Computer Automated Multi-Paradigm Modeling in Control System Design

Pieter J. Mosterman[1]
Institute of Robotics and Mechatronics
DLR Oberpfaffenhofen, D-82230 Wessling, Germany
Pieter.J.Mosterman@dlr.de, http://www.op.dlr.de/~pjm

Hans Vangheluwe
School of Computer Science
McGill University, Montreal, Canada H3A 2A7
hv@cs.mcgill.ca, http://www.cs.mcgill.ca/~hv

Abstract

The complete control system design effort involves many stages during which partial design tasks are completed. Each of these tasks requires different modeling paradigms and different tools. Furthermore, the designed embedded control system entails a wide variety of implementation technologies that all require different specification formalisms. To handle such a multitude of modeling paradigms and different support tools: (i) a unifying generic standard language can be applied, and (ii) the required modeling paradigms can be modeled by a meta model using a shared meta language. An overview of the required parts and structure of a modeling environment and of the two approaches is given. The advantages with respect to multi-paradigm modeling are discussed.

1 Introduction

The analysis and design of engineered systems involves expertise from many disciplines and entails a variety of implementation technologies (e.g., embedded software, microelectromechanical systems, analog circuits, and digital circuits) and the *heterogeneous* nature of these systems invariably combines with an architecture of different concurrent components that interact through continuous signals or discrete message passing. The corresponding complexity has led to the use of more formal approaches to system design through realization that apply dedicated modeling formalisms to different aspects and/or components of the system. Consequently, the complete system specification process combines several modeling, design, implementation, and realization paradigms such as differential equation modeling, continuous time signal processing, and discrete event controllers. Decomposition of the entire specification task allows teams of experts to concurrently work on their domain of expertise, e.g., control law design, simulation, optimization, modeling, and verification.

To comprehensively handle control system design in such a heterogeneous environment, multiple approaches based on different paradigms have to be combined. In this paper, the following definition is used [23] "A *modeling paradigm* is a set of requirements that governs how **any** system in a particular domain is to be modeled. These modeling requirements specify the types of entities and relationships that can be modeled; how best to model them; entity and/or relationship attributes; the number and types of views or aspects necessary to logically and efficiently partition the design space; how semantic information is to be captured by, and later extracted from, the models; any analysis requirements; and, in case of executable models, run-time requirements."

A tool that 'understands' each of the corresponding formalisms (i.e., has a model of them) can be used to ensure consistency between different formalisms, allow for quick adaptation to changing needs, exchange information, and efficiently provide tailored modeling environments that are maximally constrained with respect to the domain of operation. For example, a designed control law that is automatically translated into its implementation, i.e., the hardware binding. Here the control language focuses on stability and other control characteristics, whereas the implementation has to deal with issues such as schedulability, reliability, and security, which requires different analysis formalisms. If consistency and cross coupling across these languages is ensured, implementation choices (e.g., the 'time for space' trade-off) can be conveniently conveyed back to the control design engineer.

Multi-paradigm modeling is also critical for reconfigurable systems as the supervising mechanisms that combine with a flexible control architecture are based on different modeling formalisms (even different plant models) and need to integrate with the control architecture. One solution is model integrated computing [25], which allows changes in the system model/specification and translates these automatically into software (or even reconfigures hardware).

Control system design is achieved by using many soft-

[1] Pieter J. Mosterman is supported by a grant from the DFG Schwerpunktprogramm KONDISK.

ware tools, sophisticated development techniques and methodologies relying on library components and automatic (code) generation approaches. Specialized computer automated tools for each of these domains are very helpful or even indispensable to carry out the related tasks as the process of control design requires the integration of, e.g., modeling, simulation, control law design, dynamic control law integration with safety and redundancy management control logic (e.g., surveillance functionality), and control robustness assessment. Typically, there is no one tool that addresses all these issues, and, therefore, a suite of tools is used throughout the design process. Because these tools hardly ever are compatible, the sharing and coordinating of information flow between project teams inevitably leads to a lot of overhead in terms of collaboration and is very error prone, inefficient, and expensive. Moreover, similar tasks may be carried out multiple times and even simultaneously.

All these issues are addressed by adopting a meta modeling approach to data exchange [4, 8, 9, 11] and modeling paradigm and environment specification [7, 23]. This paper intends to provide an overview of the use of meta modeling concepts as used in control system design. Section 2 reviews the specification requirements for multi-paradigm modeling environments. Section 3 discusses the use of a generic unifying language. Section 4 gives an overview of an alternative approach, the use of meta modeling, and shows how this supports the required flexibility and specificity needed for multi-paradigm modeling. Section 5 presents the conclusions.

