Enhancing Graphical Model-Based System Design—An Avionics Case Study

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

***Abstract*—Graphical model-based system design is very appealing. However, there exist many different formalisms, with different semantics—as far as they do have well-defined semantics—and differing capabilities of the accompanying tools. In this paper, we present a case study from the avionics domain and report on the experiences in using different modeling languages and tools. The focus here is on the *pragmatics* of modeling, *i.e.* , the practical process of building and inspecting graphical models. The underlying application is a high-lift flap system, which is highly safety-critical and served as a demonstrator within the *Dependable Embedded Components and Systems* (DECOS) project that explored the design of distributed dependable systems build on time-triggered architectures.**

Specifically, we compare a realization in the Safety Critical Application Development Environment (SCADE), a commercial tool from Esterel-Technologies, with a design in the Kiel Integrated Environment for Layout (KIEL), a research tool that allows to explore novel model handling paradigms. Hence we compare traditional graphical drag-and-drop WYSIWYG modeling with alternative, productivity enhancing approaches. We conclude with a brief outlook on future extensions which will tightly integrate with existing tools based on the Eclipse platform.

I. Introduction

The use of graphical representations for abstract specification of problems or solutions in computer science well established. The most well-known—*graphs*—implicitly go back to Euler in the 18th century, cf. [23].

Visually enriched and equipped with source code generation one might regard them as the next logical step in developing systems just as the steps from object code to assembler and to higher-level textual programming languages. In programming most developers avoid to go back to the assembler roots unless the application requires squeezing out last performance optimizations by manual tricks. So higher programming languages are ruling the development processes and one might ask why graphical representations have not yet taken over the scepter.

The problem is that there exists a growing set of modeling languages and development environments for them. In the control engineering well known is Matlab/Simulink [27] or LabView [28] while control flow is better expressed with Statecharts introduced by Harel in 1987 [22]. These days are many different incompatible Statecharts semantics known [31]. The Unified Modeling Language (UML) [39] tries to cope with these differences by standardization. However, this was done with a focus on language syntax and with a lack of precise semantics [13], which makes it difficult to describe system behavior unambiguously. This lack of semantics is accompanied by a quite bewildering variety of graphical syntaxes.

Consequently, the UML is often regarded as too general, which has lead to the concept of Domain Specific Modeling Languages (DSMLs) [40]. It is by now standard practice to create new DSMLs and to build custom editors and tools for these. Often, this is still done manually; alternatively, one may employ a framework that supports the generation of new graphical languages. The Eclipse Platform [11], with its subproject Graphical Modeling Framework (GMF) [18], allows to synthesize customized graphical interactive drag-and-drop editors for new DSMLs from some basic specifications. Hence there now emerges a variety of new graphical languages from the Eclipse community, each for a special purpose or a special domain. With this diversity of graphical formalisms without a real standard, the different technologies get developed and evolve rather independently. This includes different approaches to the pragmatics of model handling, *i.e.* , how models are created, edited, visualized, inspected, simulated, compared and so on.

In this paper, we present a case study from the avionics domain and report on the experiences in using different modeling languages and tools. The focus here is on the pragmatics of modeling, here specifically on how models are created, structured, and visualized during simulation.

While within the Dependable Embedded Components and Systems (DECOS) project the demonstrator was mainly developed with the commercial modeling environment SCADE, afterwards we transferred some of the models to the KIEL tool in order to compare their usability. These modelling environments are presented briefly in the remainder of the introduction. The rest of this paper is structured as follows. Section II presents the application from the avionics domain considered in this paper, including a brief description of the DECOS project where it served as a demonstrator. Section III then describes the modeling of the application with SCADE, using traditional modeling approaches as far as their pragmatics is concerned. This is compared in Section IV with alternative editing and simulation visualization paradigms

provided by KIEL. We conclude with an outline of prospective future developments in modeling pragmatics within the Eclipse framework.

A. SCADE

The *Safety Critical Application Development Environment* (SCADE) is a modeling tool of Esterel-Technologies [12]. It allows to graphically define dataflow for control loops like Matlab/Simulink and control-flow like Statecharts (*e.g.* Matlab/Stateflow), but uses a precise formal semantics built on the synchronous model of computation. Hence, it can be seen as a graphical editor on top of synchronous, textual languages [4], [32]. In SCADE up to version 5, which is the version used for the DECOS aerospace project, it was mainly Lustre for dataflow [21] and Esterel for control-flow [5]. Since version 6 SCADE employs its own SCADE textual language, which tries to merge the two basic paradigms as described by Colaço *et. al.* [9].

A certified code generator can generate C-Code to be used in aerospace safety-critical systems that must be developed according to DO-178B [37].

B. KIEL

The *Kiel Integrated Environment for Layout* (KIEL) project is a test bed to experiment with novel system modeling paradigms. It has a focus on the *pragmatics* on how graphical models are created, inspected, analyzed and visualized [41], and employs the Statechart formalism. A main concept is to leverage automatic assistance by the computer to relieve the developer of tedious tasks in practical model handling [35]. For example, the *editing* of graphical models with the traditional WYSIWYG drag-and-drop editors can be quite time consuming, compared to performing similar edits to a textual program. Changing a Statechart by inserting a new state at some position typically requires the developer to first manually make space for the new state by enlarging the parent state and moving all surrounding objects. Additionally inexperienced users often come up with rather unstructured models difficult to comprehend, just due to their manual layout.

The basic idea is to separate the graphical representation of a model from the model itself, and to let the designer work on the model, nots its representation. This is akin to the Model-View-Controller familiar from software engineering [36]; the critical difference is that here we employ MVC not to develop the modeling tool, but we let the user of the modeling tool employ MVC to develop some application.

The key enabler for providing MVC at the tool user level is the systematic application of *automatic layout* for graphical models. Modeling itself is no more a drag-and-drop interaction where the developer manually positions items on the screen, instead the system performs the full layout automatically. The user can employ a macro-based structural approach to perform changes of the model *structure* and the layout is decided by the framework [34]. For example, a single command may add a successor state to an existing state, or upgrade a simple state to a composite or a parallel state. An alternative editing paradigm is provided by a fully synchronized text-editor that allows to edit a textual representation of the diagram whose changes are immediately reflected in the graphical representation [33].

