Computer Automated Multi-Paradigm Modeling: An Introduction

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

Hans Vangheluwe
Modelling, Simulation and Design Lab
School of Computer Science
McGill University
McConnell Engineering Bldg. room 328
3480 University Street
Montreal, Quebec, Canada
hv@cs.mcgill.ca

Modeling and simulation are quickly becoming the primary enablers for complex system design. They allow the representation of intricate knowledge at various levels of abstraction and allow automated analysis as well as synthesis. The heterogeneity of the design process, as much as of the system itself, however, requires a manifold of formalisms tailored to the specific task at hand. Efficient design approaches aim to combine different models of a system under study and maximally use the knowledge captured in them. Computer Automated Multi-Paradigm Modeling (CAMPaM) is the emerging field that addresses the issues involved and formulates a domain-independent framework along three dimensions: (1) multiple *levels of abstraction*, (2) *multiformalism* modeling, and (3) *meta-modeling*. This article presents an overview of the CAMPaM field and shows how *transformations* assume a central place. These transformation are, in turn, explicitly modeled themselves by graph grammars.

Keywords: Multi-paradigm modeling, multi-formalism, multi-abstraction, meta-modeling, model transformation

1. Introduction

Modern engineered systems have reached a complexity that requires systematic design methodologies and model-based approaches to ensure correct and competitive realization. In particular, the use of digital controllers has proven to be difficult to manage as small errors in their design may lead to catastrophic failures. In addition, the interdependencies in the software that implements the control algorithms are difficult to oversee, which only exacerbates with the increasing size of embedded software. Similarly, the interdependencies between controllers scattered about the control system are difficult to manage. Their effects, as well as the subtle interaction between information-processing components and the physical environment, are difficult to analyze.

This article uses a *power window* system, as typically found in modern automobiles [1], as a running example. Figure 1 illustrates how a worm gear is used to rotate the main lever of a scissor-type lift mechanism (which contains a supporting rod in addition to the lever). This mechanism moves the window up and down between the bottom and top of the window frame. A DC motor connects to the worm gear to power the rotation, as commanded by the controller. An important consideration in the design of this system is the potential presence of an obstacle (such as a passenger's arm) between the window and the frame. The design of such a system progresses through a number of stages that may or may not use *different* models of the components and subsystems. For example, in the initial design stages, discrete event models may be used to design the hierarchical control structure of the main behavior (i.e., the passenger can command the window to move; moving decomposes into up and down). In more advanced design stages, the models become increasingly detailed (e.g., adding data acquisition effects) and may include continuous-time and power effects (e.g., to simulate the current drawn by the DC motor). In addition to this, system integration requires increasingly comprehensive analyses that involve different models used in designing different aspects of the system's functionality. For example, the model of the lift mechanism may be designed using *bond graphs* [2], while the main controller may be modeled using *Statecharts* [3], and the pulse-width modulation of the DC motor may be modeled using *time-based block diagrams* [4].

Comprehensive design and analysis is the main topic of new holistic design paradigms such as mechatronics [5] and System-on-Chip [6]. These approaches aim to avoid overspecification and to attain optimal performance. The corresponding design paradigms require many different levels of explanation, different theories, and modeling

SIMULATION, Vol. 80, Issue 9, September 2004 433-450
©2004 The Society for Modeling and Simulation International
DOI: 10.1177/0037549704050532

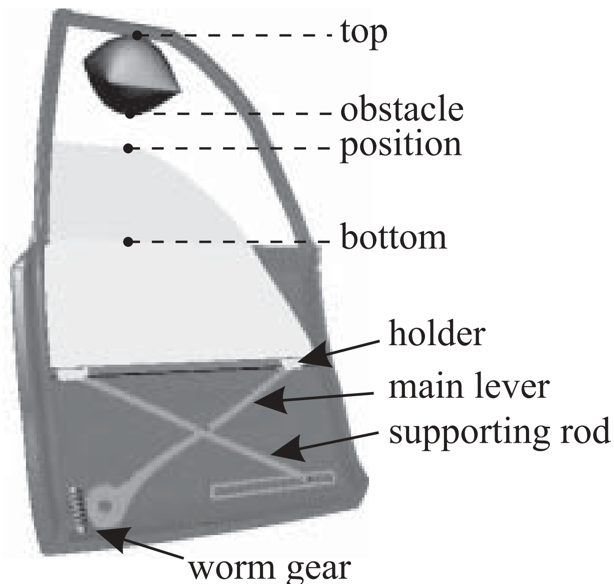

Figure 1. A power window system

languages. In general, complex systems are becoming increasingly heterogeneous because of the integration of different implementation technologies in the modern design process. In addition, the many engineering disciplines that are involved in system design all have developed domain- and problem-specific (often proprietary) formalisms to match their needs optimally.

To address these complex systems issues, designers turn to modeling and simulation technologies. Whereas in the early history of the field of control engineering, differential equation models could still be directly derived from the system, the complexity of systems has increased far beyond that. For example, the need to defer expensive prototyping while obtaining maximum confidence in the design requires models with extreme detail that incorporate many implementation effects. Sophisticated modeling languages facilitate these requirements as model design can be done at a high conceptual level. This trend is very evident in software design, in which there is a shift from programming software to modeling software. In particular, the model-driven architecture (MDA) [7] focuses on the explicit modeling of software design specifications as well as on their transformation from a platform-independent model (PIM) abstraction level, via a platform-specific model (PSM) abstraction level, to the code level.

Multiparadigm techniques have been successfully applied in the field of software architectures [8], control system design [9], model-integrated computing [10], and tool interoperability [11]. To advance the state of the art and to accumulate knowledge scattered across domains, a domain-independent framework for complex systems development is needed. The emerging field of Computer Automated Multi-Paradigm Modeling (CAMPaM) [12-14] aims to achieve this by addressing and integrating three orthogonal directions of research:

1. *model abstraction* deals with the different levels of detail of models and the relationship between these models;
2. *multiformalism modeling* deals with coupling of and transforming between the manifold of formalisms used;
3. *meta-modeling* deals with the description of modeling formalisms and their domain-specific aspects.

CAMPaM explores the possible *combinations* of these notions to provide an application- and domain-independent framework; to combine, transform, and relate formalisms; to generate maximally constrained domain- and problem-specific formalisms, methods, and tools; and to verify consistency between multiple views. This is a powerful approach that allows the generation (instantiation) of domain- and problem-specific methods, formalisms, and tools. Thanks to a common meta-language, the models that use different formalisms (instances of the different family of models that each of the formalisms embodies) can be integrated by combination, layering, heterogeneous refinement, and multiple views [9, 15-17]. When extended with model transformation, multiparadigm modeling leads to a suite of technologies and applications that convert a model into a different representation, possibly changing the abstraction, partitioning, and hierarchical structure.

This article gives an overview of CAMPaM. It first presents the separate *dimensions of CAMPaM* in section 2, which will repeatedly highlight the importance of transformations. In section 3, the different dimensions are then explicitly related to the ubiquitous *transformation* concept. Next, section 4 concentrates on the *execution of heterogeneous models*. Section 5 then presents the conclusions of this contribution.

2. CAMPaM: The Three Dimensions

A *conceptual* (as opposed to a physical) model is the cross-product of the system under scrutiny, the level of abstraction, and the formalism used. In the following, when a model is referred to, a conceptual model is meant.

2.1 Abstraction

A model is designed to solve a problem. How well it suits this purpose determines its quality. As such, a system has infinitely many models that each can be best for a given task. This task notion is captured by the *level of abstraction* determined by the perspective one has on a system, the problem to be solved, and the background of the model designer.

For example, to investigate the requirement of the power window in Figure 1—that the window be rolled down 10 cm in case of an object between the window and

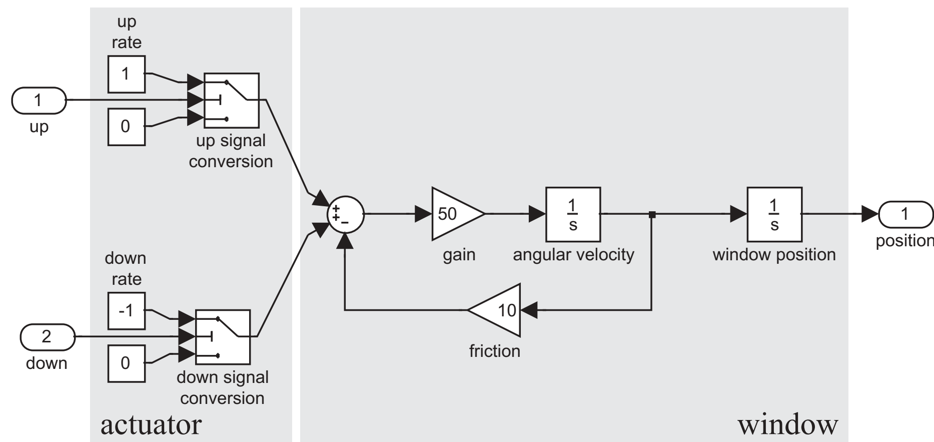

Figure 2. Low-order model of power window behavior

the frame—a continuous-time model is needed.This can be a model of low order, as shown in Figure 2, where a Simulink® [4] *continuous-time-based block diagram* is shown that is of second order. It consists of an actuator part that converts the control signals up and down before they enter the window part that consists of a gain, force integrator, and then angular velocity integrator. Viscous friction determines the force from the velocity and feeds back negatively. To evaluate the requirement that the force on the object shall not exceed 100 N, a more detailed model is required. A possible model is shown in Figure 3. The actuator takes the two control signals and integrates their voltage into an angular velocity. This angular velocity drives the worm gear, which rotates with a different velocity. The difference between the two, through some gain, is the torque acting on the gear. This torque passes through two gains; the first one is because of the gear ratio, and the next one is because of the effect of the main lever. The result is a force moving the window. The friction force that also acts on the window is computed as a nonlinear function of the window velocity and position. Note that this model is at a different level of abstraction (more detailed), yet it is still formulated using the same formalism: a *continuous-time-based block diagram*. Systematically and automatically deriving models of different complexity significantly increases productivity as well as quality of models. It also cross-correlates different modeling efforts. Note that changes between levels of abstraction may involve using different formalisms but not necessarily so, as illustrated by the power window example.