2 Modeling Environment Requirements

To facilitate computer automated control system design, modeling, and analysis, computer based environments need to be available that are tailored to the particular task at hand. The most efficient and flexible approach is to model the modeling environment and automatically generate a complete specification that can be directly compiled into an integrated development environment. This requires the specifications to be well structured, to define the precise syntax and semantics of a language, and to not be mixed with language implementation details.

A clean separation in concepts leads to [2, 23]: *Syntactic specifications* that can be divided into (i) the concrete syntax, which captures the actual representation, e.g., a textual language specified by Backus-Naur Form (BNF) constructs, and (ii) the abstract syntax, the language syntax devoid of implementation details, which allows for the representation of the essential constituents of a formalism. *Semantic specifications* that may include model composition constraints to capture domain specific concepts and constraints. These can be classified as (i) static semantics that can be checked during model composition (e.g., in logical circuits the number of loading components allowed to connect to one output), and (ii) dynamic semantics that can only be checked during execution (e.g., whether a certain state is reached). The distinction between static and dynamic semantics is not related to representation, but rather to the availability of sufficient information to assert the validity of certain constraints before model execution. If this is not the case, the constraints need to be passed on to a model execution environment. Such constraints can be represented in the meta model structure or by a constraint language (e.g., first order predicate logic). *Presentation specifications* that are critical for specification of the complete modeling environment and that specify the appearance of entities, relationships, and attributes. *Interpreter specifications* that are necessary to extract information from each of the models to allow, e.g., documentation and execution.

The first two specify the modeling language and the latter two complete the specification of a modeling environment. In a graphical language, the syntax is a collection of modeling object types, possible relations between them, and their allowed attributes. Static semantics pertain to the well-formedness of language constructs, they represent an invariant that must hold across the family of models that can be designed using the modeling language. Dynamic semantics relate to the interpretation of the model constructs and cannot be specified by language constructs.

3 Multi-Paradigm Modeling With a Generic Standard

One approach to deal with the issues of tool interoperability and multi formalism approaches is to develop a unifying generic standard, e.g., Modelica [10] and VHDL-AMS [12]. If such a standard allows for the use of multiple formalisms in one environment it corresponds to a unifying super-formalism that can be used for model exchange and mitigate the interoperability problems.

Because of the variety of formalisms that address different aspects and types of specification that are used throughout a control system design lifecycle, if possible it is difficult at best to establish one such a generic formalism. For example, it would have to include the rather different syntax, semantics, representation and interpretation specifications of data flow diagrams, control flow diagrams, formalisms such as statecharts, Grafcet, and Petri nets, physical modeling formalisms such as bond graphs and object diagrams, block dia-

grams, and process diagrams [5, 13, 14, 16, 21]. For example, in terms of their interpretation, computational models such as differential equations, state/event, discrete event, synchronous/reactive, and (a)synchronous message passing [6] are used. To capture all of these would require one underlying computational model that subsumes all others. However, modeling is not just a question of whether it is possible with a certain formalism, it critically depends on whether it can be done elegantly and intuitively [17, 18].

A standard unifying formalism works well if the area of application is sufficiently restricted [19, 20]. For example, Modelica concentrates on physical system modeling and builds on the combined differential equation and state/event computational models. This allows for a comprehensive language well suited to its purpose. However, because it does not separate abstract from concrete syntax, it may result in an overly rigid formulation. For example, the abstract syntax of an iterator construct contains an initial value, final value, and step size. A design decision is required whether to use the concrete syntax that corresponds to `i=0: 10: 2` or to use `i=0: 2: 10`. This choice will be incompatible with particular domains and cause an increased threshold to acceptance of the standard. Note that it is impossible to allow both variants, a common solution in case of such design decisions, which leads to a bloated language specification.

The use of standards relies heavily on the concept of libraries, i.e., sets of predefined components typically related to a specific domain. Each library embodies a particular modeling paradigm. This implies that any one particular control design tool is required to contain a compiler for each of the included formalisms even those not applicable to the particular task at hand. Also, in Modelica, the presentation semantics are dissociated from the language syntax and semantics. Therefore, the choice between appearance of, e.g., an electrical resistor is possible by constructing two separate libraries. Because of the inheritance construct, each of these components can inherit the same functionality only specializing the graphical appearance. However, if one imports an electrical circuit designed with a particular library, the graphical presentation is fixed, i.e., no automatic transformation to the desired presentation occurs because no knowledge is available of what a resistor is.

