Pragmatics Twelve Years Later: A Report on Lingua Franca

Reinhard von Hanxleden[1(✉)], Edward A. Lee[2], Hauke Fuhrmann[3], Alexander Schulz-Rosengarten[1], Sören Domrös[1], Marten Lohstroh[2], Soroush Bateni[4], and Christian Menard[5]

[1] Kiel University, Kiel, Germany
{rvh,als,sdo}@informatik.uni-kiel.de
[2] University of California, Berkeley, USA
{eal,marten}@berkeley.edu
[3] Scheidt & Bachmann, Kiel, Germany
Fuhrmann.Hauke@scheidt-bachmann-st.de
[4] University of Texas at Dallas, Richardson, USA
soroush@utdallas.edu
[5] Technical University Dresden, Dresden, Germany
christian.menard@tu-dresden.de

Abstract. In 2010, Fuhrmann et al. argued for enhancing modeler productivity by providing tooling that, put simply, combines the best of textual and graphical worlds. They referred to this as *pragmatics*, and argued that a key enabler would be the ability to automatically synthesize customized graphical views from a (possibly textual) model. The model would be the "ground truth" used, for example, for downstream code synthesis and simulation; the graphical views would typically be abstractions from the model serving various purposes, including documentation.

Twelve years later, we reflect on their proposal, and illustrate the current state with the recently developed polyglot coordination language Lingua Franca (LF). LF has been designed with pragmatics in mind since early on, and some characteristics of LF make it particularly suited for pragmatics-aware programming and modeling. However, the underlying pragmatic principles are broadly applicable, and by now a set of mature open source tools is available for putting them into practice.

Keywords: Pragmatics · Lingua Franca · Model-driven engineering · Diagram synthesis · KIELER

1 Introduction

Visual modeling, by some seen as synonymous with modeling altogether, offers the advantage of naturally providing visual diagrams that, ideally, self-document a system under development. However, practicioners often experience authoring with diagrams only as less efficient than working on textual artefacts, which

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13702, pp. 60–89, 2022.
https://doi.org/10.1007/978-3-031-19756-7_5

for example has given high momentum to textual Domain Specific Languages (DSLs). Twelve years ago, Fuhrmann et al. [8] proposed to enhance designer productivity by employing *pragmatics*, interpreted, roughly speaking, as combining the best of textual and visual modeling worlds. They argued that key enablers would be the separation of model and view, and the automatic construction of (graphical) customized views. The graphical views would not be necessary in a technical sense, for example to simulate a model, to formally analyze it, or to synthesize code. However, they would be valuable artefacts for the human modeler for various purposes, a primary purpose being the documentation of the model. As proof of feasibility, they presented the open-source KIEL Integrated Environment for Layout Eclipse Rich Client (KIELER) framework for *pragmatics-aware modeling*. At the time, von Hanxleden et al. [17] envisioned that by 2020, the use of pragmatics in software engineering tools would be widespread.

While it is hard evaluate the level of adoption of the approach, there are several projects centered around pragmatics that certainly have had a major impact. The most popularized contribution from the KIELER initiative might be the Eclipse Layout Kernel (ELK)[1] Java code library that implements sophisticated graph layout strategies and thus is a key driver for many of the pragmatics improvements in the KIELER approach. The transpiled ELK for JavaScript (ELKJS) library has, at the time of this writing, received around 1,000 stars on GitHub[2].

As an official project hosted at the Eclipse Foundation it has some visibility, leading to its adoption in quite a few downstream tools. Just recently, ELK has been adopted in the Graphical Language Server Platform (GLSP)[3] meta-framework for graphical modeling, which follows Microsoft's approach for a textual integrated development environment (IDE) with the language server protocol (LSP)[4]. It extends the LSP philosophy of having a clear separation between a backend for language and semantics and a frontend for GUI features to the domain of graphical modeling languages. Hence, the complex tasks of the backend can be reused in different frontend implementations and open up for broader technology mixes. Also other established meta-frameworks adopted ELK, such as Sirius[5], the Graphical Modeling Framework (GMF), and Graphiti[6].

More than a decade of employing pragmatics in various languages and tool kits, both in academic and industrial contexts, has also created a host of users that can reflect on their experience with pragmatics. As one practitioner from Bosch Siemens Hausgeräte GmbH put it after using KIELER for some time:

> "In our experience, over many years, my colleagues and I concluded that textual modeling is the only practical way, but that a graphical view of the models is a must-have as well. Your technology closes exactly that gap" [44].

[1] https://www.eclipse.org/elk/.
[2] https://github.com/kieler/elkjs.
[3] https://www.eclipse.org/glsp/.
[4] https://microsoft.github.io/language-server-protocol/.
[5] https://blog.obeosoft.com/a-picture-is-worth-a-thousand-words.
[6] https://www.eclipse.org/modeling/gmp/.

In this paper, we demonstrate the state of the art and the value of pragmatics-aware modeling with the recently developed Lingua Franca (LF) language serving as prime example, while also taking the opportunity to reflect on the broader developments and feedback received over time.

1.1 Contributions and Outline

We will first review the key concepts of pragmatics and its model-view separation (Sect. 2). For several reasons, including the hierarchical structure and the separation of coordination language and target language, LF lends itself particularly well to pragmatics-aware modeling, and its tool chain has incorporated that from its beginning. We elaborate on this with several practical examples in Sect. 3. An important component of pragmatics is the choice of graphical syntax, including its amenability to automatic layout; case in point, we discuss some of our design choices for LF in Sect. 4. The issues that arise from a graph drawing/diagram synthesis perspective and some ongoing work are covered further in Sect. 5. For readers interested in possibly employing modeling pragmatics in other tools/languages than presented here, Sect. 6 briefly covers the underlying technology, which is all available open-source. Feedback from users on their experience with pragmatics and an outlook on the challenges ahead is summarized in Sect. 7. Some related work is covered in Sect. 8, we wrap up in Sect. 9.

2 Pragmatics in Linguistics and Modeling

In linguistics, the study of how the meaning of languages is constructed and understood is referred to as *semiotics*. It divides into the disciplines of syntax, semantics, and pragmatics [32]. These categories can be applied both to natural as well as artificial languages (i. e., languages that are consciously devised), such as programming languages. In the context of artificial languages, *syntax* is determined by formal rules defining expressions of the language [13] and *semantics* determines the meaning of syntactic constructs [18]. "Linguistic *pragmatics* can, very roughly and rather broadly, be described as *the science of language use*" [14]. This definition can be applied verbatim to our domain of interest; programming languages, or modeling languages, as they are often referred to in the context of Model-Driven Engineering (MDE). However, before we further discuss and define pragmatics in the realm of software engineering and MDE, we first introduce some more terminology.

The main artifacts in MDE are *models*, which adhere to two main concepts. First, a model *represents* some software artifact or real world domain. Second, it *conforms* to a *metamodel* or *grammar* [19], defining its *abstract syntax*. The *concrete syntax*, on the other hand, is the concrete rendering of the abstract concepts. Concrete syntax can be textual or displayed in a structured way, such as a tree view extracted from an Extensible Markup Language (XML) representation of the abstract syntax. Graphical syntax is often used to visualize model structure; the Unified Modeling Language (UML) encompasses several graphical

example languages. According to Gurr, a *visual language* is any language that is expressed to the reader's visual sense. Therefore, diagrams as well as textual programs or mathematical models, are visual languages. One special characteristic of diagrams is that they exhibit intrinsic properties, and these properties directly correspond to properties in the represented domain [13], such as containment relations of boxes.

A *graphical model* is a model that can have a graphical representation, like a UML class model. A *view* of 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, a *model* conforms to the abstract syntax, while a *view* conforms to a concrete syntax.

A view can represent any subset of the model, which in some frameworks is used to break up complex models into multiple manageable views. Hence, there is no fixed one-to-one relationship between model and view.

In linguistics, *pragmatics* traditionally refers to how elements of a language should be used, e. g., for what purposes a certain statement is eligible, or under what circumstances a level of hierarchy should be introduced in a model. It denotes the "relation of signs to (human) interpreters" [32] and therefore Gurr calls it the "real semantics" of a language [13]. Pragmatics addresses questions concerned with avoiding ambiguities or misleading information in graphical representations, for example by improper usage of layout conventions, termed *secondary notation* by Petre and Green [11]. As Gurr states:

> "A major conclusion of this collection of studies is that the correct use of pragmatic features, such as layout in graph-based notations, is a significant contributory factor to the effectiveness of these representations." [13]

In the context of MDE, Selic [45] (somewhat implicitly) uses "pragmatics" in a rather broad sense, referring to practical aspects such as code synthesis. We here slightly extend the linguistic interpretation of pragmatics as follows:

Pragmatics concern all practical aspects of handling a model in its design process. This includes practical design activities such as editing and browsing of graphical and/or textual model representations in order to construct, analyze, and effectively represent a model's meaning. Pragmatics aim to increase modeler productivity and product quality. Pragmatics-aware tooling helps achieve a separation of model and view, in line with the well-established Model-View-Controller (MVC) paradigm [36]. A key enabler is the ability to automatically construct customized graphical views of a model.