In general, the abstraction process can be considered a type of transformation that is preserving some invariant properties (usually behavioral) of the system. The challenge is to model these transformations and use such transformation models to automate model abstraction and refinement as well as abstraction-level selection. This facilitates many applications. For example, in optimization, increasingly complex models may be more likely to find a global optimum [18]. Another example is the use of one base model that embodies as much detail as possible for any given task. Less detailed models can be automatically derived from it for the different design and operation tasks (e.g., control design, performance assessment, and model-based diagnosis) [19]. Another possible application is in numerical solvers that adapt the complexity of the model to the efficiency requirements (e.g., real-time simulation constraints) [20]. In reactive learning environments (microworlds), increasingly adding detail to the world model leads to a challenging environment for students at different levels of proficiency [21]. Note that, in general, it may be possible to automatically add model detail as well as to automatically reduce complexity of a base model [22].

2.2 Formalism

Independent from the changes in abstraction level, changes in the *modeling formalism* can be made. A change in formalism may induce a change in abstraction level, but this is not necessary. Which formalism to use depends not only on the desired level of abstraction but also on the data available to calibrate the model, the available numerical solvers (or, more generally, what tools facilitate the desired analyses), and, as indicated earlier, what problem needs to be solved.

For example, the design of the power window controller is most naturally expressed in the *Statecharts* formalism [23, 24]. A possible implementation using *DCharts* [25, 26] is shown in Figure 4. The controller is in its Neutral, movingUp, or movingDown state. Hierarchy is used, for example, to transition from either movingUp or movingDown to Neutral when the cmdStop event occurs.

Because the hierarchical nature of Statecharts may hamper analysis, it is often desirable to transform the hierarchical state transition diagram into a flat *state transition*

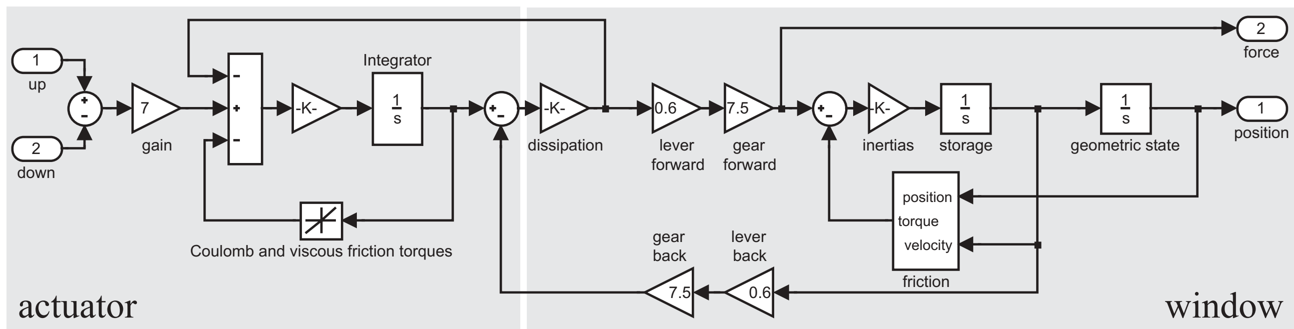

Figure 3. Higher order model of power window behavior

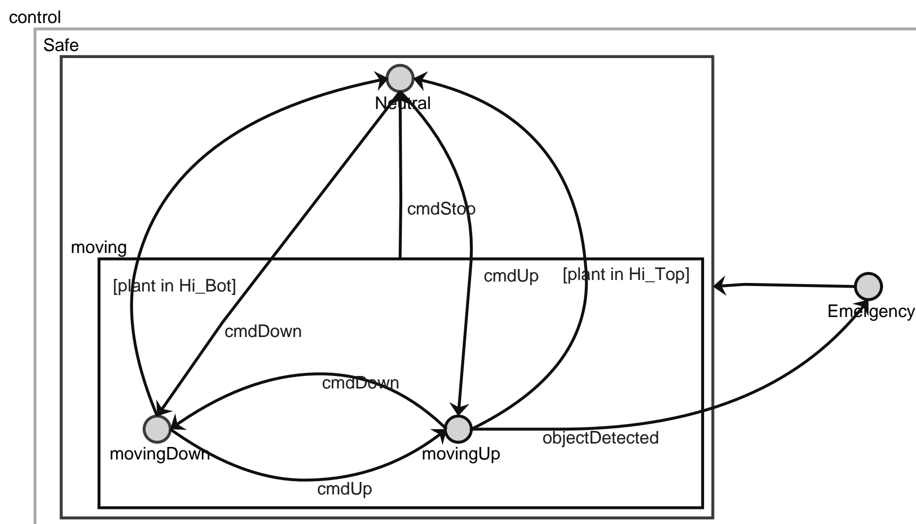

Figure 4. Statechart of the power window control

diagram [27].[1] The equivalent state transition diagram is shown in Figure 5. Since state transition diagrams are a proper subset of Statecharts, a more illustrative formalism transformation is given by the subsequent transformation to an equivalent Petri net [28], shown in Figure 6. The transformation into a Petri net representation may allow for different types of analysis (such as static deadlock checks) that are otherwise not possible.

Figure 6 shows the transformed controller in the top-right corner. States in the state transition diagram have been mapped onto Petri net Places. Transitions in the state transition diagram have been mapped onto Petri net Transitions. The input events cmdUp, cmdDown, and cmdStop are represented as Places in the Petri net model. This provides an explicit *interface* to the controller model. Placing a token in an interface Place is equivalent to generating the corresponding input event in the state transition diagram. Transitions triggered not by an external event but rather by a change in system state, such as `[in S]`, have been mapped onto a two-way arc between the Place representing `S` and the Petri net Transition. The return arc is necessary as *testing* state `S` should not modify the current state.

Thanks to the inherent concurrency in Petri nets, not only the controller but also other concurrent parts of the system can be easily modeled. The small part of the plant behavior pertaining to the presence of an object is needed at this stage of the design process. This is modeled in the top-left part of Figure 6. In the bottom-right part of Figure 6, an *environment* that generates exactly one of cmdUp, cmdDown, or cmdStop, exactly once, is modeled. In the bottom-left corner of the same figure, the possible insertion and subsequent removal of an object are modeled. Note how the controller does not directly observe the

1. Note that some tools actually exploit the hierarchical nature to apply more efficient analysis algorithms or to synthesize more efficient code.

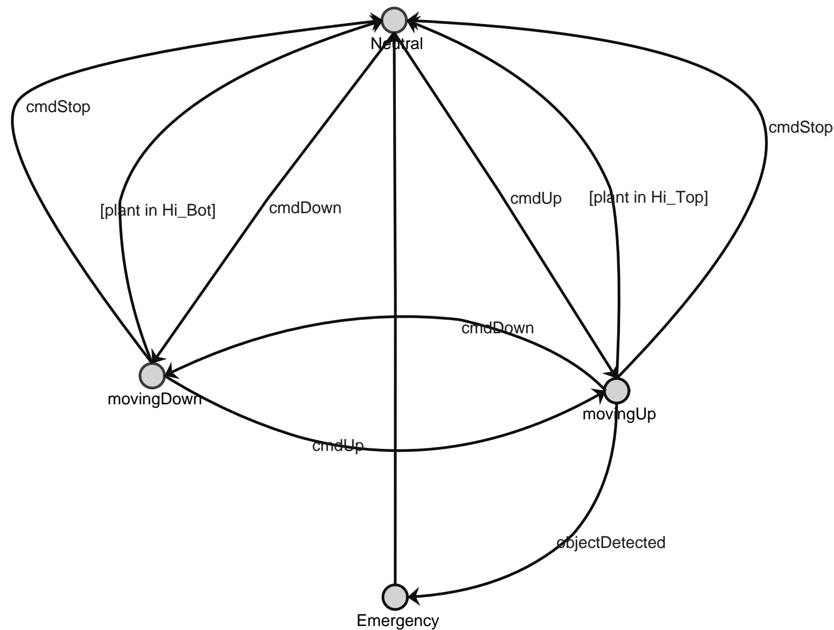

Figure 5. State transition diagram of the power window control

presence of an object but rather indirectly, via the plant's power consumption, as will be discussed later. The model does not specify *when* (i.e., in what order) any of the above events occur. This makes Petri nets more suited than Statecharts to model and subsequently analyze a whole *class* of reactive behaviors.

A formalism consists of a *syntactic* part and a *semantic* part. The syntactic part deals with the form and structure of valid models. It is typically separated into a *concrete* part and an *abstract* part. The former pertains to the actual appearance (which can be, for example, textual or graphical), while the latter is about how the language elements may be connected: for example, that an algebraic assignment has a left-hand side and a right-hand side or that a Petri net Place may only be connected to a Petri net Transition.

The semantic part concerns the *meaning* of the formalism's syntactic constructs. It can be specified in an *operational* manner, which captures explicitly how a model can be executed. Alternately, a *denotational* or *transformational* specification can be given by providing rules to map a model in a given formalism onto a model in a different formalism for which a semantics is available. For example, the state transition model in Figure 4 can be mapped onto the behaviorally equivalent Petri net in Figure 6. Note that, in a sense, the operational approach is also one of transformation as it transforms an executable specification into a simulation trace.

Formalism transformations allow one to:

- generate a functional model from software or even an execution trace (e.g., a solver procedure can be synthesized from the concepts that are part of the domain-specific ontology [i.e., function calls] and their respective execution ordering);
- automate the generation of different views on a system (e.g., scenario diagrams from a functional model) or even an implementation model;
- automate design by generating specifications from requirements, ultimately leading to automated code synthesis (or at least stub generation), which, in turn, can be integrated in an automated optimization and runtime architecture reconfiguration scheme for hardware and software or for software only (e.g., for System-on-Chip applications [6]);
- automatically derive a reconfiguration model for guiding runtime system changes from functional and architectural models [29, 30];
- use best-of-class methods and tools by generating the required data and model representation format to prevent inconsistencies at the boundaries between engineering teams, engineering software, and multiple modeling paradigms, as well as to enable the sharing and coordinating of information flow with minimal overhead [31].

In addition to facilitating usage of multiple formalisms in isolation, it should be possible to combine and even integrate models that use a variety of different formalisms by means of coupling and transformation. This multiformalism modeling is often facilitated on the semantic level by providing a sufficiently general execution mechanism onto which many different formalisms can map their semantics. Examples of this are the DEVS formalism [7, 32, 33], Ptolemy [34], and S-functions [4].