The use of a standard works well for modeling affinitive domains. In case of Modelica, this is the structure of a physical system. Behavioral models, such as block diagrams and statecharts, require a graphical notation and semantics that may differ significantly, and, therefore, may be hard to capture in the standard. For example, in physical systems, models based on energy flow have no computational causality, and, therefore, the representation does not concern direction of connections. In block diagrams, on the other hand, causality of input and output signals is inherent. Typically, this is indicated by adorning the relation with an arrowhead. The semantics of this cannot be easily added to a non causal relation. For example, in the Modelica block diagram library, the relations are still non causal, and the input-output behavior is specified by the connected objects. This specification is not related to the graphical representation, though. So, for each port instance it is specified separately, whether it operates on input, output, or both. The graphical representation then is drawn as an arrowhead without this having a direct implication on the semantics.

In conclusion, the flexibility required for a standard calls for increasingly generic constructs such as undirected relations. Furthermore, given that the language needs to be sufficiently powerful to specify a multitude of formalisms, its genericity makes it hard to use it for specific analyses, i.e., it becomes hard to proof certain characteristics of a model. Also, these languages typically lack a constraint language to limit the family of models one has to deal with. the requirement that an electrical circuit includes at least one ground node cannot be specified. This allows for an entire class of electrical circuits (infinitely many) that can be modeled but that cannot be executed. Finally, the rigidity of such a generic standard makes it hard to keep up with state of the art and disallows users to define additional model specifications that they need themselves.

4 Meta Modeling

A proven method to achieve the required flexibility for a modeling language that supports many formalisms and modeling paradigms is to model the language itself. This is exemplified by, a.o., the *domain modeling environment* (DOME) [1, 7] and the *multigraph architecture* (MGA) [25]. To illustrate this notion, consider the state transition diagram in Fig. 1. When in the ON *state*, a *transition* to OFF occurs when the *condition* $t > 2$ is true and this generates an alarm *action*. The state, transition, condition, and action elements are part of any state transition diagram and their dependencies can be modeled as shown in Fig. 2. This model specifies a family of state transition diagrams where each instantiation has states that are connected by transitions. Each state can have any number of exit transitions, indicated by the $1 : 1$ and $1 : N$ cardinality on the downward arrow in the figure, i.e., each state can have between 1 and N transitions and each transition has to exit between 1 and 1 states (it has to be connected to one and only one state), where N represents infinitely many. Each transition can enter only one state and each state may have any number

of entering transitions (indicated by cardinality on the upward arrow). The transitions between states have two attributes, *viz.*, one condition that allows the state transition to be taken and one optional (indicated by the $0:1$ cardinality) action.

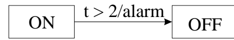

Figure 1: A state transition diagram model.

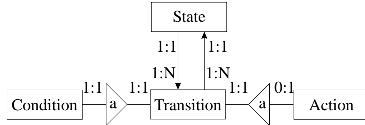

Figure 2: A model of state transition diagram models.

The model in Fig. 2 can be used to specify different state transition diagram formalisms, e.g., the action attribute can be made mandatory for each transition by changing the cardinality from $0:1$ to $1:1$. Furthermore, actions can be associated with states as well and hierarchical state machines can be modeled by giving each state a state attribute (i.e., a relation with itself).

Such a model of the modeling language is called a *meta* model. It prescribes the possible mathematical structures (formalisms) that can be expressed in the modeling language and can be tailored to specific needs of particular domains. From the meta model specification, the modeling language can then be instantiated automatically. This requires the meta model modeling formalism to be sufficiently rich and support the constructs needed to define a modeling language. To allow for easy extension, the meta model modeling formalism can be modeled by a *meta meta* model. This meta meta model specification captures the basic elements that can be used to design a meta model modeling formalism. In case new concepts and structures are required, these can be conveniently modeled at a meta meta level.

For example, the state transition diagram meta model in Fig. 2 is limited to the family of state transition diagrams. This restriction can be lifted by modeling the model of state transition diagrams in Fig. 2 by the meta meta model in Fig. 3. It contains an abstract representation of the mechanisms that are part of the state transition diagrams meta model, i.e., entities (states, transitions), attributes (actions and conditions), and relations between them. This meta meta model groups entities and relations by an *object* model component and each of them optionally has any number of attributes. It also shows that each relation connects to one entity as its source (marked src) and one entity as its destination (marked dst), and, therefore, directed links can be used. The meta meta model specification language has to consist of entities, attributes to specify the cardinality and relations to specify the three types, (i) source relations (src), (ii) destination relations (dst), and (iii) attribute relations (atr).

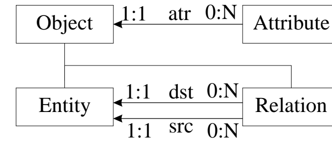

Figure 3: A state transition diagram meta meta model.