3 Diagrams for Development of Lingua Franca Programs

LF (LF) is a recently-developed polyglot coordination language for concurrent and possibly time-sensitive applications ranging from low-level embedded code

to distributed cloud and edge applications [29]. An LF program specifies the interactions between components called *reactors*. The emphasis of the framework is on ensuring deterministic interaction with explicit management of timing. The logic of each reactor is written in one of a suite of target languages (currently C, C++, Python, Rust, or TypeScript) and can integrate (possibly legacy) code in those languages. A code generator synthesizes one or more programs in the target language, which are then compiled using standard toolchains. If the application has exploitable parallelism, then it executes transparently on multiple cores. This happens without compromising determinacy because data dependencies between reactors translate into scheduling constraints at runtime. A distributed application translates into multiple programs and scripts to launch those programs on distributed machines. The communication fabric connecting components is synthesized as part of the programs.

There are a number of features of LF that make its programs particularly well suited to diagram representations:

1. The LF language expresses only the structure of programs, not their detailed logic. The latter is done in chunks of target-language code that are ignored by the LF parser.
2. The language encourages the use of hierarchy, where components contain other components. This lends itself well to the KIELER mechanisms for handling hierarchy, where hierarchical components can be expanded or collapsed to focus attention.
3. LF components (reactors) are concurrent, lending themselves naturally to being rendered as side-by-side boxes coexisting in a space.
4. The interactions between components are explicit. Each reactor has input and output ports, and a container reactor explicitly connects the ports of its contained reactors. This contrasts with most actor frameworks, such as Akka [38], where the existence and interconnections of actors is buried in the logic of target-language code.

LF's interactive diagram synthesis capability is an integral part of its IDE support. An Eclipse-based LF IDE called Epoch is available at GitHub[7]. An LSP-based Visual Studio Code Extension for LF is installable from both the Visual Studio Marketplace[8] and the Open-VSX Registry[9].

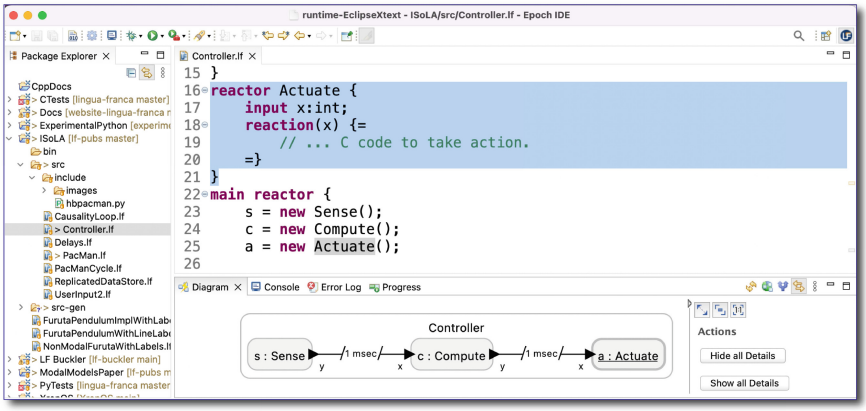
3.1 Data Dependencies

We begin with the use of automatically generated diagrams that emphasize reasoning about data dependencies in a LF program. Consider the program in Fig. 1a. The program is constructed textually and is hopefully reasonably easy to read. It defines three reactor classes, `Sense`, `Compute`, and `Actuate`, creates one

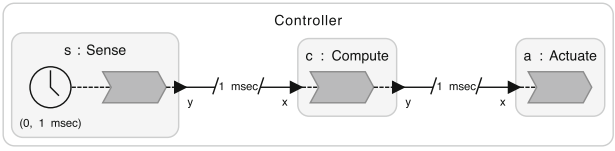
[7] https://github.com/lf-lang/lingua-franca/releases.
[8] https://marketplace.visualstudio.com/items?itemName=lf-lang.vscode-lingua-franca.
[9] https://open-vsx.org/extension/lf-lang/vscode-lingua-franca.

```
target C;
reactor Sense {
    output y:int;
    timer t(0, 1 msec);
    reaction(t) -> y {=
        // ... C code to produce output.
    =}
}
reactor Compute {
    input x:int;
    output y:int;
    reaction(x) -> y {=
        // ... C code to process data.
    =}
}
reactor Actuate {
    input x:int;
    reaction(x) {=
        // ... C code to take action.
    =}
}
main reactor {
    s = new Sense();
    c = new Compute();
    a = new Actuate();

    s.y, c.y -> c.x, a.x after 1 msec;
}
```

(a) Textual LF code



(b) An interactive rendered diagram in the Epoch IDE; selecting `Actuate` in the diagram highlights the definition and instantiation in the textual code.



(c) Expanded diagram, exposing internals of reactors.

Fig. 1. A simple pipeline in LF.

instance of each, named `s`, `c`, and `a`, respectively, and connects their two output ports to their two input ports (with a logical delay of 1 ms) on a line that reads like this:

```
s.y, c.y -> c.x, a.x after 1 msec;
```

Although such a statement is not difficult to read, it is dramatically easier to see what is going on with the automatic diagram rendering provided through LF's IDE support. To underscore this difference, we had an interaction with a user

who had built a program similar to this. This person filed a bug report saying that the logical delay introduced was 2 ms rather than the intended 1 ms. It turned out that this individual was using the command-line tools rather than one of the IDEs. Upon opening the program in Epoch, rendering the diagram reproduced in Fig. 1b, it was immediately obvious that the specified end-to-end delay was in fact 2 ms.

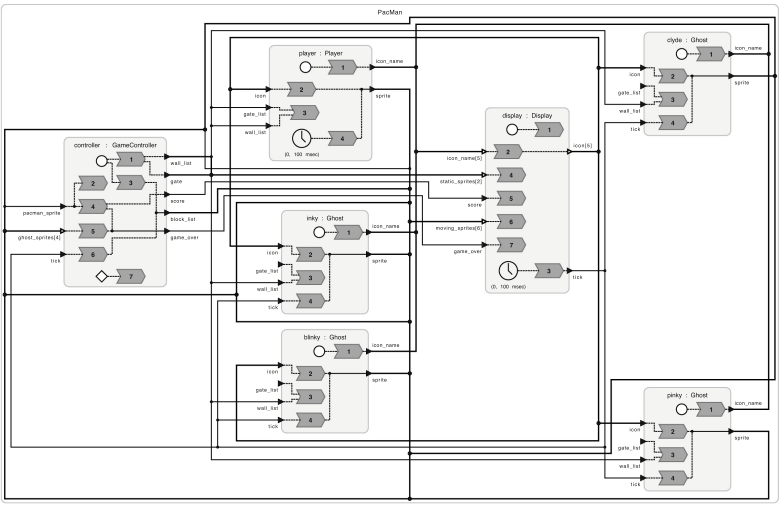
Figure 1b illustrates a number of features that make building and understanding LF programs much easier. For example, the diagram can be used to efficiently navigate in the textual source code. The box labeled `a : Actuate`, which represents the instance `a` of class `Actuate`, is highlighted, by a thick border, having been clicked on by the user. The corresponding instantiation and class definition are both highlighted, by non-white backgrounds, in the source code. Double clicking on any of these boxes will expand it, showing the inner structure of the reactor class. Figure 1c shows all three components expanded, revealing the internal timer in the `Sense` component and the reactions (represented by the chevron shape) in each reactor.

One phenomenon that we have noted is that the automatic diagram synthesis tends to expose sloppy coding and encourages the programmer to modularize, parameterize, and reuse components. Without the diagram synthesis, sloppy structure is much less obvious, and the diagram synthesis gives considerable incentive to clean up the code. As an exercise, we developed a small Pac-Man game based on the 1980 original released by Namco for arcades. We started with straight Python code written by Hans-Jürgen Pokmann and released in open-source form on Github.[10] Our goal was to capture the dynamic aspects of the game in components that could be individually elaborated to, for example, replace the player with an AI. Our first working version of the game rendered as the diagram in Fig. 2a. A more refined version rendered as the diagram in Fig 2b. The incomprehensibility of the first diagram was a major driving factor for the improvements. When executed, this program creates a user interface like that shown in Fig 2c.

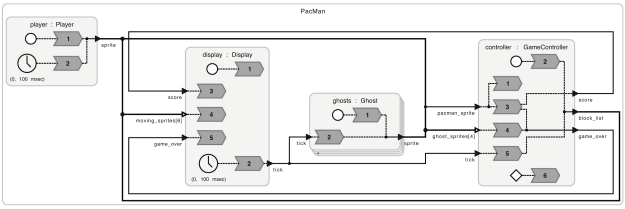
The two programs have identical functionality. The differences between them fall into two categories. First, we used a feature of LF that enables creating a bank of instances of a reactor class using a compact textual syntax that also renders more compactly in the diagram. The textual syntax in this example is the following:

```
ghosts = new[4] Ghost(
    width = {= ghost_specs[bank_index]["width"] =},
    height = {= ghost_specs[bank_index]["height"] =},
    image = {= ghost_specs[bank_index]["image"] =},
    directions = {= ghost_specs[bank_index]["directions"] =},
    character_class = ({= pacman.Ghost =})
)
```

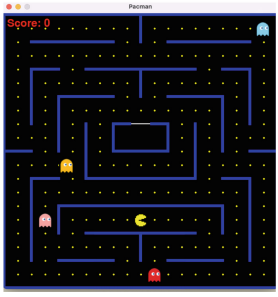
[10] https://github.com/hbokmann/Pacman.



