Model-Based Design for System Integration

Pieter J. Mosterman
3 Apple Hill Drive
The MathWorks, Inc.
Natick, MA 01760
pieter.mosterman@mathworks.com

Jason Ghidella
3 Apple Hill Drive
The MathWorks, Inc.
Natick, MA 01760
jason.ghidella@mathworks.com

Jon Friedman
39555 Orchard Hill Place Suite 280
The MathWorks, Inc.
Novi, MI 48375
jon.friedman@mathworks.com

Abstract

System partitioning is essential to the design of complex systems such as automobiles. Complexity can be because of compounded phenomena or because of intricate behavior. This paper focuses on the compounding effect of integrating the different subsystems that result from the partitioning. In particular, it concentrates on subsystems, or *features*, that have an embedded computation part to them. Three types of feature integration are classified: (i) shared resources, (ii) communicating features, and (iii) interacting control. The integration is addressed from a Model-Based Design perspective. It is discussed how this allows managing the complexity of integrating computations because it is operative at a higher level of abstraction than even high-level programming languages.

1 Introduction

With the advent of the microprocessor, computing power has been taking on an increasingly prominent role in the automotive industry (e.g., [10, 19]). Computing power is quickly becoming an important factor that adds value and differentiates between brands and models.

As a result, the amount of computing power in vehicles is quickly increasing. In addition, the complexity of the computations is keeping up a steady pace. Complexity has many aspects, varying from the nested control flow in computations to the integration of separate computations. Two types of complexity will be distinguished: (i) complexity because of size and (ii) complexity because of complication. Where complexity because of size requires understanding ramifications of *compounded* phenomena, complexity because of complication requires understanding semantics of *intricate* phenomena.

In order to manage the two types of complexity, different means of realizing the desired computations have been developed. At present, high-level programming languages such as C and C++ are prevailingly used in the automotive industry to capture the embedded computations. However, even using such high-level programming languages, software is notoriously difficult to design, as it is difficult if not impossible to prevent programmers from introducing defects into the code.

The error prone nature of software increases manifold with increasing complexity of both size as well as complication of the computations that it implements. The main reason for this increasing error-proness is that the language constructs of programming languages do not keep trend with the semantic notions that the design engineers are trying to capture.

Progress has been made in developing language constructs that help manage the complexity because of size of software and its reusability. In this respect, the object oriented technology [9, 15] has been a great step forward. Similarly, templates in C are a great help to being more efficient, and, therefore, going by the maxim that elegant models are the best models, making less mistakes, especially in the maintenance of software.

In spite of all these advances, programming lan-

guages have never achieved the level of domain specificity that provides the semantic notions desired and needed for a complex design task. Those semantic notions are, in turn, embodied by sophisticated modeling languages such as Simulink® [21], Stateflow® [22], CAMeL-View [3], VHDL [4], its analog extension VHDL-AMS, and Modelica™ [7]. The challenge and opportunity then lies in providing engineers with such languages to design the desired functionality and make the models that are designed using such languages computable.

Translating the high-level domain specific semantic notions into computer code then becomes the task of the language experts rather than the embedded control experts (e.g., [6]). Compiler technology such as, e.g., graph grammars [2, 20], can be efficiently applied to systematically generate the software, thus removing any coding errors that may result from human intervention. In addition, optimization techniques can be applied on intermediate representations of the computations. Since computers scale well, if such optimization algorithms are of at most order three, computers can even handle large applications of the algorithm better than humans and so produce more optimal code (e.g., [12]).

The use of such Model-Based Design is increasingly exploited in embedded control system design. Model-Based Design has shown to be of great value in fostering innovation, improving productivity, producing better quality features, reducing cost, and allowing a shorter time to market.[1,2]

However, the isolated use of Model-Based Design to design a feature does not unlock the full potential that can be reaped by enterprise-wide usage. A feature is typically only a subsystem of the entire system that makes up the product. While Model-Based Design improves the subsystem design and realization, its isolated use does not address the integration of the different subsystems. In an enterprise-wide adoption of Model-Based Design, the integration can be done at a model level as well, rather than at an implementation level, providing additional efficiencies and advantages over traditional development processes.

