Meta-Modelling Hybrid Formalisms

Simon Lacoste-Julien, Hans Vangheluwe, Juan de Lara, and Pieter J. Mosterman

Abstract—This article demonstrates how meta-modelling can simplify the construction of domain- and formalism-specific modelling environments. Using AToM³ (A Tool for Multi-formalism and Meta-Modelling developed at McGill University), a model is constructed of a hybrid formalism, HS, that combines Event Scheduling constructs with Ordinary Differential Equations. From this specification, an HS-specific visual modelling environment is synthesized. For the purpose of this demonstration, a simple hybrid model of a bouncing ball is modelled in this environment. It is envisioned that the future of modelling and simulation in general, and more specifically in hybrid dynamic systems design lies in domain-specific Computer Automated Multi-Paradigm Modelling (CAMPaM) which combines multi-abstraction, multi-formalism, and meta-modelling. The small example presented in this article demonstrates the feasibility of this approach.

I. INTRODUCTION

The ability to model complex physical as well as control systems and to experiment with them using simulation can be greatly enhanced when an appropriate, possibly visual, modelling and simulation environment is available. Such an environment will only be useful if it supports the most appropriate modelling formalism for the task at hand. Appropriateness is context dependent and depends on the goals of the user of the tool as well as on the information available about the system. In particular, appropriateness of a formalism depends on the type of system under study, on the aspects of the structure and behaviour of the system one is interested in, and on the kind of queries one wishes to make regarding the system.

Hybrid models combine discrete (time/event) and continuous model constructs in a single model. The reasons for this combination vary. Often, certain aspects of a system’s continuous behaviour can be abstracted and represented as an instantaneous discrete event as they happen on a very small time scale compared to the rest of the system’s behaviour. A side-effect of such abstraction is an improvement in simulation performance. A discussion of different types of physically meaningful abstraction is found in [1].

A plethora of discrete-time, discrete-event as well as continuous modelling formalisms exist. This allows for a large number of possible combination (hybrid) formalisms.

Depending on the modeller’s needs, a modelling and simulation tool should support the most appropriate combination. As modellers’ needs may vary widely, it is desirable to support many different combinations. Constructing one tool that supports all formalism combinations is not feasible nor efficient.

The first section of this article describes a particular hybrid formalism that combines Event-Scheduling (ES) with Ordinary Differential Equations (ODEs). The Event-Scheduling formalism [2] was chosen as it allows describing queueing problems elegantly. A visual syntax for this modelling formalism is presented. The formalism’s syntax, its meaning, as well as a prototype simulator for it implemented in Python1 are introduced by means of the “bouncing ball” example.

In the second section, it is shown how meta-modelling can be used to describe the (abstract as well as concrete visual) syntax of the hybrid formalism. Meta-modelling is the explicit modelling of a class of models in an appropriate formalism – Entity-Relationship Diagrams in this case. Using AToM³ to encode this meta-model allows the automatic generation of a visual modelling environment specific to the described formalism.

The third section presents the notion of model transformation. Different types of transformation are possible. Simulation consists of a series of transformations that modify time and the state. Simplification transformations modify the structure of the model. Code-generating transformations produce a textual representation of the model suitable for processing by an appropriate solver/simulator. Graph grammar models that allow for declarative modelling of model transformations are briefly introduced. The code generator for the modelling and simulation environment is modelled as a graph grammar.

II. A HYBRID FORMALISM: HS=ES+ODE

To set the stage for the subsequent presentation of meta-modelling of a domain-specific visual modelling environment, a visual formalism (named HS) is introduced, combining Event-Scheduling with Ordinary Differential Equations. To introduce the formalism, Figure 1 models a bouncing ball that can get stuck on the ground after a certain time. Two modes are used: when the ball is in free fall (mode Free_Ball) and when it is stuck (mode Stuck). When in free fall, the ODE describing the ball’s behaviour is simply \[ \frac{dv}{dt} = -g \] and \[ \frac{dy}{dt} = v \] where \( y \) is the height of the ball; \( v \) its speed; and \( g \) the gravity constant. When in the Stuck mode, the ball is

1http://www.python.org
at rest: \( \frac{dv}{dt} = 0 \). The first event in the model is the Initialize_Model event, which is labelled in Figure 1 as the START_EVENT. In any HS model, there needs to be exactly one START EVENT which will indicate where to start the simulation. This event is typically used to initialize the state variables in the model; here, the action code in the event handler states \( y = y_0; \ v = v_0 \) so the height and velocity of the ball are initialized to the value given by parameters \( y_0 \) and \( v_0 \), which are defined in a global attribute for the model (namely, in Parameters_List - see Figure 4 for a visual environment defining those attributes). There is a Mode_Transition going from the start event to the Free_Ball mode indicating which mode is used as the initial continuous mode for the simulation. In this mode, the behaviour of the ball is governed by the ODE

\[
\begin{align*}
\frac{dy}{dt} &= v \\
\frac{dv}{dt} &= -g
\end{align*}
\]

For correct simulation of the model, it is mandatory that the START_EVENT be linked to a (possibly empty) continuous mode.