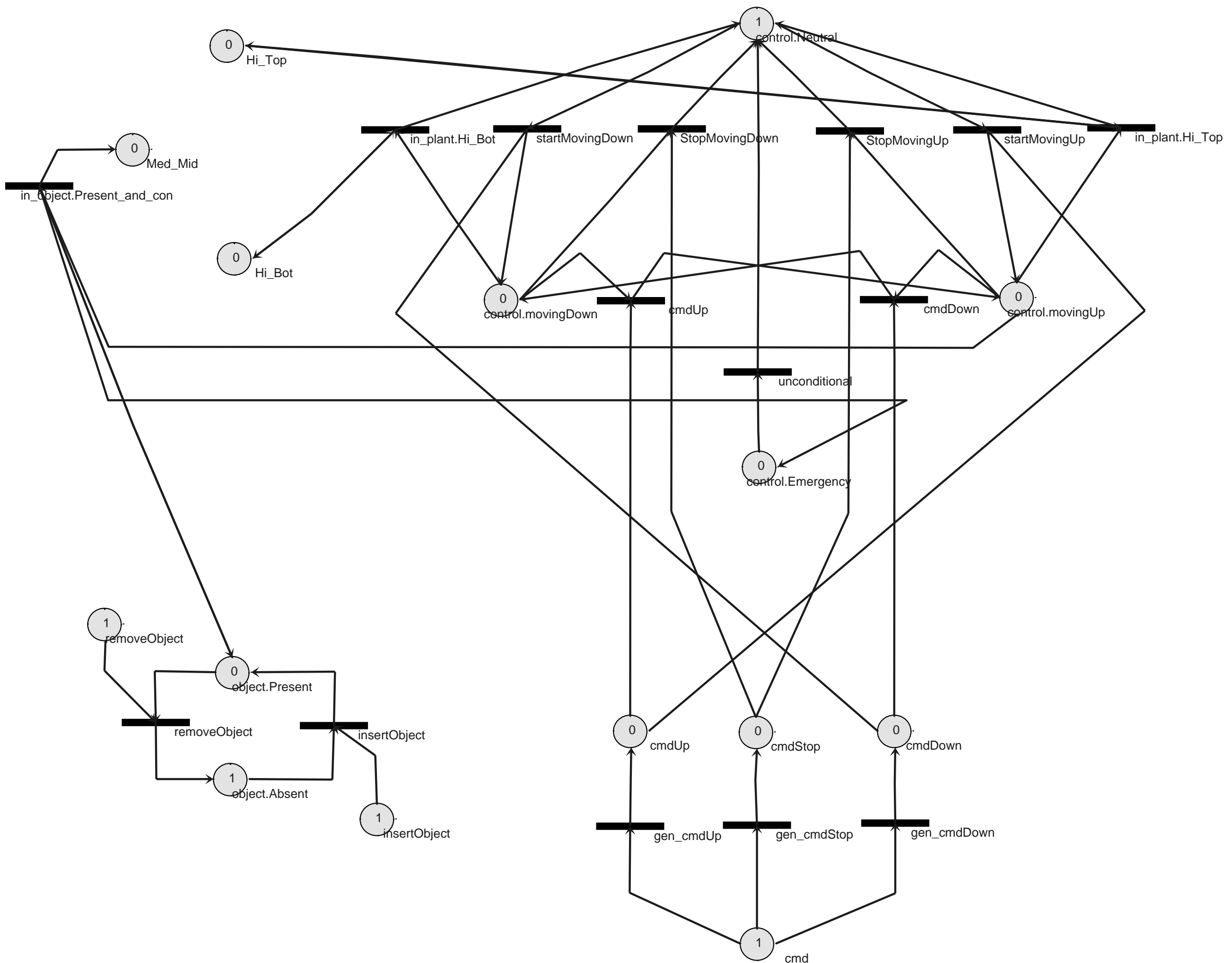

Figure 6. Petri net model of the power window control

2.3 Meta-Modeling

The third CAMPaM dimension concentrates on the modeling of modeling formalisms (i.e., meta-modeling) [35-38]. To quickly construct domain-specific and tailored formalisms and their (visual) editors, explicitly modeling the formalisms is the most efficient approach, provided that a meta-model compiler is available to synthesize the tools.

Meta-modeling is the explicit modeling of a class of models (i.e., of a modeling language). A meta-model $M_{\mathcal{L}}$ of a modeling language $\mathcal{L}$ is a model (with textual or visual syntax) in its own right, which *specifies* precisely which models m are elements of $\mathcal{L}$.

Modeling environments based on meta-modeling will *check*, by means of a meta-model $M_{\mathcal{L}}$, whether a given model m is in $\mathcal{L}$, or they will *constrain* the modeler during the incremental model construction process such that only elements of $\mathcal{L}$ can be constructed. Note how the latter approach, though possibly more efficient due to its incremental nature of construction and consequently of checking, may render certain valid models in $\mathcal{L}$ unreachable through incremental construction.

The advantages of meta-modeling are numerous. First, an *explicit* model of a modeling language can serve as *documentation* and as *specification*. Such a specification can be the basis for the *analysis* of properties of models in the language. From the meta-model, a modeling environment may be *automatically* generated. The flexibility of the approach is tremendous: new languages can be designed by simply *modifying* parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modeling language is apparent. Above all, with an appropriate meta-modeling tool, modifying a meta-model and subsequently generating a modeling tool for possibly visual languages is orders of magnitude *faster* than developing such a tool by hand. The tool synthesis is *repeatable* and *less error prone* than hand crafting.

As meta-models are models in their own right, they must be elements of a modeling language (or, put differently, expressed in a particular formalism). This modeling language can be specified in a so-called meta-meta-model. Note how the *meta* qualifier is obviously *relative* to the original model.

Although an arbitrary number of meta-levels are possible in principle, in practice, some modeling languages/formalisms such as entity-relationship diagrams (ERD) and UML class diagrams are expressive enough to be expressed in themselves. That is, the meta-model of such a language $\mathcal{L}$ is a model in language $\mathcal{L}$. From the implementation point of view, this allows one to *bootstrap* a meta-modeling environment. This is often referred to as *meta-circular* interpretation.

To illustrate these concepts, consider a meta-model of the Statechart formalism that was used in Figure 4 to model part of the power window control. This meta-model, shown in Figure 7, is itself a model in the UML class diagram formalism. The basis of the Statechart formalism is the state transition diagram, which consists of States, Transitions, Conditions, Actions, and an Initial state transition. These entities are marked by rectangles in the meta-model. Transitions connect states, indicated by the directed relations. Each state may have 0 or more transitions exiting it, as marked by the $0 : N$ cardinality. Similarly, each state may have $0 : N$ transitions entering it. On the other hand, a transition can exit from one and only one state and enter one and only one other, indicated by the $1 : 1$ cardinalities. Also, a state may have one initial transition connected to it, and the initial transition can connect to one and only one state. Each transition may contain a condition and an action, indicated by the diamond connection.

To extend this state transition diagram meta-model to include Statecharts,[2] the State, Initial, and Transition entities are all derived from one Element entity. Now, by allowing states to contain elements, hierarchy is introduced. Furthermore, each state is specialized into an AND state or an OR state. All states in an AND state are active when the containing state is active, whereas in an OR state, only one of the contained states is active (the traditional state transition diagram notion). These constraints are not explicitly modeled here for clarity.

In many research endeavors, meta-modeling is applied to capture the abstract syntax of a class of models. Models of transformation allow a generalization of this to include the semantics as well. This then becomes the enabling technology for (1) the design of tailored formalisms and tools by constituting an infinitely fine-grained spectrum of formalisms; (2) the use of domain-specific formalisms and tools to facilitate high-level, model-based programming; (3) including domain constraints within formalisms; and (4) finding analogies, similarities, and differences between models of different system views and aspects.

An illustrative example of meta-modeling is its use to facilitate exchange of models and data between tools for computer-aided software/systems engineering (CASE). The corresponding CDIF (CASE Data Interchange Format) project [41] proved meta-modeling to be an industrial-strength technology.A crucial aspect of CDIF was its extensibility. New formalisms could be developed and used in exchange transactions by first making the model of the formalism available using the meta-formalism of CDIF, an entity-attribute-relationship (EAR)–type formalism (a proven powerful formalism for modeling the syntax of many types of formalisms). The CASE tool first processes the meta-model so it "understands" the data that follow. The only formalism that needs to be shared between tools is the EAR one that specifies the meta-models.

2.4 Relating the Dimensions

So far, the independent dimensions of CAMPaM have been presented. In general, however, these dimensions interact, and their full benefits are reaped only when the different dimensions are cross-correlated.

An example of this is the translation of a continuous-time model into a finite state representation to validate the control structure designed in Figure 4. This approach is common in analysis and verification approaches that require a finite state space (e.g., Lunze [42] and Preußig et al. [43]). For example, Figure 8 shows a trajectory in the force-position phase space where the window is commanded from its bottom-most position to the top. This trajectory was generated from the model in Figure 3. Based on this, a finite state discrete model can be derived using a grid, as shown by the dashed horizontal and vertical lines in Figure 8. The finite state machine abstracted from this trajectory is shown in Figure 9. Note that during a normal closing operation, the system moves through the sequence of states Low_Bot, Low_Mid, Low_Top, Med_Top, Hi_Top, Med_Top, Low_Top.

When an object is present, the system is required to detect and roll down the window before the force exceeds 100 N. A continuous-time phase space trajectory that stops movement without reversal is shown in Figure 10. The additional state Med_Mid that may be traversed is an emergency state and should not be reachable in normal operation. It is entered when the object event occurs. The state Hi_Mid is a violation of system requirements as it corresponds to a state where there is too much force exerted on the object. This state should not be reachable given a proper control algorithm. The nonreachability of the unwanted state can be *verified* once a discrete event model of the plant behavior is available. It has to be verified that the control model in Figure 4, in which the cmdDown command always follows objectDetected, renders the control *safe* (i.e., the state Hi_Mid cannot be reached).

2. Note that the actual Statechart formalism, such as used in Stateflow® [39], is much more complex than the reduced form discussed here. A more detailed meta-model of Statecharts can be found in Pereira Remelhe et al. [40].