In this paper, the subsystem or feature integration is presented from the perspective of Model-Based Design. Section 2 discusses why partitioning is useful and what benefits it yields. In Section 3, three different types of integration are presented and the issues involved in achieving the respective forms of integration are elaborated. Section 4 provides an overview of how to address those issues. Section 5 discusses moving subsystem integration into the early design phases. In Section 6, the conclusions of this work are presented.

2 Why Partition a System

The development of a complex system such as an automobile involves the integration of many different disciplines. For example, to model the physical part of the device under control, the *plant*, requires engineers who are well-versed in physics areas such as thermal, mechanical, hydraulic, and electrical systems. In addition, control system design requires feedback control engineers to work together with reactive control engineers and diagnostic engineers. Furthermore, the system integration calls for system engineers, and the implementation of control algorithms on a microcontroller is typically done by software engineers. Moreover, the use of communication buses in automobiles necessitates the involvement of network communication engineers.

Optimizing the entire system all at once is simply too complex as the sheer number of different components would take so much coordination between the different teams of engineers that the schedule to accomplish this optimization would be far too long to be acceptable. Furthermore, all of these different domains are typically best addressed by domain specific tools. To support the need for dedicated design approaches and to manage the complexity because of the size of the overall system, partitioning, hierarchy, and modularization are applied [24]. This allows engineers with different backgrounds and experience to all contribute, reuse, and take advantage of the same models

A good partitioning minimizes the dependencies of

[1] http://www.mathworks.com/industries/aerospace/miadc/presentations/10_F35_Flight_Control_DevelopmentDaveNixon.pdf

[2] http://www.mathworks.com/company/events/programs_de/iac2004/agenda.html

the components and optimizes the controllers individually, including verification and validation[3] of these components.

The partitioning prevents the need for a comprehensive design which would be much more complicated than the cumulative design of the parts. Such a partitioning approach does have the drawback that it cannot facilitate cross-controller optimizations. The benefit of one larger and more comprehensive design is then the possibility of more optimal control, but the drawback is that the complexity increases as well.

In general, a number of reasons to partition can be listed:

- To support re-use.
- To avoid having to work with the largest assembly, which results in large overhead on development resources.
- To isolate errors and defects.
- To support multi-user development.
- To reduce complexity
- To be more efficient, as incremental technologies can be applied
  - Code generation
  - Code and model compilation
  - Verification & Validation (V&V)
  - Test vector design
  - Failure Mode, Effects, and Criticality Analysis (FMECA)

More computationally complex analysis and design methods can be applied to the smaller components, but this necessitates well-defined interfaces and context to eliminate dependencies and to ensure that the results still are valid in the integrated system.

Once a system is partitioned, the parts can be put under version control separately. Furthermore, it allows configuration management to be put in place. In this context, it supports a useful business paradigm that is established in the Electronic Design Automation (EDA) community, where suppliers provide the original equipment manufacturers (OEM) with models of their components as an 'electronic data sheet'. This facilitates quick experimentation with different types of components to identify their suitability in meeting the overall system requirements.

Note that the decomposition of the design process is truly along two dimensions as illustrated in Fig. 1. In one dimension, the decomposition is structural, partitioning the overall system in subsystems that can be further partitioned till the component level is reached. In the other dimension, the design process is functionally decomposed into the different phases that one subsystem passes through such as the continuous time feedback control, its discretized and sampled time version, and then the scheduled control functionality to be implemented on the microcontroller [14].