(a) First attempt, with four individual `Ghost` reactors.



(b) Second version, with bank of `Ghost` instances.



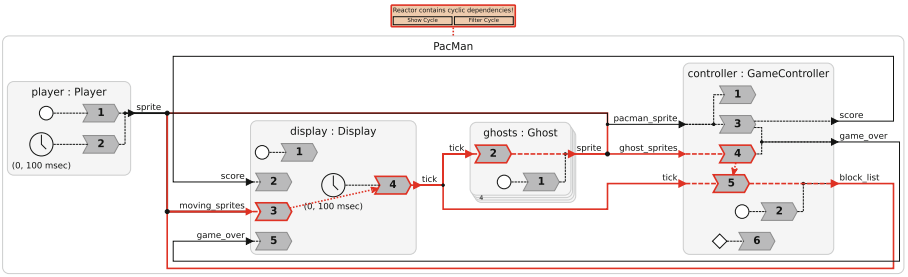
(c) User interface.

Fig. 2. Pac-Man game, with different `Ghost` instantiation strategies shown at same scale, and UI.

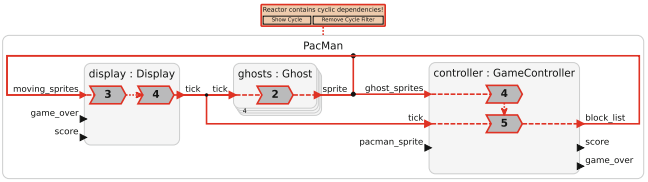
This creates four instances of class `Ghost` and sets their parameters via a table lookup. This version of the program renders much more compactly, as shown at the bottom center of Fig. 2b.

The second significant change was to avoid using messages to pass around static information that does not change during runtime, such as the configuration of the walls. Such information is converted to parameters (when the information differs for distinct instances) or shared constant data structures (when the information is identical across the whole game). We believe that the automatic diagram rendering was *the* driving factor for improving the design.

The diagram also aids during debugging. Take Fig. 3a as an example. This is a small variant of the Pac-Man program that simply moves reaction 2 of the `Display` reactor, which produces the `tick` output, from position 2 to position 4. This move has semantic implications, since in LF, the order in which reactions



(a) Unfiltererd view, showing all components.



(b) Filtererd view, showing only components in cycle.

Fig. 3. Pac-Man game with cyclic dependencies, different views shown at same scale.

are declared affects the order in which they are invoked at each logical time. This is a key property that helps ensure determinacy, as the scheduling of reactions and their accesses to shared variables is determined, at compile time, by their textual order. In the example, however, the scheduling change implied by this move creates a problem because, now, reaction 3 must be invoked before the reaction that produces `tick`, but reaction 3 is ultimately triggered by `tick`. This "scheduling cycle," sometimes also referred to as "causality loop," is extremely difficult to see in the textual source code. To help identify such problems, LF diagrams highlights the offending causality loops in red, as shown in Fig. 3a. The tools also adds dashed arrows that indicate exactly where the flaw may be due to reaction ordering, in this case indicating a path from reaction 3 to 4 in the `Display` reactor.

When LF programs have feedback, such causality loops commonly arise during development and require careful reasoning to get the intended behavior. The LF tools provide a further filter that can help understand the root cause and suggest fixes. By clicking on the "Filter Cycle" button at the top, a new diagram is rendered that includes only the components that are involved in the causality loop, as shown in Fig. 3b. Simply by inspecting this diagram, it is clear that swapping the order of reactions 3 and 4 in the `Display` reactor could resolve the cyclic dependency. The ability to generate such a filtered view automatically depends on the model-view-controller architecture of the system and on its sophisticated automated layout.

3.2 Control Dependencies

So far, the diagrams emphasize the flow of data between components and the ensuing execution dependencies. Decision making, i.e., whether to perform action A or action B, is hidden in the target language code and not rendered in the diagrams. Indeed, rendering the detailed logic of the reaction bodies would likely lead to unusably cluttered diagrams.

Nevertheless, sometimes, the logic of decision making is truly essential to understanding the behavior of a program, even at a high level. Consider a program to control a Furuta pendulum [9], a classic problem often used to teach feedback control. As shown in Fig. 4, it consists of a vertical shaft driven by a motor, a fixed arm extending out at 90° from the top of the shaft, and a pendulum at the end of the arm. The goal is to rotate the shaft to impart enough energy to the pendulum that it swings up, to then catch the pendulum and balance it so that the pendulum remains above the arm. Each of these steps requires a different control behavior, which makes a controller a prime candidate for a *modal model*. It cycles through the three modes, which we will name `SwingUp`, `Catch`, and `Stabilize`.

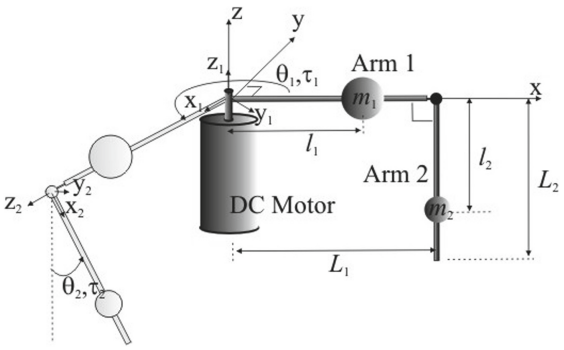


Fig. 4. Schematic of the Furuta pendulum from Wikipedia by Benjamin Cazzolato | CC BY 3.0.

Figure 5a shows an LF program for such a controller based on the design of Eker et al. [28]. This program uses a newly added feature of LF to explicitly represent modal models, programs with multiple *modes* of operation and switching logic to switch from one mode to another. The overall program consists of three connected reactors `Sensor`, `Controller`, and `Actuator`. The diagram in Fig. 5c shows these modes very clearly.

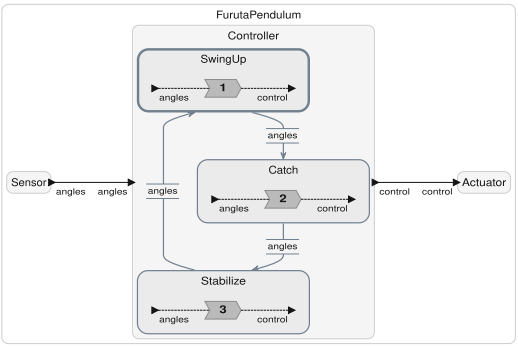
Of course, such modal behavior could easily be written directly within reactors in target-language code without using the modal models extension of LF. Such a realization, shown in Fig. 5b, in this case, is even slightly more compact. Notice that the existence of modes is now hidden in the control logic of the imperative target language, C in this case. As covered in the related work

```
reactor Controller {
  input angles:float[];
  output control:float;

  initial mode SwingUp {
    reaction(angles) -> control, Catch {=
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        SET_MODE(Catch);
      }
    =}
  }

  mode Catch {
    reaction(angles) -> control, Stabilize {=
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        SET_MODE(Stabilize);
      }
    =}
  }

  mode Stabilize {
    reaction(angles) -> control, SwingUp {=
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        SET_MODE(SwingUp);
      }
    =}
  }
}

import Sensor ...
import Actuator ...
main reactor {
  s = new Sensor();
  c = new Controller();
  a = new Actuator();
  s.angles -> c.angles;
  c.control -> a.control;
}
```

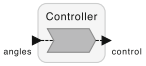
(a) Sketch of the `Controller` reactor code with LF modes

```
target C;
preamble {=
  typedef enum {SwingUp, Catch, Stabilize} modes;
=}
reactor Controller {
  input angles:double[];
  output control:double;
  state my_mode:modes;

  reaction(angles) -> control {=
    if (self->my_mode == SwingUp) {
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        self->my_mode = Catch;
      }
    } else if (self->my_mode == Catch) {
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        self->my_mode = Stabilize;
      }
    } else {
      ... control law here in C ...
      SET(control, ... control value ... );
      if ( ... condition ... ) {
        self->my_mode = SwingUp;
      }
    }
  =}
}
```

(b) `Controller` without LF modes



(c) Diagram for code in Fig. 5a



(d) Diagram for code in Fig. 5b

Fig. 5. Alternative LF realizations to drive the Furuta pendulum, with and without using LF-level modes.

in Sect. 8, there are several approaches for extracting state/modal structures for various programming languages. However, it is notoriously difficult for tools to discern modal structure in such code, and the rendered diagram, shown in Fig. 5d gives no hint.

4 On the Graphical Syntax in Lingua Franca

An important design decision for LF diagrams is how data flow and control flow should visually relate to each other. In LF, reactors are denoted with rounded rectangles and their data flow is visualized with rectangular edge routing. Reactions are depicted by chevrons, a choice that was made intentionally to reduce visual clutter. Because the shape of the reaction already implies a direction (left-to-right), it is unnecessary to show arrows on incoming and outgoing line segments that connect to triggers/sources (attached on the left) and effects (attached on the right).

