Model Integrated Computing in Robot Control to Synthesize Real-time Embedded Code

Robert Höpler and Pieter J. Mosterman

Institute of Robotics and Mechatronics
DLR Oberpfaffenhofen, P.O. Box 1116
D-82230 Wessling, Germany
[Robert.Hoepler|Pieter.J.Mosterman]@dlr.de
http://www.op.dlr.de/~[hoepler|pjm]

Abstract— **Manufacturing robots present a class of embedded systems with hard real-time constraints. On the one hand controller software has to satisfy tight timing constraints and rigorous memory requirements. Especially nonlinear dynamics and kinematics models are vital to modern model-based controllers and trajectory planning algorithms. Often this is still realized by manually coding and optimizing the software, a labor intensive and error-prone repetitive process. On the other hand shorter design-cycles and a growing number of customer-specific robots demand more flexibility not just in modeling. This paper presents a model-integrated computing approach to automated code synthesis of dynamics models that satisfies the harsh demands by including domain and problem specific constraints prescribed by the robotics application. It is shown that the use of such tailored formalisms leads to very efficient embedded software, competitive with the hand optimized alternative. At the same time it combines flexibility in model specification and usage with the potential for dynamic adaptation and reconfiguration of the model.**

I. Introduction

Industrial processes and consumer products (e.g., robots, washers, and cellular phones) increasingly rely on the use of embedded software. This allows more functionality to be implemented in software which is less expensive and introduces more flexibility in design and optimization of operation, e.g., because of functional redundancy in smart sensors and fault detection, isolation, and reconfiguration applications. The ultimate incarnation of this notion is the System-on-a-Chip paradigm that supports high integration with a number of benefits: (i) shorter time to market, (ii) less power consumption, (iii) higher reliability, and (iv) lower cost [1].

Because embedded code is restricted to fit limited resources and has to interface the physical world through sensors and actuators, computer science software design concepts (e.g., [2], [3]) do not apply [4]. In addition, whereas the sense of time is thoroughly banished from the computer science domain, it is one of the most important factors in embedded reactive systems given the harsh timing constraints that mandate functionally correct, safe, and optimized code. All this has led to the common practice of domain engineers being prevalently responsible for code design. These engineers are trained in their specific application domain (e.g. mechanics, electronics) and because not coding experts, a systems view on software as provided by computer aided software engineering (CASE) tools based on structured analysis methods [5], [6], [7] is invaluable. This represents a paradigm shift from software engineering to systems engineering for which software architectures and code synthesis tools are key enablers.

Embedded code will increasingly consist of interacting software components [8], which requires generating the most applicable domain specific software architectures. This supports a model-based approach to program synthesis, so-called *model integrated computing* (MIC) which introduces a number of advantages over traditional manual coding [9]:

- Because the physical environment and the synthesized software, i.e., the information processing component, are both described by models, the conceptual gap between these two vanishes and a more comprehensive view on the generated application becomes possible. This enables one to deduce and manipulate run-time properties at the model level, such as timing and system requirements, and reconfiguration because of a changing environment.
- The use of models can be exploited by generating different implementations of the same model to support various analyses and computations. It also facilitates design and evolution of the generated applications. The introduced additional level of abstraction is amenable to formal analyses. This is exploited, e.g., in analyses of synchronous languages [10], [11] to ensure verifiably correct code, notably in the aircraft industry.
- Coding is a labor intensive process and because error-prone it requires many 'debugging' iterations. The use of automatic synthesis minimizes this stage of manual interaction, and, therefore, results in much shorter lead times on new products.
- The chance of introducing errors in software when making a change is proportional to the size of the code, not the size of the change [12]. This prohibits quick system adaptation to new demands and requirements. Using automatic code synthesis, model integrated computing supports changes at the model level, which is much less susceptible to errors.

In order to reap these benefits, efficient program synthesis is paramount. In general, manual coding will have a marked edge in terms of efficiency over automatically generated code. To mitigate this drawback, domain specific

constraints have to be included as much as possible in the used modeling paradigm and, even more important, in the involved model interpretation and synthesis steps.