II. The Application: A High-Lift Flap System

The application considered here was developed within the *Dependable Embedded Components and Systems* (DECOS) project, an Integrated Project within the European Union Framework Program 6 [10], [19], [24]. The project explored approaches to build a system based on *components of the shelf* (COTS) components and employs different layers of abstractions and tools to develop and deploy the application. Aiming at mixed criticality systems, including safety-critical ones, it was built upon the Time-Triggered Architecture (TTA) [25] with different possible lower level implementations, including the TTP [26] and FlexRay [3] communication protocols.

Next to the basic technology development in DECOS, there were three demonstrator sub-projects, each employing the new technology in a different domain. One domain is industrial control [20], another domain is the automotive industry [8]. The third domain, which centered on the application considered here, is the aerospace sector [14]–[16], [38], represented in DECOS by Airbus Germany [1].

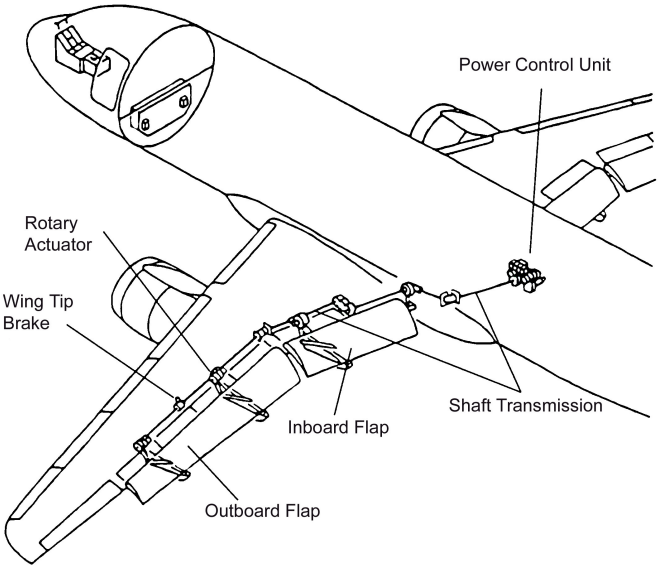


Fig. 1. Schematical View of a High-Lift Flap System [29]

The aerospace demonstrator implements a high-lift flap system, which is motivated by the state-of-the-art of such systems [29]. The main purpose of a flap system as depicted in Fig. 1 is to increase the size and concavity of the wings and by this to increase the high-lift temporarily. This is used at low speed for landing and take off purposes only. There are several reasons why the high-lift system is safety-critical.

- First, its proper function is required during landing procedures. Hence a system freeze during flight might cause severe landing problems.
- Second, it is required that all flaps on both wing sides are at all times perfectly synchronized. Otherwise it

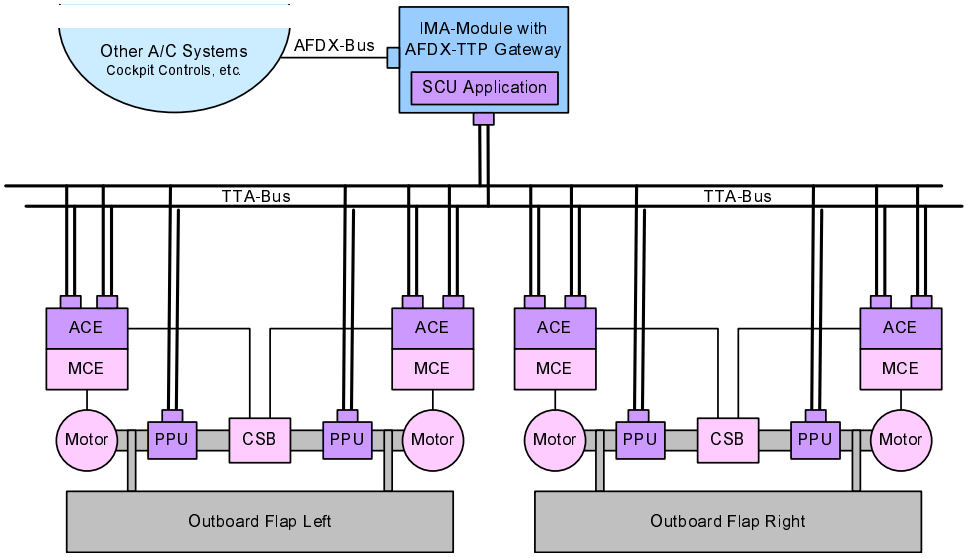


Fig. 2. Electronically Synchronized System Schematics

would influence the flight attitude and would most likely ultimately lead to a crash.

- Third, the surface area of one flap panel is quite large, much larger for example than the ailerons. For this reason the mechanical forces on the panels are tremendous. If the holding mechanisms (brakes, motors) of the panel shafts would fail, the high forces would feedback onto the actuator shafts and cause them to spin. Ultimately centrifugal forces might then destruct the installation and seriously damage the wings.

In the following, we consider two deployment alternatives, which are the traditional, mechanical approach and a novel, electrical approach.

A. Mechanical Synchronization

For the perfect synchronization of all flap panels, state-of-the-art mechanisms perform the alignment purely mechanically. Every flap panel is driven by a rotary actuator that gets its actuation by a rotating shaft lengthwise within the wing. Another long cross-shaft lies across the whole fuselage and connects the left wing flap panel with the right wing flap panel. A central power control unit actuates the shaft with the help of two electrical motors. If one motor fails, the other side will move the whole shaft with half the speed by a speed summing differential. This way both sides are always perfectly synchronized unless a shaft breaks or blocks, whereupon the shaft brakes freeze the system. However this scenario is highly unlikely.

The drawback of this solution is obvious: The shaft across the fuselage is very inflexible and the development of the tip-to-tip shaft transmission very laborious and includes the internal construction of the fuselage into the development of the flap system, which makes the system neither modular nor reusable.

