

# Bridging Engineering and Formal Modeling: WebGME and Formula Integration

Tamas Kecskes   Qishen Zhang   Janos Sztipanovits

Department of EECS, Vanderbilt University, Nashville, TN

{*tamas.kecskes, qishen.zhang, janos.sztipanovits*}@vanderbilt.edu

**Abstract**—Emergence of heterogeneous engineering domains that cross disciplinary boundaries lead to design flows that span multiple Domain Specific Modeling Languages (DSML). Analyzing system level behavior and pursuing cross-domain trade-offs requires the semantic integration of modeling domains. Driven by the specific needs of and our experience with design automation tool suites for Cyber-Physical Systems (CPS), this paper focuses on using model integration languages as a flexible way for modeling cross-domain interactions. The primary challenge in specifying and supporting model integration languages is that both rapid evolvability and semantic precision are required. This challenge is mapped into a meta-level model integration problem, the integration of a meta-modeling language used for specifying DSMLs and configuring the underlying meta-programmable modeling tool WebGME, and a formal framework that uses algebraic data types and Constraint Logic Programming - Formula - devoted to formally specifying the semantics of DSMLs and model transformations. The primary contribution of the paper is the deep semantic integration of WebGME and Formula that keep the engineering view of an evolving model integration language and its formal representation tightly synchronized.

**Index Terms**—WebGME, Formula, DSML, Model Transformation, Constraint Programming.

## 1. Introduction

Modeling tools are key enablers of model-driven engineering. They are responsible for offering intuitive engineering interface (usually graphical) for model developers and they provide a range of services supporting safe model engineering practices including composing/decomposing, visualizing, modifying, checking well-formedness, versioning, and storing large models. Adaption of model-driven methods in new cross-disciplinary fields such as Cyber-Physical Systems (CPS) challenges modeling tool developers with heterogeneity: design flows require the integration of multi-physics, multi-abstraction and multi-fidelity models expressed in a rich set of domain-specific modeling languages (DSML). Since design flows and the tools involved in their implementation change, the suite of relevant modeling languages is not static; they evolve continuously. Our approach to manage model heterogeneity has been the introduction of *model integration languages* [1] that

are restricted to modeling language constructs limited to modeling the interactions among different and changing modeling domains. The cost of introducing easy-to-evolve model integration languages - that are themselves DSMLs - is the requirement for explicit representation of their formal semantics, which is necessary to preserve the semantic integrity of design flows [2].

In our previous work, developing the OpenMETA design automation tool suite for DARPA's Adaptive Vehicle Make program [3], we addressed these needs by pursuing and coordinating two parallel paths in model integration. As a first step, we specified and continuously evolved the CyPhyML model integration language focusing on the integrated OpenMETA design flow targeting ground vehicle design [3]. Since the OpenMETA design flow extended to multiple physical and cyber domains, CyPhyML itself proved to be a complex DSML requiring the use of our meta-programmable modeling tool, WebGME[4]. To satisfy the need for representing and evolving the formal semantics of CyPhyML, we developed the OpenMETA Semantic Backplane[2]. It provided a formal representation of the interacting physical and cyber domains. Although the two-pronged approach satisfied the basic needs, the challenge of synchronizing the CyPhyML and Formula models, and meta-models decreased the benefits of the Semantic Backplane.

In this paper we discuss our recent work on establishing a deep integration of WebGME and Formula. Deep integration means keeping the two representations - the CyPhyML meta-models and models in WebGME and their FORMULA equivalent - fully synchronized and providing seamless access to complementary services of the tools. WebGME services include [4]:

- meta-programmability with prototypal inheritance that allows smooth language integration and evolution,
- graphical concrete syntax that is highly customizable,
- multiple, well defined APIs for model interpretation and tool integration,
- version control with branch support to allow modeling in large, and
- collaborative, distributed modeling via web-interface.

FORMULA services include:

- formal representation of structural semantics of modeling languages [5] as strongly-typed, open-World logic programs (OLP) [6] offering specifications that are highly declarative and executable, they can express static, dynamic, and transformation semantics of DSMLs,
- program synthesis and automated reasoning enabled by the symbolic execution of logic programs into quantifier-free sub-problems, which are dispatched to the state-of-the-art SMT solver Z3 [7],
- modular reuse of DSMLs via the composition of OLPs in a strong category theoretic sense [8].

