A Common Integrated Framework for Heterogeneous Modeling Services

Anastasia Mavridou, Tamas Kecskes, Qishen Zhang, and Janos Sztipanovits
Vanderbilt University
firstname.lastname@vanderbilt.edu

ABSTRACT

Modeling languages are often used for designing complex systems. To simplify the modeling process, system designers can use dedicated modeling frameworks, which provide all necessary tools and services in order to facilitate system development and decrease costs by eliminating errors early at design time. Nowadays, the emergence of heterogeneous engineering domains has led to design flows that span multiple Domain Specific Modeling Languages (DSMLs). As a result, analyzing system behavior requires the semantic integration of modeling domains through languages that are supported by formal semantics, are easily adaptable, and provide scalable and efficient correctness-checking services. To this end, we present a framework that provides heterogeneous modeling services that are based on the combination of two formalisms, 1) logics and 2) graphs. On one hand, we use the FORMULA tool to formalize the semantics of DSMLs and provide a common integrated semantic background. On the other hand, we use graph queries specified in Gremlin to employ efficient query and model analysis services. Our framework offers a complete model-based engineering solution that includes graphical modeling and version control services, which are integrated into a web-based collaborative environment.

1 INTRODUCTION

Modeling frameworks are key enables of model-driven engineering. They provide necessary tools and services to system designers in order to facilitate system development and eliminate errors early at design time. The adoption of model-engineering methods in new cross-disciplinary fields such as Cyber-Physical Systems (CPS) brings new challenges in modeling frameworks [17], which are now required to provide heterogeneous services for capturing multi-abstraction and multi-fidelity models expressed in many different DSMLs. The list of DSMLs used is not static; it changes and evolves depending on the application needs.

Analyzing system structure and behavior, and pursuing cross-domain trade-offs requires the semantic integration of modeling domains. To do that, modeling frameworks must be extensible and adaptable to allow modeling in different DSMLs in a way that their semantics are captured in a formal manner to allow rigorously specifying model transformations. Additionally, modeling frameworks are called to provide heterogeneous services that include 1) engineering services for graphical modeling, metamodeling, version-control, etc.; 2) a common integrated semantic background represented in a formal language; 3) efficient and scalable correctness-checking mechanisms.

On one hand, even though logic-based languages [5, 6, 15] can be very effective for formally capturing constraints, model transformations and proof generations, they might not scale. On the other hand, even though graph-based query languages [1, 16] can be very efficient for model analysis and clustering, they cannot be used for model synthesis and proof generation. We believe that a framework that combines both languages, and thus, all their advantages, can provide the expressiveness and the efficiency needed for modeling and analyzing large, heterogeneous systems.

Our work is based on previous preliminary work presented in [8]. The primary contribution of [8] was the deep semantic integration of WebGME [11], a meta-programmable modeling tool, with FORMULA [6], a formal framework and tool for specifying DSMLs. Deep integration requires that the engineering view of an evolving model and its formal representation to be tightly synchronized. The approach taken in [8] for the formalization of the WebGME metamodeling language with FORMULA was specific to WebGME and thus, cannot be generalized for other metamodeling languages. Additionally, it did not support automatic derivation of conformance constraints nor efficient querying through graph-database languages. There are other works in the field [1, 18] that show the integration of several aspects of our work, but they all lack some of the functionalities we provide - either the capability of visual editing and user defined constraints, or they are tied to a single domain. Our contributions are threefold:

(1) We study and formally present the semantics of the WebGME metamodeling language;
(2) We study a formal, abstract, easy-to-reason model based on typed graphs for capturing the semantics of metamodeling languages. Our approach eliminates as much as possible error-prone user input by automatically deriving consistency conditions that must be met. Additionally, our approach is generic; we have applied it on the WebGME metamodeling language but it can be applied to any other metamodeling language.
(3) We develop a framework that provides heterogeneous services, which are based on the combination of different formalisms. We use logic-based and graph-based languages and study their relation.