This paper discusses a model integrated approach to robot modeling for control systems. The software is synthesized from a high level model specification using an ontology of primitive and aggregate components in the robotic domain. In this domain, the executable model is needed in different forms for different tasks such as loop control, path planning, calibration, fault detection, and reconfiguration. This is facilitated by a carefully selected model and software structure that results in code with (i) fast, (ii) modular, and (iii) reconfigurable characteristics. Section II gives an overview of the manufacturing robot domain and its specific constraints. Section III presents a robot modeling paradigm for real-time embedded models of the mechanical part of an extensive class of robots. The realization of a C++ library based upon the paradigm and results are discussed in section IV.

II. The Context of the Robotic Domain

Nowadays robots are used extensively in industrial and hazardous environments to perform repetitive, dangerous, unpleasant, and heavy works, and are more and more entering everyday life as toys and service robots. Physically, they consist of mechanical, electrical, and sometimes hydraulic components (e.g., links, drive trains, induction motors, hydraulic actuators, encoders), often controlled by embedded software running on a central processing unit, or a dedicated digital signal processor.

A crucial part in the control of a robot manipulator is the model of its mechanical structure, providing the kinetic and dynamical state of the robot, which is vital to control functionality such as trajectory generation and loop control. This paper concentrates on implementing the model as embedded code.[1]

A. Requirements for Robot Models

The design of manufacturing robots has to account for a number of domain specific characteristics: The mass/power ratio is one of the most important factors in robot design. To increase precision and performance, manipulators are usually overdesigned in terms of structural properties so as to enable operation and analyses as stiff objects, causing them to be overly heavy. To reduce power consumption, it is desirable to have minimal robot weight, however, this requires light-weight materials and actuators which in turn affects performance. Optimal controllers can achieve similar performance while reducing weight, e.g. by compensating for elasticity in lighter drive-trains [13].

From an operational perspective, the sample rate at which the controller operates strongly affects its performance. The control bandwidth is not just constrained by data acquisition resources, but also by the control output computation time. For economic reasons low cost digital processors are preferred, but this increases computation time and the sample rate puts strict demands on resources available for control algorithm implementation. This problem is exacerbated by the desire to use model-based approaches, e.g., computed torque and path planning, because of their superior performance characteristics but required computing resources. All this pleads for models that can be evaluated at a high sample rate and that can be stored in little memory.

From a design and implementation perspective, there is a frequently recurring need for model changes. These are invoked by evolution of the robot manufacturer's product line, both in a rigorous sense when new robot designs are introduced but also when smaller changes to a particular type are made in case of accessories and options. The modeling formalism must reflect this demand for modularity. Furthermore, the trend towards more flexible production units requires on-line reconfiguration of models in case of e.g. multi-purpose tooling or cooperating robots.

B. Domain Specific Constraints and Model Aspects

This work focusses on the mechanical part of a robot. This implies the mechanical domain is dynamically decoupled from the electrical and possible other physical domains, a valid assumption given the design of sensors and actuators. The first decouple power domains by transfer into low power signals and the second have well determined causal input-output behavior.

Treating the mechanical parts as a rigid or sometimes elastic multi-body system (MBS) is a sufficient approximation for control purposes of most robotic manipulator. Investigating the common MBS-model topologies of manufacturing robots reveals that one can limit oneself to a fixed base, and primarily tree-structured topology possibly including some kinematic loops. These models consist of a finite set of primitive components such as joints, links, springs, and drives. Much of the strength of a formalism stems from restricting its functionality to its application domain. In our case, complex model computations such as system simulation, task analysis or digital mock-up can be ruled out to arrive at the dedicated formalism. In particular, a robot model is a small MBS that has to compute non-linear kinematics and dynamics used in

- computed torque for drive control,
- path planning, and to some extent, in
- calibration and parameter identification.

These restrictions within the mechanical domain allow the use of dedicated algorithms, such as recursive Newton-Euler and related algorithms [14], [15] to operate on the model.