The primary contribution of this paper is the semantic integration between the tools, detailing the relationship between WebGME meta-models and Formula domain specifications. The following sections presents the integrated use of WebGME and Formula services using constraint specification as example.

## 2. Modeling in WebGME

WebGME[4] is the next generation of Vanderbilt’s Generic Modeling Environment(GME) [9] providing many newly designed features such as web-based deployment, version control, real-time collaborative editing and prototypical inheritance to add more scalability and extensibility for large, real world applications. WebGME is a response to the limitations of GME uncovered by the widespread application of our model-integrated computing (MIC) tools.

Although the WebGME advanced the modeling capabilities of GME and provided a highly customizable and meta-programmable framework; it still lacks an expressive and easy-to-use platform for well-formedness checking. In WebGME, the models could be checked with script defined in JavaScript language, but for model-developers and engineers its use is cumbersome and lacks expressiveness which makes it hard to maintain. Elimination of this gap was one of the key motivations for the deeper integration between WebGME and Formula.

## 3. Formal Specification of Modeling Languages in Formula

Microsoft Formula[10] is a constraint logic programming tool developed by Jackson at Vanderbilt and later at Microsoft Research [5]. Formula is based on algebraic data types and first-order logic with fixed-point semantics. It has found many application in Model-Based Engineering such as reasoning about meta-modeling [11] or finding specification errors by constraints [12].

The theoretical foundation for Formula reasoning is based on Non-recursive Horn Domain (a well-known decidable subset of first-order logic) to reason about meta-model as mentioned in a previous paper[13]. Formula is also capable of generating automatic proofs by the help of the

state-of-the-art satisfiability modulo theories (SMT) solver Z3 [14].

In summary, the Formula tool complements WebGME with a range of essential services: verifying if domain constraints are consistent (the domain is not empty), completing partially defined models that are part of the domain, checking well-formedness of models, formally specifying (and executing) model transformations, and formally composing domains.

## 4. Integration Architecture

To provide a seamless and tight integration that combines the modeling power of WebGME with the reasoning engine of Formula, we have created an integration architecture presented in Figure 1. As it is shown, the WebGME client interface has a single communication channel towards the server, that helps in providing a unified platform where the user is able to harness the capabilities of both tools.

To keep the WebGME project and its Formula representation synchronized, the system uses the *webhook* feature of WebGME. It generates a notification towards a configured machine - in our case the Formula machine - at every event of interest. For every new commit in the WebGME model store, the Formula machine will receive a request to make the model translation and constraint evaluation. After finishing, it will store the results in a separate database so they can be queried later as well. With the help of an additional *middleware*, the Formula editor is capable of continuously querying the results from the server and provide it to the user - whether it is the evaluation of constraints or some syntax error.

Another aspect of the integration is that the WebGME meta-model represented as a Formula domain is always available in a synchronous manner. Modifications to the meta-model are immediately reflected as Formula *domain* specifications and can be extended with Formula constraints without interaction with server.

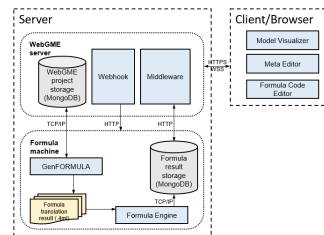
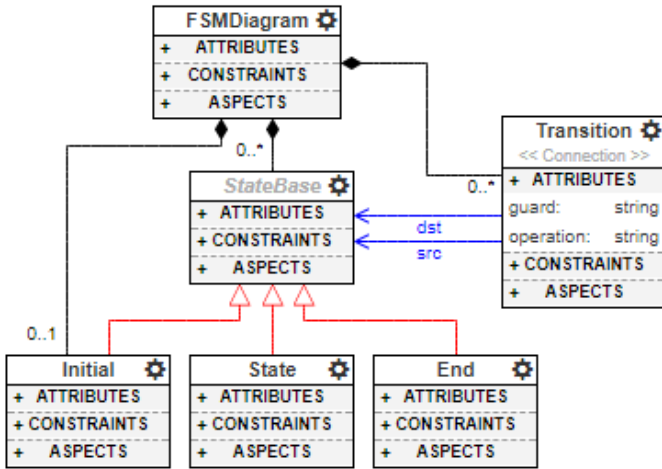


