and their attributes.

Ptolemy II In Ptolemy one can define arbitrary parameters of actors. They are represented by an unconnected node only showing the key and the

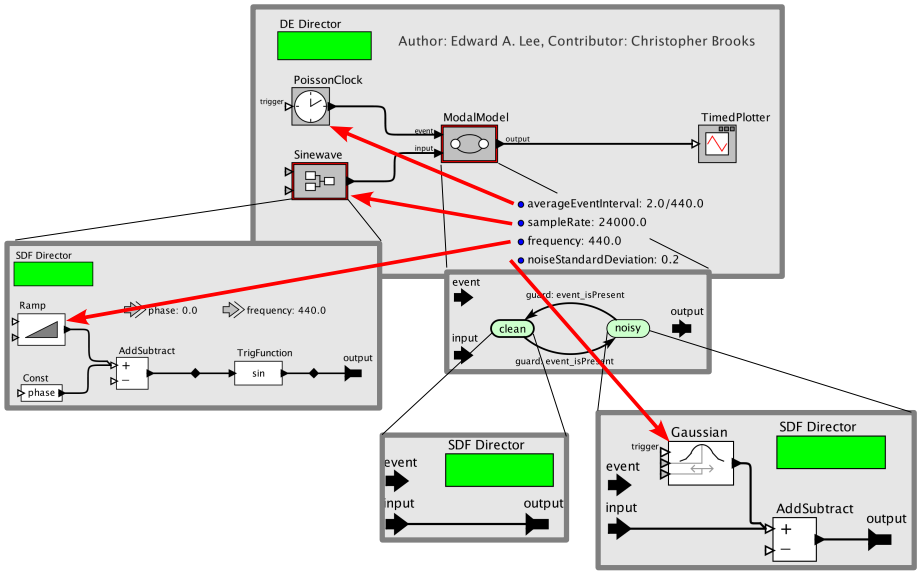


Fig. 3. A dual model for Ptolemy could show where parameters of an actor are used (from Fuhrmann [7]).

value of the parameter. Then they get referenced by arbitrary expressions in Ptolemy's expression language, which is just text. They are often used to map parameters of lower-level actors to the top-level actor. The dual model could explicitly show which objects use which parameters. An example montage is shown in Fig. 3. Technically this would work best if the editor would use visible hierarchy, which the Ptolemy editor Vergil does not.

Note that the structural view in Fig. 2c is also a kind of hybrid view that combines drawings of individual model components with an overall drawing (using gray lines) of how these components are related to each other. As of today, creating such drawings is again a manual, rather laborious process, which severely compromises designer productivity and thus goes against pragmatics-aware modeling. To do so automatically in a well-readable, compact fashion is an interesting layout problem that we are currently investigating, which leads to the concept of automatic layout also addressed in the next section.

4 Trend 2: Novel Input Devices

If we may believe innovation-leading companies in the field of ergonomic human-machine interaction, we are in the decade of "post-PC devices" [13]. Improvements in touch-display technology foster the success of smartphones and even

new device categories like tablet computers that convince users with intuitive interaction paradigms. In professional environments such handheld devices or also bigger devices like computerized white boards may assist collaboration in team meetings and ease both the group access to data and capturing group results. Nonetheless the modeling community maintains traditional interaction paradigms for creating, navigating and maintaining models, notably What-You-See-Is-What-You-Get (WYSIWYG) Drag-and-Drop (DND) freehand editing that requires a precise instrument like the mouse.

4.1 2020 vision: Touch-based editing and browsing

To take advantage of these novel input devices and to increase designer productivity, we propose to adapt novel design entry and browsing mechanisms that are less dependent on precise pointing devices. As a first enabling step, this requires to enhance today's modeling tools with reliable, high-quality *automatic layout* capabilities that can arrange diagram elements in a compact, well-readable fashion. As of today, visual models are traditionally drawn manually. However many modeling tools have some auto-layout capabilities already, and the insight that designers should be freed from the burden of doing manual place-and-route work as part of their modeling activity slowly seems to gain acceptance. E. g., one of the advertised new features for IBM's Rational Software Architect includes a variety of automated layout algorithms. To quote from their announcement: *These automated layouts also make it easier to understand complex models and to build abstractions by viewing the model in a well-laid-out way. Most importantly, they should reduce the overall amount of time you need to spend on hand-formatting diagrams, thereby increasing your productivity and freeing more of your time for higher-value activity.*[6]