To dwell on the functional decomposition, consider a fuel injection controller. The specification of the controller may first start out as simply defining the interface, the I/O. Then the feedback control design engineers, working in the continuous time domain with Bode plots or root locus plots, develop the control algorithm. In this stage, concerns are the number of poles and zeros in the left-hand side of the root locus and phase and gain margins. Then, the control engineers will discretize the control laws in order to start thinking about how their controller design can be implemented on a processor, so their concerns are now about working with sample times. The systems engineers then integrate the controller together with the rest of the system, and so they need to account for correct interfaces, data integrity, filtering, data transformation, partitioning the controller, and mode switching. Finally, software engineers working on the implementation of the system on an actual target are concerned with data types, threads, scheduling, rates, and I/O drivers. So, in the functional decomposition dimension, it is desired to relate the different stages for a system so that the effort can be reused and leveraged without having to redesign or rework the part from the very beginning at each stage (e.g., [23]).

[3]Verification is determining whether the system has been designed correctly, whereas validation is determining whether the correct system has been designed.

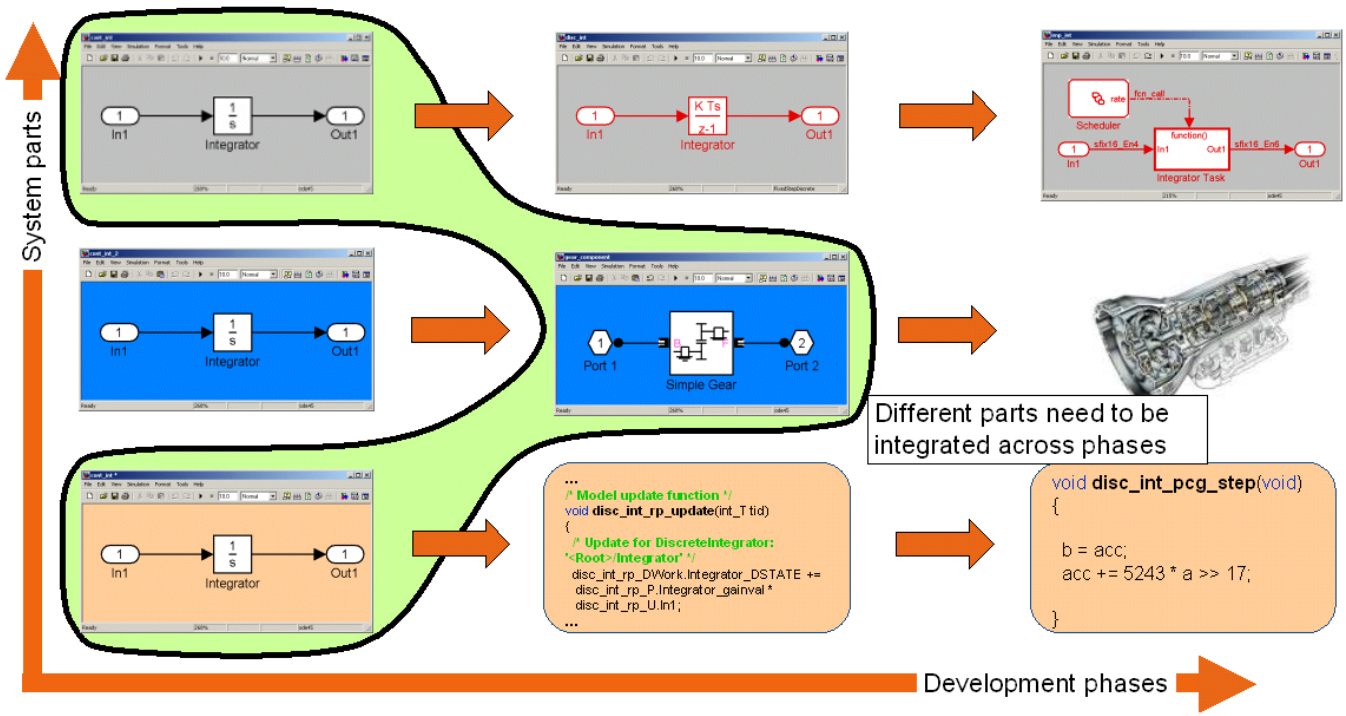

Figure 1: Stages and system decomposition in system development.