The restriction to MBS also benefits hierarchical decomposition of models. From a domain engineer's point of view this has several advantages: The modularity concept helps reducing model complexity and offers a familiar system view as being composed of parts and components. Coding and testing efforts can be minimized by a model database with reusable parts and submodels. From a code synthesis perspective, the component view provides the potential to generate code that easily adapts to changing environments,

[1] This definition is used consistently in the proceedings to refer to the specific domain under investigation.

even during run-time [16]. Finally, as the synthesized program has to be embedded in different real-time environments the possible programming languages used for code generation are *de facto* limited to Assembler, C and C++.

C. Existing Formalisms

Over the last decades, many approaches have evolved that deal with the complex process of modeling MBS and the numerical solution of the governing equations, especially in the field of robotics [17], [18]. State-of-the-art in modeling complex systems applies object-oriented notions such as inheritance and hierarchy. This is also included in domain-specific formalisms for MBS as well as more general formalisms for physical modeling [19], [20]. Those formalisms often use graphical or textual model descriptions to symbolically derive the differential and algebraic equations of motion of the parts or to immediately generate executable applications. The physical properties are obtained from object-oriented databases, including, e.g., computer-aided-design data. In case of equations these are transformed to the desired input-output form and are finally converted to source code (e.g. C, Fortran). Utilizing order(n)-formalisms is also a powerful technique for numerical solution, as it can handle many possible mechanisms and creates very efficient code [19], [15].

However, this approach is not completely satisfying for embedded code in robot control. When fast reconfiguration of the model is needed, e.g., when a change of robot tooling occurs, the compile-to-code step may take too long in a real-time system while the combinatorial explosion tends to prohibit exhaustive analysis of all configurations and storing these in memory. In fact, even for few configuration changes, as memory resources are limited in embedded systems, it is not practical to store models of all possible configurations. This is aggravated by the fact that the generated executable models are usually available in one fixed causal, input-output, form. If different exogeneous variables correspond to model input and output for different tasks, a variety of models has to be run in parallel for the different causal assignments.

To meet the needs for a multi-purpose and easily reconfigurable model the model-description can be directly interpreted in terms of data flow and computations. Unfortunately, this approach lacks performance commensurate with the real-time requirements as has been shown by some MATLAB implementations [21].

An object-oriented programming language such as C++ can form the basis of a solution that meets most requirements. It can be used directly as embedded code and a carefully chosen design allows computational performance that satisfies the real-time constraints demanded by the robotics application [22]. Detailed knowledge about the domain enables establishing a highly restricted yet sufficiently powerful paradigm, needed to meet all requirements.

III. Paradigm for a C++ Embedded Robot Model

Model component interaction is based on the port concept, a versatile technique that is sufficiently powerful to handle the MBS domain. A component is called a *transmission object*, a term coined in [23] to express the distinct properties of MBS parts, when viewed as abstract kinematical and dynamical transformations. A transmission may aggregate other transmission objects and contain several *connector* objects, the ports.

The robot is restricted to be constructed from a finite set of primitive domain components that is sufficient to cover a large class of robots

- joints (revolute, prismatic, etc.),
- rigid links and bodies,
- springs, dampers, and external forces,
- drive-trains, and
- custom-specific or more abstract parts, e.g., a frame parametrized by Denavit-Hartenberg convention [24].

Relations between transmission objects are established by a *connection* between two or more ports. The MBS interpretation of a connection is a modeled physical interaction between two or more components, e.g., the direct physical contact of two bodies in one certain coordinate frame. This interconnection of several components is illustrated in Fig. 1 for a generic robotic structure.