B. Electrical Synchronization

The approach is to perform the flap panel synchronization electrically instead of mechanically and to remove the cross-shaft through the fuselage. The system architecture used in the project is depicted in Fig. 2. For the sake of simplicity only two panels were developed, one for each wing. Each panel is still actuated by an individual cross-shaft, but they are not mechanically connected anymore. Additionally, each shaft is actuated by a set of two motors at the ends of the shaft. Hence the two panels could be moved independently, and the electrical synchronization is responsible for avoiding this. The two motors on each panel are still mechanically connected for redundancy reasons: if one motor would fail, the other one could still rotate the shaft with half-speed. Due to this direct mechanical connection, the motors must be precisely synchronized in order to avoid a torque-fight. Each shaft is equipped with a *cross-shaft brake* (CSB) to freeze the system when it is not moving, and two redundant sensors, called *position pick-off unit* (PPU) . Each motor is controlled on low level by its *motor control electronics* (MCE) and on application level by an additional *actuator control electronics* (ACE). MCE and ACE are closely coupled and for simplicity could be regarded as one unit. The control loops are closed between the ACEs and PPUs so that the shafts can be moved to any exact position with high precision. Additionally, one central *system control unit* (SCU) communicates with the cockpit and takes pilot flap position requests on the one hand, and calculates set-point values for the shafts and monitors the system and commands fault reaction strategies on the other hand.

For this safety-critical application with small control loop periods (2ms), the field bus connecting SCU, ACEs and PPUs is a time-triggered architecture as proposed by the DECOS project described above. The communication between SCU and other

aircraft systems use the standard Airbus Avionics Full Duplex Switched Ethernet (AFDX) [2] communication, which extends standard Ethernet with enhancements for predictability and dependability.

III. The Initial Model of the Application—SCADE

The high-lift flap system basically separates in two different functions:

A. Closing the control loop between actuator and sensors, influenced by control commands of the central control unit communicating with the cockpit and
B. monitoring of the system, detecting faults and react on them appropriately to get as much fault-tolerance as possible.

A. Basic Control

The basic control algorithms are implemented rather straight forward. The basic loop is shown in Fig. 3. The whole system is for safety reasons purely time-triggered, and so is the field bus. Hence the central system control unit (SCU) sends to the actuator control electronics (ACE) periodically set-points for the cross-shaft position at every time-cycle. To synchronize all four actuator stations, the SCU does not send a final destination value, but instead calculates the whole movement trajectory (including in- and decreasing of speed) of the cross-shaft. At each time cycle only the position and speed that the four stations should have at that time are supplied. To calculate the correct trajectory, the SCU considers the current plane angle and speed and the flap angle requested by the pilot via the control levers. The latter two values are used to do auto-retraction of the panels in certain flight modes in order to avoid damage to them.

The ACEs control the speed command to their motors such that the given set-point values are always best matched. A simple proportional controller with differential additions is implemented in the ACE module.

The whole control loop is implemented in SCADE and spans over many modules and many hierarchy levels. Due to its standard navigation and view mechanisms, its graphical representations are too complex to be presented fully in this paper. To give an example, the proportional controller of the ACE is shown in Fig. 4. This pure dataflow diagram shows the model inputs on the left—the set-point values for motor speed and shaft position and the measured current shaft position as obtained by the PPU—and its output to the motor control electronics (MCE) on the right.

The control loops are implemented mainly with SCADE dataflow and little control-flow. The more challenging part is the monitoring and fault reaction implementation presented in the following.

B. Fault Detection and Reaction

1) Local Monitoring: To determine what can go wrong in the system, first a *Failure Modes and Effects Analysis* (FMEA) was performed by the mechanical engineers within the DECOS aerospace sub-project. Basic failure modes were identified, the effects on the system, how the modes could be detected by the system (without additional sensors) and what an appropriate reaction would be. Main failure classes are classified as follows:

1) loss of motor power or disconnection from shaft,
2) powered runaway in both directions,
3) jam of motor, gear or cross-shaft,
4) breakage of shaft at different positions,
5) cross-shaft brake (CSB) failure, either released or set, and
6) communication failure.

Some of these cases the system can tolerate and still be operational with reduced performance. *E.g.* when one motor gets disconnected somehow, the other motor of that flap panel is still able to drive the cross-shaft and hence the whole flap panel with reduced speed. A loss of the CSB can be coped with by the motors by actively holding and controlling the cross-shaft to certain positions where the system should be held. While the whole system is designed following the *single fault hypothesis* [30] such that it tolerates every single failure, it still can tolerate certain multiple concurrent faults as well. For example one motor may fail at each flap panel and both brakes could go out of order and the flaps could still be maneuvered. Nevertheless a jam of the system or two motor failures at the same panel would disable the proper operation and hence the system must be completely frozen on both sides to avoid asynchronous states.

The implementation in SCADE mixes both dataflow and control-flow aspects. The first version was implemented in SCADE version 5 and hence the two paradigms are explicitly separated (just the same as in Simulink). The first part is schematically shown in Fig. 5.

The main state of each flap panel is modelled by a complex Statechart which gets described in more detail in the next section. States represent detected error modes and the currently active reaction strategy. Hence transitions between these states are responsible for the error detection and reaction decision.

The guards of the transitions must therefore somehow relate to the sensor readings of the PPUs and the set-point values of the SCU. To determine failure modes, those values need to be processed and compared. Doing this in transition trigger expressions of pure Statecharts would lead to very long and incomprehensible textual transition guards. As in SCADE version 5 it is not possible to mix dataflow expressions into the Statecharts, the dataflow got pulled out of the chart and the calculations are done in advance. So the relevant variables in the monitors are fed into complex comparison and check operators in the dataflow level of the model and get translated into simple Boolean triggers for the control Statechart. The Statechart then can decide by querying simple Boolean signals in what system modes it should switch and output this mode and some other commands or failure codes.