Given the meta meta model in Fig. 3, a broader family of meta models can be described. For example, a Petri net as illustrated in Fig. 4 consists of *places* (shown as transparent circles) and *transitions* (shown as solid rectangles). Each place has *connections* to transitions and each transition to places. A transition may have a condition and when this condition is true and all its input places, i.e., places that are the source to the transition, contain a *token* (shown as a black dot inside a place), it may fire (the corresponding transition may be executed). The Petri nets meta model shown in Fig. 5 specifies places and transitions that can connect to one another. Furthermore, the tokens are specified as optional attributes of a place and conditions as an optional attribute of a transition. The family of Petri nets that can be instantiated from this meta model allows places with multiple tokens. However, in some modeling paradigms, only one token per place is allowed, which can be conveniently changed in the meta model by specifying $0:1$ cardinality, and, consequently, constraining the family of Petri nets. Note that this represents both a static and dynamic constraint. When the Petri net is initialized, it can be ensured by the model editor that each place has been assigned at most one token. However, a dynamic check is still required whether this constraint is violated during execution.

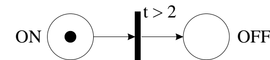

Figure 4: A Petri net model.

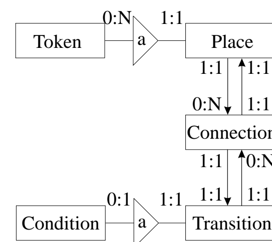

Figure 5: A model of Petri net models.

In addition to the elements of the graphical language, often a *constraint language* is facilitated by a meta modeling language to specify domain specific constraints that are hard to incorporate otherwise in the

Table 1: Four layer meta modeling structure.

Order	Description	Example
meta meta model	Modeling language for specifying meta models.	`Relation hasDestiny Entity`
meta model	Modeling language for specifying models, an instantiation of a meta meta model.	`State connectsTo Transition`
model	The model of an object (that could be a model), an instance of a meta model.	`when t > 2 transition from ON to OFF`
object data	An instance of a model.	$0 < \mathtt{t} \leq 2$ $alarm = F$ $\mathtt{t} > 2$ $alarm = T$

meta model. This language can be used to rule out semantically incorrect models and greatly reduce the family of models that can be modeled [23]. For example, a model of a family of Petri nets could include the constraint that there should never be more than ten tokens in the net (to model a resources limit) by an additional specification `Token.allInstances->size < 10`.

The outlined meta modeling approach leads to the four layer structure in Table 1 [2]. The object data row represents the data generated from a particular model, e.g., the simulation results of a physical system model in the time domain. This data is one instance of the set of data that can be generated by the model. The model row represents the particular model such as the state transition diagram in Fig. 1. At a meta level, the meta model row represents a class of models, e.g., the model of the family of state transition diagrams in Fig. 2. This meta model is described by a language that is specified by a meta meta model, the meta meta model row. This could be the model in Fig. 3.

The apparent advantage of this approach is the tremendous flexibility that can be achieved. Consider the iterator construct discussed in Section 3, at a meta level it can be specified to consist of an initial value, final value, and step size, but the concrete syntax can be instantiated as desired. This does not affect the abstract syntax nor the semantics, though, and exchange of models with this construct is inherently supported. Moreover, the concrete syntax of the iterator construct will be automatically adapted to the desired form when loaded by a different tool.

This flexibility also manifests in the ease with which new formalisms can be designed. By adapting the model of a modeling formalism, and automatically generating a prototype of the modeling environment, design choices can be rapidly evaluated. Furthermore, if the same language for meta model specification is used, consistency between different formalisms can be achieved. For example, if a component in a block diagram has certain output signals, these values have to be computed internally. In case the particular component is modeled by a state transition diagram, the output of this model has to correspond with the block diagram output at a higher level.

To allow deeper specification of the semantics of a modeling formalism, it is often meaningful to express how a model structure (meta model) can be mapped onto other model structures. An invariant of this mapping must obviously be the modeled system's dynamic behavior. Examples of mappings are the transformation between a bond graph and a corresponding system of differential and algebraic equations (DAE), or between a statechart and an equivalent state transition diagram. When analyzing models, or when generating code, transformations are used. It is meaningful to chart possible formalism transformations in a *formalism transformation graph* (FTG) [26], and to allow for the specification of transformations which enables the automatic generation of model transformers/compilers.