Following established practices for state machine models, modes are also represented using rounded rectangles. As illustrated in Fig. 5c, modes are distinguished from reactor instances via a differing color scheme, and state transitions are drawn as splines. Initial states are indicated with a thicker outline. Transition labels are drawn on top of the edge, instead of the more common label placement alongside the transition, which is prone to ambiguities. LF diagrams offer various ways for a user to influence the appearance and level of detail. For example, transitions may be shown without labels, or with labels that indicate which inputs and/or actions could cause a mode transition to be taken.

Labeling itself is also subject to design considerations. Traditionally, transition labels include triggers and effects. However, this would require an analysis of target code, and the conditions that actually lead to taking a transition, as well as the effects that result from taking that transition, might become arbitrarily complex. We therefore opted to restrict the transition labels to the events that *may* trigger a reaction, omitting whatever further logic inside the host code determines whether a transition will actually be taken. This appears adequate in practice so far, but if one would want to visualize triggers further, it would for example be conceivable to let the user control what part of the program logic should be shown by some kind of code annotation mechanism. If labels are filtered out, we bundle multiple transitions between the same modes into one to further reduce diagram clutter.

After defining the basic visual syntax for data flow and control flow, the next question is how to combine these different diagram types. One option would be to fully integrate these diagrams. This would mean, for example, that a data flow edge would cross the hierarchy level of the mode to connect any content of the mode. We created several visual mockups for that. However, all variants that included some form of cross-hierarchy edges were considered confusing as soon as they exceeded a trivial size. Additionally, the interaction for collapsing modes to hide their contents and the feasibility with respect to automatic layout algorithms seemed non-trivial. In the end we opted for breaking up these outside connections on the level of each mode. This makes some connections more

implicit, but leads to cleaner diagrams and simplifies the layout task. For ports, we duplicate those used in a mode and represent them by their arrow figure. The name is used to create an association to the original reactor port. In Fig. 5c one can see this in the `angles` input triggering reactions.

5 Auto-Layout of Lingua Franca Diagrams

As explained in Sect. 2, a key to pragmatics-aware modeling and programming is the ability to synthesize graphical views of a model, which is commonly also referred to as "automatic layout". As discussed further in Sect. 7, key to acceptance is a high-quality layout, which should meet a number of aesthetic criteria such as minimal edge crossings.

While the average user should not be required to have a deep understanding of the graph drawing algorithm engineering employed by an IDE, just like the average programmer should not need to know the inner workings of the compiler used, it is helpful to understand some of the basics. This in particular when one wants to tweak the layout in some ways, e. g., via the model order covered below. We consider this basic understanding also essential for tool developers who want to harness layout libraries effectively. Even though these libraries may produce reasonable results out of the box, they typically have numerous parameters that one may adjust to fine-tune the results.

A natural candidate for the synthesis of LF diagrams is the "Sugiyama" algorithm, also known as *layered algorithm* [51], which is, for example, employed in the well-known GraphViz package [5]. The LF tooling uses an extension of the Sugiyama algorithm that can handle hierarchical graphs, hyperedges, and port constraints necessary for LF [43]. That extension is provided by ELK[11], which provides automatic layout for a number of commercial and academic visual tool platforms, including, e. g., Ptolemy II [35]. All the LF diagrams shown in this paper have been generated automatically by ELK, using the tooling described further in Sect. 6. In this section, we very briefly review the basics of the underlying algorithm and point out some of the issues that arise in practice.

The Sugiyama algorithm is divided into five *phases* to break down complexity: 1) The graph is made acyclic, by reversing a set of edges. One tries to minimize the number of reversed edges, to create a clear left-to-right data-flow, which makes it easier to follow edges. 2) Nodes, which correspond to reactors, reactions, actions, and timers, are assigned to vertical *layers* such that edges only occur between layers, which can be seen in the `Display` reactor in Fig. 3a. Here, reactions 1 and 4 are in the second layer, the other nodes are in the first (leftmost) layer. 3) Edge crossings are minimized by changing the node order inside a layer and the port or edge order on a node. With that, the "topology" of the layout, consisting of the assignments of nodes to layers and node ordering within the layers, is fully defined. 4) Nodes are assigned coordinates within layers, trying to minimize edge bends. 5) Layers are assigned coordinates, edges are routed.

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

This basic approach is unchanged since the beginnings of KIELER and the underlying ELK algorithmics. However, a current development, also spurred by the LF effort, is to give the modeler more and easier control over the layout than is traditionally the case. Specifically, the *model order*, which refers to the order in which nodes (e. g., reactors or modes) or edges (e. g., transitions) are declared in the textual LF code, should have a direct influence on the layout topology. This is not the case in standard graph drawing practice, which considers a graph to consist of an *unordered* set of nodes and an *unordered* set of edges. Thus, when computing the layout topology, if the Sugiyama algorithm has multiple solutions of the same aesthetic quality to choose from, it usually picks one of them randomly, which may not be what the modeler wants. One common way to give the modeler influence over the layout topology is to allow *layout annotations* in the textual code, and this is also possible in KIELER. However, this requires some extra effort, also in communicating and familiarizing oneself with these annotations, whereas the model order is a concept that is already inherent in the process of modeling.

For example, concerning Phase 1 of the Sugiyama algorithm, if the graph contains a cycle, then one may randomly choose any edge to be reversed. At least from a graph drawing perspective that only tries to minimize the number of reversed edges, that solution will be as good as any other that only reverses one edge. However, when considering the model order, here the order of declaration of the states, it seems desirable to break cycles such that the model order is preserved. For example, the mode transitions in the `Controller` in Fig. 5c form a cycle[12], meaning that (at least) one transition has to be rendered in the reverse direction. The model order suggests to order `SwingUp` first, and hence to reverse the edge that leads from `Stabilize` back to `SwingUp`. Thus, we say that model order should serve as "tie breaker" whenever there are several equally good solutions to choose from. For modal models, there is also the convention to place initial states first, which in Fig. 5c also happens to be `SwingUp`.

The model order also concerns the textual order of reactor instantiations. This would, e. g., in Fig. 2a suggest to place `clyde` and `pinky` in the same vertical layer as `inky`, `blinky`, and the `player`, which are currently only placed in the rightmost layer because of a random decision of the used edge reversal heuristic.

Also, the order of reactions in the same layer should be taken into account during crossing minimization, as seen in Fig. 3a. The startup and shutdown actions, which are not in the textual model, should be placed at the top and bottom of the first layer, as seen in the `Display` and `GameController` reactor. Furthermore, reactions should be ordered based on their order inside their respective vertical layers, as their numbering suggests.

Reactors are laid out left-to-right, resulting in vertical layers. Modes are laid out top-to-bottom, which turned out to result in a better aspect ratio. Therefore, horizontal layers are created, as seen the `Controller` reactor in Fig. 5c. The greedy cycle breaker reverses edges based on the difference of inputs and outputs, called the outflow. As discussed, if several modes have the same outflow,

[12] The reactions do not form a causality loop in this case, so the program is well-formed.

the model order should be taken into account. The model order of the modes can also be used as a tie-breaker during crossing minimization, as described by Domrös and von Hanxleden [4].

6 Diagram Synthesis Tooling

Providing a comprehensive modeling experience that combines textual editing and interactive diagrams with automatic layout requires sophisticated tooling. The tool has to account for model parsing and editing, diagram synthesis, diagram layout, rendering, and providing a user interface. The KIELER project [8] has been a testbed and birthplace of some key technologies in pragmatics-aware modeling. In the past, it was primarily build upon Eclipse, a versatile and extensible Java-based IDE. Recent development also allows a simultaneous support for Visual Studio Code using the LSP. A major factor in supporting both IDEs is the Xtext framework[13] [7] that enables the development of DSLs with multi-platform editor support. Based on a grammar specification, it automatically creates a parser, as well as editor support for syntax highlighting, content-assist, folding, jump-to-declaration, and reverse-reference lookup across multiple files, in the form of either Eclipse plugins or a language server. Furthermore, Xtext follows a model-based approach that uses the ecore meta-model of the Eclipse Modeling Framework (EMF) to represent the parsing result in a more abstracted format that is easier to process.

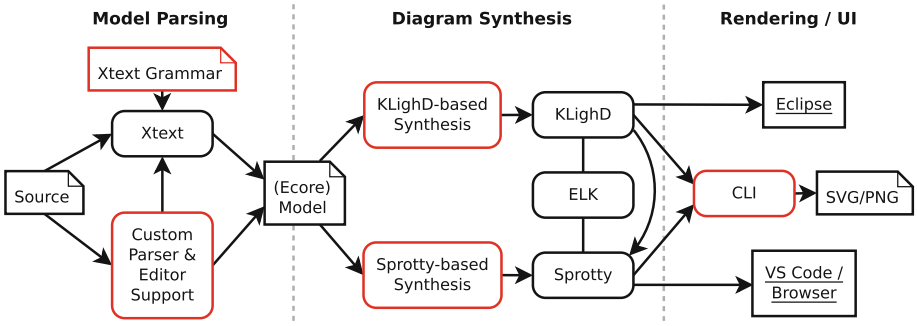


Fig. 6. Overview of the frameworks involved in and around the KIELER project. Sticker-like boxes indicate (in-memory) artifacts, rectangles are IDEs. Rounded boxes represent frameworks or modules, where red ones are language-specific. (Color figure online)

We will now take a look at how these technologies come together in the tool chain created around the KIELER project. Figure 6 gives a schematic overview of the frameworks and steps involved in turning a textual source into an interactive

[13] https://www.eclipse.org/Xtext/.

diagram. There are different paths, we start with the one used by LF. We will discuss alternatives afterwards, as they represent options to utilize this tool chain for a new project. Boxes in red indicate where a user has to provide a model-specific implementation.