This is the main point where the application would benefit from the new mix of paradigms introduced in SCADE version 6. In SCADE 5 it is only possible to embed a Statechart into a dataflow diagram and use simple dataflow variables as signal inputs for the Statechart. The other way round is only possible

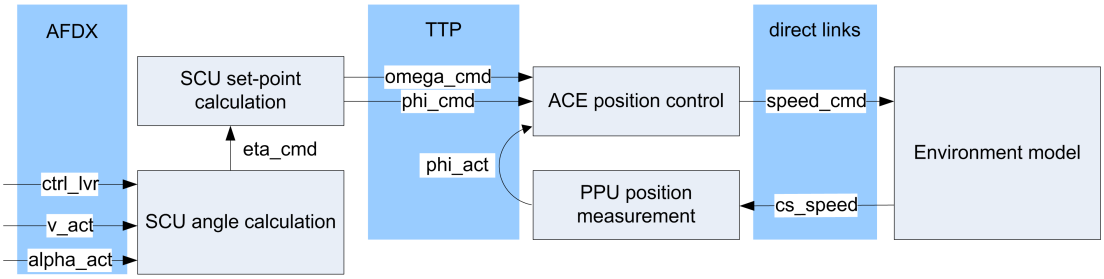


Fig. 3. Basic Control Loop

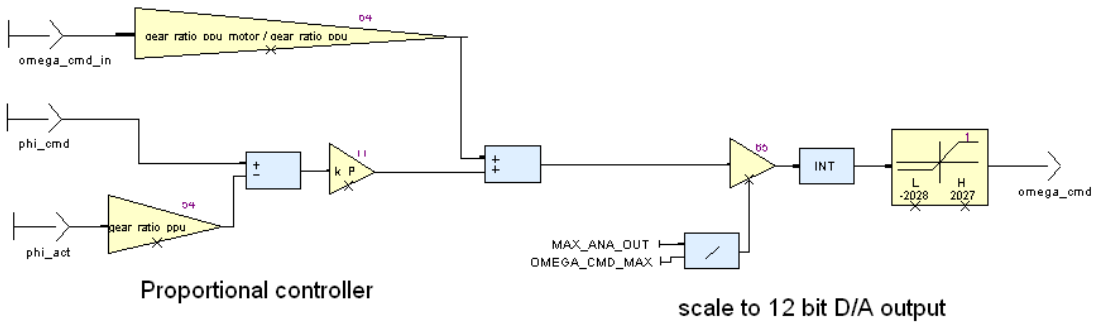


Fig. 4. Simple Loop Controller in SCADE

in SCADE 6, where one may mix dataflow and control-flow in any hierarchy level. Dataflow components may be placed within states for example. A state of a state machine can define some kind of modal dataflow, which means the dataflow is only active when the state is active. Hence in every state the outputs of the subsystem are required to be written at most once. If an output is not written in some state, it holds its old value.

With these mechanisms our monitor could be implemented much more intuitively. Certain dataflow operations as checks and calculations on data could reside within the system states that require those checks to be done. This would lead to simpler development and simulation as the *mental map* of the model is better preserved by this locality.

2) Global Commanding: The local monitoring results of each flap panel get combined and compared in a high-level control module schematically depicted in Fig. 6. The relevant variables get processed by the wing monitors described above. Important synchronization messages are passed between the two monitors for the basic events, *i.e.* the other side is either driving, holding or switched to emergency mode. Determined failure codes and states of one flap panel are processed by two respective sub-command modules that decide to activate motors or the CSBs.

A high-level command unit decides upon the rotation speed according to the error state of all flaps and passes this information to the main set-point calculation unit that combines it with the destination flap angle and outputs the set-points for speed and position to the ACEs.

3) System Setup: The whole system is modeled in SCADE and augmented with additional architecture configuration that is specified by the DECOS internal tool chain. This especially concerns the time-triggered communication and operating system setup. SCADE generates C-code that is deployed on the target platform, a physical test-rig at the Technical University Hamburg-Harbug. Additionally the system can be simulated prior to its physical integration. For that a continuous-time high precision model of the mechanical parts of the system is available that can be connected to the SCADE controllers. SCADE is a purely discrete modeling environment following the strict synchronous semantics and does not directly support continuous-time modeling. Hence the environment model was designed in Matlab/Simulink. During simulation Simulink and SCADE are executing in parallel while they exchange simulation data in each simulation step. This interaction and setup is explained in more detail elsewhere [16].

Structuring the models in SCADE with hierarchy requires to create a completely new sub-model for each hierarchical operator. Graphically it is not possible to directly display one's interior within the same graphical view. Hence if the user might want to see the context of many operators, he or she might end up with a screen cluttered as shown in Fig. 7. However, to arrange all components in this fashion on the screen is not only time consuming (it took about 20 minutes of opening, positioning and resizing windows to create this view), but ultimately rather useless as the overall structure is still unclear and the individual charts become too small to be legible. Hence, this is not the way one would