Our web-based, open source[1] framework provides a set of engineering services. It allows real-time collaboration between multiple developers. Project changes are committed and versioned, which enables branching, merging and viewing the history of a project.

The organization of the paper is as follows. Section 2 provides background information. Section 3 presents the theoretical foundation of our approach, which is based on labeled and typed graphs and first-order logic conformance conditions. Section 4 and 5 presents the translation of the graphs and conformance conditions in FORMULA and Gremlin languages, respectively. Section 6 discusses

[1] https://github.com/kecso/comif-hems

the translation of specific metamodel and model instances in FORMULA and Gremlin, while also provides implementation details. Section 7 concludes the paper.

2 METAMODELING AND WEBGME

There exists a plethora of metamodeling languages and tools; Kern et al. provide a comparative analysis [9]. In this paper, we use the Web Generic Modeling Environment (WebGME) [11] metaprogrammable modeling tool. The metamodeling language of WebGME is shown in Figure 1 as a UML Class diagram [14].

A project in WebGME is a collection of metamodels and models. When a user creates a new project, it contains the `First Class Object` (`FCO`), which comprises the root of the inheritance hierarchy. `FCO` has a fixed attribute `name` of type `String`, which is inherited by all other objects. Additionally, objects may contain several other attributes. An `Attribute` defines a property of an object, which can be typed as `String`, `Integer`, `Float`, `Enumeration`, etc.

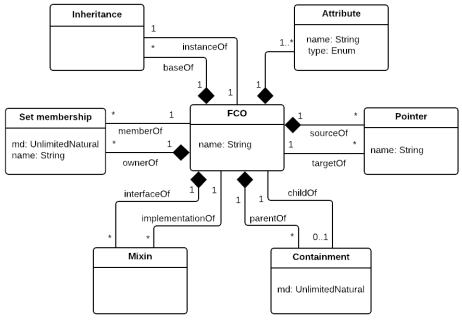


Figure 1: The WebGME metamodeling language.

Relations between objects include `Pointer`, `Set membership`, `Mixin`, `Inheritance`, and `Containment`. `Inheritance` and `Mixin` are similar to UML inheritance and UML realization, respectively. `Inheritance` and `Mixin` are both associated with `FCO` by: 1) a strong containment for their "sources", i.e., `baseOf`, `interfaceOf` and 2) a weak association for their "destinations", i.e., `instanceOf`, `implementationOf`. The strong association denotes the fact that if the base (resp. interface) of an inheritance (resp. mixin) relation is deleted, the inheritance (resp. mixin) relation is deleted as well.

Furthermore, the `Containment` and `Pointer` are similar to a UML containment and UML binary directed named association, respectively. In the same manner, `Set membership` is similar to a UML directed named association between an instance of an *owner* object type and an unordered set of *member* objects of another type. `Set membership` can be considered as a collection of pointers. For all three relations their source is associated with `FCO` through a strong containment association and their destination is associated with `FCO` through a weak association. All three relations have additional user-defined properties. `Set membership` and `Pointer` include property `name` of type `String`. Additionally, `Containment` and `Set membership` include propery `md` of type `UnlimitedNatural` that denotes the destination multiplicity of the child. The source multiplicities of `Containment` and `Set membership` are always 1. Notice that `Inheritance`, `Pointer`, and `Mixin` do not have the `md` property because the multiplicities of their instances are not user defined but fixed. The source multiplicities of each `Inheritance`, `Pointer`, and `Mixin` instance are all equal to 1. The destination multiplicities of each `Inheritance`, `Pointer`, and `Mixin` instance are equal to $*$, 1, and $*$, respectively.

3 SEMANTIC FOUNDATIONS

Before we provide the formal definitions let us first give the intuition behind our approach. We aim at describing the semantics of a metamodeling language with an abstract, high-level language that enables easy reasoning. Graphs provide such an abstract language and have been studied extensively for representing metamodels and models [10]. Additionally, we aim at describing the generic conformance conditions at the meta-meta model level. So instead of creating sets of rules — one set per relation instance — we generalize these rules, by describing them at the meta-meta model level.