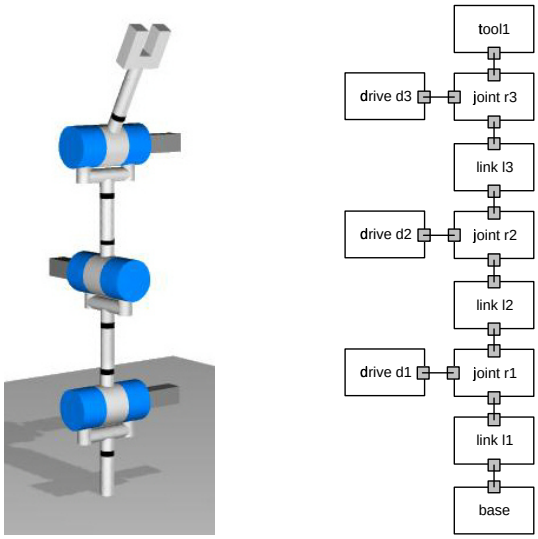

Fig. 1. *Left:* A generic 3 degree of freedom robot. *Right:* One possible corresponding graphical MBS model representation. The boxes depict transmission objects (joints, etc.), the grey squares the ports (narrow black rings), lines symbolize a (mechanical) connection.

To alleviate the modeling burden, higher level components, so-called *assemblies*, contain a hierarchy of aggregate connected primitive objects. In turn, this composition embodies the properties of a transmission. Along with the primitives, these assemblies are made available to the modeler as part of the domain library. In general, formalism migration, i.e., the continuous change in modeling primitives, is an important factor in the model integrated computing approach. However, in the robotics domain the set of primitives is not subject to much change. Therefore this issue is not discussed any further.

A. Static Semantics

One of the goals of including domain specific constraints is to prevent the definition or modification of models from being not physical or not solvable. To this end, complementary connectors ('A' and 'B') are introduced that only permit connections between certain connectors of A and B type. Figure 2 shows the relations between all entities in the domain model in a UML-like class diagram [25]. Every transmission object is composed of a certain number of connector ports, where the A-typed are restricted to a number of two. This constrains the model graph to a directed tree-structure with restricted types of loops, as required by the Newton-Euler algorithm [14]. Note that there are (currently) two different types of connectors, *frame* and *state*. The first represents the contact of two or more components in one 3-dimensional coordinate frame, the latter a one-dimensional variant, e.g., for modeling drive-trains. Every transmission contains variables and parameters describing some of its distinct physical properties, e.g., the angle or the mass of a revolute joint.

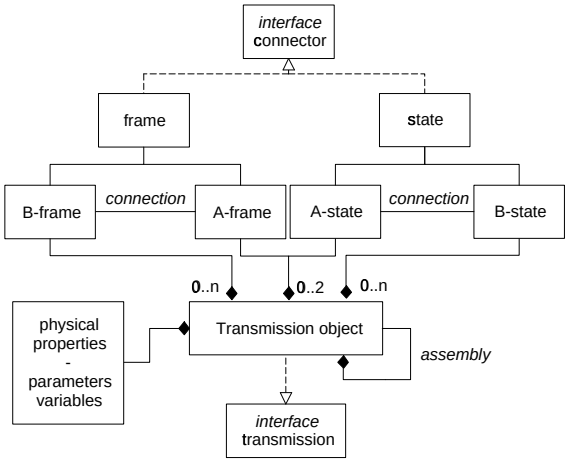

Fig. 2. UML-like meta-model specification of the robot domain model formalism for an arbitrary transmission object.

Note that Fig. 2 includes objects that are not part of the robot as such, e.g., an inertial system that relates the robot movements to a certain coordinate frame.

B. Dynamic Semantics

Concepts from [23] were adopted to represent the physical components of the robot as primitives or assemblies in the modeling formalism in order to map them onto a software-architecture of well-defined C++ interfaces and class-hierarchies. To be more explicit, the following code fragment gives the definition of the generic robot in Fig. 1, built of three rigid links connected by three revolute joints each driven by some arbitrary drive. The '. . .' arguments furnish physical properties and the operator '<<' has been overloaded as a connection operator.

```
main() {
/* component definition */
  InertialSystem base(0,0,9.81);
  RevoluteJoint r1(...);
  DriveTrain d1(...);
  Link l1(...);
  RevoluteJoint r2(...);
  DriveTrain d2(...);
  Link l2(...);
  RevoluteJoint r3(...);
  DriveTrain d3(...);
  Link l3(...);
  ToolType1 tool1(...);
  ToolType2 tool2(...);
/* connections */
  base << r1 << l1 << r2 << l2 << r3 << l3 << tool1;
  r1.axis << d1.flange;
  r2.axis << d2.flange;
  r3.axis << d3.flange;
  ... // etc.
```