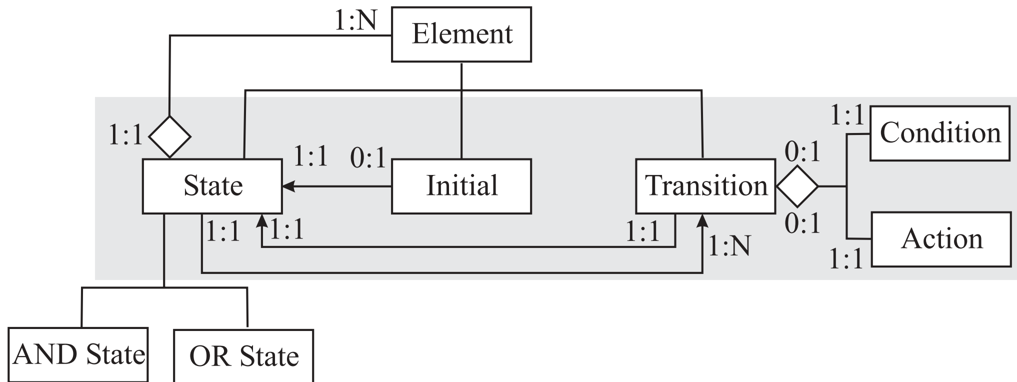

Figure 7. Statecharts meta-model

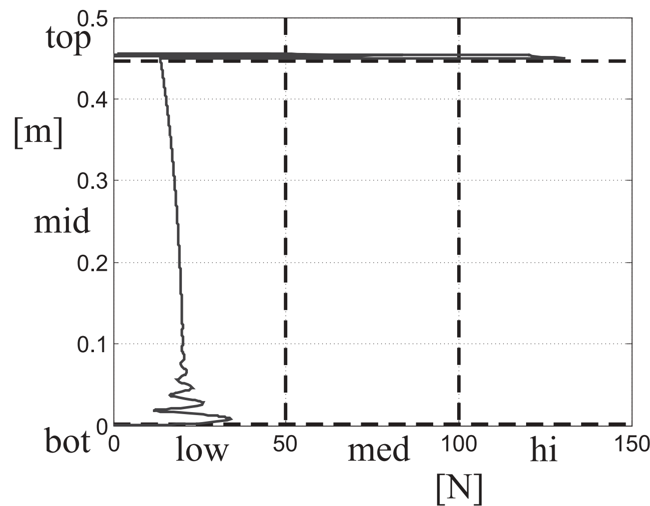

Figure 8. Window force versus position when the top (at 45 cm) is reached

The finite state machine model of the system behavior was automatically transformed into a behaviorally equivalent Petri net. This Petri net was subsequently combined with the controller and environment model shown in Figure 6. The resulting Petri net model is shown in Figure 11. From this model, the CAMPaM tool AToM[3] has computed in Figure 11 the coverability graph [28, 44], shown in Figure 12 as depicting all reachable states. The design of AToM[3] has been described in de Lara and Vangheluwe [45] and de Lara Jaramillo, Vangheluwe, and Moreno [46].

Figure 13 shows a simple computation tree logic (CTL) [47] model expressing that Hi_Mid is unwanted.

The results of a simple model checker included in AToM[3] in Figure 14 show that this state is not reachable. Furthermore, after constructing an appropriate integer linear programming (ILP) problem from the coverability graph, the following conservation laws (that were intuitively expected) were inferred:

$$\left\{\begin{array}{rcl} x[\text{object.Absent}] + x[\text{object.Present}] & = & 1 \\ x[\text{Low_Bot}] + x[\text{Low_Mid}] + x[\text{Low_Top}] + & & \\ x[\text{Med_Bot}] + x[\text{Med_Mid}] + x[\text{Med_Top}] + & & \\ x[\text{Hi_Bot}] + x[\text{Hi_Top}] & = & 1 \\ x[\text{control.Neutral}] + x[\text{control.movingUp}] + & & \\ x[\text{control.movingDown}] + & & \\ x[\text{control.Emergency}] & = & 1 \end{array}\right.$$

3. Transformation

In section 2, the notion of the *transformation* of models has been a recurring concept. It is a crucial element in model-based endeavors. It forms the glue between the three orthogonal directions of CAMPaM: formalisms, abstraction, and meta-modeling.