We define labeled graphs, which are directed graphs edge-labeled with names and multiplicities, to specify (meta-)metamodels.

Definition 3.1. **(Labeled graph).** A labeled graph is a quadruple $L = \langle V, E, \lambda_v, \lambda_e \rangle$, with a set of vertices V, a set of directed edges $E \subseteq V^2$, and labeling functions 1) $\lambda_v : V \mapsto N$ and 2) $\lambda_e : E \mapsto I \times N \times I$, where N is a set of names and I is a set of intervals of the form $= \mathbb{N}^2 \cup \mathbb{N} \times \{*\}$.

For ease of presentation, we define the notation $\forall e \in E$, $\forall v \in V$:

- $src(e) \in V$ denotes the source vertex of e,
- $dst(e) \in V$ denotes the destination vertex of e,
- $ms(e) \in I$ denotes the source multiplicity of e,
- $md(e) \in I$ denotes the destination multiplicity of e,
- $n(e)$, $n(v) \in N$ denote the names of e and v, respectively.

We require the following uniqueness conditions:

- $\forall v_1, v_2 \in V$, if $n(v_1) = n(v_2)$ then $v_1 = v_2$,
- $\forall e_1, e_2 \in E$, if $src(e_1) = src(e_2)$, $n(e_1) = n(e_2)$, and $dst(e_1) = dst(e_2)$ then $e_1 = e_2$ with source and destination multiplicities $ms(e_1) \cup ms(e_2)$ and $md(e_1) \cup md(e_2)$, respectively.

Next, we define Model graphs, which we use to specify models.

Definition 3.2. **(Model graph).** A model graph is a Labeled graph $M = \langle V, E, \lambda_v, \lambda_e \rangle$ such that for all edges $e \in E$, $ms(e) = md(e) = [1, 1]$.

Typed graphs have been previously studied by the model transformation community [4, 7]. We propose an extension of typed graphs to check the model conformance, which allows to capture `inheritance` defined between nodes of the metamodel.

Definition 3.3. **(Typed graph).** A typed graph is a quadruple $T = \langle L, M, \tau_v, \tau_e \rangle$ where L and M are labeled and model graphs, respectively; $\tau_v : V_M \cup V_L \mapsto 2^{V_L}$, $\tau_e : E_M \mapsto E_L$

Definition 3.3 describes the inheritance relation 1) between vertices of the model or labeled graph and vertices of the labeled graph and 2) edges of the model graph and edges of the labeled graph. The L graph specifies a metamodel with node types and edge types. The M graph is an instance model referencing these types. The type(s) of each vertex v and edge e, of M, is $\tau_v(v)$ and $\tau_e(e)$, respectively.

Definition 3.4. **(Model conformance).** For a typed graph $T = \langle L, M, \tau_v, \tau_e \rangle$, a model, represented by the model graph M, conforms

to a metamodel, represented by the labeled directed graph L if a set of conditions hold:

$$conforms(T) = \bigwedge_{i=1}^{n} (\text{i}),$$

where (i) represents logical formulæ that are either 1) generic and automatically derived from T or 2) application-specific and thus, user-defined.

3.1 Automatically-derived Conditions

For a typed graph $T = \langle L, M, \tau_v, \tau_e \rangle$, we derive the following conformance conditions:

$$(1) \triangleq \forall v_M \in V_M,\ \exists v_L \in V_L : v_L \in \tau_v(v_M)\,.$$

Meaning of (1): for each vertex in the model graph there exists a vertex in the labeled graph that characterizes its type.

$$(2) \triangleq \forall e_M \in E_M,\ \exists e_L \in E_L :$$
$$\tau_e(e_M) = e_L\ \wedge\ src(e_L) \in \tau_v(src(e_M))\,.$$