In order to synthesize a complete application from the above model several steps are necessary. An additional object is introduced, the *Manipulator*, that

- creates a set of run-time primitive objects and connects them according to the model description given either by code or by a textual representation,
- interprets the description to prohibit non-physical configurations in addition to the input-output semantics shown in Fig. 2,
- detects special configurations such as kinematic loops and singularities,
- forms a graph describing the connections between the run-time primitives as required for the parametrization of the domain specific solvers. These manage computations and data-flow in the graph of executable instances of the model primitives in order to compute the desired quantities.

This Manipulator is the domain specific code generator that maps the model description to an executable application as sketched in Fig. 3. A corresponding C++ code fragment to create an instance of a manager object and to perform the steps explained above might look like:

```
Manipulator ExampleRobot(base); ExampleRobot.init();
```

The necessary flattening of the model hierarchy down to the level of executable primitives is performed by the compiler or the internal model interpreter which operates on the textual description. It makes use of the model database and, e.g., decompresses the `DriveTrain` objects in motor, gear and shaft objects.

C. Execution Semantics

Execution is facilitated by a small set of routines declared in the transmission interface that have to be implemented in each transmission object primitive, which comprise the fundamental domain specific kinematic and dynamic computations (i.e., most prominently `do_position`, `do_velocity`, `do_acceleration`, `do_force`). These primitives capture the domain knowledge of a mechanical engineer and implement highly optimized code to meet the real-time demands and to significantly reduce the testing effort. This is similar to the abstract functions in Ptolemy [26] that have to be implemented in a user-defined way to facilitate simulation, but they are much more domain specific in this implementation, leading to more efficient code.

In summary, a directed graph of computational nodes (executable instances of transmission objects) and data-flow edges, together with the solver objects prepared by the manipulator, which in fact are mere data-flow managers, forms the run-time computational model (see Fig. 3). The manipulator keeps all structural information about the model. It is therefore able to perform model reconfiguration during run-time, a domain requirement, and to save and recall the current status.

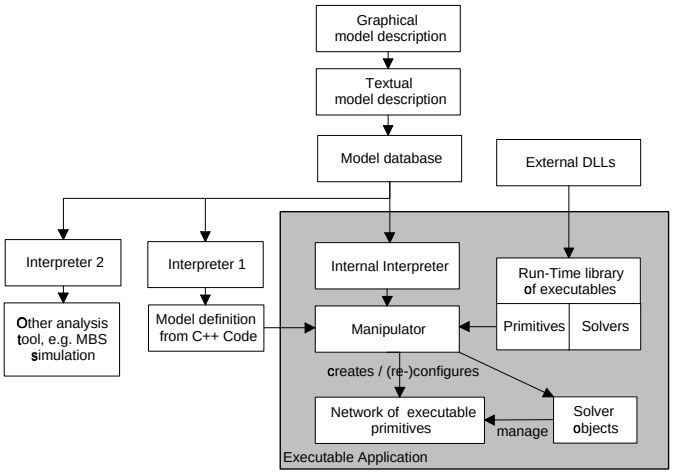

Fig. 3. Stages of the domain specific code synthesis process.

In other work [27], the Multi-Graph Architecture (MGA) accomplishes similar tasks, but chooses a different point of view. Three different levels are involved in the domain specific code synthesis process. An application level that represents a network of executable components to provide fast adaption of applications. A program synthesis level that comprises tools for building, storing and testing domain specific models and translating them to applications, in order to facilitate evolution and re-synthesis of models. A meta level provides the definition of modeling paradigms for different domains. This definition is used to automatically generate the tools utilized on the program synthesis level from formal specifications. In contrast the approach presented in this paper starts from the model definition to include as many MBS specific constraints as possible, and integrates several parts of the code generator in one object, the Manipulator. This results in a small and handy library, that is inherently applicable on all platforms providing a C++-compiler and that can be integrated easily in existing applications.