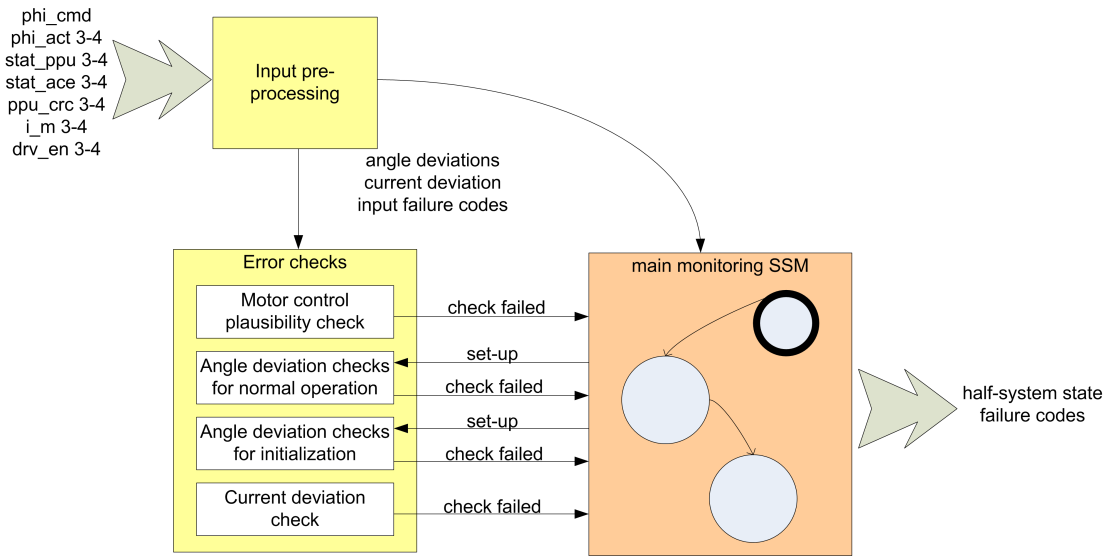


Fig. 5. Schematical View on Local Monitoring Concept

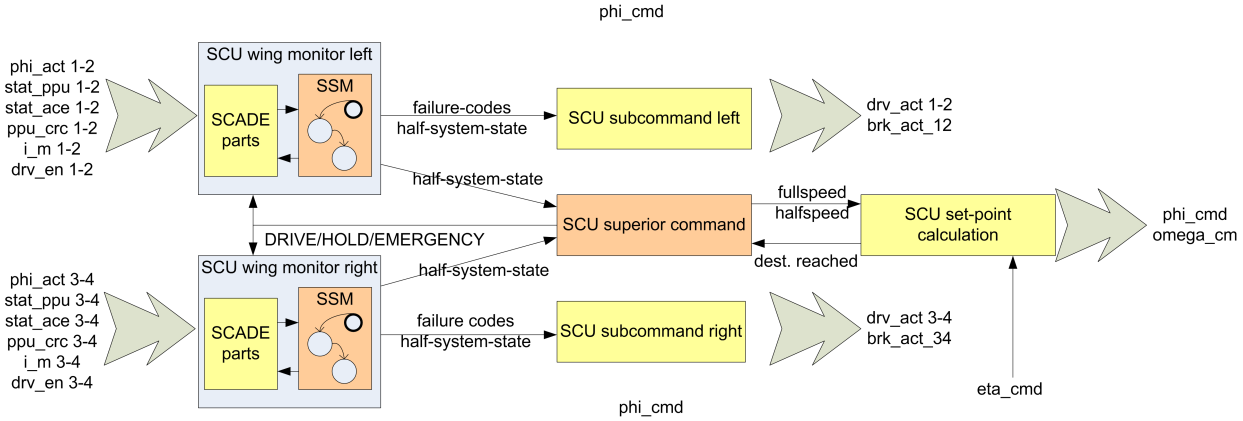


Fig. 6. Schematic Overview of the Global Command Unit

actually work with the tool. Usually one displays only one or a few operators at a time, and there is a lot of time-consuming navigation between windows, going back and forth in the model to learn about the system and how the dataflow interplays between the components. Hence to identify the context in which one operator is located is not trivial and slows down the development process.

IV. Alternative Approaches—Kiel Integrated Environment for Layout (KIEL)

The main monitoring Statechart of the local monitoring subsystems originally has been modeled in SCADE, *i.e.* the Safe State Machine (SSM) editor of the SCADE suite. This is a separate tool and only partly integrates into the dataflow editor. In the newer version the new mixture of control- and dataflow is tightly integrated into one development tool.

For documentation and demonstration purposes this Statechart has been re-modeled using the *Kiel Integrated Environment for Layout* (KIEL) tool. While the graphical representation of the original SSM is hardly adequate to briefly describe in a paper, the dynamic view structure of KIEL allows to generate multiple views with different complexity of the chart.

Fig. 8 is a screenshot of the Statechart from SCADE's SSM editor tool. It shows the most important states and even shows some hierarchy. However, because of the complexity, some of the lower level states (*brake, activehold, onemot, twomot*) were defined as hidden macro states, *i.e.* they are refined by further substates, but this is not shown in this view. The decision which states are hidden and which are visible within a diagram is more or less final and static and therefore becomes an important design decision by the developer for the goal of creating a comprehensible statechart.

Here we have chosen this level of detail in this view, in order to give the user a good first overview of the diagram. However if he or she has to dig deeper in the diagram, only the lower former hidden macrostates are shown and you lose

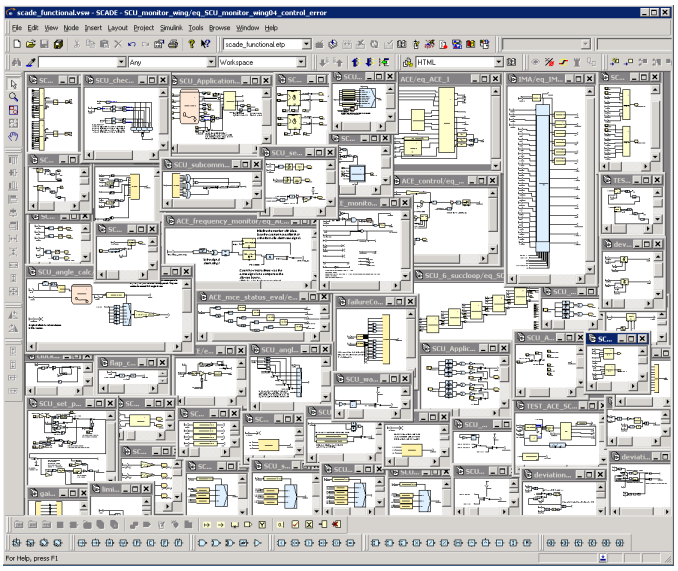


Fig. 7. Seeing the big picture in SCADE

the view of the surrounding context.