Meaning of (2): for each edge e_M in the model graph, there exists an edge e_L in the labeled graph, such that e_M is of type e_L and the source vertex of e_M is of type of the source vertex of e_L.

$$(3) \triangleq \forall e_M \in E_M,\ \exists e_L \in E_L :$$
$$\tau_e(e_M) = e_L\ \wedge\ dst(e_L) \in \tau_v(dst(e_M))\,.$$

Meaning of (3): for each edge e_M in the model graph, there exists an edge e_L in the labeled graph, such that e_M is of type e_L and the destination vertex of e_M is of type of the destination vertex of e_L.

$$(4) \triangleq \forall v_A \in V_M, \forall e_L \in E_L, \forall V_{MS} \subseteq V_M, \exists v_B \in V_{MS},$$
$$\forall e_M \in E_M : src(e_L) \notin \tau_v(v_A)\ \vee\ src(e_M) \neq v_A\ \vee$$
$$dst(e_M) \neq v_B\ \vee\ \tau_e(e_M) \neq e_L\ \vee\ |V_{MS}| \in md(e_L).$$

Meaning of (4): each vertex v_A of the model graph is correctly connected to a subset of the vertices of the model graph according to the destination multiplicities of all the edges of the labeled graph that are connected to the vertex that corresponds to the type of v_A.

$$(5) \triangleq \forall e_L \in E_L, \forall v_M \in V_M, \exists e_M \in E_M : src(e_L) \notin \tau_v(v_M) \vee$$
$$0 \in md(e_L) \vee\ (src(e_M) = v_M\ \wedge\ e_L = \tau_e(e_M))\,.$$

Meaning of (5): for each edge e_L of the labeled graph there exists at least an edge e_M, which is an instance of e_L in the model graph if the corresponding destination cardinality of e_L does not include zero and there exists at least a node v_M such that $src(e_M) = v_M$.

$$(6) \triangleq \forall v_A \in V_M, \forall e_L \in E_L, \forall V_{MS} \subseteq V_M, \exists v_B \in V_{MS},$$
$$\forall e_M \in E_M : src(e_L) \notin \tau_v(v_A)\ \vee\ src(e_M) \neq v_B\ \vee$$
$$dst(e_M) \neq v_A\ \vee\ \tau_e(e_M) \neq e_L\ \vee\ |V_{MS}| \in ms(e_L).$$

Meaning of (6): each vertex v_A of the model graph is correctly connected to a subset of the vertices of the model graph according to the source multiplicities of all the edges of the labeled graph that are connected to the vertex that corresponds to the type of v_A.

$$(7) \triangleq \forall e_L \in E_L, \forall v_M \in V_M, \exists e_M \in E_M : dst(e_L) \notin \tau_v(v_M) \vee$$
$$\wedge\ 0 \in ms(e_L)\ \vee\ (dst(e_M) = v_M\ \wedge\ e_L = \tau_e(e_M))\,.$$

Meaning of (7): for each edge e_L of the labeled graph there exists at least an edge e_M, which is an instance of e_L in the model graph if the corresponding source cardinality of e_L does not include zero and there exists at least a node v_M such that $dst(e_M) = v_M$.

4 LOGIC-BASED FORMULA SPECIFICATION

FORMULA (Formal Modeling Using Logic Programming and Analysis) [6] is a specification language and tool that can be used to formally model and verify specifications. It has been successfully used to verify critical drivers in Windows [3].

Next, we show the equivalent specifications of a labeled, model and typed graph in FORMULA. The complete specification is wrapped in a `domain` block, which delimits a domain-specific abstraction.

The specification of a *labeled graph* is as follows:

```
MetaNode ::= new (name: String).
MetaEdge ::= new (name: String, src: MetaNode,
     dst: MetaNode, ms: Multiplicity, md: Multiplicity).
Multiplicity ::= new (low: Natural,high: Natural + {"*"}).
```

FORMULA supports algebraic data types and these are used to encode user defined relations. For example, the first line of the labeled graph specification declares a data type constructor `MetaNode()` for instantiating meta-level nodes (V_L). This constructor produces `MetaNode` instances that have a field called `name` of type `String`.