The Check_Collision State_Guard connects the Free_Ball mode to the Collision Event_HS, with the meaning that when a zero is detected in the monitoring function given in the State_Guard attributes (here, \( y \)) in the correct direction (here, from + to -), then the given Event_HS is triggered. That is, its action code, mode transition and schedule events are executed. In this case, Collision contains only \( v = -k \cdot v \) as action code, which reverses the speed of the ball with a coefficient of restitution \( k \) (smaller than 1 for inelastic collision; equal to 1 for elastic collision; and greater than 1 for superelastic collision). The Mode Transition to the Free_Ball mode indicates that after this (instantaneous) event, the system is put back in the Free_Ball continuous mode. Note there is also a Schedule relationship that connects the Collision event to the getStuck event. The condition for the scheduling to occur is shown graphically with \( \text{IF } t > t_{\text{stuck}} \), i.e., the event getStuck will be scheduled after 0 time-units (because of the AFTER 0) if \( t \) (time) is greater than the parameter \( t_{\text{stuck}} \). The event getStuck simply sets the speed to 0 and then moves the system in the continuous Stuck mode. In this mode, the behaviour of the ball is governed by the ODE

\[
\begin{align*}
\frac{dy}{dt} &= v \\
\frac{dv}{dt} &= 0
\end{align*}
\]

This hybrid model defines a piecewise continuous behaviour. The model in Figure 1 is automatically compiled by AToM³ (see the next section) into BouncingBallStuck.py, a representation suitable for numerical simulation in a Python Hybrid Simulator developed at McGill University [3]. The simple user interface of the simulator is shown at the top of Figure 2. A model file (generated from the modelling environment) can be loaded. This displays the model parameters and variables. Model parameters as well as the most appropriate numerical solver may be selected by the user. The result of the simulation is shown at the bottom of Figure 2.
III. META-MODELLING IN AToM³

A. Domain/Formalism-Specific Modelling

Domain- and formalism-specific modelling and simulation environments have the potential to greatly improve productivity. They are able to exploit features inherent to a specific domain or formalism. This may for example enable specific analysis techniques or the synthesis of efficient code. They also maximally constrain the users, allowing them, by construction, to only build syntactically and (for as far as this can be statically checked) semantically correct models. Furthermore, the specific, possibly visual syntax used in domain-specific modelling environments matches the users’ mental model of the problem domain.

The time required to construct such domain/formalism-specific modelling and simulation environments can be prohibitive. Thus, rather than using such specific environments, generic environments are typically used. Such generic environments are necessarily a compromise.

B. Meta-Modelling

A modelling language/formalism $L$ is (by definition) the set of all valid models in the language/formalism. The more specific a language/formalism is, the smaller the set will be. Note that a modelling language may contain an infinite (but countable) number of models. The models in the language may have a textual concrete syntax, a visual concrete syntax, or a combination of the two.

Meta-modelling [4], [5], [6] is the explicit modelling of a class of models, i.e., of a modelling language. A meta-model $M_L$ of a modelling language $L$ is a model in its own right which specifies concisely and precisely which models $m$ are elements of $L$.

Modelling environments based on meta-modelling will either check, by means of a meta-model $M_L$ whether a given model $m$ is in $L$, or they will constrain the modeller during the incremental model construction process such that only elements of $L$ 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 $L$ unreachable through incremental construction.

The advantages of meta-modelling are numerous. Firstly, an explicit model of a modelling 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 modelling 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 modelling language is apparent. Above all, with an appropriate meta-modelling tool, modifying a meta-model and subsequently generating a possibly visual modelling tool is orders of magnitude faster than developing such a tool by hand. Ultimately, even the transformations between meta-models (of variants of modelling languages) may be explicitly modelled in the form of graph grammar models.

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

Though an arbitrary number of meta-levels are possible in principle; in practice, some modelling languages/formalisms such as Entity-Relationship diagrams (ER) and UML Class Diagrams are expressive enough to be expressed in themselves. That is, the meta-model of such a language $L$ is a model in language $L$. From the implementation point of view, this allows one to bootstrap a meta-modelling environment.