Since the very start, LF was based on an Xtext grammar, so it was relatively quick and easy to integrate with the KIELER framework. As Fig. 6 illustrates, Xtext produces an ecore model. Next is a model-specific diagram synthesis that defines the graphical elements and their appearance. In case of LF, the KIELER Lightweight Diagrams (KLighD)[14] [41] framework is used. Based on the internal representation (also an ecore model), KLighD performs layout by using ELK and is able to display the diagram in Eclipse. In case of LF, the result is a custom Eclipse product, called Epoch. It is also possible to bundle the Xtext and KLighD infrastructure into a command line tool to export diagrams without an IDE. To incorporate diagrams in VS Code, KLighD utilizes the Sprotty framework[15], a more basic diagramming framework built with web technologies. The final VS Code extension for LF only needs to bundle KLighD into the language server and depends on a KLighD extension.

A project that does not rely on Xtext, Eclipse, or even Java can still utilize the given tool chain. For example the Blech language[16] [12] has a custom parser written in `F#` but uses the Sequentially Constructive StateCharts (SCCharts) language [16] as an exchange format with diagram support, to visualize an abstracted view of its structure [30]. In a Java-based context, even non-ecore-based model representations can be used to create KLighD diagram syntheses. For a fully web-based solution targeting only VS Code and browsers, Langium[17] could be a viable replacement for Xtext, and instead of KLighD, a Sprotty-based synthesis would be written. Sprotty is likewise capable of performing layout using ELKJS.

7 Modeling Pragmatics—Obstacles, Opportunities and Outlook

As explained, the practicality and value of modeling pragmatics has been validated now for some time. However, it is still far from standard practice, which begs the question of what is holding progress back. We believe the underlying issues are as much of psychological as of technical nature. In the following, we review some of the impediments, but also cover some (overall rather positive) user feedback beyond the LF context and reflect on what we see as the main challenges ahead.

[14] https://github.com/kieler/KLighD.
[15] https://github.com/eclipse/sprotty.
[16] https://www.blech-lang.org/.
[17] https://langium.org/.

7.1 A Priori User Concerns

When first presented with the concept of pragmatics-aware modeling, the reactions typically range from all-out enthusiasm to pronounced scepticism. Here are some of the more common reactions.

"I Want Full Control" As Gurr states, "people like having feedback and control" [13]. There is the fear to lose control when handing the layout problem to the machine. Of course, there is some truth in this—when using a compiler, people cannot fully control how the assembler is written anymore. In a similar vein, Taylor commented on What You See is What You Get (WYSIWYG) for typesetting and states its two-faced nature [53]:

> "Why has WYSIWYG succeeded so spectacularly, while other typesetting approaches have languished? I think WYSIWYG's main appeal is that it appears to offer its users superior cybernetics—i. e., feedback and control. To the extent that you can trust its authenticity, the screen gives immediate feedback. Acting on that feedback, the user then has immediate control. And people like having feedback and control. [. . .]
> It is worth remarking in this context that while WYSIWYG may have won the hearts and minds of designers through "superior cybernetics," the degree of control that such programs offer may be more illusory than real. Or perhaps it is more accurate to say that desktop publishing programs let you fiddle interactively with the details of your typography until the cows come home, but they do not let you control the default behaviors of the composition algorithms in a way that efficiently and automatically delivers the kind of quality typography that was formerly expected of trade compositors."

However, in practice, experience shows that one often is satisfied with *any* readable layout and thus does not invest efforts in making the layout sound with a freehand Drag-and-Drop (DND) editor. Therefore, in practice the issue of full control may be less relevant than it may seem at first. We also see an analogy here to the usage of auto-formatters in coding that is well-accepted by now and that helps to achieve a consistent "look" for textual code. Still, when propagating automatic layout, one should listen carefully to the potential users and try to extract what it really is that they want control of. Often this is not the individual pixel-by-pixel placement, but something more abstract, that might even be integrated into automatic layout. We see the incorporation of model order into the layout sketched in Sect. 5 as a prime example for that.

Graphical ≠ Informal. "I'm not a graphical person, I'm a formal person" is another, not untypical comment. However, graphical vs. textual is a question of syntax, and both diagrams and text can have arbitrarily formal or informal semantics. Therefore, the proposal here is to combine the best of both worlds and not to play text off against diagrams.

Layout Algorithms Not Good Enough. One claim of this work is that automatic layout must be so good that people are willing to replace manual placing and routing by it. However, a common opinion is that the layout algorithms today do not meet this requirement. And indeed, there have been several examples of tools that provided some auto-layout functionality that produced rather unsatisfactory results. However, there also are positive examples, also in commercial tools; e. g., LabView has employed sophisticated layout algorithms in its "clean-up" functionality [39], which seems quite satisfactory (see also a quote citing LabView further below).

Not Aware of Productivity Loss. Many decision makers seem to be unaware of the productivity losses of the usual freehand DND editing and manual static view navigation for diagrams. Practitioners of graphical modeling realize the benefits more immediately. Numbers from industry partners indicate that about 30% overhead is induced by manual layout. However, current trends towards textual DSL modeling might indicate that some people already see drawbacks with the traditional graphical modeling. Still, the consequences should not be to replace diagrams by text but to enhance the pragmatics of diagrams.

Loose the Mental Map. A spontaneous fear expressed for automatic layout is about losing the mental map that one may already have of a diagram. However, for rather stable layouters and employing them consistently from the beginning, the mental map can be kept very well. Conversely, for radical model changes the value of preserving the mental map seems overrated. Experience shows that a clean and reproducible auto layout results in more comprehensible models than a very effort-prone manual incremental layout that tries to change as little as possible. Especially when working with different people, maybe in different roles, adherence to a consistent layout style may be more important than preserving the mental map of an individual developer throughout the design process. And as discussed in Sect. 5, incorporating the model order into the layout process may help to align a modelers mental map with the automatically created layout.

7.2 Feedback After Usage and Lessons Learned

Get Used to It—Don't Want to Miss it Anymore. Even if it may be unfamiliar to work only with auto layout in the beginning, users get used to it. Finally they find it hard to go back to other tools employing manual layout again. In a survey conducted by Klauske [21] with around thirty practicioners at Daimler that used a version of Simulink enhanced with automatic layout [23], users reported massive time savings and expressed the wish for keeping that functionality. That feedback was also put forward to the tool supplier.

Interactive Layout Overrated. First, some users request ways to interactively influence the layout, either by specific means to configure the layout algorithm or by simply tagging regions as "manually laid out, don't touch!" Therefore

many configuration options have been added to the KIELER Infrastructure for Meta Layout (KIML) including an interactive mode for some layout algorithms and tagging regions as manual layout. While this seems to ease initial acceptance, in the long term people often get used to full automatic layout, such that these interactive features are only rarely used any longer.

User Interface Must be Simple. Users usually have no sense about the layout approach of a specific layout algorithm or requirements of the tool. Therefore, the user interface to call layout must be simple and intuitive. Otherwise, people tend to not use it at all. One example is the first approach to the routing problem in the Ptolemy II editor Vergil. It introduced five buttons, all changing the diagram massively in different ways while users without any background usually did not understand what the differences were. Therefore, the functionality was barely adopted and required a different approach allowing a cleaner interface with only one button. While configurability is very good and important, this tells us that user interfaces have to be very clear and simple. They should always provide meaningful default configurations that lead to good results if the user is not willing to spend any efforts in understanding all customization options.

7.3 A Short Experience Report from the Railway Domain

An early adaptor in industry of SCCharts [16] and its pragmatics-aware modeling support based on KIELER is the German railway signalling manufacturer Scheidt & Bachmann System Technik GmbH (S&B). Just like LF, SCCharts are edited in a textual form and get embedded into host code (Java or C++). Engineers at S&B widely recognize the benefits of textually editing models right next to the host code, and are accustomed to using automatically generated diagrams for communication purposes. Typical communication tasks are documentation of the system architecture and detailed behavior. An important use case where the automatically synthesized, always up-to-date diagrams are particularly useful, is the onboarding of new team members working on the SCCharts models. A major advantage is that complex domain logic is abstracted such that it can be read and written and understood by domain experts who are not familiar with the host programming languages. This takes advantage of the fact that SCCharts models, like LF models, can be written and visualized before filling in host code. This allows a good division of work where domain experts can directly contribute domain logic as real technical artifacts, without a laborious and error-prone manual transition of domain knowledge into code. In fact, this practical aspect was a main driver for adopting the SCCharts tooling, and we consider this a great success.