Figure 1: Integration Architecture view of the WebGME Formula integration

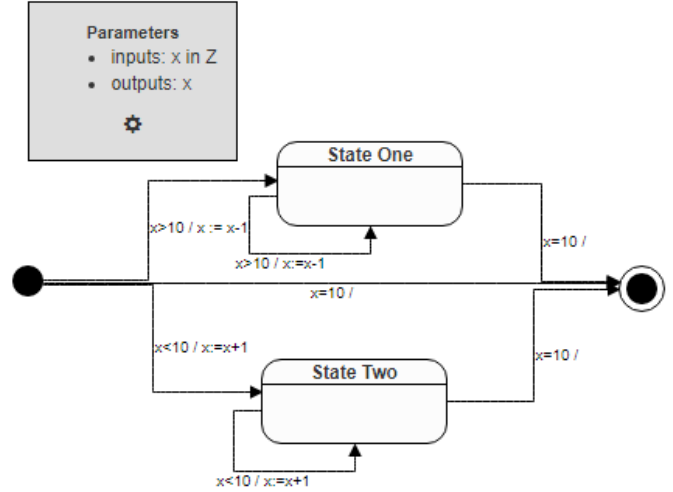
However, the constraint evaluation and syntax checking requires server interaction with reasoning in Formula engine.

## 5. Semantic Integration Between WebGME and Formula

The cornerstone of WebGME and Formula integration is the transformation of WebGME metamodels and models into Formula domain and instance representations. We explain this translation in two steps. First, we discuss a simple example to demonstrate the two alternative representations, and second we provide a simplified version of the model transformation rules.



(a) FSM Meta-model in WebGME



(b) Up-Down Counter model in WebGME

```

domain FiniteStateMachines {
  Transition ::= new (id: String,
    parent: any FSMDiagramTYPE + {NULL},
    attributes: any Attr_Transition,
    pointers: any Ptr_Transition).
  Attr_Transition ::= new (guard: String,
    name: String,
    operation: String).
  Ptr_Transition ::= new (base: any FCOTYPE + {NULL},
    dst: any StateBaseTYPE + {NULL},
    src: any StateBaseTYPE + {NULL}).

  Initial ::= new (id: String,
    parent: any FSMDiagramTYPE + {NULL},
    attributes: any Attr_Initial,
    pointers: any Ptr_Initial).
  Attr_Initial ::= new (name: String).
  Ptr_Initial ::= new (base: any FCOTYPE + {NULL}).

  StateBase ::= new (id: String,
    parent: any FSMDiagramTYPE + {NULL},
    attributes: any Attr_StateBase,
    pointers: any Ptr_StateBase).
  Attr_StateBase ::= new (name: String).
  Ptr_StateBase ::= new (base: any FCOTYPE + {NULL}).

  State ::= new (id: String,
    parent: any FSMDiagramTYPE + {NULL},
    attributes: any Attr_State,
    pointers: any Ptr_State).
  Attr_State ::= new (name: String).
  Ptr_State ::= new (base: any FCOTYPE + {NULL}).

  End ::= new (id: String,
    parent: any FSMDiagramTYPE + {NULL},
    attributes: any Attr_End,
    pointers: any Ptr_End).
  Attr_End ::= new (name: String).
  Ptr_End ::= new (base: any FCOTYPE + {NULL}).

  FSMDiagram ::= new (id: String,
    parent: any {NULL},
    attributes: any Attr_FSMDiagram,
    pointers: any Ptr_FSMDiagram).
  Attr_FSMDiagram ::= new (name: String).
  Ptr_FSMDiagram ::=
    new (base: any FCOTYPE + {NULL}).
}
  