Note that when providing an automatic layout capability, one must also ensure that automatic layout does not destroy the *mental map* of a user when editing a model; for example, morphing mechanisms can help here significantly. We also acknowledge that designers, when confronted with the idea of automatic layout, are often at first reluctant to defer the drawing of a model to some algorithm that does not have any understanding of the application. As a compromise, there is also option of performing only *incremental* automatic layout, or to provide some *intentional* layout capability that allows the modeler to guide the automatic layout algorithm in certain ways. However, it is our experience that after getting used to a tool with high-quality automatic layout capabilities, designers are quite happy to make use of this capability, and become frustrated whenever they have to use a modeling tool without such a capability. This pattern is common whenever designers are asked to give up control of certain design aspects, and indeed it is often advisable to provide some escape mechanism. An analogy in the programming world is the capability of embedding assembler in a high-level language. However, carrying this analogy further, we also observe

[6] `http://www.ibm.com/developerworks/rational/library/10/whats-new-in-rational-software-architect-8/index.html`

that today, most programmers appear to be glad to have been mostly freed from the task of manual assembler programmer, and are happy with the results that a compiler generates for them.

Note that the automated diagram drawing is by no means trivial, as many rather unusable auto-layout buttons can attest to, and there is an active research community that works on improving the state of the field [3]. However, the challenge here lies not only in the fundamental drawing problem, but also in smoothly integrating layout capabilities into the modeling tool. Here, the actor-oriented tooling approach outlined in Sec. 5 might also help. With automatic layout capabilities, it is possible to post-process imprecise drawing commands into high-quality diagram drawings. For a nice illustration of this approach, consider the Instaviz "pocket whiteboard,"[7] which uses advanced shape-recognition (Recog) and automatic drawing (GraphViz) capabilities. From the product description: *Sketch some rough shapes and lines, and Instaviz magically turns them into beautifully laid-out diagrams.* We are not aware of hard experimental data on the productivity of this software, but the subjective impression is that with this approach, working with a phone-size touch-sensitive display, one is faster to create a usable diagram than with a traditional model editor without layout capabilities installed on a full-size PC. This is not to advocate smart phones for productive system design, but the technologies developed there might very well be helpful. Multi-touch displays might allow more efficient and intuitive model manipulation and navigation than traditional pointing devices. For example, one might borrow from the effective navigation techniques that allow to browse photo libraries or web pages with very little screen real estate. Other examples of such inspiring innovations are dictionary-based predictive text entry (T9) or motion-based text entry (Swype).

4.2 Structure-based editing

Next, given a modeling platform that provides automated drawing capabilities, we can raise the abstraction level of editing activities to work on the structure of the model itself, rather than working on its representation. This *structure-based editing* [9] does not require precise pointing any more, so for example it does not require shape recognition. Instead, it suffices to select existing model elements and to specify the operation to apply to it, such as "add a successor state" or "invert transition direction".

Such higher level, semantically oriented editing capabilities could also enhance traditional editing paradigms. For an example, consider the copy&paste operation, which originally was made possible by computer-based editing, but remains rather primitive until today. In a usual freehand editing environment, copy&paste requires numerous *enabling steps*. The user has to 1. select all objects to copy, 2. call the copy operation, 3. choose a target space, 4. free space at the target location, 5. select the target place (however, selecting an empty location usually is not possible in most tools), 6. call the paste operation, 7. move