In the early days, the generated diagrams were quite often printed on large "wallpapers" to collaboratively reason about design and find flaws in the implementation. Bugs in the models still get analyzed mainly in static diagrams rather manually. However, in practice is also desirable to inspect actual run time behavior, and although simulation of SCCharts is possible, the problems

usually appear when used in the complex contexts in which they are embedded in some distributed application. Thus, the simpler the context outside of the model is, the better are the possibilities of analyzing the behavior of the whole system. That experience has prompted the development of a run-time debugging capability that seamlessly integrates SCCharts and host code [6]. However, this is still rarely used because the design problems are often too timing dependent and depend on the interaction of multiple distributed models, making them unlikely to surface in debugging sessions. Instead, when a rare unexpected behavior occurs, logged event traces of the distributed models are used to reproduce the state transitions to find the model flaw. Thus, the S&B use case might very much benefit from explicitly modeling the distribution of the system parts and the messaging between them with the LF approach.

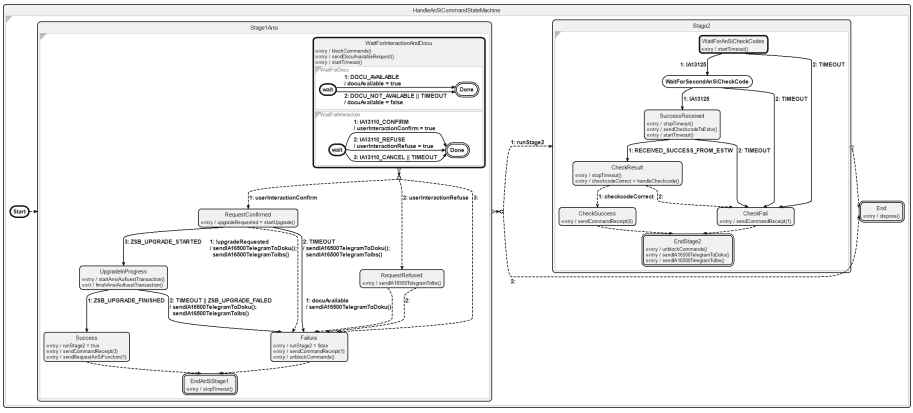


Fig. 7. A typical reactive SCChart diagram for the railway domain, illustrating label management.

Despite its limitations, the high level of abstraction of SCCharts and the possibility to view the models as diagrams has increased productivity. Especially textual editing next to automatic diagram generation, with diagrams that link elements back to the textual source, facilitates rapid changes such as refactorings and additions to models. In a classic what-you-see-is-what-you-get editing approach, these operations might be considered too costly to do, which tends to deteriorate code quality and increase technical debt on the long run. A typical, still relatively small SCChart is shown in Fig. 7. One feature that can be seen here is *label management* [42], which in this case is configured such that transition labels are shortened by introducing line breaks between triggers and actions and between actions. As most of our SCCharts tend to have even longer labels than in this example, label management—which also allows truncating labels at a freely defined width—is an essential feature to keep diagrams manageable.

One word of caution, however, about tooling in general. Fancy complex modeling environments with a shiny Graphical User Interface (GUI) are nice for the

editing experience, but may be counterproductive for solving the obsolescence problem. In industry, one needs tools that last for a long period of time. We need to make sure that we are able to work on our technical source artifacts for the whole life-span of our product, i. e., at least 15–20 years in the railway domain. While it is remarkable that the Eclipse-based tooling of KIELER is in active development and use for over a decade now, the successor generation with web-based, cloud-ready tooling is already there, as outlined in Sect. 6. Hence, it seems essential in practice to create modular tooling, where the core functionality can be used even when the GUI originally developed is no longer technically supported by operating systems or hardware platforms and might be replaced by some next generation frontend. For SCChart, this prompted the development of a commandline tool for SCChart compilation and essential diagram generation, while we use any additional tooling frontend as "pragmatic sugar".

To conclude, the reception of the overall approach within the S&B engineers is positive, despite the aforementioned complexities and caveats. We also observe that by now several generations of additional on- and offboarding freelancers have come to value the pragmatics-aware SCCharts tooling and thus might spread the word in their next projects.

7.4 The Challenges Ahead

LF illustrates a comprehensive example of a pragmatics-aware modeling experience. From the fist step the modeller has an abstract visualization of the program available that can be used to illustrate and document the model[18].

From the modelling perspective, this approach can be extended to include simulation, as in the GMOC debugging and simulation tool for LF [3] that feeds live data into the diagram, or visualizing intermediate steps in the compilation, such as dependencies determined by the compiler, e. g. as available in the KIELER tool for SCCharts [49]. With the support of the LSP and Visual Studio Code, the pragmatics-aware modeling support is also available in the latest IDEs (see Sect. 6). We hope that the the pragmatic development idea will flourish in this large and vibrant ecosystem.

From the documentation perspective, pragmatics-aware diagrams can improve the effectiveness of static documentation and maybe ease classical certification processes. Ongoing work automatically synthesizes diagrams for System-Theoretic Process Analysis (STPA) [56]. The commercial EHANDBOOK tool[19] uses ELK and other technologies described in this paper to serve interactive documentation for electronic control unit (ECU) software in the form of Simulink diagrams. Web-based technologies (Sect. 6) combine the classical written documentation with embedded interactive diagrams, including dedicated views or links that navigate the reader into different locations and configurations of the model and that support collaborative browsing.

[18] https://www.lf-lang.org/docs/handbook/overview.
[19] http:/www.etas.com/ehandbook.

While the fundamental concepts and technologies for pragmatics-aware modeling are already realized and validated, there are still challenges ahead that need to be addressed to further improve the usability and ultimately the acceptance of this approach.

Finding the Right Abstraction Level. Having support for a pragmatics-aware modeling is one side, but creating effective views for complex models is another aspect. Here, abstraction is key. Visualizing the general coordination structure of a program, such as in LF diagrams, is probably more economic and helpful than, e. g., generating a huge control-flow diagram that displays all underlining machine instructions. Also, while we in this paper focussed on synthesizing abstract views from implementation-level textual models, system architects may want to start with abstract models as well. As mentioned earlier, the separation of high-level coordination language and low-level host code is one step in that direction.

Browsing Complex Systems. Related to the abstraction challenge, in particular large and complex models require further development of filtering and browsing techniques. We see an approach inspired by Google Maps as a promising direction to tackle this issue [15].

Configuring Automatic Layout Effectively. Layout algorithms typically provide many options to influence the way the layout works and thus the aesthetic of the final result. In our experience, achieving very good results for non-trivial diagram types requires an at least basic understanding of how the underlying graph drawing algorithm works, just like crafting high-performance software requires some basic understanding of modern compilers and computer architectures. Also, layout documentation and configuration guidance should be improved. Luckily, diagrams can aid in this, to interactively showcase options and their effects.

Lowering the Entry Barrier. Creating pragmatic concepts for new DSLs is easier than integrating it into existing design methodologies and tools. Yet, from a commercial and psychological perspective, low invasiveness is a circuital factor in the acceptance of these concepts. Hence, it is important to provide low entry barriers and allow augmenting existing languages and tools, rather than trying to replace them. For example, mode diagrams were added to Blech [30] in a way that did not impact the existing tool chain, but augmented it as an add-on. Again, the change into web-based technologies is an important step in this direction. A related challenge is to provide pragmatics capabilities in a way that one the one hand integrates well into existing tools and work flows, but on the other hand is robust against technology changes, see the obsolescence issue raised earlier. Technology-independent standards like LSP and the aforementioned more recent GLSP might help.

Establishing a Diagrammatic Modeling Mindset. In our experience, users do not normally consider the automatic diagram synthesis as an option for visualizing internal or conceptual structures of their model. Addressing this requires a change in the mindset of the tool developers, but we are convinced the effort will pay off.

8 Related Work

The Interactive Programmatic Modeling proposed by Broman [2] also advocates to combine textual modeling with automatically generated graphical views. He argues for an MVC pattern as well, albeit with different roles for model and controller than in our case, as for him the controller concerns the parameterization and the model is an execution engine. He identifies several problems with standard modeling approaches, including the *model expressiveness problem*, implying that graphical models become less intuitive when trying to capture more complex models. While we argue here that model complexity is actually an argument in favor of (automatically generated) graphical models, we concur with his point that graphical models are particularly helpful when they do not aim to capture a system in full detail, but rather provide abstractions.

The *intentional programming* paradigm proposed by Simonyi, in a sense, also advocates a separation of model and view [47]. There, a developer should start with formulating rather abstract "intantions" that successiveley get refined working on a "program tree", which can be viewed using an arbitrary and non-permanent syntax. One may argue that we here take the inverse approach, where the developer directly authors a (detailed) model, but gets assisted by a continuously updated visual, abstract documentation.

Computational notebooks, such as Jupyter[20], follow the *literate programming* paradigm proposed by Knuth that integrates documentation and source code [24]. They are increasingly popular in data science and are related to modeling pragmatics in that they also advocate a mix of representations.

The Umple framework also follows a pragmatic approach in that it also explicitly aims to provide the best of textual and diagrammatic worlds [27]. It allows both textual and graphical editing of UML models, and it can automatically synthesize class diagrams and other diagram types, using GraphViz layouts. However, these diagrams appear to be rather static, without further filtering/-navigation capabilities as we propose here.