IV. Application to a C++ Library

A C++ class library was built according to the paradigm presented in Section III to investigate whether including domain-specific constraints in the modeling formalism allows automatic program synthesis of embedded software that meets real-time control requirements.

Comparison with the standard solutions for robot-models in terms of computational cost is presented in Fig. 4. A typical benchmark is a single evaluation of rigid-body inverse dynamics of a common 6 degree of freedom manufacturing robot, comprising 6 revolute joints, 6 rigid links including dense inertia tensors, and 6 simple drive-train objects [22]. To produce the results in Fig. 4 a 200Mhz Pentium and the Microsoft Visual C++-Compiler (Version 6.0) were used.

Used approach	Time [μs]
a) Symbolically generated code (C)	12
b) Manually coded C	23
c) C++-ROBOOP library [22]	1200
d) Presented approach (C++)	36

Fig. 4. Comparison of the computational cost needed for one evaluation of the inverse dynamics of a rigid MBS model (6 degree of freedom manufacturing robot).

The comparison of computational costs in Fig. 4 shows some remarkable differences:

a) The symbolically generated code represents a lower boundary of the run-time for two reasons. On the one hand the equations are optimized algebraically and therefore no redundant intermediate variables are computed. On the other hand the equations are reduced to one certain input-output form of this model, which in general has superior performance compared to a multi-purpose computational approach as the one presented.

b) The hand coded C-program tailored to this model relies on nested loops and circumvents the need for function calls. This results in 50% less computing time compared to the automatically synthesized code. This is to be expected compared to the dispatches needed in the automated approach described in this paper: The solver object has to access the code of the run-time objects through an interface.

c) The other object-oriented library can fulfil the demands of modern day controllers, but still lacks performance indispensable for complex tasks such as path planning when many evaluations are required. One reason for this is because not all domain specific constraints had been included in the modeling paradigm. This results in insufficient flexibility of the model description and therefore less potential to use optimized code.

d) In the presented approach optimized code implements the transmission interface of the different components to gain maximum performance. The generated graph of computational nodes was handled by a `InverseDynamics` solver object to compute the joint torques. The obtained performance shows that the MIC formulation can indeed fulfil the real-time demands of modern controllers.

An additional benefit of the approach is that it allows run-time reconfiguration as can be illustrated by the robot in Fig. 1. The manager of the computational model, the Manipulator, holds all information about the run-time computational model. Run-time reconfiguration of the model (i.e., during execution time), e.g., to change the robot's tool, can be achieved by the following code fragment

```
l3 | tool1;  // cut off tool1
l3 << tool2; // connect tool2
ExampleRobot.init();
```

This describes that `tool1` is disconnected and a new connection between `tool2` and the robot has been established. Also, the manipulator has checked, whether the changes made resulted in a physically correct model. Fast dynamic reconfiguration of a model can be exploited not only to adapt model structure to a changing physical environment but to adjust model complexity to specific applications or computational resources. As in our domain no system simulation has to be performed, state information residing in the objects is not relevant, and a mapping of information while reconfiguration is facilitated, indicating this procedure is practicable in a real-time system.

V. Conclusions

Embedded code is increasingly used in industrial processes and consumer products. Software design, however, is a notoriously difficult, time-consuming, and error-prone process. Moreover, modern software design approaches typically do not apply to the design of embedded code because of its specific requirements such as to fit into limited resources.

The model-integrated computing approach avoids these difficulties by automatically synthesizing programs from their models. This bridges the conceptual gap between information processing parts and the physical components of an embedded system. It also allows high level analysis to operate on the model to verify its functional correctness. Note that timing constraints are a critical requirement to ensure correctness, safety, and optimal implementation.

In this paper, model-integrated computing (MIC) concepts are applied to the manufacturing robot domain. The harsh real-time constraints in robot control demand highly optimized code. To still automatically generate this from a model, many domain specific constraints are incorporated in the synthesis process. This results in code that is sufficiently efficient to support model-based implementations of such various tasks as loop control, path planning, and calibration, while facilitating run-time reconfiguration of the robot. This can be exploited, e.g., when exchanging tools, but also is conceivable when on-line adapting the level of detail of the robot model to the desired precision of the real-world task to perform.