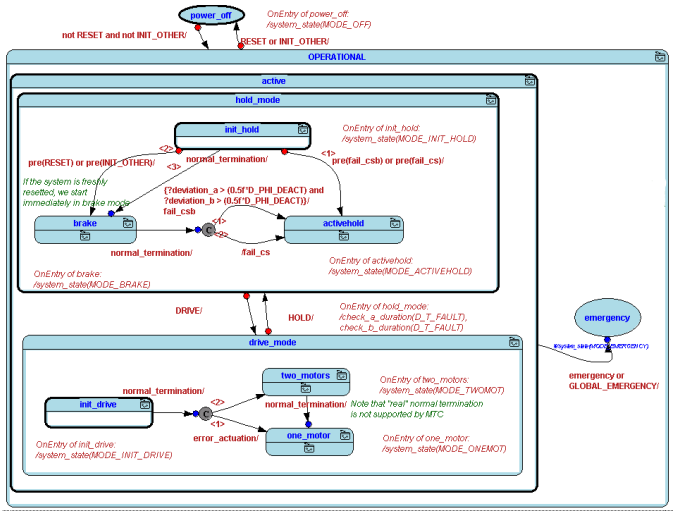


Fig. 8. Monitor SSM in SCADE Editor

Opposing to that Fig. 9 shows almost the same Statechart in the KIEL tool. It represents the monitoring Statechart of one flap panel on the highest level with all states unfolded. This allows to get a first impression of the complexity of the whole chart, but it is certainly difficult to discern any details.

KIEL has a built-in Statechart simulation engine based on SSM semantics. During a simulation run one can leverage the *Dynamic Charts* paradigm [34]. The idea is to use *focus and context* mechanisms to display the currently interesting parts of the diagram in full detail in the focus. This "interesting" part of the model in a Statechart is naturally the currently active state with all its ancestors. The other states, the context, is visible only with reduced detail. The sibling states are still present but get folded so that their contents is not visible anymore.

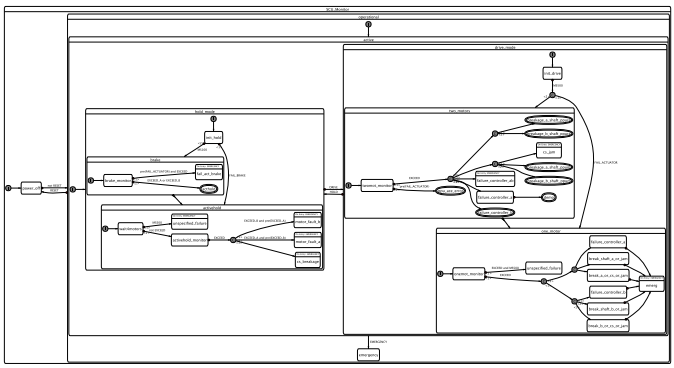


Fig. 9. Overview of the Monitor SSM in KIEL

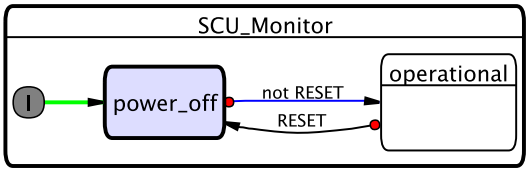


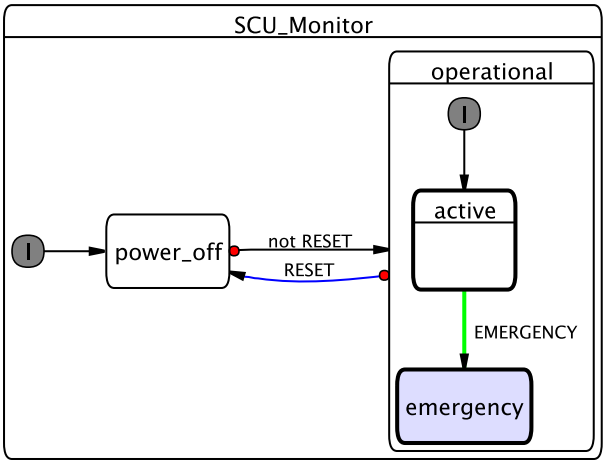
Fig. 10. First Dynamic View on the Model

Fig. 10 shows this dynamic focus and context in action when the simulation is started for the Statechart. As you can see, the chart gets simplified very much and its meaning becomes obvious: In the beginning the system is in a power off state and it might switch eventually into an operational state.

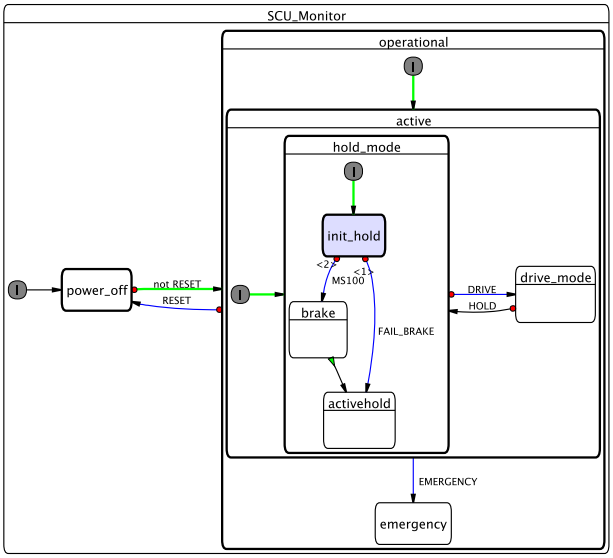
The system in operational state is shown in Fig. 11a. This reveals the interior of this state to the next hierarchy level. The panel might either be active, *i.e.* still capable of being moved, or in emergency mode, which means the whole system needs to be frozen, because some intolerable fault (most likely more than one consecutive fault at a time) has happened. In the latter case there are no further refinements and hence no other substates.

During normal operation the system will be in the active state as shown in Fig. 11b. This reveals two substates: The hold state in which the flap panel is not moving and the drive state in which the flap panel is moving in one direction. Either of the states is active and the hold mode is the initial one. It itself has two basic substates and a simple init state. When the system is held, it can either be held by the brake (brake state) or if the brake has failed it can actively be held by the motors (activehold state).