3 Issues in System Partitioning

The first issue to address in the design of an automobile is how to partition the overall effort so that each team can work on separate, well defined tasks. The next issue is naturally the integration of the separate parts.

3.1 How to Partition

In partitioning, the challenge is how to minimize the dependencies between the tasks when they are inherently related and even coupled. This is typically addressed by designing an architecture of the overall system, the parts of which are designed by dedicated teams of engineers, where the parts may be further decomposed based on an architecture themselves [11].

A consequence of this is that the behavior of the parts is context-independent, i.e., their behavior is not affected by the parts that they are connected to. Otherwise, there truly are dependencies between the connected parts and their design can only be effectively and correctly performed by considering them in conjunction with one another. This comprehensive analysis, of course, devoids the purpose of partitioning.

A subsystem can be regarded as a more or less stand-alone unit, and often it is developed by a third party supplier. The original equipment manufacturer (OEM) draws up the requirements of the interface and functionality of the subsystem. The supplier then develops the subsystem, referred to as a feature, according to these requirements.

3.2 How to Integrate

The integration of features can be such that: resources have to be shared, features have to communicate, and controllers interact through the plant.

3.2.1 Resource Sharing

The most commonly shared resource in automotive features that include embedded computations is the battery. Power management is, therefore, an important part of feature integration.

For example, consider the power window in an automobile [18]. A reactive controller issues commands to a dc motor to roll the window up and down. This dc motor draws power from the battery, and so when starting the engine, the movement of the window may be halted to provide maximum power to the ignition subsystem.

Another important shared resource is the controller area network (CAN) [1]. The power window commands may be communicated to the electronic control unit (ECU) that controls the dc motor over the CAN bus. This allows the window controls to be located elsewhere in the vehicle, such as the center console. Note that features with different requirements based on the data transmission rate and even different levels of criticality can all use the same CAN bus.

Another resource that is increasingly shared is the ECU. With the computing power that is available nowadays, it is more efficient to share this resource. So, whereas features used to exclusively own one or more dedicated ECUs, there is a move to have fewer ECUs with multiple features. For example, the Autosar project[4] aims to integrate multiple features on one ECU [10].

For the power window example, this means the window controller could be implemented on the same ECU as the door mirror controller as well as the door lock controller, even though these features may each be developed by different suppliers.

3.2.2 Feature Communication

Different features may communicate with each other directly, i.e., at a computational level. This requires their interfaces to be compatible, where compatibility is at a syntactic as well as semantic level. So, not only do function calls have to have the same signature, it is critical that the communicated values are restricted to the appropriate domain. In this context, domain is a broad notion and includes characteristics such as timing constraints, value ranges, and frequency limitations. For example, a part of a controller may only take positive values as input.

In addition, the interface has to be adhered to strictly, meaning that advanced notions such as polymorphism are disallowed. Note that this includes functionality such as sample time propagation as it can be found in Simulink. When sample times are propagated, a function may change its sample time based on the context in which it is used. This change in sample time may severely affect performance and be detrimental to the point where the functionality could become damaging to the automobile.

3.2.3 Control Interaction

Many of the features interact with one another because they operate on the same plant. For example, the spark advance controller and the air intake controller both directly affect the performance of the engine. Another example is the door lock control that may cause the windows to be closed as well, and so it operates on the same plant as the power window control.

Studying interaction of features is an essential part of the system integration. In the worst case, if such interacting control is not comprehensively analyzed and designed appropriately, undesired and potentially damaging behavior may occur.

4 Facilitating System Partitioning

Modeling the architecture of a system requires a number of aspects to be addressed so the parts that plug into it are indeed composable.

4.1 Defining Interaction

Clearly for any type of integration, the interaction between the parts needs to be defined rigorously. This interaction consists of two parts: the interface of each of the parts and the allowed behavior.