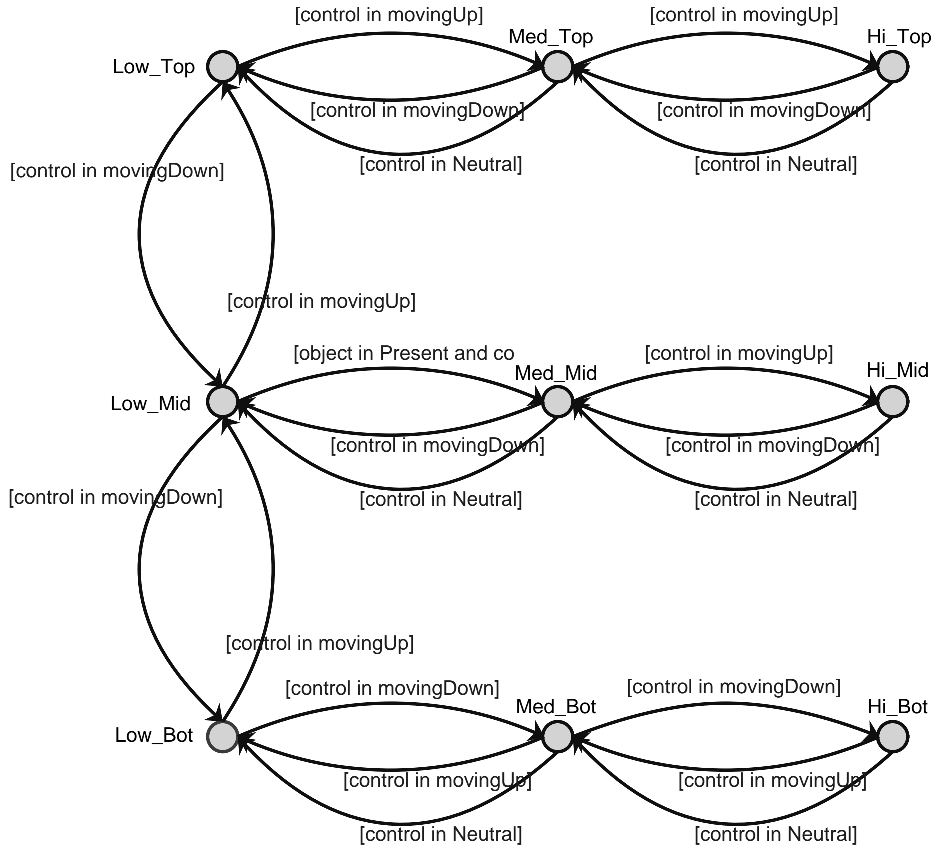

Figure 9. Discrete event model of plant behavior

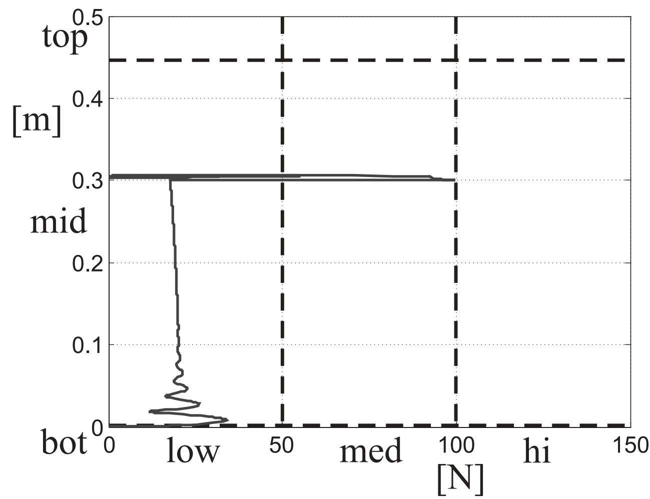

Figure 10. Window force versus position when an object (at 30 cm) is detected

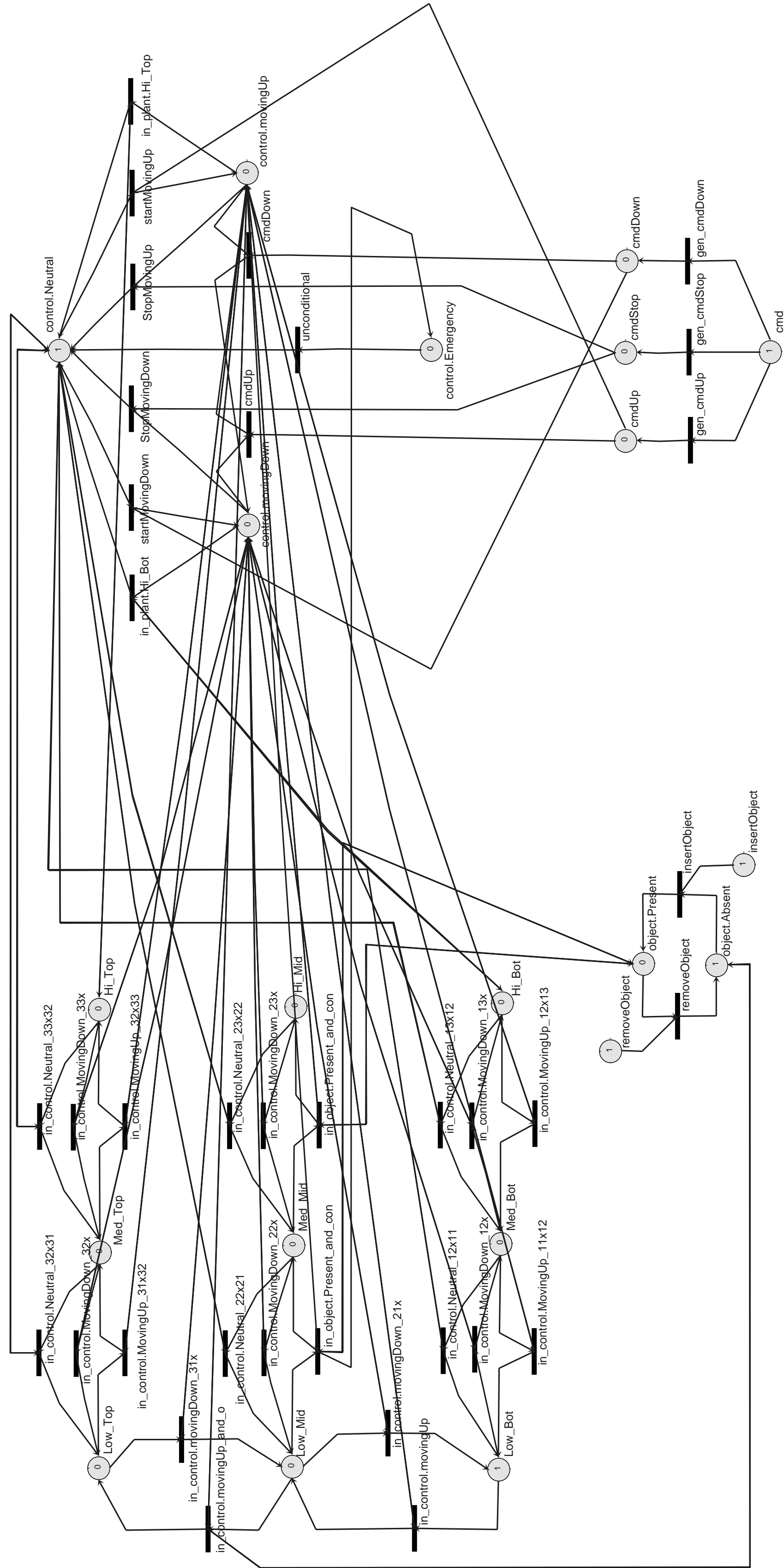

Figure 11. Petri net model of plant, controller, and environment

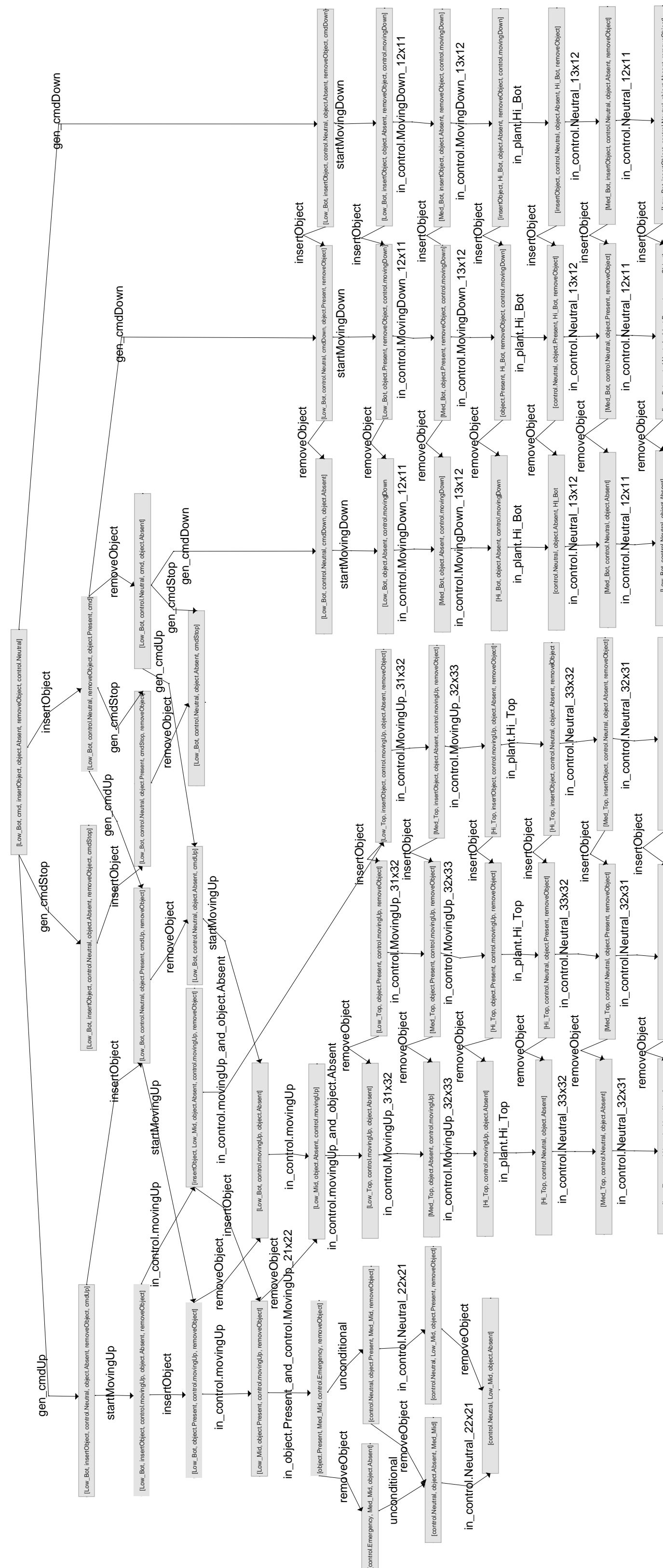

Figure 12. Reachability graph for the Petri net model

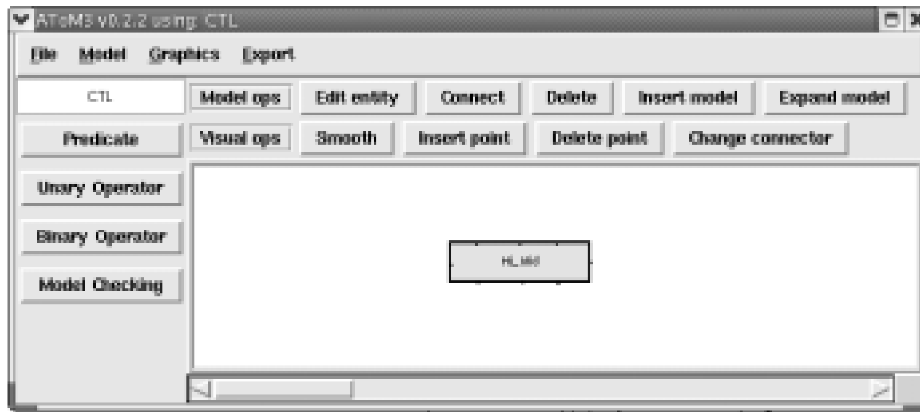

Figure 13. A model of unwanted patterns

As in many cases, models, meta-models, and meta-meta-models are all attributed, typed graphs. These graphs can be transformed by means of graph rewriting. The rewriting is specified in the form of models in the graph grammar [48] formalism.

Graph grammars are a generalization of Chomsky grammars for graphs [48, 49].Graph grammars are composed of production rules, with each having graphs on their left- and right-hand sides (LHS and RHS). In the *double-pushout approach* (DPO), productions have the following form: $p : L \stackrel{l}{\longleftarrow} K \stackrel{r}{\longrightarrow} R$, where L (left-hand side), K (interface graph), and R (right-hand side) are graphs, and l and r are (usually injective) morphisms. That is, K is the set of nodes and edges that are preserved by the production, $L \setminus K$ is the set of nodes and edges that are deleted, and $R \setminus K$ is the set of nodes and edges that are created by the production. The diagram in Figure 15 sketches the application of a rule on a graph G, resulting in graph H.

Thus, to apply a production on a graph G, a match m should be found between the production's LHS L and the graph G. This can be either an injective or noninjective morphism. The next step is to delete all the elements in G matched with elements of $L \setminus K$. Finally, the elements of $R \setminus K$ are added. Note how this process can be expressed in terms of category theory as two pushouts in category *Graph* [49]. Furthermore, the double-pushout approach needs two additional conditions. The *dangling condition* specifies that if an edge is not deleted, its source and target nodes should be preserved. The *identification condition* specifies that if two nodes or edges are matched into a single node or edge in the host graph (via a noninjective morphism), then both should be preserved. Productions can be extended with sets of *application conditions* (AC) [50] of the form $\{P \stackrel{c_i}{\longrightarrow} Q\}$ and a morphism x from L to P. This means that to apply the rule, if an occurrence of P is found, then an occurrence of Q must be found for the rule to be applicable. Note that if c_i is empty, there is a *negative application condition* (NAC). In this case, if an occurrence of graph P is found, then the rule is not applicable. If $x = id_L$, then we have a *positive application condition*.

Some graph rewriting systems have control mechanisms to determine the order in which rules are checked. When multiple matches are found, nondeterminism occurs. This nondeterminism may be resolved in three ways. Evaluation can be sequentialized, a random matching rule may be chosen, or, if no conflicts exist, rules may be evaluated in parallel. Having these three possibilities gives one the power to model a variety of operational semantics of formalisms.

Three kinds of transformations of models are typically of interest. The first is model *execution* (defining the operational semantics of the formalism). The second is model transformation into *another formalism* (expressing the semantics of models in one formalism by mapping onto a known formalism). A special case of this is when the target formalism is textual. In this case, it is possible to describe, by means of meta-modeling, the *abstract syntax graph* of the textual formalism (i.e., the intermediate representation used by compilers once they parse a program in text form), in such a way that models in textual formalisms can then be processed as graphs. The third one is model *optimization*, for example, reducing its complexity (maintaining pertinent invariants, however).