[7] `http://instaviz.com/`

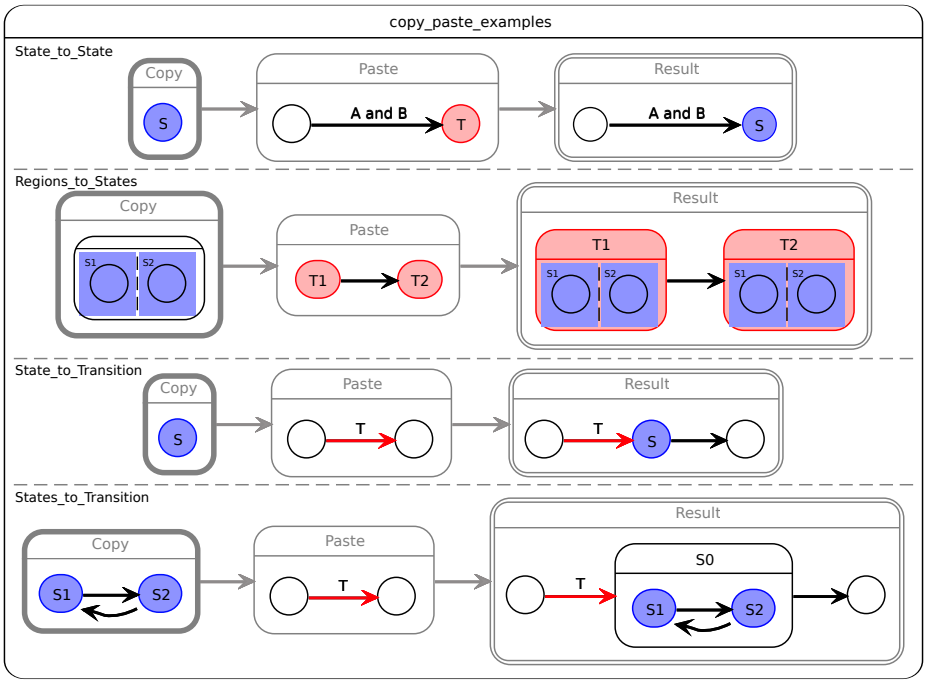


Fig. 4. Examples for copy&paste operations on a Statechart diagram. Each operation is illustrated with a sequence of three states: 1) the Copy state with a selected source to copy (e. g., state S), 2) the Paste state with the selected target (e. g., state T) into which the source should be pasted, and 3) the Result into which Paste gets transformed.

the pasted set of objects to the new empty space and finally 8. rearrange the surroundings such that the new objects seamlessly integrate. Especially steps 4, 7 and 8 may be arbitrarily effort-prone, and step 7 may be frustrating when the pasted objects do not appear at the target space of step 3 and the tool does not state explicitly about its target space policy. However, structure-based editing employing automatic layout can improve the situation considerably [7]. The editing steps would boil down to 1. select all objects to copy, 2. call the copy operation, 3. select a target *object*, and 4. call the paste operation. With automatic layout, the user should not specify any target *location*, but only a target *object* where the contents should be pasted. A generic transformation description should then specify how the elements are pasted *into* the target object and the automatic layout would do the rest.

To illustrate, Fig. 4 presents some possible copy&paste operations for Statecharts. Each transformation rule has to consider the *copy sources* (labeled "S" in Fig. 4), i. e., the selected elements which get copied, and the *copy targets* ("T"). For Statecharts these objects may be states, regions, and transitions, and each

set may be of arbitrary size. A good example is "copy multiple states to one transition". In a usual freehand editor, this is not possible and would do nothing. As implemented in KIELER, the transformation 1. cuts the target transition into two transitions, 2. adds a new state in-between both transitions, and 3. adds the selected nodes into a new region of the new state. Other similar transformations are possible, which the toolsmith would have to define according to experience in the context of the given DSL. Selecting multiple target objects is a fast way to replicate objects multiple times.

As a word of caution, these copy&paste effects go considerably beyond what designers are familiar with today. Also, some of these effects are probably needed only rarely, such as the "copy transitions to transitions". Still, extending the copy&paste paradigm in this fashion may significantly increase productivity, and is yet another example of the possibilities for harnessing automatic layout towards pragmatics-aware modeling.

5 Trend 3: The Move to the Cloud

Activities traditionally done locally become increasingly distributed and are moved to "the cloud." For example, to generate the class diagram drawing in Fig. 1, we did not install a UML tool, but visited a web page and pasted the textual description of the diagram into a text box. Not having to undergo lengthy installation procedures and always having a current tool version at one's disposal is appealing. We believe that this applies in particular to the world of MDE with its typically quite complex tool environments, and this also applies to other cloud-benefits such ease of design sharing (leading to *model mashups*) and designer mobility (consider google docs etc. that are already commonly integrated into mobile OSs such as Android). As another example, National Instruments' LabVIEW Web UI Builder is a cloud-based Rich Internet Application (RIA), which is hosted by Amazon Web Services and is basically a light-weight version of LabVIEW that allows to interface with hardware and/or web services. Similarly, NI offers a cloud version of a compiler that deploys LabVIEW models onto an FPGA. This application can be very compute-intensive, and there is a large variety of possible compilation targets; both factors make it attractive to move away from the local desktop into the cloud.