[4] http://www.autosar.org

4.1.1 Interface Definition

The interface between two parts is the most conspicuous aspect of interaction. First, the data that is provided has to match the data that is accepted. This match has to be syntactically correct in a sense that in case of a set of data that is communicated, the ordering of the elements in this set is important. In addition, the type of the individual data elements that are communicated has to match between the provider and the acceptor.

Because the interface definition is critical for system integration, and because in automotive systems there tend to be many of such interfaces [10], it is imperative that convenient means are available to define interfaces. In particular, to negotiate the size of interfaces, support for hierarchy is a necessity.

In software, such hierarchy is provided by `struct` elements. At the model level, modeling tools such as Simulink provide buses, that can be constructed hierarchically as well. Such hierarchical buses can be automatically translated into `struct` elements in the generated code, where facilities are available to enforce contiguous memory to be used. This is especially important if large amounts of data have to be transferred across the interface often.

An example of a structure generated from a hierarchical bus that can be used in a power train model of an automobile could be as follows:

```
33 typedef struct {
34    transmission transmission;
35    engine engine;
36    real_T throttle;
37    real_T brake;
38    real_T speed;
39 } CANB;
```

Here, in the model, the `CANB` bus consists of the signals `throttle`, `brake`, and `speed`, as well as two other buses, `transmission` and `engine`. These buses consist of signals, again, where the generated struct for the `engine` bus is:

```
24 typedef struct {
25    real_T w;
26    real_T EGO;
27    real_T MAP;
28 } engine;
```

This struct shows the signals that comprise the `engine` bus to be `w` for angular velocity, `EGO` for estimated generated oxygen and `MAP` for manifold air-pressure.

4.1.2 Context Definition

An issue in interaction between parts that is perhaps discussed less in literature is the need to match the behavioral aspects of the communicated data [8]. Still, it is critical to ensure the data that is provided by a part matches the assumptions based on which the accepting part was designed.

These assumptions can be static semantics such as the maximum and minimum value of communicated variables. However, dynamic semantics may be equally important. For example, the frequency components that are allowed in an input signal may be limited. Another example is the sample rate of the providing and accepting parts, which has to match. At the model level such dynamic semantic compatibility is often easier to enforce than once software has been generated. Often times, in software, the dynamics of a function are the result of a scheduler, and this scheduler is dissociated from the actual function code.

This discrepancy between the need for rigorously associating dynamic semantics with real-time software and the lack of options to do so has been noted in other work [16]. In the Ptolemy project [5], it is tried to make the definition of dynamic semantics of interface variables part of their type. It appears to be convenient to address compatibility of dynamic semantics at the model level, where this information is provided in a language that is well-suited and especially designed for it.

Using assertions at the model level, design assumptions can be enforced. For example, in Fig. 2 a model of a transmission controller is shown. This controller takes as one of its input a throttle signal and the design is based on the assumption that the throttle signal takes on values between 0 and 100. To enforce the usage of this controller in the correct context, an assertion is thrown if the throttle signal value violates these bounds.

Assertions can be enforced in design stages, but for

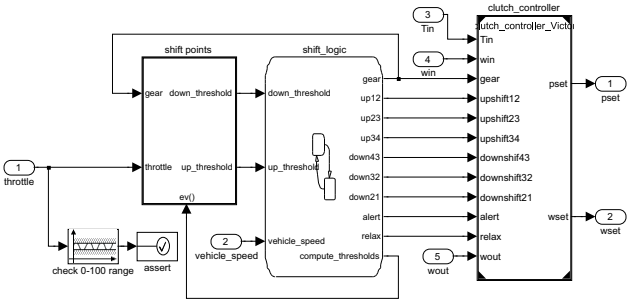

Figure 2: Transmission controller model.

the production code, they can be removed by turning them off so as to minimize memory usage. This is very similar to the use of assertions in software development, where they may be included in a DEBUG build of the code, but removed for the eventual version.

4.2 Human Communication

Since ultimately the different features are shared between teams of engineers, it is important that the deliverables support quick understanding by humans. This is achieved by standardized, complete, and comprehensive documentation as well as the use of standardized manners to design models and generate code.