On one hand, graph grammars have some advantages over specifying the computation to be done in the graph using a traditional programming language. Graph grammars are a natural, formal, visual, declarative, and high-level representation of the computation. Computations are thus specified by means of high-level models, expressed in the graph grammar formalism. The theoretical foundations of graph rewriting systems may assist in proving correctness and convergence properties of the transformation tool. On the other hand, the use of graph grammars is constrained by efficiency. In the most general case, subgraph isomorphism testing is NP-complete. However, the use of small sub-

```
NODE LABELS:

0['Low_Bot', 'cmd', 'insertObject', 'object.Absent', 'removeObject', 'control.Neutral', 'True']
1['Low_Bot', 'control.Neutral', 'removeObject', 'object.Present', 'cmd', 'True']
2['Low_Bot', 'control.Neutral', 'cmd', 'object.Absent', 'True']
3['Low_Bot', 'control.Neutral', 'object.Absent', 'cmdUp', 'True']
4['Low_Bot', 'control.movingUp', 'object.Absent', 'True']
5['Low_Mid', 'object.Absent', 'control.movingUp', 'True']
6['Low_Top', 'control.movingUp', 'object.Absent', 'True']
7['Med_Top', 'object.Absent', 'control.movingUp', 'True']
8['Hi_Top', 'control.movingUp', 'object.Absent', 'True']
9['Hi_Top', 'control.Neutral', 'object.Absent', 'True']
10['Med_Top', 'control.Neutral', 'object.Absent', 'True']
11['Low_Top', 'control.Neutral', 'object.Absent', 'True', 'deadlock']
12['Low_Bot', 'control.Neutral', 'object.Absent', 'cmdStop', 'True', 'deadlock']
13['Low_Bot', 'control.Neutral', 'cmdDown', 'object.Absent', 'True']
14['Low_Bot', 'object.Absent', 'control.movingDown', 'True']
15['Med_Bot', 'object.Absent', 'control.movingDown', 'True']
16['Hi_Bot', 'object.Absent', 'control.movingDown', 'True']
17['control.Neutral', 'object.Absent', 'Hi_Bot', 'True']
18['Med_Bot', 'control.Neutral', 'object.Absent', 'True']
19['Low_Bot', 'control.Neutral', 'object.Absent', 'True', 'deadlock']
20['Low_Bot', 'control.Neutral', 'object.Present', 'cmdUp', 'removeObject', 'True']
21['Low_Bot', 'object.Present', 'control.movingUp', 'removeObject', 'True']
22['Low_Mid', 'object.Present', 'control.movingUp', 'removeObject', 'True']
23['object.Present', 'Med_Mid', 'control.Emergency', 'removeObject', 'True']
24['control.Emergency', 'Med_Mid', 'object.Absent', 'True']
25['control.Neutral', 'object.Absent', 'Med_Mid', 'True']
26['control.Neutral', 'Low_Mid', 'object.Absent', 'True', 'deadlock']
27['control.Neutral', 'object.Present', 'Med_Mid', 'removeObject', 'True']
28['control.Neutral', 'Low_Mid', 'object.Present', 'removeObject', 'True']
29['Low_Bot', 'control.Neutral', 'object.Present', 'cmdStop', 'removeObject', 'True']
30['Low_Bot', 'control.Neutral', 'cmdDown', 'object.Present', 'removeObject', 'True']
31['Low_Bot', 'object.Present', 'removeObject', 'control.movingDown', 'True']
32['Med_Bot', 'object.Present', 'removeObject', 'control.movingDown', 'True']
33['object.Present', 'Hi_Bot', 'removeObject', 'control.movingDown', 'True']
34['control.Neutral', 'object.Present', 'Hi_Bot', 'removeObject', 'True']
35['Med_Bot', 'control.Neutral', 'object.Present', 'removeObject', 'True']
36['Low_Bot', 'control.Neutral', 'object.Present', 'removeObject', 'True']
37['Low_Bot', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'cmdUp', 'True']
38['Low_Bot', 'insertObject', 'control.movingUp', 'object.Absent', 'removeObject', 'True']
39['insertObject', 'Low_Mid', 'object.Absent', 'control.movingUp', 'removeObject', 'True']
40['Low_Top', 'insertObject', 'control.movingUp', 'object.Absent', 'removeObject', 'True']
41['Med_Top', 'insertObject', 'object.Absent', 'control.movingUp', 'removeObject', 'True']
42['Hi_Top', 'insertObject', 'control.movingUp', 'object.Absent', 'removeObject', 'True']
43['Hi_Top', 'object.Present', 'control.movingUp', 'removeObject', 'True']
44['Hi_Top', 'control.Neutral', 'object.Present', 'removeObject', 'True']
45['Med_Top', 'control.Neutral', 'object.Present', 'removeObject', 'True']
46['Low_Top', 'control.Neutral', 'object.Present', 'removeObject', 'True']
47['Hi_Top', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'True']
48['Med_Top', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'True']
49['Low_Top', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'True']
50['Med_Top', 'object.Present', 'control.movingUp', 'removeObject', 'True']
51['Low_Top', 'object.Present', 'control.movingUp', 'removeObject', 'True']
52['Low_Bot', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'cmdStop', 'True']
53['Low_Bot', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'cmdDown', 'True']
54['Low_Bot', 'insertObject', 'object.Absent', 'removeObject', 'control.movingDown', 'True']
55['Med_Bot', 'insertObject', 'object.Absent', 'removeObject', 'control.movingDown', 'True']
56['insertObject', 'Hi_Bot', 'object.Absent', 'removeObject', 'control.movingDown', 'True']
57['insertObject', 'control.Neutral', 'object.Absent', 'Hi_Bot', 'removeObject', 'True']
58['Med_Bot', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'True']
59['Low_Bot', 'insertObject', 'control.Neutral', 'object.Absent', 'removeObject', 'True']

  --> FORMULA = Hi_Mid
  --> LIST OF NODES WHICH SATISFY THE FORMULA = []
```

Figure 14. Result of model checking

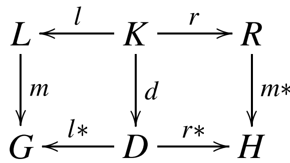

Figure 15. Application of a rule on graph *G*

graphs on the LHS of graph grammar rules, as well as using node and edge types and attributes, can greatly reduce the search space. This is the case with the vast majority of formalisms of interest. It is noted that a possible performance penalty is a small price to pay for explicit, reusable, easy-to-maintain models of transformation. In cases in which performance is a real bottleneck, graph grammars can still be used as an *executable specification* to be used as the starting point for a manual implementation.

The above CAMPaM concepts have been implemented in $AToM^3$ (A Tool for Multi-formalism and Meta-Modeling), in which multiabstraction, multiformalism, and meta-modeling are combined. The power of $AToM^3$ has been demonstrated by modeling the DEVS formalism [51], Petri nets and Statecharts [52], GPSS [53], causal block diagrams [54], and flow diagrams [55].

4. Hybrid Dynamic Systems

To *execute* (simulate) models designed using the sophisticated and domain-specific languages discussed so far, interpreters/compilers are required that translate the high-level modeling constructs into low-level specifications. The simulated behavior can be continuous or of a discrete event nature, and often these two types of behavior are combined, which allows a hybrid execution.

4.1 Combining Executable Formalisms

At the execution level, there are again many different formalisms, especially in the discrete event domain. For example, Petri nets (and their variants such as timed, colored, and stochastic nets) have operational semantics that allow simulation. VHDL [56] allows for simulation, where the event-driven nature of the simulator is of critical importance because of the typically large set of possible events of which only a minor subset is active. DEVS is another language, with operational semantics for which simulators exist. In the continuous domain, differential equations can be simulated using numerical solvers. Continuous behavior generation is often based on discretization in time, and in the control engineering domain, typically straightforward simulation of continuous behavior is applied by implementing some form of a forward integration algorithm.

The continuous and discrete event formalisms are fundamentally different, though. Dedicated continuous-time numerical solvers for differential and algebraic equations, such as used in plant modeling, apply numerical algorithms that are based on continuity assumptions. In addition, such numerical solvers may support implicit model formulations, which leads to conceptually simpler and more elegant models in certain cases [57]. Although the model itself may be simpler, its transformation into a trajectory is more complex, which demonstrates how the *complexity of the model-solver combination* is invariant under behavior-preserving formalism transformations.

In its most general form, execution can be achieved by producing computer code that may even be optimized by weaving the numerical solver code and model execution code together (e.g., [58]). The CAMPaM technologies can be applied to have model interpretation automatically produce highly optimized code that integrates solver and model characteristics.

Dedicated solvers have their advantage, though: they also allow independent selection of an appropriate numerical integration method, depending on the characteristics of the simulation trajectories (e.g., particular types of stiffness). This is not possible when the solver is built into the model of computation.

The combination of continuous behavior with discrete state changes leads to so-called hybrid dynamic systems,[3] which have been investigated extensively, driven by the increasing need for comprehensive controller/plant behavior analysis [59-61]. As such, hybrid dynamic systems are a key technology in the field of CAMPaM. Advances are immediately reflected in the usefulness of higher level CAMPaM notions. Therefore, it is meaningful to give a brief overview of the two basic perspectives. The combined behavior of hybrid dynamic systems introduces issues in many aspects such as modeling, simulation, sensitivity analysis, and optimization [62]. In particular, issues specific to simulation include (1) event detection and location, (2) sequences of discrete transitions, (3) consistent semantics of hybrid dynamic systems formalisms, (4) sensitivity to initial conditions, and (5) sliding mode behavior [63].

4.2 State-Centered Execution Model

A canonical representation of hybrid dynamic systems is in terms of *hybrid automata* [64]. These models combine continuous behavior in certain discrete states with transitions between them. Behavior in each state is then captured by a set of differential equations, while an invariant specifies the allowed values of continuous variables in this state. Transitions between states can be enabled based on continuous variables crossing thresholds. When enabled, they are not enforced to be taken immediately, but they do have to be executed before the invariant would be violated (note

3. To clearly distinguish between hybrid intelligent systems that mix neural nets and fuzzy logic and also to avoid confusion with hybrid vehicles that mix electric motors and combustion engines.

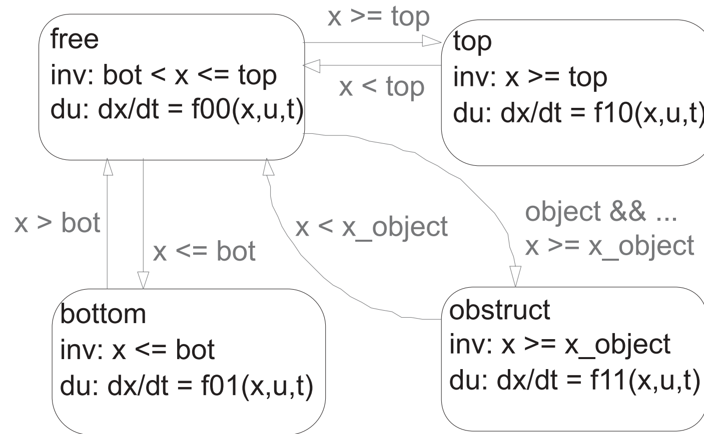

Figure 16. Hybrid automata model of window behavior

that the complexity of the differential equations in each state as well as the invariant may differ between states). A state transition may discontinuously change the values of the variables used in the differential equations and even the set of continuous state variables itself.

To illustrate, consider the state in which the window in the power window example reaches the top of the frame. Four states can be identified:

- *free*, the window moves with only actuation and friction forces acting on it;
- *bottom*, the window is at the bottom of the frame with a large reaction force acting on it;
- *top*, the window is at the top of the frame with a large reaction force acting on it;
- *obstruct*, the window moves between the top of the frame and the bottom with an object stuck between the window and the frame.