There are several approaches for extracting, and typically also visualizing, state structures from textual sources. As mentioned in Sect. 6, the tooling for the textual Blech language [12] allows to automatically visualize an abstracted modal view [30]. Unlike in LF, modes are not explicit in the textual Blech source, but must be derived from Esterel-like await statements and overall imperative control flow. Kung et al. [26] and Sen and Mall [46] present ways to extract state machines from object-oriented languages. They analyze the behavior of classes and infer state machines describing the class behavior. Giomi [10] and Said et al. [40] describe state machine extractions based on control flow to represent the program's state space.

Less common is the automatic synthesis of actor diagrams, as proposed by Rentz et al. for legacy C/C++ code [37]. Ishio et al. [20] have investigated interprocedural dataflow for Java programs. They propose Variable Data-Flow Graphs (VDFGs) that represent interprocedural dataflow, but abstract from

[20] https://jupyter.org/.

intraprocedural control flow. There are further tools and frameworks to reverse engineer diagrams from C code. CPP2XMI is such a framework as used by Korshunova et al. [25] for extracting class, sequence, and activity diagrams, or MemBrain for analyzing method bodies as presented by Mihancea [31]. UML class models are extracted from C++ by Sutton and Maletic [52], and another framework for the analysis of object oriented code is presented by Tonella and Potrich [54]. Smyth et al. [48] implemented a generic C code miner for SCCharts. The focus was to create semantically valid models from legacy C code, whereas in modeling pragmatics, the aim typically is to synthesize graphical abstractions. However, all these tools and frameworks show the importance of reverse engineering and presenting views to programmers.

There are several pragmatics-oriented proposals that go beyond the basics presented here. Early on, Prochnow et al. proposed *dynamic focus and context* views that highlight the run-time state of a system by presenting active regions in detail and collapsing others [33], realized in the KIEL Integrated Environment for Layout (KIEL) tool, a predecessor or KIELER. KIEL also included *structure-based editing*, which combines WYSIWYG editing (without a textual source) with continuous automatic layout [34]. More recent work includes the *induced data flow* approach that synthesizes actor-oriented diagrams from SCCharts [55], and interactive compilation models that visualize intermediate transformation results, which can be helpful for users and compiler developers alike [50].

Finally, as explained, a key enabler for the pragmatics approach presented here is the large body of work produced by the graph drawing community. Conversely, pragmatics has become a significant use case that prompted advancements in graph drawing [1]. Interested readers may look at the publication lists of the ELK contributors, which include flagship venues and several dissertations. However, like writing a compiler, the authoring of a graph drawing library that works well in practice is a significant piece of engineering. Fortunately, the contributors to ELK and other projects covered in Sect. 6 not only strived for publishable algorithms, but also made good software engineering a priority. For example, one of the lessons learned over the years is that one has to strike a good balance between functionality and maintanability, and if in doubt, one should probably favor the latter.

9 Summary and Conclusions

To recapitulate, the main driver of modeling pragmatics as presented here is to enhance developer productivity by combining the best of the textual and visual modeling worlds. Traditionally, visual models 1) are manually created, often in a rather time-consuming process in particular when one wants to maintain good readability in an evolving model, 2) have a fixed level of detail, and 3) may get out-of-date with respect to an implementation model. In the pragmatic approach, visual models 1) are created automatically, 2) can be customized and apply filtering, and 3) are consistent with the model from which they are synthesized. Clearly, the capability to automatically create abstract, up-to-date visual

diagrams from textual models is a boon for documentation. This encompasses documentation in a very broad sense, where it is an integral, supportive part of the design process itself. We postulate that whenever developers may need to document or communicate a model aspect that cannot be readily gleaned from looking directly at source code, a viable approach might be to automatically generate a diagram that fits on a screen. Even though automatic graph drawing is not a trivial process and a still a research topic in its own right, the primary challenge in modeling pragmatics might be less the task of computing good layouts, but rather to filter/abstract the model appropriately; i. e., once one knows *what* to visualize, the question of *how* to visualize it can be answered quite well with today's technology.

Looking back at the story of modeling pragmatics so far, the main conclusion may be that it takes some perseverence to change tools and habits, just like it probably took quite some convincing to wean programmers off their assembly writing when compilers came about. In some way, it seems a bit like a chicken-and-egg problem; the average user is not aware of its potential, and thus the average tool provider sees no sufficient demand for it. Also, for an existing language with existing, traditional tooling, the conversion barriers seem significantly higher than for a newly created language without any such legacy.

However, progress seems possible, also for existing languages and tools that already have a large user base. As exemplary point in case, consider the following (abbreviated) exchange, on a Mathworks user forum[21]. On Feb. 23, 2012, User K E inquired: "Is there an automatic way to rearrange a Simulink block diagram so that it is easy to read?" Staff Member Andreas Gooser responded March 14, 2012: "A year ago, I worked with users and developers (I called it myself Simulink Beautifier :-)) to find out if such things are possible. I found myself convinced that this is a non-trivial undertaking, if you try this in a generic way, as there are too many criteria/rules." On that, User Ben noted Nov. 14, 2012: "Mathworks should invest the energy to develop an auto-cleanup feature. Tools like these are expected for serious and relevant 21st century software. Yes, it is non-trivial, but take a look at National Instrument's LabView—they've implemented such a feature beautifully and it saves hours of aggravation especially if you are developing complex code." That thread then fell silent for eight years. However, Staff Member Anh Tran posted Jan. 31, 2020: "From MATLAB R2019b, you can improve your diagram layout and appearance by opening the FORMAT tab on the toolstrip and click on Auto Arrange. This command can realign, resize, and move blocks and straighten signal lines". We have not evaluated the quality of the layout ourselves and have not heard of user feedback yet. However, already back at the 2010 MathWorks Automotive Conference, Klauske and Dziobek of the Daimler Center for Automotive IT Innovations (DCAITI) presented a Simulink extension for doing automatic layout that received quite positive user feedback, from actual users and during the workshop presentation, which makes us hopeful [21–23].

[21] https://www.mathworks.com/matlabcentral/answers/30016-clean-up-simulink-block-diagram.

In case of LF, the situation was certainly quite different from an established commercial tool that already has a very large user base accustomed to doing things in a certain way. We used that chance to harness modern, state-of-the art tooling infrastructure, building on insights won over many years in earlier projects such as Ptolemy and SCCharts. The pragmatics-aware approach presented here, with the automatic synthesis of abstract diagrams, is part of the LF toolchain since early on. As it turned out, while that approach of continuous, automatic diagram synthesis is still anything but standard practice, it does not seem to command much attention in that is basically "just there" and taken for granted. And when somebody is unhappy with certain aspects of some specific diagram, it is usually not an actual tool user working on some LF application, but somebody from the "pragmatics team" themselves with a close eye on graphical detail, who then also sets about finding a fix for it.

Thus, to conclude, we think that it certainly has helped in the case of LF to have considered pragmatics from the start, rather than as an afterthought. Also, as explained, some properties of LF such as its separation of coordination language and target language make it particularly natural to automatically synthesize abstract diagrams, as part of the design process and for documentation purposes. However, we also believe that much of the underlying pragmatics concepts are also transferrable to other languages and contexts. We thus conclude this report with an open invitation to try out the approaches presented here, and to share experiences and direct inquiries to the authors or to one of the public message boards associated with the open-source tools presented here.

References

1. Binucci, C., et al.: 10 reasons to get interested in graph drawing. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 85–104. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_6
2. Broman, D.: Interactive programmatic modeling. ACM Trans. Embed. Comput. Syst. **20**(4), 33:1–33:26 (2021). https://doi.org/10.1145/3431387
3. Deantoni, J., Cambeiro, J., Bateni, S., Lin, S., Lohstroh, M.: Debugging and verification tools for LINGUA FRANCA in GEMOC studio. In: 2021 Forum on specification Design Languages (FDL), pp. 1–8 (2021). https://doi.org/10.1109/FDL53530.2021.9568383
4. Domrös, S., von Hanxleden, R.: Preserving order during crossing minimization in Sugiyama layouts. In: Proceedings of the 14th International Conference on Information Visualization Theory and Applications (IVAPP 2022), Part of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022), pp. 156–163. INSTICC, SciTePress (2022). https://doi.org/10.5220/0010833800003124
5. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–484. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45848-4_57
6. Eumann, P.: Model-Based Debugging. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science (2020). https://rtsys.informatik.uni-kiel.de/biblio/downloads/theses/peu-mt.pdf

7. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA 2010, Reno/Tahoe, Nevada, USA, pp. 307–309 (2010). https://doi.org/10.1145/1869542.1869625
8. Fuhrmann, H., von Hanxleden, R.: Taming graphical modeling. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 196–210. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_14
9. Furuta, K., Yamakita, M., Kobayashi, S.: Swing-up control of inverted pendulum using pseudo-state feedback. Proc. Inst. Mech. Eng. Part I: J. Syst. Control Eng. **206**(4), 263–269 (1992). https://doi.org/10.1243/PIME_PROC_1992_206_341_02
10. Giomi, J.: Finite state machine extraction from hardware description languages. In: Proceedings of Eighth International Application Specific Integrated Circuits Conference, pp. 353–357. IEEE (1995). https://doi.org/10.1109/ASIC.1995.580747
11. Green, T.R.G., Petre, M.: When visual programs are harder to read than textual programs. In: Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics) (1992)
12. Gretz, F., Grosch, F.J.: Blech, imperative synchronous programming! In: Proceedings of Forum on Specification Design Languages (FDL 2018), pp. 5–16 (2018). https://doi.org/10.1109/FDL.2018.8524036
13. Gurr, C.A.: Effective diagrammatic communication: syntactic, semantic and pragmatic issues. J. Vis. Lang. Comput. **10**(4), 317–342 (1999). https://doi.org/10.1006/jvlc.1999.0130
14. Haberland, H., Mey, J.L.: Editorial: linguistics and pragmatics. J. Pragmat. **1**, 1–12 (1977)
15. von Hanxleden, R., Biastoch, A., Fohrer, N., Renz, M., Vafeidis, A.: Getting the big picture in cross-domain fusion. Inform. Spektr. (2022). https://doi.org/10.1007/s00287-022-01471-2
16. von Hanxleden, R., et al.: SCCharts: sequentially constructive statecharts for safety-critical applications. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014), pp. 372–383. ACM, Edinburgh (2014). https://doi.org/10.1145/2594291.2594310
17. von Hanxleden, R., Lee, E.A., Motika, C., Fuhrmann, H.: Multi-view modeling and pragmatics in 2020—position paper on designing complex cyber-physical systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 209–223. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_11
18. Harel, D., Rumpe, B.: Meaningful modelling: what's the semantics of "semantics"? IEEE Comput. **37**(10), 64–72 (2004). https://doi.org/10.1109/MC.2004.172
19. Hoffmann, B., Minas, M.: Defining models–meta models versus graph grammars. In: Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010). Electronic Communications of the EASST, vol. 29. Berlin, Germany (2010). https://doi.org/10.14279/tuj.eceasst.29.411
20. Ishio, T., Etsuda, S., Inoue, K.: A lightweight visualization of interprocedural dataflow paths for source code reading. In: Beyer, D., van Deursen, A., Godfrey, M.W. (eds.) IEEE 20th International Conference on Program Comprehension (ICPC), pp. 37–46. IEEE, Passau (2012). https://doi.org/10.1109/ICPC.2012.6240506
21. Klauske, L.K.: Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus. Ph.D. thesis, Technische Universität Berlin (2012)

22. Klauske, L.K., Dziobek, C.: Improving modeling usability: automated layout generation for simulink. In: Proceedings of the MathWorks Automotive Conference (MAC 2010) (2010)
23. Klauske, L.K., Schulze, C.D., Spönemann, M., von Hanxleden, R.: Improved layout for data flow diagrams with port constraints. In: Cox, P., Plimmer, B., Rodgers, P. (eds.) Diagrams 2012. LNCS (LNAI), vol. 7352, pp. 65–79. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31223-6_11
24. Knuth, D.E.: Literate programming. Comput. J. **27**(2), 97–111 (1984). https://doi.org/10.1093/comjnl/27.2.97
25. Korshunova, E., Petković, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In: 13th Working Conference on Reverse Engineering (WCRE 2006), pp. 297–298. IEEE Computer Society, Benevento (2006). https://doi.org/10.1109/WCRE.2006.21
26. Kung, D., Suchak, N., Gao, J.Z., Hsia, P., Toyoshima, Y., Chen, C.: On object state testing. In: Proceedings Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 1994), pp. 222–227. IEEE (1994). https://doi.org/10.1109/CMPSAC.1994.342801
27. Lethbridge, T.C., et al.: Umple: model-driven development for open source and education. Sci. Comput. Program. **208**, 102665 (2021). https://doi.org/10.1016/j.scico.2021.102665
28. Liu, J., Eker, J., Janneck, J.W., Lee, E.A.: Realistic simulations of embedded control systems. IFAC Proc. Vol. **35**(1), 391–396 (2002). https://doi.org/10.3182/20020721-6-ES-1901.00553
29. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a Lingua Franca for deterministic concurrent systems. ACM Trans. Embedd. Comput. Syst. (TECS) **20**(4), Article 36 (2021). https://doi.org/10.1145/3448128. Special Issue on FDL'19
30. Lucas, D., Schulz-Rosengarten, A., von Hanxleden, R., Gretz, F., Grosch, F.J.: Extracting mode diagrams from Blech code. In: Proceedings of Forum on Specification and Design Languages (FDL 2021). Antibes, France (2021). https://doi.org/10.1109/FDL53530.2021.9568375
31. Mihancea, P.F.: Towards a reverse engineering dataflow analysis framework for Java and C++. In: Negru, V., Jebelean, T., Petcu, D., Zaharie, D. (eds.) 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, pp. 285–288 (2008). https://doi.org/10.1109/SYNASC.2008.7
32. Morris, C.W.: Foundations of the Theory of Signs, International Encyclopedia of Unified Science, vol. 1. The University of Chicago Press, Chicago (1938)
33. Prochnow, S., von Hanxleden, R.: Comfortable modeling of complex reactive systems. In: Proceedings of Design, Automation and Test in Europe Conference (DATE 2006), Munich, Germany (2006). https://doi.org/10.1109/DATE.2006.243970
34. Prochnow, S., von Hanxleden, R.: Statechart development beyond WYSIWYG. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 635–649. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_43
35. Ptolemaeus, C.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org, Berkeley (2014). http://ptolemy.org/books/Systems
36. Reenskaug, T.: Models – views – controllers (1979). Xerox PARC technical note

37. Rentz, N., Smyth, S., Andersen, L., von Hanxleden, R.: Extracting interactive actor-based dataflow models from legacy C code. In: Basu, A., Stapleton, G., Linker, S., Legg, C., Manalo, E., Viana, P. (eds.) Diagrams 2021. LNCS (LNAI), vol. 12909, pp. 361–377. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86062-2_37
38. Roestenburg, R., Bakker, R., Williams, R.: Akka in Action. Manning Publications Co. (2016)
39. Rüegg, U., Lakkundi, R., Prasad, A., Kodaganur, A., Schulze, C.D., von Hanxleden, R.: Incremental diagram layout for automated model migration. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), pp. 185–195 (2016). https://doi.org/10.1145/2976767.2976805
40. Said, W., Quante, J., Koschke, R.: On state machine mining from embedded control software. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 138–148. IEEE (2018). https://doi.org/10.1109/ICSME.2018.00024
41. Schneider, C., Spönemann, M., von Hanxleden, R.: Just model! - Putting automatic synthesis of node-link-diagrams into practice. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2013), San Jose, CA, USA, pp. 75–82 (2013). https://doi.org/10.1109/VLHCC.2013.6645246
42. Schulze, C.D., Lasch, Y., von Hanxleden, R.: Label management: keeping complex diagrams usable. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2016), pp. 3–11 (2016). https://doi.org/10.1109/VLHCC.2016.7739657
43. Schulze, C.D., Spönemann, M., von Hanxleden, R.: Drawing layered graphs with port constraints. J. Vis. Lang. Comput. Spec. Issue Diagram Aesthet. Layout **25**(2), 89–106 (2014). https://doi.org/10.1016/j.jvlc.2013.11.005
44. Seibel, A.: Personal communication (2017)
45. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003). https://doi.org/10.1109/MS.2003.1231146
46. Sen, T., Mall, R.: Extracting finite state representation of Java programs. Softw. Syst. Model. **15**(2), 497–511 (2014). https://doi.org/10.1007/s10270-014-0415-3
47. Simonyi, C.: The death of computer languages, the birth of intentional programming. Technical report MSR-TR-95-52, Microsoft Research (1995)
48. Smyth, S., Lenga, S., von Hanxleden, R.: Model extraction for legacy C programs with SCCharts. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), Doctoral Symposium. Electronic Communications of the EASST, Corfu, Greece, vol. 74 (2016). https://doi.org/10.14279/tuj.eceasst.74.1044. With accompanying poster
49. Smyth, S., Schulz-Rosengarten, A., von Hanxleden, R.: Guidance in model-based compilations. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018), Doctoral Symposium. Electronic Communications of the EASST, Limassol, Cyprus, vol. 78 (2018). https://doi.org/10.1007/978-3-030-03418-4_15
50. Smyth, S., Schulz-Rosengarten, A., von Hanxleden, R.: Towards interactive compilation models. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11244, pp. 246–260. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03418-4_15

51. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybern. **11**(2), 109–125 (1981). https://doi.org/10.1109/TSMC.1981.4308636
52. Sutton, A., Maletic, J.I.: Mappings for accurately reverse engineering UML class models from C++. In: 12th Working Conference on Reverse Engineering (WCRE 2005), pp. 175–184. IEEE Computer Society, Pittsburgh (2005). https://doi.org/10.1109/WCRE.2005.21
53. Taylor, C.: What has WYSIWYG done to us? Seybold Rep. Publ. Syst. **26**(2), 3–12 (1996)
54. Tonella, P., Potrich, A.: Reverse Engineering of Object Oriented Code. Springer, New York (2005). https://doi.org/10.1007/b102522
55. Wechselberg, N., Schulz-Rosengarten, A., Smyth, S., von Hanxleden, R.: Augmenting state models with data flow. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling. LNCS, vol. 10760, pp. 504–523. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95246-8_28
56. Young, W., Leveson, N.G.: An integrated approach to safety and security based on systems theory. Commun. ACM **57**(2), 31–35 (2014). https://doi.org/10.1145/2556938