4.2.1 Style Guidelines

Once teams of engineers are facilitated to work on different subsystems during different stages in the design process, it becomes of paramount importance that style guidelines are followed. The need for following style guidelines holds for coding, where linting tools can check style guidelines such as MISRA[5] compliance, but also for modeling. For example, the style guidelines defined by the worldwide MathWorks Automotive Advisory Board (MAAB)[6] lay down rules for designing Simulink and Stateflow models. An example MAAB style rule is given in Table 1.

[5] http://www.misra.org.uk/

[6] http://www.mathworks.com/applications/controldesign/MAAB_Style_Guide_html_v1_00/MAAB_v1p00.htm

Table 1: 4.5.1.2.db_0086: Basic block interface signals

ID: Title	**db_0086: Basic block interface signals**
Priority	mandatory
Scope	MAAB
Automation	possible
Prerequisites	
Description	A basic block ... • has no input busses. • has no output busses. • may have an input vector, if the block is vectorized. • may have an output vector, if the block is vectorized.
Benefit	Respecting the guideline ensures ... • accessible signals. • a clear system structure.
Penalty	Breaking the guideline ... • may cause problems with non-accessible signals. • may cause a lot of redesign work. • may cause problems or errors with custom tools.
Author	Daniel Buck
Last Change	07.02.2000, Daniel Buck

In other work [13], style guidelines for statecharts in different tools such as Stateflow can be defined and automatically checked. Modeling guidelines help everyone easily understand one another and prevent complex modeling constructs that are unreadable, error prone or could result in inefficient code being generated.

4.2.2 Documentation

Similarly, standardized formats for company reports make the reports much easier to understand and amalgamate. Again, automatic report generation ensures such formats are being applied and improves the consistency and efficiency of documentation.

5 The Role of Different Targets in the Integration

To assess the overall system performance, the separate components are integrated and the integrated

system is optimized. However, because the functionality of the individual components has been locked down, changes to the parts are difficult to make. Consequently, ultimately a sub-optimal design may result. So, when should integration of the different parts be started so that an optimal design can be achieved?

An important aspect of system design is the early identification of eventual behavior so anomalies are detected and remedied as early as possible.

Feature integration effects are by their very nature typically established in the later system design phases and so the liability in case of failure is significant. As such, potential integration issues should be investigated in the early design stages, if possible.

Traditional approaches to design verfication and validation throughout the development process include rapid prototyping, processor-in-the-loop simulation, and hardware-in-the-loop simulation [17]. To make the system integration amenable to such analyses, it has to fit within Model-Based Design. The obvious requirement for that is the delivery of features as models. This requirement facilitates the generation of code for different configurations and platforms.

For example, during the design of a system, subsystems that are in different design phases can be tested in conjunction with each other. Floating point code can be generated for integration testing in a prototyping stage where, for example, an MPC 555 may be used to test the generated code, while in production a fixed-point processor such as the Infineon C166 may be used. If the feature were only available in its final fixed-point form, the integration testing would be complicated by additional communication between the floating-point and fixed-point processors.

6 Conclusions

The application of embedded computing power in the automotive industry is increasing at a steady pace. The main benefit of embedded computations are the relatively low cost for the design flexibility that it provides.

With the flexibility comes a complexity, though, that can be difficult to negotiate. This complexity can be because of size, but also because of complicatedness. In this paper, it is concentrated on the former, and it is discussed how partitioning aids in managing the complexity because of the compounded phenomena.

With partitioning comes the inevitable integration, and three classes of integration are identified: (i) shared resources, (ii) communicating features, and (iii) interacting control. It is argued that high-level programming languages such as C and C++ do not suffice in addressing those three integration issues.

Instead, Model-Based Design is introduced as a means to support the subsystem integration at a much higher semantic level, that is closer to the domain specific notions. Some of the benefits of Model-Based Design for subsystem integration are presented. The models that are used are better suited to address intricate matters such as interface compatibiliy, not just from a syntactic, but also from a static and dynamic semantics perspective.