Note that in the above presentation of meta-modelling, a meta-model must specify all elements of a modelling language. Traditionally, the concrete syntax of textual (programming) languages has been specified in the form of grammars. Similarly, grammars may be used to specify a visual concrete syntax [7]. Often, a distinction is made between concrete and abstract syntax. The latter only specifies the core structure of models. What goes into the abstract syntax (and what not) is typically determined by what is needed to describe model semantics. This separation between concrete and abstract syntax allows one (i) to specify model semantics once for a whole class of languages, in terms of abstract syntax and (ii) to craft a variety of concrete textual and visual syntaxes onto a single abstract syntax.

Model semantics can be expressed in a variety of ways. At the core of all specification lies model transformation. If model transformations are modelled explicitly (by means of graph grammars for example), model semantics too can be expressed inside the (meta-)modelling framework.

C. Modelling HS in AToM³

The above meta-modelling concept has been implemented in AToM³. A Tool for Multi-formalism and Meta-modelling. AToM³ is a vehicle for Computer Automated Multi-Paradigm Modelling (CAMPaM) [8], [9], [10] in which multi-abstraction, multi-formalism, and metamodelling are combined. The design of AToM³ has been described in [11], [12]. The power of AToM³ has been demonstrated by meta-modelling the DEVs formalism [13], Petri Nets and Statecharts [14], GPSS [15], Causal Block Diagrams [16], and flow diagrams [17].

Figure 3 shows the model of the HS formalism (the meta-model of HS models) in AToM³. In the top left corner of the main window we notice that the Entity-Relationship (ER) formalism is used to model the HS formalism. This means that only valid ER models will be accepted by the tool. Only two icons appear: Entity (a rectangle) and Relationship (a diamond). The HS model is hence composed of only these two constructs as well as of connections between
In particular, one entity, Mode, is used to model continuous modes whereas another, Event_HS, is used to model discrete events. The relationships between the entities (Mode_Transition, State_Guard, and Schedule) together with their cardinalities determine which connections between instances of the entities are valid. To check validity, a homomorphism between a model and its meta-model must be checked: instances of entities must be connected by instances of appropriate relationships. Note how in Figure 3, both at a global (model) level and at the
level of individual entities and relationships, typed attributes can be specified.

Up to now, only abstract syntax has been specified. In AToM³, concrete visual syntax is specified as shown in the bottom of the figure for the Mode and Event_HS entities. Note how entity attributes may be rendered in the concrete syntax.

From the above specification of the HS formalism (the meta-model), a HS-specific visual modelling environment is compiled. The environment was shown in Figure 1; Figure 4 shows examples of its dialog boxes. The environment only allows the creation of valid HS models. With similar forms in which the meta-model attributes were typed, the compiled modelling environment can present the user with appropriate forms for the user to enter the attributes in.

IV. MODEL TRANSFORMATION

The transformation of models is a crucial element in model-based endeavours. As models, meta-models and meta-meta-models are all in essence attributed, typed graphs, we can transform them by means of graph rewriting. The rewriting may be specified in the form of graph grammar [18] models. These are a generalization, for graphs, of Chomsky grammars. They are composed of rules. Each rule consists of left-hand side (LHS) and right-hand side (RHS) graphs. Rules are evaluated against an input graph, called the host graph. If a matching is found between the LHS of a rule and a sub-graph of the host graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to determine the order in which rules are checked. In AToM³ for example, rules are ordered according to a user-assigned priority, and are checked from higher to lower priority. After a rule matching and subsequent application, the graph rewriting system starts the search again. The graph grammar execution ends when no more matching rules are found.

Three kinds of transformations of models are 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-modelling, the Abstract Syntax Graph of the textual formalism (that is, 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 the 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 subgraphs 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, re-usable, easy to maintain models of transformation. In cases where 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. In the case of the HS formalism, a graph grammar is used to specify the mapping of a HS model onto a representation suitable for numerical simulation by a Python Hybrid Simulator that was developed previously [3].

V. CONCLUSION

This paper demonstrated how meta-modelling can make the construction of domain/formalism specific modelling environments straightforward. Using AToM³, a model of a simple, rather contrived HS formalism was constructed, which combines Event Scheduling constructs with Ordinary Differential Equations. From this specification, an HS-specific visual modelling environment was synthesized. For the purpose of this demonstration, a particularly simple hybrid model of a bouncing ball was designed in this environment. It is envisioned that the future of modelling and simulation in general, and more specifically in hybrid systems design lies in domain-specific Computer Automated Multi-Paradigm Modelling (CAMPaM) which combines multi-abstraction, multi-formalism and meta-modelling. The small example presented in this article demonstrates the feasibility of this approach. Another hybrid systems example of CAM-PaM, with focus on model transformation can be found in [19] and [20].

REFERENCES

Fig. 4. Editing the bouncing ball model.