Similarly, the specification of a *model graph* is as follows:

```
Node ::= new (name: String, type: MetaNode).
Edge ::= new (name: String, type: MetaEdge,
     src: Node, dst: Node).
```

For simplification, we have omitted the source and destination multiplicities of a model graph since they are always equal to 1.

The *inheritance relation* between the nodes and edges of the labeled and model graphs is defined through the *typed graph*. We use the following transitive closure relation to specify node inheritance:

```
NodeInheritance ::= new (base: MetaNode,
     instance: MetaNode + Node).
NodeInstanceOf ::= (MetaNode, MetaNode + Node).
NodeInstanceOf(b,i) :- NodeInheritance(b,i) ;
     NodeInheritance(b,m), NodeInstanceOf(m,i).
```

Edge inheritance can be directly checked through the `type` argument of each `Edge`. We additionally define the `WrongMultiplicity` condition, which follows directly from the labeled graph definition.

```
WrongMult:-Multiplicity(low,high), high !="*", low > high.
```

4.1 Automatically-derived Conditions

To generate the conditions presented in Section 3.1 in FORMULA, for a typed graph $T = \langle L, M, \tau_v, \tau_e \rangle$, we take the negation of the formulas. Due to space limitations we show the equivalent specification for a subset of conditions. The negation of (2) is translated to FORMULA as follows:

```
not2 :- e is Edge, no {m | m is MetaEdge,
    m = e.type, NodeInstanceOf(m.src,e.src)}.
```

The negation of (4) is translated to FORMULA as the conjunction of `(not4a)` and `(not4b)`, which are defined as follows:

```
not4a :- n is Node, m is MetaEdge,
    NodeInstanceOf(m.src,n),
    count({s|s is Node, e is Edge (_,m,n,s)}) < m.md.low.
```

```
not4b :- n is Node, m is MetaEdge,
    NodeInstanceOf(m.src,n), m.md.high != "*",
    count({s|s is Node, e is Edge (_,m,n,s)}) > m.md.high.
```

The underscores denote “dont care” variables.

5 GRAPH-BASED GREMLIN SPECIFICATION

Graph databases use graph structures to represent and store data. They allow to retrieve data of complex hierarchical structures in a simple and fast manner, in comparison with relational databases. We use the Gremlin traversal machine and language [16] by the Apache TinkerPop Project. Gremlin provides a general graph database interface that can be used on top of various industrial graph database implementations. Our Gremlin graphs have vertices and edges with a dedicated *label* property and a number of other properties. The MetaNodes and MetaEdges of the labeled graph are specified in Gremlin as follows:

```
graph.addVertex('class','MetaNode','name',
'theNameOfTheMetaNode');
graph.addVertex('class','MetaEdge','name',
'nameOfTheMetaEdge');
ME.addEdge('src',sMN,'min':0[,'max':1]);
ME.addEdge('dst',dMN,'min':0[,'max':1]);
```

For every MetaEdge and MetaNode, a vertex is created in the graph specification. These vertices have an extra `class` property for identifying their origin. To represent the `src` and `dst` properties of the MetaEdge, we use labeled edges in the graph (`ME` is the MetaEdge while `sMN` is the source MetaNode, and `dMN` is the destination MetaNode). In the edge specifications, the property *max* is not defined if the interval does not have an upper bound. Model graph specifications are identical to Meta graph specifications with the only difference being that their `class` properties are either set to `Node` or `Edge`. To represent the `type` property and the inheritance relation among node types, we specify the additional edges:

```
nodeOrMetaNode.addEdge('type',metaNode);
edge.addEdge('type',metaEdge);
```