References

[1] CAN specification. Technical Report, 1991. Robert Bosch GmbH.

[2] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. In *Proceedings of the International Workshop on Graph Transformation and Visual Modeling Techniques*, Barcelona, Spain, Mar. 2004.

[3] CAMeL-View. *CAMeL-View Virtual Engineering Workbench Reference Guide.* iXtronics GmbH, Paderborn, Germany, 2004.

[4] E. Christen. The VHDL 1076.1 language for mixed-signal design. *EE Times*, June 1997.

[5] J. Davis, II, R. Galicia, M. Goel, C. Hylands, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy II – heterogeneous concurrent modeling and design in

java. http://ptolemy.eecs.berkeley.edu, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999. version 0.1.1.

[6] J. de Lara, H. Vangheluwe, and P. J. Mosterman. Modelling and analysis of traffic networks based on graph transformation. In E. Schnieder and G. Tarnai, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)*, pages 120–127, Braunschweig, Germany, Dec. 2004.

[7] H. Elmqvist *et al.* Modelica$^{\text{tm}}$–a unified object-oriented langauge for physical systems modeling: Language specification, Dec. 1999. version 1.3, http://www.modelica.org/.

[8] B. Falkenhainer, A. Farquhar, D. Bobrow, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, and B. Kuipers. CML: A compositional modeling language. Technical Report KSL-94-16, Knowledge Systems Laboratory, Stanford University, Stanford, CA, Jan. 1994.

[9] A. Goldberg. *Smalltalk-80: The interactive programming environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[10] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *EMSOFT'04*, pages 203–210, Pisa, Italy, Sept. 2004.

[11] D. J. Hatley and I. Pirbhai. *Strategies for Real-Time Systems Specification*. Dorset House Publishing Co., New York, New York, 1988.

[12] G. Hodge, J. Ye, and W. Stuart. Multi-target modelling for embedded software development for automotive applications. In *2004 SAE World Congress*, Detroit, MI, Mar. 2004. 2004-01-0269.

[13] M. Huhn, M. Mutz, K. Diethers, B. Florentz, and M. Daginnus. Applications of static analysis on UML models in the automotive domain. In E. Schnieder and G. Tarnai, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2004)*, pages 161–172, Braunschweig, Germany, Dec. 2004.

[14] R. A. Hyde. Fostering innovation in design and reducing implementation costs by using graphical tools for functional specification. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference*, Monterey, CA, Aug. 2002.

[15] Krasner and Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *ParcPlace Systems*, 1988.

[16] E. A. Lee. What's ahead for embedded software. *Computer*, 33(9):18–26, Sept. 2000.

[17] P. J. Mosterman, S. Prabhu, and T. Erkkinen. An industrial embedded control system design process. In *Proceedings of The Inaugural CDEN Design Conference (CDEN'04)*, Montreal, Canada, July 2004. CD-ROM: 02B6.

[18] P. J. Mosterman, J. Sztipanovits, and S. Engell. Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*, 12(2):223–234, Mar. 2004.

[19] K. D. Müller-Glaser, G. Frick, E. Sax, and M. Kühl. Multi-paradigm modeling in embedded systems design. *IEEE Transactions on Control System Technology*, 12(2):279–292, Mar. 2004.

[20] M. Pezzé. Cabernet: A customizable environment for the specification and analysis of real-time systems. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1994.

[21] Simulink. *Using Simulink*. The MathWorks, Natick, MA, 2004.

[22] Stateflow. *Stateflow User's Guide*. The MathWorks, Natick, MA, 2004.

[23] S. Vestal. Software architecture workshop, July 1994.

[24] K. Wijbrans. *Twente Hierarchical Embedded Systems Implementation by Simulation: a structured method for controller realization.* PhD dissertation, University of Twente, Enschede, The Netherlands, 1993. ISBN 90-9005933-4.