5 Conclusions

Control system design is a process that involves many task specific activities that rely on dedicated formalisms and tailored tools. These formalisms are part of widely differing modeling paradigms and may differ considerably in their syntax, semantics, computational model, and representation. To achieve a comprehensive design approach, it is desirable to have these modeling paradigms understand each other so data and model fragments can be exchanged between formalisms and tools. Two basic approaches handle this problem: (i) the use of a generic standard and (ii) the use of a meta modeling approach. To unify all aspects in one generic standard is difficult to achieve in case of heterogeneous modeling paradigms. Often compromises cannot be avoided while any such diminishes the usuability of the standard.

The meta modeling approach models each formalism that is used by meta models that capture the family of models that can be designed using a given formalism. The model of the formalism then represents a meta meta model that captures the concepts and structures in a formalism. By using a common modeling language for these meta meta models a tool that understands this language can automatically instantiate any desired modeling formalism that can be constructed from the meta meta concepts and structures. By choosing these sufficiently abstract (entities, attributes, relations), the family of formalisms that can be captured ranges from data flow diagrams to energy based physical system modeling formalisms. Furthermore, meta meta model descriptions allow for easy experimenting with new formalisms and highly constrained and tailored domain

specific formalisms can be developed.

The interpretation of multi-paradigm models relies on different computational models where a distinct difference between continuous and discrete event behavior exists: Continuous behavior is typically generated by discrete points on interpolation polynomials with a certain degree of smoothness and communication requirements are much more stringent than for discrete event message passing. This distinction between continuous and discrete behavior is fundamental to analyses such as simulation [3, 15] and verification [24, 22] of heterogeneous models and requires methodologies especially designed for mixed continuous/discrete, *hybrid*, dynamic systems.

References

[1] DOME guide. http://www.htc.honeywell.com/dome/, Honeywell Technology Center, Honeywell, 1999. version 5.2.1.

[2] OMG unified modeling language specification, June 1999. version 1.3, http://www.omg.org/.

[3] Paul I. Barton. Modeling, simulation, and sensitivity analysis of hybrid systems: Mathematical foundations, numerical solutions, and sofware implementations. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[4] EIA/CDIF Technical Committee. CDIF CASE data interchange format – overview, January 1994. EIA Interim Standard EIA/IS-106.

[5] René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992.

[6] John Davis, II *et al.* Ptolemy II – heterogeneous concurrent modeling and design in java. http://ptolemy.eecs.berkeley.edu, Dept. of EECS, University of Califoria at Berkeley, 1999. version 0.1.1.

[7] Eric Engstrom and Jonathan Krueger. A metamodeler's job is never done: Building and evolving domain-specific tools with DOME. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[8] Johannes Ernst. Data interoperability between CACSD and CASE tools using the CDIF family of standards. In *1996 Intl. Symp. on CACSD*, pp. 346–351, Dearborn, MI, Sep. 1996.

[9] Johannes Ernst and Scott Washburn. Zero-latency engineering^{tm} for control design. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[10] Hilding Elmqvist *et al.* Modelica^{tm}–a unified object-oriented langauge for physical systems modeling: Language specification, December 1999. version 1.3, http://www.modelica.org/.

[11] Michael Fisher. Zero-latency engineering^{tm}. Aviatis Corp., White Paper, 1999.

[12] IEEE 1076.1 Working Group. IEEE standard 1076.1-1999, March 1999. http://www.vhdl.org.

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

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

[15] Karl Henrik Johansson, John Lygeros, Shankar Sastry, and Jun Zhang. Hybrid automata: A formal paradigm for heterogeneous modeling. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[16] D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley and Sons, New York, 2 edition, 1990.

[17] Edward A. Lee. Embedded software – an agenda for research. Technical Report M99/63, Dept. of EECS, University of California, Berkeley, CA 94720, 1999.

[18] Jie Liu and Edward A. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[19] Dieter Moormann, Pieter J. Mosterman, and Gert-Jan Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, (3):115–126, 1999.

[20] Pieter J. Mosterman, Martin Otter, and Hilding Elmqvist. Modeling Petri Nets as Local Contraint Equations for Hybrid Systems Using Modelica. In *SCSC'98*, pp. 314–319, Reno, Nevada, July 1998.

[21] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[22] Simin Nadjm-Tehrani. Formal methods for analysis of heterogeneous models of embedded systems. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

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

[24] Taeshin Park. Verification of large-scale hybrid systems using implicit model representation. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.

[25] J. Sztipanovits *et al.* MULTIGRAPH: An architecture for model-integrated computing. In *ICECCS'95*, pp. 361–368, Ft. Lauderdale, Florida, Nov. 1995.

[26] Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid system modeling. In *IEEE Intl. Symp. on CACSD*, Anchorage, Alaska, Sep. 2000.