There already exist standards for web service interfaces, e. g., the Web Services Business Process Execution Language (WS-BPEL) [18] to describe business process activities as web services. However, such (mostly syntactic) standards are not enough, as they still exhibit semantic ambiguities that hamper tool compatibility. And, as Lapadula et al. state, the *design of WS-BPEL applications is difficult and error-prone also due to the presence of such intricate features as concurrency and race conditions, forced termination, [etc.]* [14].

5.1 2020 vision: actor-oriented, cloud-based modeling tools

The idea of actor-oriented modeling is to break down complexity by decomposing a system into *actors* that communicate through well-defined interfaces [4].

The components interact not via control flow (such as a method-call in object-oriented design), but via data. This approach sidesteps many difficulties in the design of complex systems and supports the clean handling of concurrency [15].

We here claim that many of the arguments for actor-oriented design also apply to the modeling tools, and that this aligns well with the cloud-computing infrastructure already in place. This would not only make modeling tools more robust and versatile, but would also allow toolsmiths to focus on particular services, such as simulation or visualization, and not on having to re-develop everything else that is needed for a complete design environment. This would also go hand in hand with the trend towards more agile, customized design processes described earlier.

An interesting initiative in this regard is the ModelBus [11], which is built upon Web Services and follows a Service Oriented Architectures (SOA) approach. ModelBus provides an interaction pattern in order to enable model sharing in a distributed and heterogeneous model-driven development process. In comparison, actor-oriented design of modeling tool does not necessarily entail model sharing, but model sharing could be combined with the actor-oriented approach advocated here.

5.2 Example of a service: simulation

For example, as explained in Sec. 3, the KIELER modeling environment leverages Ptolemy as simulation engine. This is currently implemented by first transforming a KIELER model into a Ptolemy model. Then a Ptolemy instance is run in the background that processes simulation requests coming from KIELER and communicates simulation data back for proper visualization in KIELER.

One might as well move this simulation capability to a server that communicates through a standardized interface, e. g., based on XML. A non-trivial question here is what kind of information should be communicated. Traditionally, one is interested in the input/output behavior of the simulated component, and this is what most APIs (if tools have APIs for this purpose at all) offer. However, when using such a simulation service from within a modeling tool, one typically would like to know about the internal states of the simulated system as well. For example, the Ptolemy-SyncChart does communicate to KIELER the current state of the simulation; however, a modeler would typically also like to know which transition was taken to get to that state, which is not communicated. KIELER does remember the previous state, which can help to deduce the taken transition—but not if there are multiple transitions between the previous state and the current state. Conversely, one may not want to execute complete, externally visible reaction steps at once, but would like finer control over the simulation.

The lesson to be learned from there is that modeling frameworks should have open simulation interfaces, both for exporting and for importing simulations. These interfaces should not be limited to the externally visible behavior of the system under development (SUD), but should also include internal information that might be of interest to the modeler.

5.3 Example of a service: automatic layout

As another example of a possible service to be provided in the cloud, KIELER provides layout capabilities to Ptolemy. A non-trivial issue there was to find a suitable user interface to access the auto-layout capabilities. E. g., initially, the user interface consisted of five buttons of different functionality. This proved too complicated to handle for the uninitiated. The current interface has just one button, which lead to much better user acceptance. The deeper reason for the initially too complicated user interface for the automatic layout was that

As is customary for today's editors, Ptolemy's graphical Vergil editor was not developed with externally provided automatic layout in mind. E. g., after the modeler has placed the nodes of a model, Vergil uses some heuristic to automatically route edges. This is a certain help to the human layouter, but conflicts with automatic layout, which needs control of both the node and the edge placement. The solution was to enhance Vergil to consider layout-annotations added by the KIELER layouter to the Ptolemy model.

Another issue turned out to be hyper edges. The Ptolemy way of connecting more than two actors is to add a relation node to the model, and adding a connection from each of the to-be-connected actor to the relation node. From the perspective of a generic layout algorithm, however, the relations look just like another actor. This typically leads to less compact layouts than would result from hyper edges that would directly connect the actors.

The lesson to be learned there is that editors should be developed with automatic layout in mind, and should provide simple interfaces to these. As a notable example in this direction, one of the five stated objectives of the Eclipse Graphiti project proposal was to provide *the ability to use any existing layout algorithms for auto layouting a diagram*[8]. There are further issues not discussed here, such as hyper edges, the handling of comments, and the efficient incorporation of layout results into a model (as it turns out, this is often more time consuming than the actual layout computation) [7].