An additional advantage of the model-integrated computing approach is its support for smoothly migrating embedded code between robot models and types without the need for manually modifying and tuning existing software.

Acknowledgments

Robert Höpler is supported by Amatec Robotics and Pieter J. Mosterman is supported by a grant from the DFG Schwerpunktprogramm KONDISK. R.H. gratefully acknowledges many helpful discussions with Martin Otter.

References

[1] Kenneth H. Peters. Migrating to single-chip systems. *Embedded Systems Programming*, 12(4):30–45, April 1999.

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

[3] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[4] Edward A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, September 2000.

[5] Derek J. Hatley and Imtiaz Pirbhai. *Strategies for Real-Time Systems Specification*. Dorset House Publishing Co., New York, New York, 1988.

[6] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[7] K.C.J. 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.

[8] Janos Sztipanovits, Gabor Karsai, and Ted Bapty. Self-adaptive software for signal processing. *Communications of the ACM*, 41(5):66–73, 1998.

[9] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *IEEE Computer*, 4:110–112, 1997.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] Nicolas Halbwachs and Pascal Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *Asian Computing Science Conference (ASIAN'99)*, Phuket, Thailand, December 1999. LNCS 1742, Springer Verlag.

[12] Gregory G. Nordstrom. *Metamodeling – Rapid Design and Evoluion of Domain-Specific Modeling Environments*. PhD dissertation, Vanderbilt University, Electrical Engineering, May 1999.

[13] Alin Albu-Schaeffer and Gerd Hirzinger. State feedback controller for flexible joint robots: A globally stable approach implemented on DLRs light-weight robots. In *Proceedings of the IROS 2000, Takamatsu, Japan*, 2000.

[14] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul. On-line computational scheme for mechanical manipulators. *ASME J. of Dynamic Systems Meas. and Control*, 102:69–76, 1980.

[15] M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME J. of Dynamic Systems Meas. and Control*, 104:205–211, 1982.

[16] Akos Ledeczi, Gabor Karsai, and Ted Bapty. Synthesis of self-adaptive software. In *Proceedings of the IEEE Aerospace 2000 Conference, Big Sky, USA*, 2000.

[17] Robert E. Roberson and Richard Schwertassek. *Dynamics of multibody systems*. Springer-Verlag, 1988.

[18] Roy Featherstone. *Robot dynamics algorithms*. Kluwer Academic Publishers, Boston/Dordrecht/Lancaster, 1987.

[19] Intec GmbH, Germany. Simpack - analysis and design of general mechanical systems. http://www.simpack.de.

[20] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial, Version 1.4*. Modelica Association, 2000.

[21] Peter I. Corke. A robotics toolbox for Matlab. *IEEE Robotics and Automation magazine*, 3(1):24–32, 1996.

[22] Richard Gourdeau. Object-oriented programming for robotic manipulator simulation. *IEEE Robotics and Automation Magazine*, 9:21–29, 1997.

[23] A. Kecskeméthy. *Objekt-orientierte Modellierung der Dynamik von Mehrkörpersystemen mit Hilfe von Übertragungselementen*. PhD thesis, Universität Duisburg, 1993.

[24] J. Denavit and R.S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of applied mechanics*, pages 215–221, June 1955.

[25] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley Publishing Company, 1998.

[26] John Davis, II, Ron Galicia, Mudit Goel, Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. Ptolemy II – heterogeneous concurrent modeling and design in java. http://ptolemy.eecs.berkeley.edu, Department of Electrical Engineering and Computer Sciences, University of Califoria at Berkeley, 1999. version 0.1.1.

[27] J. Sztipanovits, G. Karsai, C. Biegl, T. Bapty, A. Ledeczi, and A. Misra. MULTIGRAPH: An architecture for model-integrated computing. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 361–368, Ft. Lauderdale, Florida, November 1995.