This can be modeled by switching the system of ordinary differential equations (ODE) that govern the continuous behavior of the system, illustrated by the hybrid automaton in Figure 16. Here, the transition conditions are given along the transition (e.g., $x < top$), the invariants in a state are labeled "inv:," and the active ODE is labeled "du:" (based on the action *during* the state's active period). When an event occurs, the system moves into a different *mode* of operation. After the mode change, the state variables in the new mode have to be initialized appropriately based on the values in the previous mode; when no explicit function is given in the transition action part, the identity mapping is assumed. Note that transitions between states may cause the complexity of the ODE (i.e., the number of continuous states) to change. For example, when the window reaches the top of the frame, a stiff, dampened spring effect becomes active, which adds a continuous state to the ODE.

The hybrid automata perspective is discrete event centered and provides an explicit representation. This is beneficial to analysis and synthesis activities. However, it suffers from a combinatorial explosion of discrete states when there are many interacting local discrete state changes.

4.3 Equation-Centered Execution Models

An alternative approach relies on a system of guarded differential and algebraic equations. This approach is centered around differential equations (e.g., Modelica [65], MAsim [66], VHDL-AMS [67], χ [68]). Events are generated by continuous variables crossing thresholds, which may enable and disable equations. The different discrete states with continuous behavior are implicit, and invariants that capture the domain of continuous behavior in each state are typically not used. Instead, events have "must-fire" semantics (i.e., an enabled transition is immediately executed).

Using guarded equations, the power window system can be modeled as

$$\begin{aligned} 0 &= \alpha_{top}(v_{window} - v_{deform}) + (1-\alpha_{top})F_{object}, \\ 0 &= C_{object}\dot{F}_{deform} - v_{deform}, \\ 0 &= F_{object} - F_{deform} - R_{object}v_{deform}, \end{aligned} \tag{1}$$

where the mode selection variable α_{top} is determined by the window being at the top of the frame or not.

$$x_{window} \geq x_{top} \Rightarrow \alpha_{top}. \tag{2}$$

Here, the dampened spring parameters are C_{object} to model the spring and R_{object} to model the damping. The rate of deformation of the object is represented by v_{deform} and the corresponding force by F_{deform}. This force is the difference of the total force acting on the object, F_{object}, and the force required to compensate the dissipation, $R_{object}v_{deform}$.

Discontinuous changes in continuous variables are modeled implicitly by activating algebraic constraints that reduce the state space dimension and thus require an instantaneous projection into the new space [69]. For example, if the frame top is not modeled by a dampened

spring, the window velocity is instantly forced to zero when $x_{window} \geq x_{top}$ by replacing equation (1) with

$$0 = \alpha_{top}(v_{window}) + (1 - \alpha_{top})F_{object}. \quad (3)$$

The implicit modeling approach (both in terms of the discrete states as well as the continuous equations) allows a succinct specification of a large number of discrete state changes, but simulation is about the only analysis tool that can handle it. Besides, to perform simulation, the numerical solver has to be extended with additional operations to make the implicit jumps in continuous states explicit.

4.4 Discussion

Whether to use a state-centered or equation-centered approach, again, depends on many factors. State-centered approaches are convenient for analysis and synthesis tasks because they provide an explicit representation. On the other hand, equation-centered approaches provide more modeling power and are often more convenient to use due to their implicit nature. Therefore, the choice of which formalism to use should be made judiciously and be related to the task at hand, similar to the choice of higher level formalisms discussed earlier.

5. Conclusions

The best models are elegant models. This article has introduced the emerging field of CAMPaM, which tries to support this maxim by developing a domain-independent framework for multiparadigm modeling that consists of three dimensions: (1) multiabstraction, (2) multiformalisms, and (3) meta-modeling. Transformations, possibly modeled as graph grammars, are presented as an operator within and between the different dimensions. A classification of hybrid dynamic systems, the underlying execution mechanisms of multiparadigm models has been given.

CAMPaM is a critical enabler for domain-specific modeling and serves to facilitate the use of high-level modeling languages. As languages with high-level, domain-specific modeling elements and constructs become available, the design of applications becomes intuitive for domain experts, while a computational implementation is derived by automatic model transformation. This allows a focus on design issues rather than on implementation issues. Moreover, the implementation does not have to be structured in a human-readable and, more important, comprehensible manner anymore. This in contrast to the present-day situation in which, for example, object-oriented programming techniques are a necessity for humans to implement the complex designs that have become common.

A specific domain in which CAMPaM is increasingly applied is in the field of embedded control systems. The interested reader is referred to a special issue of *IEEE Transactions on Control System Technology* on CAMPaM [70].[4] More about the theory and methodology of CAMPaM can be found in a special issue of *ACM Transactions on Modeling and Computer Simulation* [13] on the same topic.

6. Acknowledgments

The authors wish to acknowledge extensive discussions on the topic of CAMPaM with Juan de Lara.

Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Innovation Grant.

Finally, Adelinde Uhrmacher and Ernie Page are thanked for their efforts in organizing the Dagstuhl seminar on Grand Challenges for Modeling and Simulation.

7. References

[1] Mosterman, Pieter J., Janos Sztipanovits, and Sebastian Engell. 2003. Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*.

[2] Karnopp, D. C., D. L. Margolis, and R. C. Rosenberg. 1990. *Systems dynamics: A unified approach*. 2nd ed. New York: John Wiley.

[3] Harel, David. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8:231-74.

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

[5] van Amerongen, Job. 2000. Mechatronic design. In *The 7th Mechatronics Forum International Conference*, Atlanta, GA.

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

[7] Warmer, Jos, and Anneke Kleppe. 2003. *The object constraint language: Getting your models ready for MDA*. 2nd ed. Reading, MA: Addison-Wesley.

[8] Garlan, David, Robert T. Monroe, and David Wile. 1997. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, Toronto, Ontario, pp. 169-83.

[9] Vestal, Steve. 1994. Software Architecture Workshop. Retrieved from http://www.htc.honeywell.com/projects/dssa/ftp/papers/arch-wkshop

[10] Sztipanovits, Janos, Gabor Karsai, and Hubertus Franke. 1996. Model-integrated program synthesis environment. In *IEEE Symposium on Engineering of Computer Based Systems*, Friedrichshafen, Germany.

[11] Ernst, Johannes. 1996. Data interoperability between CACSD and CASE tools using the CDIF family of standards. In *Proceedings of the 1996 International Symposium on Computer Aided Control System Design*, Dearborn, MI, pp. 346-51.

[12] Mosterman, Pieter, and Hans Vangheluwe. 2000. Computer automated multi paradigm modeling in control system design. In *IEEE International Symposium on Computer-Aided Control System Design*, Anchorage, AK, pp. 65-70.

[13] Mosterman, Pieter J., and Hans Vangheluwe. 2002. Computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation* 12 (4): 1-7.

[14] Vangheluwe, Hans, and Juan de Lara. 2003. Computer automated multi-paradigm modelling: Meta-modelling and graph transformation. In *Winter Simulation Conference*, New Orleans, LA, pp. 595-603.

[15] Fishwick, Paul A. 1991. Heterogeneous decomposition and inter-level coupling for combined modeling. In *1991 Winter Simulation Conference*, Phoenix, AZ, pp. 1120-8.

4. See also http://msdl.cs.mcgill.ca/people/mosterman/campam/ for a number of special sessions on CAMPaM at control-oriented conferences.

[16] Jourdan, M., F. Lagnier, F. Maraninchi, and P. Raymond. 1994. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL)*, Toulouse, France.

[17] Ledeczi, Akos, Greg Nordstrom, Gabor Karsai, Peter Volgyesi, and Miklos Maroti. 2001. On metamodel composition. In *Proceedings of the IEEE International Conference on Control Applications*, Mexico City, Mexico.

[18] Gelsey, Andrew, Mark Schwabacher, and Don Smith. 1998. Using modeling knowledge to guide design space search. *Artificial Intelligence* 101:35-62.

[19] Mann, Heřman. 1996. A versatile modeling and simulation tool for mechatronics control system development. In *1996 IEEE Symposium on Computer Aided Control System Design*, Dearborn, MI, pp. 524-9.

[20] Kamel, M. S., K. S. Ma, and W. H. Enright. 1993. ODEXPERT—an expert system to select numerical solvers for initial value ODE systems. *ACM Transactions on Mathematical Software* 19 (1): 44-62.

[21] Mosterman, Pieter J. 1999. Towards model manipulation for efficient and effective simulation and instructional methods. In *Distributed Modelling and Simulation of Complex Systems for Education, Training and Knowledge Capitalisation*, Eze, France.

[22] Breunese, Arno P. J., Theo J. A. de Vries, Job van Amerongen, and Peter C. Breedveld. 1995. Maximizing impact of automation on modeling and design. In *ASME Dynamic Systems & Control Div. '95*, San Francisco, pp. 421-30.

[23] Harel, David. 1988. On visual formalisms. *Communications of the ACM* 31 (5): 514-30.

[24] Harel, David, and Amnon Naamad. 1996. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5 (4): 293-333.

[25] Huining Feng, Thomas. 2003. An extended semantics for a Statechart Virtual Machine. In *Summer Computer Simulation Conference: Student Workshop*, edited A. Bruzzone and Mhamed Itmi, pp. S147-66. Montréal, Canada: Society for Computer Simulation International (SCS).

[26] Huining Feng, Thomas. 2004. Dcharts, a formalism for modeling and simulation based design of reactive software systems. M.Sc. diss., School of Computer Science, McGill University.

[27] Kohavi, Zvi. 1978. *Switching and finite automata theory*. New York: McGraw-Hill.

[28] Murata, Tadao. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77 (4): 541-80.

[29] Bapty, Ted, Sandeep Neema, Jason Scott, Janos Sztipanovits, and Sameh Asaad. 2000. Model-integrated tools for the design of dynamically reconfigurable systems. Technical Report ISIS-99-01, Vanderbilt University, Nashville, TN.

[30] Kumar, Sanjaya, Devesh Bhatt, Steve Vestal, Bill Wren, John Shackleton, Hazel Shirley, Rashmi Bhatt, John Golusky, Mark Vojta, John Fischer, Steve Crago, Brian Schott, Robert Parker, and Gary Gardner. 1999. ADAPTERS. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, Laurel, MD.

[31] Fisher, Michael. 1999. Zero-latency engineering™. White paper, Aviatis Corp.

[32] Vangheluwe, Hans. 2000. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*, Anchorage, AK, pp. 129-34.

[33] Vangheluwe, Hans, and Ghislain C. Vansteenkiste. 2000. The cellular automata formalism and its relationship to DEVS. In *14th European Simulation Multi-conference (ESM)*, Ghent, Belgium, pp. 800-10.