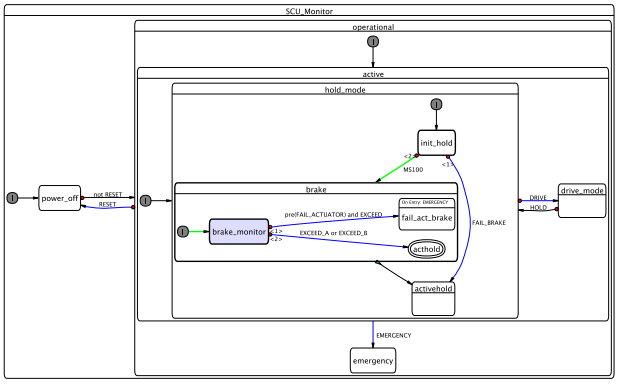
The brake mode is illustrated in Fig. 11c and depicts the lowest hierarchy level of this Statechart: The brake monitor state is the default state here and the other states indicate severe error conditions. An exceed signal of one of the control loops would indicate that this station has moved too far from its set-point position where the flap should be held. In this case it can be assumed that the brake is not working properly. The brake state switches to its following acthold state, which is marked *final* and hence would follow the *normal termination* transition to exit the parent brake state and switch to the activehold



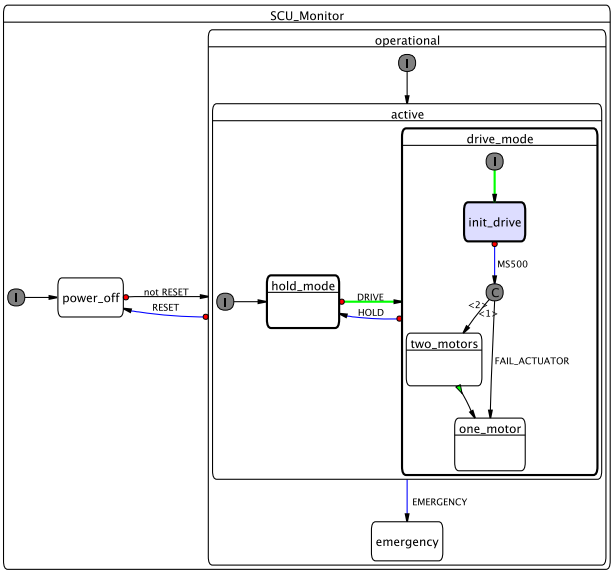
(a) The Operational State here in Emergency Mode



(b) The Operational State entering Active Mode



(c) The Brake Mode



(d) The Drive Mode and its Direct Children

Fig. 11. Different Dynamic Views in KIEL

state (which also has some internal monitoring states that are not explicitly shown here).

Given a flap movement command of the pilot, the panels will start to move in the desired direction. Therefore the system switches from hold mode to drive mode. This also has two main states and an initializer. The main states twomot and onemot say whether one or two motors of the panel are still operational. This triggers different movement speeds and also different foregoing monitoring states and respective guards to their error states. The drive state with its direct children is shown in Fig. 11d.

In this paper for every new dynamic view we need to present a new image and for better readability need to adjust the zoom level. During simulation in KIEL this is morphed smoothly between the dynamic views in order to preserve the user's *mental map* of the system. Hence it is quite comprehensible to follow from one view to the next. The dynamic views are computed on the fly, using a hierarchical layout engine that recursively performs a bottom-up layout across the hierarchy levels. The computation times for the layout are in practice negligible and typically well in the sub-second range.

To create the models, KIEL's interaction mechanisms were employed, namely the *macro-based* editing and the synchronized *textual representation*. With these methods, the developer does not need to manually move around boxes and arrows, think about what makes a comprehensible layout or

spend time making space for new objects. Hence creating those models—though they are a bit simplified—took only a fraction of time compared to build the models in the classical drag-and-drop fashion in SCADE.

V. Conclusion

We presented a safety-critical aerospace application that was graphically modeled in the commercial tool SCADE and the experimental KIEL platform.

SCADE has its strengths in its rigorous, formal basis as it builds upon synchronous languages and supports qualified code generation. In questions of user interaction it reflects the current state-of-the-practice, which means to be a drag-and-drop static view graphical editing and simulation environment. Either during editing of a diagram or simulation the user is busy with manually navigating to the views of interest or manually arranging or fixing the graphical layout for the diagram.

The KIEL tool shows the idea of systematically employed automatic graphical layout and relieves the developer of many laborious mechanical tasks. Additionally it is the enabler for further techniques, as illustrated with the dynamic charts for this application. Those can help to better understand not only the *structure* of an application but also its *behavior*.

Despite its good approaches, the KIEL tool has some major restrictions. Its prototype monolithic Java implementation does yet integrate only with a small set of other tools (Esterel Studio, Stateflow, ArgoUML). The only supported graphical modeling language is Statecharts, though it supports multiple dialects of those. Only a small set of concrete layout engines is integrated, that sometimes do not lead to optimal layouts. Especially the fact that one whole diagram is layouted at once gives sometimes bad results for big diagrams.

VI. Future Work

We consider the KIEL results to be a promising start, but there is much space for improvement. We are currently undertaking a complete redesign, with the aims of added functionality, support for a broader set of modeling languages, and better integration with other tools. This new platform integrates into the rich-client application framework of *Eclipse* [11], which leads to its name *Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform* (KIELER) [17].

The Eclipse integration is done for multiple reasons. First it allows to build upon an existing platform and leverage common already existing frameworks and therefore let this project concentrate on its own topics. Second it can try to integrate the new interaction mechanisms of graphical modeling in a more generic way. It gets integrated into the common platform so that its contributions can be used by a big, already existing and still growing community. Hence the idea of consistent employed automatic layout gets interfaced with the existing graphical modeling projects in Eclipse, the *Graphical Editing Framework* (GEF). This way every graphical editor that is built upon GEF can leverage the KIELER functionalities.

Another major difference between KIEL and KIELER lies in the approach towards providing simulation capabilities. In KIEL, simulation is either done with an integrated simulator, as done for SSMs, or by an interface to an external tool that centers on a specific modeling language, such interfaces exist for Stateflow and ArgoUML. Either method is rather labor intensive and ultimately provides only limited simulation capabilities. In KIELER, we follow an alternative approach, which seeks for a clean separation of modeling pragmatics and semantics, including simulation capabilities. The aim is to allow a smooth integration with other modeling frameworks that augment the KIELER capabilities in the realm of pragmatics with other capabilities such as simulation and code synthesis. We are currently investigating such an integration with the Ptolemy framework [7], also with the aim of tool supported multi-modeling [6].