```

(c) Domain in Formula

```

model M of FiniteStateMachines {
  TenMachine_attr is Attr_FSMDiagram("TenMachine").
  TenMachine_ptr is Ptr_FSMDiagram(NULL).
  TenMachine is FSMDiagram("TenMachine",
    NULL, TenMachine_attr, TenMachine_ptr).
  Initial_attr is Attr_Initial("Initial").
  Initial_ptr is Ptr_Initial(NULL).
  Initial is Initial("Initial",
    TenMachine, Initial_attr, Initial_ptr).
  End_attr is Attr_End("End").
  End_ptr is Ptr_End(NULL).
  End is End("End", TenMachine, End_attr, End_ptr).
  State_One_attr is Attr_State("State One").
  State_One_ptr is Ptr_State(NULL).
  State_One is State("State One",
    TenMachine, State_One_attr, State_One_ptr).
  State_Two_attr is Attr_State("State Two").
  State_Two_ptr is Ptr_State(NULL).
  State_Two is State("State Two",
    TenMachine, State_Two_attr, State_Two_ptr).
  Tr_I_to_E_attr is
    Attr_Transition("x=10", "Transition", "").
  Tr_I_to_E_ptr is Ptr_Transition(NULL, Initial, End).
  Tr_I_to_E is Transition("Tr_I_to_E",
    TenMachine, Tr_I_to_E_attr, Tr_I_to_E_ptr).
  Tr_I_to_1_attr is
    Attr_Transition("x>10", "Transition", "x:=x-1").
  Tr_I_to_1_ptr is Ptr_Transition(NULL, Initial, State_One).
  Tr_I_to_1 is Transition("Tr_I_to_1",
    TenMachine, Tr_I_to_1_attr, Tr_I_to_1_ptr).

  Tr_I_to_2_attr is
    Attr_Transition("x<10", "Transition", "x:=x+1").
  Tr_I_to_2_ptr is Ptr_Transition(NULL, Initial, State_Two).
  Tr_I_to_2 is Transition("Tr_I_to_2",
    TenMachine, Tr_I_to_2_attr, Tr_I_to_2_ptr).

  Tr_1_to_1_attr is
    Attr_Transition("x>10", "Transition", "x:=x-1").
  Tr_1_to_1_ptr is Ptr_Transition(NULL, State_One, State_One).
  Tr_1_to_1 is Transition("Tr_1_to_1",
    TenMachine, Tr_1_to_1_attr, Tr_1_to_1_ptr).

  Tr_1_to_E_attr is Attr_Transition("x=10", "Transition", "").
  Tr_1_to_E_ptr is Ptr_Transition(NULL, State_One, End).
  Tr_1_to_E is Transition("Tr_1_to_E",
    TenMachine, Tr_1_to_E_attr, Tr_1_to_E_ptr).

  Tr_2_to_2_attr is
    Attr_Transition("x<10", "Transition", "x:=x+1").
  Tr_2_to_2_ptr is Ptr_Transition(NULL, State_Two, State_Two).
  Tr_2_to_2 is Transition("Tr_2_to_2",
    TenMachine, Tr_2_to_2_attr, Tr_2_to_2_ptr).

  Tr_2_to_E_attr is Attr_Transition("x=10", "Transition", "").
  Tr_2_to_E_ptr is Ptr_Transition(NULL, State_Two, End).
  Tr_2_to_E is Transition("Tr_2_to_E",
    TenMachine, Tr_2_to_E_attr, Tr_2_to_E_ptr).
}
  
```

(d) Model in Formula

Figure 2: Meta-Model and Model representation in WebGME and Formula




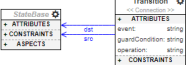
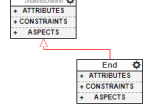
Concept description	WebGME (Meta) representation	Formula translation
Component		<code>FSMDiagram ::= new (id: String, parent: any {NULL}, attributes: any Attr__FSMDiagram, pointers: any Ptr__FSMDiagram).</code>
Containment		<code>StateBase ::= new (...parent: any FSMDiagramTYPE + {NULL},...).</code>
Attribute		<code>Transition ::= new (...attributes: any Attr__Transition...).</code> <code>Attr__Transition ::= new (guard: String, name: String, operation: String).</code>
Pointer (one to one association)		<code>Transition ::= new (...pointers: any Ptr__Transition).</code> <code>Ptr__Transition ::= new (...dst: any StateBaseTYPE + {NULL}, src: any StateBaseTYPE + {NULL}).</code>
Inheritance		<code>StateBase ::= new (...).</code> <code>End ::= new (...).</code> <code>StateBaseTYPE ::= StateBase + ... + End.</code>