[34] Davis, John, 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. 1999. Ptolemy II—heterogeneous concurrent modeling and design in Java. Retrieved from http://ptolemy.eecs.berkeley.edu

[35] Atkinson, Colin. 1997. Metamodeling for distributed object environments. In *First International Enterprise Distributed Object ComputingWorkshop (EDOC'97)*, Brisbane, Australia, pp. 90-101.

[36] Geisler, R., M. Klar, and C. Pons. 1998. Dimensions and dichotomy in metamodeling. Technical Report 98-5, TU Berlin, Berlin, Germany.

[37] Engstrom, Eric, and Jonathan Krueger. 2000. A meta-modeler's job is never done: Building and evolving domain-specific tools with DOME. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, Anchorage, AK, pp. 83-88.

[38] Karsai, Gabor, Greg Nordstrom, Akos Ledeczi, and Janos Sztipanovits. 2000. Specifying graphical modeling systems using constraint-based metamodels. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, Anchorage, AK, pp. 89-94.

[39] Stateflow. 2004. *Stateflow user's guide*. Natick, MA: The MathWorks.

[40] Pereira Remelhe, Manuel A., Sebastian Engell, Martin Otter, André Deparade, and Pieter J. Mosterman. 2002. An environment for the integrated modelling of systems with complex continuous and discrete dynamics. In *Modelling, analysis, and design of hybrid systems*, edited by S. Engell, G. Frehse, and E. Schnieder. Berlin: Springer-Verlag.

[41] Flatscher, Rony G. 2002. Metamodeling in EIA/CDIF meta-metamodel and metamodels. *ACM Transactions on Modeling and Computer Simulation* 12 (4)322-342.

[42] Lunze, Jan. 2000. Diagnosis of quantised systems by means of timed discrete-event representations. *Lecture Notes in Computer Science* 1790:258-71.

[43] Preußig, J., O. Stursberg, and S. Kowalewski. 1999. Reachability analysis of a class of switched continuous systems by integrating rectangular approximation and rectangular analysis. *Lecture Notes in Computer Science* 1569:209-22.

[44] Cassandras, Christos G. 1993. *Discrete event systems*. Homewood, IL: Irwin.

[45] de Lara, Juan, and Hans Vangheluwe. 2002. AToM^3: A tool for multi-formalism and meta-modelling. *Lecture Notes in Computer Science* 2306:174-88.

[46] de Lara Jaramillo, Juan, Hans Vangheluwe, and Manuel Alfonseca Moreno. 2003. Using meta-modelling and graph grammars to create modelling environments. In *Electronic notes in theoretical computer science*, vol. 72, edited by Paolo Bottoni and Mark Minas. New York: Elsevier.

[47] Clarke, E. M., and E. A. Emerson. 1981. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science* 131:38-55.

[48] Ehrig, H., G. Engels, H.-J. Kreowski, and G. Rozenberg. 1999. *Handbook of graph grammars and computing by graph transformation: Vol. 2: Applications, languages, and tools*. New York: World Scientific.

[49] Rozenberg, G. 1997. *Handbook of graph grammars and computing by graph transformation*. Vol. 1. New York: World Scientific.

[50] Heckel, R., and A. Wagner. 1995. Ensuring consistency of conditional graph rewriting—a constructive approach. In *Proceedings of SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 2.

[51] Levytskyy, Andriy, Eugène J.H. Kerckhoffs, Ernesto Posse, and Hans Vangheluwe. 2003. Creating DEVS components with the meta-modelling tool AToM^3. In *15th European Simulation Symposium (ESS)*, edited by Alexander Verbraeck and Vlatka Hlupic, pp. 97-103. Delft, the Netherlands: Society for Modeling and Simulation International (SCS).

[52] de Lara, Juan, and Hans Vangheluwe. 2002. Computer aided multi-paradigm modelling to process Petri-nets and statecharts. *Lecture Notes in Computer Science* 2505:239-53.

[53] de Lara, Juan, and Hans Vangheluwe. 2002. Using meta-modelling and graph grammars to process GPSS models. In *16th European*

Simulation Multi-conference (ESM), edited by Hermann Meuth, pp. 100-7. Darmstadt, Germany: Society for Computer Simulation International (SCS).

[54] Posse, Ernesto, Juan de Lara, and Hans Vangheluwe. 2002. Processing causal block diagrams with graph-grammars in $AToM^3$. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, Grenoble, France, pp. 23-34.

[55] de Lara, Juan, and Hans Vangheluwe. 2002. Using $AToM^3$ as a Meta-CASE tool. In *4th International Conference on Enterprise Information Systems (ICEIS)*, Ciudad Real, Spain, pp. 642-9.

[56] IEEE 1076.1 Working Group. 1999. IEEE standard 1076.1-1999. Retrieved from http://www.vhdl.org

[57] Cellier, F. E., H. Elmqvist, and M. Otter. 1996. Modelling from physical principles. In *The control handbook*, edited by W. S. Levine, pp. 99-107. Boca Raton, FL: CRC Press.

[58] Schiela, Anton, and Hans Olsson. 2000. Mixed-mode integration for real-time simulation. In *Proceedings of the First International Modelica 2000 Workshop*, Lund, Sweden, pp. 69-75.

[59] Di Benedetto, Maria Domenica, and Alberto L. Sangiovanni-Vincentelli, eds. 2001. Hybrid systems: Computation and control. *Lecture Notes in Computer Science* 2034.

[60] Lynch, Nancy, and Bruce Krogh, eds. 2000. Hybrid systems: Computation and control. *Lecture Notes in Computer Science* 1790.

[61] Vaandrager, Frits W., and Jan H. van Schuppen, eds. 1999. Hybrid systems: Computation and control. *Lecture Notes in Computer Science* 1569.

[62] Barton, Paul I., and Cha Kun Lee. 2002. Modeling, simulation, sensitivity analysis and optimization of hybrid systems. *ACM Transactions on Modeling and Computer Simulation* 12 (4):256-289.

[63] Mosterman, Pieter J. 1999. An overview of hybrid simulation phenomena and their support by simulation packages. *Lecture Notes in Computer Science* 1569:164-77.

[64] Alur, Rajeev, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science* 736:209-29.

[65] Elmqvist, Hilding, et al. 1999. Modelica^{TM}—a unified object-oriented langauge for physical systems modeling: Language specification. Version 1.3. Retrieved from http://www.modelica.org/

[66] Mosterman, Pieter J. 2001. MAsim—a hybrid dynamic systems simulator. Technical Report DLR-IB-515-01/07, Institute of Robotics and Mechatronics, DLR Oberpfaffenhofen, Wessling, Germany.

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

[68] van Beek, D. A., V. Bos, and J. E. Rooda. 2002. Declaration of unknowns in DAE-based hybrid system specification. *ACM Transactions on Modeling and Computer Simulation* 13 (1):39-61.

[69] Mosterman, Pieter J. 2000. Implicit modeling and simulation of discontinuities in physical system models. In *The 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems*, pp. 35-40.

[70] Engell, Sebastian, and Pieter J. Mosterman, eds. 2004. Computer automated multi-paradigm modeling [Special issue]. *IEEE Transactions on Control System Technology* 12 (2).

Pieter J. Mosterman *is a senior research scientist at The MathWorks, Inc., in Natick, MA. Before, he held a research position at the German Aerospace Center (DLR) in Oberpfaffenhofen. He has a PhD degree in Electrical and Computer Engineering from Vanderbilt University in Nashville, TN, and a MSc degree in Electrical Engineering from the University of Twente, Netherlands. His primary research interests are in computer automated multi-paradigm modeling (CAMPAM) with principal applications in training systems and fault detection, isolation, and reconfiguration. For this, he designed several simulation environments such as the Electronics Laboratory Simulator (nominated for The Computerworld Smithsonian Award), a first version of TRANSCEND, HYBRSIM (a paper on which received the Donald Julius Groen Prize), and MASIM. Specific areas of interest are modeling of physical systems, meta-modeling, and model and formalism transformation in computer aided control system design (CACSD). An important aspect concerns the behavior generation for heterogeneous models, which requires a hybrid dynamic systems approach.*

Dr. Mosterman is the invited session chair for the 2006 IEEE International Symposium on CACSD and co-chaired the 2004 Bellairs CAMPaM Workshop and the 14th International Workshop on Principles of Diagnosis (2003). He is currently a member of the IFAC Technical Committee on CACSD, chair of the IEEE CSS Action Group on Hybrid Dynamic Systems for CACSD, editor of SIMULATION: Transactions of The Society for Modeling and Simulation International for the area of Mechatronics, and associate editor of IEEE TRANSACTIONS ON CONTROL SYSTEM TECHNOLOGY and the International Journal of Applied Intelligence. He was also guest editor of special issues of ACM TRANSACTIONS ON MODELING AND COMPUTER SIMULATION and IEEE TRANSACTIONS ON CONTROL SYSTEM TECHNOLOGY on the topic of CAMPAM.

Hans Vangheluwe *Hans Vangheluwe is an Assistant Professor in the School of Computer Science at McGill University, Montreal, Canada. He holds a DSc degree, as well as an MSc in Computer Science, and BSc degrees in Theoretical Physics and Education, all from Ghent University in Belgium. He has been a Research Fellow at the Centre de Recherche Informatique de Montreal, Canada, the Concurrent Engineering Research Center, WVU, Morgantown, WV, USA, at the Delft University of Technology, The Netherlands, and at the Supercomputing and Education Research Center of the Indian Institute of Science (IISc), Bangalore, India. At McGill University, he teaches Modelling and Simulation as well as Software Design. He also heads the Modelling and Simulation and Design (MSDL) research lab. He has been the Principal Investigator of a number of research projects focused on the development of a multi-formalism theory for Modelling and Simulation. Some of this work has led to the WEST++ tool, which was commercialized for use in the design and optimization of bioactivated sludge Waste Water Treatment Plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "Simulation in Europe", a founding member of the Modelica Design Team, and an advisor to the Flemish Institute for the Promotion of Scientific-Technological Research in Industry (IWT), as well as to the European Commission's 5th Framework programme. He is an Associate Editor for the journal Simulation: Transactions of the Society for Modeling and Computer Simulation. His current interests are in domain-specific modelling and simulation. The MSDL's tool AToM3 (A Tool for Multi-formalism and Meta-Modelling) uses meta-modelling and graph grammars to specify and generate domain-specific environments.*