Next to the simple layout interface a set of layout engines gets integrated. Especially for actor-oriented dataflow diagrams like SCADE or Simulink this is not trivial, because the layout cannot naturally be mapped to standard graph layout problems.

The ideas of dynamic views onto a model shall be expanded, especially to actor oriented dataflow languages. Here, again, it is not as obvious as in Statecharts how to trigger the different *focus and context* views of a model. The introduction of a common notion of simulation and semantics in Eclipse could help to build the necessary base for this task.

References

[1] Airbus Deutschland GmbH. Company homepage. http://www.airbus.com.

[2] ARINC, Annapolis, Maryland, USA. *ARINC 664, Aircraft Data Networks, Part 7 — Deterministic Networks*. http://www.arinc.com.

[3] Ralf Belschner, Robert Mores, Gary Hay, Josef Berwanger, Christian Ebner, Sven Fluhrer, Emmerich Fuchs, Bernd Hedenetz, Andreas Krüger, Peter Lohrmann, Dietmar Millinger, Martin Peller, Jens Ruh, Anton Schedl, and Michael Sprachmann. FlexRay: The communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Detroit, USA, March 2001. Society of automotive Engineers.

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[5] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *Lecture Notes in Computer Science (LNCS)*, pages 389–448. Springer-Verlag, 1984.

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

[7] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[8] Markus Buhlmann, Martin Schlager, and D Kant. Integrating mixed-criticality automotive subsystems. In *SAE World Congress*, Cobo Center, Detroit, USA, april 2006.

[9] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT '05)*, Jersey city, New Jersey, USA, September 2005.

[10] DECOS Consortium. Dependable embedded components and systems—research project homepage. https://www.decos.at/.

[11] Eclipse Software Foundation. Eclipse homepage, 2008. http://www.eclipse.org/.

[12] Esterel Technologies. Company homepage. http://www.esterel-technologies.com.

[13] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 State Machines. In *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[14] Hauke Fuhrmann, Hendrik Geilsdorf, Lothar Klein, and Stefan Schneele. System-Entwicklung basierend auf der DECOS-Architektur. In *Informationstagung Mikroelektronik ME 2006*, volume 43, pages 139–149, Vienna, Austria, October 2006. VIENNA-TEC.

[15] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. The aerospace demonstrator of DECOS. In *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems (ITSC'05)*, pages 19–24, Vienna, Austria, September 2005.

[16] Hauke Fuhrmann, Jens Koch, Jörn Rennhack, and Reinhard von Hanxleden. Model-based system design of time-triggered architectures—an avionics case study. In *25th Digital Avionics Systems Conference (DASC'06)*, Portland, OR, USA, October 2006.

[17] Hauke Fuhrmann and Reinhard von Hanxleden. The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage, 2009. http://www.informatik.uni-kiel.de/rtsys/kieler/.

[18] Richard Gronback. Graphical Modeling Framework, 2008. http://www.eclipse.org/gmf/.

[19] Manfred Gruber and DECOS Consortium. *Dependable Embedded Components and Systems—DECOS Technology and its Application—DECOS Book*. DECOS Consortium, 2006.

[20] M Gusenbauer, T Rittschober, and T Philipp. Deformation and vibration control in high precision production processes by smart material actuation and sensing. In *Fourth World Conference on Structural Control and Monitoring*, San Diego, CA, USA, June 2006.

[21] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[23] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988. http://doi.acm.org/10.1145/42411.42414.

[24] Wolfgang Herzner, Rupert Schlick, Martin Schlager, Bernhard Leiner, Bernhard Huber, Andras Balogh, Gyorgy Csertan, Alain Le Guennec, Thierry LeSergent, Neeraj Suri, and Shariful Islam. Model-based development of distributed embedded real-time systems with the DECOS tool-chain. In *Proceedings of the SAE Aerotech*, September 2007.

[25] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[26] Hermann Kopetz and G. Grünsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. Technical report, Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria, 1992.

[27] Mathworks Inc. *Simulink – Simulation and Model-Based Design*. The Mathworks, Inc., Natick, MA, 6.5r2006b edition, September 2006. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf.

[28] National Instruments. LabVIEW, visited 03/2008. http://www.ni.com/labview/.

[29] Tom Neuheuser, B. Holert, and Udo B. Carl. Elektrische Antriebssysteme für ein zentrales Landeklappenelement. In *Deutscher Luft- und Raumfahrtkongress*, volume DGLR-JT 2002-192, Stuttgart, 2002.

[30] Roman Obermaisser and P Peti. A fault hypothesis for integrated architectures. In *Proceedings of the 4th International Workshop on Intelligent Solutions in Embedded Systems (WISES'06)*, pages 47–64, Vienna, Austria, June 2006.

[31] Amir Pnueli and M. Shalev. What is in a step: On the semantics of Statecharts. In *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–264, London, UK, 1991. Springer-Verlag.

[32] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. The synchronous hypothesis and synchronous languages. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005.

[33] Steffen Prochnow and Claus Traulsen. KIEL—textual and graphical representations of statecharts. Presentation at the 12th Synchronous Workshop (SYNCHRON'05), Malta, November 2005.

[34] Steffen Prochnow and Reinhard von Hanxleden. Comfortable modeling of complex reactive systems. In *Proceedings of Design, Automation and Test in Europe (DATE'06)*, Munich, Germany, March 2006.

[35] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

[36] Trygve Reenskaug. Models – Views – Controllers. Technical report, Xerox PARC technical note, December 1979.

[37] RTCA/EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[38] Martin Schlager, Hauke Fuhrmann, Hendrik Geilsdorf, Stefan Schneele, Lothar Klein, and Patrice Toillon. *Dependable Embedded Components and Systems—DECOS Technology and its Application—DECOS Book*, chapter Aerospace Demonstrator, pages 178–191. DECOS Consortium, 2006.

[39] The Object Management Group. UML Homepage. http://www.uml.org/.

[40] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[41] Reinhard von Hanxleden. On the pragmatics of model-based design—position statement. In *Pre-Proceedings of the 15th International Monterey Workshop on Foundations of Computer Software, Future Trends and Techniques for Development*, Budapest, September 2008.