Figure 3: Subset of translation rules of WebGME Meta concepts into Formula domain constructs

## 5.1. Modeling Example

Finite State Machine (FSM) is used to demonstrate how WebGME meta-models are represented as Formula domains. The FSM modeling language defined as WebGME meta-model is shown in Figure 2a. The meta-modeling language (WebGME META) adopts a UML-like concrete syntax, which is detailed in [4]. A specific FSM instance using WebGME graphical interface is shown in Figure 2b. The concrete syntax is configurable as expected from meta-programmable modeling tools. The example shows a simple up-down counter that - given any  $x$  integer input value and an initial state - will check the value of input, if its value is equal to ten, a transition is triggered to move from initial state to end state denoting the goal is reached, otherwise it will jump to either State One or State Two and keep incrementing or decrementing  $x$  until it reaches ten and transfer to end state. The equivalent Formula representation are shown in Figures 2c and 2d.

The definition on Figure 2a, still allows the creation of malformed models. To avoid it, we need to introduce domain constraints - so the user can evade flaws like having states with the same name or not having an initial state at all. From tool design point of view, Formula has its clear advantages by allowing consistency check, model synthesis, and well-formedness checking. The seamless use of Formula's constraint language is enabled by the tight integration of the two tools and will be described in Section 6.

It is important to note that we leave out from this

discussion the formal specification of *behavioral semantics*. Formula supports this by its constructs for specifying model transformations. The various ways to use Formula for specifying behavioral semantics in operational or denotation style is discussed in [2].

## 5.2. Generation of Formula Domains and Models

A WebGME plugin, GenFORMULA makes the translation by using the Core API functions of WebGME. It creates a Formula domain by traversing the meta-concepts of the project. By following the rules presented in Figure 3, every Class definition is translated into three constructors in Formula. `Attr__Class` is a tuple for the available attributes, `Ptr__Class` couples the pointer definitions, and `Class(id, parent, attributes, pointers)` tuple combines the other two and adds the containment representation with the `parent` field. The plugin also defines `ClassTYPE` set, that captures the inheritance among meta-concepts of WebGME. Inheritance among models and model elements are kept in the `base` pointer definition. Finally, it specifies some helper constructs `GMENode`, `GMEContainment` and `GMEInheritance` to represent all nodes, their containment relation and inheritance relation. The user defined constraints are then added without modification. This step finalizes the Formula domain. Finally the procedure traverses the whole containment hierarchy in the WebGME project. For every node, it gathers the necessary

values with the help of the Core API, and translate them into instances in Formula.

The resulting formula file is then processed with the help of the Formula engine to get the constraint evaluations and syntax check. Being automated, the result is available after every change, so the user will be notified at the place of error.

## 6. Constraints

The constraints play important role in specifying the structural semantics of modeling languages. They restrict structural characteristics, properties and relations in models such that all instances of a modeling domain are semantically well-formed. However, the built-in constraints in WebGME's meta-modeling language are quite limited and do not support the specification and enforcement of complex constraints. The only available solution is the traversal of models using JavaScript plugin and check if they satisfy the constraints. This method is error-prone, do not provide declarative representation for the constraints to reason about their properties (e.g. consistency) and make the use of advanced service such as model synthesis impossible. Integration of WebGME with Formula eliminates these drawbacks.

### 6.1. Constraint Example for FSM Domain

In our examples shown in Figure 4, we collected a few important constraints regarding the FSM domain. These well-formedness rules are complex and cannot be expressed with the graphical suite of meta-modeling of WebGME.

For example, it is required in any FSM, that all transitions should have valid states as endpoints. The pointer concept of WebGME doesn't specify strict requirement regarding the target so it can be NULL - as in many applications that would be perfectly fine. To eliminate this gap, we define the `noSourceTr(t)` and `noDestinationTr(t)` rules for every transition. Then we combine them to get the `danglingTr(t)` rule which can be used to finally get global `NoDangling` constraint. Overall, many complicated constraints can be easily written in logic programming style Formula Language and evaluated in Formula engine.