A further issue was the handling of comments. Traditionally, comments are text boxes placed (manually, like everything else) at some convenient location in the visual model. These comments might refer to the whole diagram, e. g., to provide a general description or to identify the author. Often, however, comments refer to specific model elements. This reference is usually not anchored in the model itself, but only implicit in the spatial proximity of the comment to the referenced model element. This proximity usually gets lost when applying an automatic layout to the diagram. The lesson learned there was that comments should be anchored to model elements. This is already possible e. g. in Eclipse GEF.

6 Conclusions and Outlook

MDE, or software and systems engineering in general, keeps to be challenged by increasingly complex and powerful applications. In the past, this has fostered the

[8] `http://www.eclipse.org/proposals/graphiti/`

development of similarly complex and powerful modeling tools and processes, often with little regard for the practical needs and limitations of the human developer.

We here advocate an approach that focuses on the different, concrete design activities of the developer and provides practical support for these activities. This proposal is driven mostly by the authors' experience in the design of cyber-physical systems, but we expect that much of this is of relevance beyond CPS design as well. Key aspects here are the tool-supported creation of different views for these different activities, and pragmatic-aware model interaction paradigms. We sketched a vision, or at least fragments thereof, of how this approach might benefit from and provide support for a selection of current technological trends, and where this approach might lead to until the end of this decade. As it turns out, we here drew less from the established MDE community and more from other communities and from industry trends. So, a general conclusion might be that there is much innovation out there from which the MDE community could and should benefit from in the near future.

Acknowledgement

We thank the participants of the workshop and the reviewers for their very valuable comments.

References

1. André, C.: Computing SyncCharts reactions. Electronic Notes in Theoretical Computer Science 88, 3–19 (Oct 2004)
2. 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 (Sep 2008)
3. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Algorithms for drawing graphs: An annotated bibliography. Computational Geometry: Theory and Applications 4, 235–282 (Jun 1994)
4. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (Jan 2003)
5. Estefan, J.: Survey of model-based systems engineering (MBSE) methodologies, Rev. B. Technical report, INCOSE MBSE Focus Group (May 2008)
6. Fishwick, P.A., Zeigler, B.P.: A multimodel methodology for qualitative model engineering. ACM Trans. Model. Comput. Simul. 2, 52–81 (Jan 1992)
7. Fuhrmann, H.: On the Pragmatics of Graphical Modeling. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel (2011)
8. Fuhrmann, H., von Hanxleden, R.: On the pragmatics of model-based design. In: Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers. LNCS, vol. 6028, pp. 116–140 (2010)

9. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). LNCS, vol. 6394, pp. 196–210. Springer (Oct 2010)
10. Gurr, C.A.: Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. Journal of Visual Languages & Computing 10(4), 317–342 (1999)
11. Hein, C., Ritter, T., Wagner, M.: Model-driven tool integration with ModelBus. In: Workshop Future Trends of Model-Driven Development (2009)
12. ISO/IEC JTC 1/SC 7: Systems and software engineering — architecture description. ISO/IEC FDIS 42010, working document ISO/IEC JTC 1/SC 7 N (2011), http://www.iso-architecture.org/
13. Jobs, S.: Apple special event, keynote speech (Mar 2011)
14. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Lea, D., Zavattaro, G. (eds.) Coordination Models and Languages, LNCS 5052. pp. 199–215 (2008)
15. Lee, E.A.: The problem with threads. IEEE Computer 39(5), 33–42 (2006)
16. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (Dec 2005)
17. Motika, C., Fuhrmann, H., von Hanxleden, R., Lee, E.A.: Executing domain-specific models in Eclipse (2012), in preparation
18. OASIS WSBPEL TC: Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (Apr 2007)
19. Paige, R., Ostroff, J.: The single model principle. Journal of Object Oriented Technology 1, 2002 (2002)
20. 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). LNCS, vol. 4735, pp. 635–649. IEEE, Nashville, TN, USA (Oct 2007)
21. Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer 30(4), 110–111 (Apr 1997)
22. Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E.: A survey on UML-based aspect-oriented design modeling. ACM Comput. Surv. 43(4), 28:1–28:33 (Oct 2011), http://doi.acm.org/10.1145/1978802.1978807