### 6.2. Benefits of Using Formula Constraints in WebGME

Similar works can be found that enable adding visual or textual constraints to general modeling. For example, the Object Constraint Language (OCL) [15], Semantics Of Business Vocabulary And Rules (SBVR) and Extensible Visual Constraint Language (EVCL) [16] is widely used in the UML context. Constraint Languages like OCL have the expressiveness to describe complex constraints but they lack model transformation and model completion features that are integral to our needs. Other tools like EVCL [16] generate actual JavaScript code to check visually defined constraints. Consequently, it's not flexible enough and can

```
noSourceTr ::= new(t:TransitionTYPE).
noSourceTr(t) :- t is TransitionTYPE, k = t.parent,
k is FSMDiagramTYPE, t.pointers.src = NULL.
noDestinationTr ::= new(t:TransitionTYPE).
noDestinationTr(t) :- t is TransitionTYPE, k = t.parent,
k is FSMDiagramTYPE, t.pointers.dst = NULL.
danglingTr ::= new(t:TransitionTYPE).
danglingTr(t) :- noSourceTr(t) ; noDestinationTr(t).
HasDangling :- t is TransitionTYPE, danglingTr(t).
NoDangling :- no HasDangling.
NotExactlyOneInitState :- p is FSMDiagramTYPE,
t is FSMDiagramTYPE, p.pointers.base = t,
count({s | s is Initial, s.parent = p}) != 1.
ExactlyOneInitState :- no NotExactlyOneInitState.
NotAtLeastOneEndState :- p is FSMDiagram,
p.parent = NULL,
k = count({s | s is End, s.parent = p}), k = 0.
AtLeastOneEndState :- no NotAtLeastOneEndState.
ExistsDuplicateName :- s1 is StateBaseTYPE,
s2 is StateBaseTYPE,
s1 != s2, s1.parent = s2.parent,
s1.attributes.name = s2.attributes.name.
UniqueName :- no ExistsDuplicateName.
reachable ::= new(x: StateBaseTYPE, y: StateBaseTYPE).
reachable(x,y) :- x is StateBaseTYPE, y is StateBaseTYPE,
t is TransitionTYPE, x.parent = y.parent,
x.parent = t.parent,
t.pointers.src = x, t.pointers.dst = y.
reachable(x,y) :- reachable(y,x).
reachable(x,z) :- reachable(x,y), reachable(y,z).
DisjointDiagram :- f is FSMDiagramTYPE,
x is StateBaseTYPE,
y is StateBaseTYPE, x.parent = f, y.parent = f,
x != y, no reachable(x,y).
NoDisjointDiagram :- no DisjointDiagram.
```

Figure 4: Formula Constraint Example

be cumbersome to specify large complex set of constraints with visual blocks. Formula stands in the middle of these two categories of constraint specification methods by using declarative constraint specification style, but also providing a reasoning engine to automatically derive the result by repeatedly applying rules to a set of initial constants.

### 6.3. Constraint Editor in WebGME Visualizer

In WebGME environment, a visualizer for code editing is implemented for users to write constraints in Formula syntax. The embedded constraint editor is also used to render evaluation results directly in the code by changing the background color of the head of the constraint based on the result as shown in Figure 5.

Furthermore, the editor also provides syntax highlighting and syntax check to help users come up with correct constraints. Whenever there is a syntax error in the text, an exclamation mark - left of the faulty line - will be shown and by hovering over it the user can see the source of the error. Also, the user can turn on the Formula view, that will result in a split view shown in Figure 5, where the upper portion with white background shows the Formula translation of the Language domain to help the user as those definitions should make the foundation of the constraints.

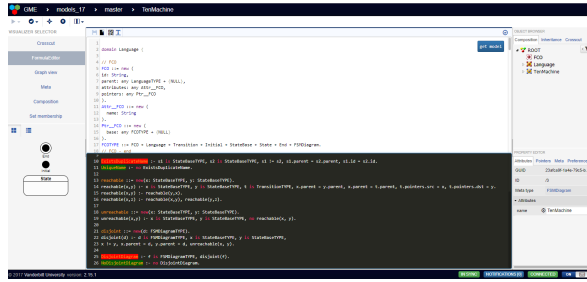


Figure 5: Formula Code Editor

## 7. Conclusion and Future Work

This paper discusses the integration of WebGME, a meta-programmable modeling tool with Formula, a formal framework and tool for specifying domain specific modeling languages. The purpose of the integration has been the construction of an advanced modeling tool that provides extensive model engineering services, such as graphical modeling interface, scalable model repository, web-based implementation architecture and concurrent modeling as well as rigorous formal foundations and tool for specifying modeling language semantics, model transformation semantics, consistency checking, and model synthesis. Seamless integration of these services are enablers for the safe use of domain specific modeling language (DSML) technologies in application domains, where heterogeneous and changing modeling domains need to be integrated and evolved.

As described in the paper, the current level of integration focused on automated and fully synchronized generation of Formula representation from WebGME models. In the next step we will address the opposite direction: the auto-generation of WebGME plugins from Formula specifications for constraint checking and model transformation. The primary goal is to preserve the advantages of the Formula-based declarative specification of model transformations, such as conciseness, semantic precision and preservation of domain invariants after the transformation - but without losing the efficiency of an imperative language implementation, which is particularly important for large models.

## References

[1] J. Sztipanovits, T. Bapty, S. Neema, X. Koutsoukos, and E. Jackson, "Design tool chain for cyber-physical systems: Lessons learned," in *Proceedings of DAC15*. IEEE, 2015, pp. 1–8.

[2] G. Simko, T. Levendovszky, S. Neema, E. K. Jackson, T. Bapty, P. Joe, and J. Sztipanovits, "Foundation for model integration: Semantic backplane," in *Proceedings of the ASME 2012 IDETC/CIE 2012*. ASME, 2012, pp. 1–8.

[3] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, and E. Jackson, *OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber Physical Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 235–248. [Online]. Available: [https://doi.org/10.1007/978-3-642-54848-2\\_16](https://doi.org/10.1007/978-3-642-54848-2_16)

[4] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurác, T. Levendovszky, and Á. Lédeczi, "Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure." in *MPM@ MODELS*, 2014, pp. 41–60.

[5] E. Jackson and J. Sztipanovits, "Formalizing the structural semantics of domain-specific modeling languages," *Software & Systems Modeling*, vol. 8, no. 4, pp. 451–478, Sep 2009.

[6] E. Jackson, W. Schulte, and N. Bjorner, "Open-world logic programs: A new foundation for formal specifications," Tech. Rep., May 2013. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/open-world-logic-programs-a-new-foundation-for-formal-specifications/>

[7] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011. [Online]. Available: <http://doi.acm.org/10.1145/1995376.1995394>

[8] E. K. Jackson, "A module system for domain-specific languages," *CoRR*, vol. abs/1405.4041, 2014. [Online]. Available: <http://arxiv.org/abs/1405.4041>

[9] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi, "The generic modeling environment," in *Workshop on Intelligent Signal Processing, Budapest, Hungary*, vol. 17, 2001, p. 1.

[10] E. K. Jackson and W. Schulte, "Formula 2.0: A language for formal specifications," in *Unifying Theories of Programming and Formal Engineering Methods*. Springer, 2013, pp. 156–206.

[11] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about metamodeling with formal specifications and automatic proofs," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 653–667.

[12] E. K. Jackson, W. Schulte, and N. Bjørner, "Detecting specification errors in declarative languages with constraints," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, 2012, pp. 399–414.

[13] E. K. Jackson and J. Sztipanovits, "Constructive techniques for meta- and model-level reasoning," in *MoDELS*, vol. 7. Springer, 2007, pp. 405–419.

[14] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[15] M. Richters and M. Gogolla, "On formalizing the uml object constraint language ocl," in *International Conference on Conceptual Modeling*. Springer, 1998, pp. 449–464.

[16] B. Broll and Á. Lédeczi, "Extensible visual constraint language," in *Proceedings of the Workshop on Domain-Specific Modeling*. ACM, 2015, pp. 63–70.