5.1 Automatically-derived Conditions

Similarly, we generate the conditions presented in Section 3.1 as Gremlin queries. Due to space limitations we show the equivalent specifications for conditions (2) and (4):

```
not2 = g.V().has('class','Edge').not(
        match(__.as('s').out('type').out('src').as('a'),
        __.as('s').out('src').out('type').as('b')
        .where('b',eq('a')))).hasNext();
```

Query `not2` uses the `match` step where multiple traversals can be checked. For every `Edge` vertex it checks whether there is no `match` based first on `type` and second on `src` edges, and continues by checking for no `match` in the opposite order, i.e., first on `src` and second on `type`. Similarly, queries `not4a, not4b` are as follows:

```
not4a = g.V().has('class','MetaEdge').match(__.as('m').
        in('type').groupCount().by(out('src')).
        order(local).by(values,incr).select(values).
        limit(local,1).as('actual'),__.as('m').
        outE('src').properties('min').   value().
        as('allowed').where('allowed',gt('actual')).
        hasNext();
not4b = g.V().has('class','MetaEdge').where(outE('src').
        has('max')).match(__.as('m').in('type').
        groupCount().by(out('src')).order(local).
        by(values,decr).select(values).limit(local,1).
        as('actual'),__.as('m').outE('src').
        properties('max').value().as('allowed').
        where('allowed',gt('actual')).hasNext();
```

6 INTEGRATION INTO THE DESIGNBIP TOOL

We apply our framework on the DesignBIP tool [13]. Behavior-Interaction-Priority (BIP) is a component-based framework that has been effectively used for the correct-by-construction development of large systems [2, 12]. DesignBIP is a WebGME-based tool that allows to graphically model and generate BIP systems.

6.1 Part of the DesignBIP Metamodel

The DesignBIP metamodel is partly shown in Figure 3. A `Project` contains zero or more `ComponentType` objects. Each `ComponentType` contains at least one `StateBase`, at least one `TransitionBase`, and zero or more `Guard` instances. A `ComponentType` contains exactly one `InitialState`. `StateBase` is an abstract object, meaning that it can only be instantiated as `InitialState` or `State`. Similarly, `TransitionBase` is also an abstract object. The `TransitionBase` object points to `StateBase` through the `src` and `dst` pointers.

6.1.1 DesignBIP metamodel in FORMULA. Our dedicated WebGME-based FORMULA code editor parses the MetaNodes and then automatically creates the corresponding FORMULA rules. The MetaNodes are specified in FORMULA as shown in the second row of Figure 2. Next, the plugin parses the relations specified in the metamodel and their multiplicities and creates multiplicity rules, e.g.,:

```
starMultiplicity is Multiplicity(0,"*").
exactlyOneMultiplicity is Multiplicity(1, 1).
```

As you can see in Figure 3, WebGME concepts have attributes of type `String` and `asset`. The types of these attributes are modeled as MetaNodes in the labeled graph, while the containment of an attribute by a concept is modeled as a MetaEdge, as follows:

```
String is MetaNode("String").
metaAttr1 is MetaEdge("guardName", TransitionBase, String,
    starMultiplicity, exactlyOneMultiplicity)
```

Next, the plugin creates FORMULA rules for the additional `MetaEdge` instances. Pointers are specified as shown in the fourth row of Figure 2. Similarly, containments are specified as shown in the second row of Figure 2. Set memberships are specified as shown in the sixth row of Figure 2. Inheritance is specified with the constructor `NodeInheritance`, as shown in the last row of Figure 2.

6.1.2 DesignBIP metamodel in Gremlin. To generate the graph database queries, we have created a second translator. The equivalent Gremlin specifications for the WebGME concepts and relations are shown in Figure 2. A notable difference between the FORMULA and Gremlin specifications is that the cardinalities are defined in the form of `min` and `max` properties of edges of the graph. Additionally, the unbounded cardinality intervals are simply represented by not defining the `max` property. Another difference is the `type` edges. They represent the NodeInstanceOf relationship which means that every base of a MetaNode is directly connected and we do not have to make long traversals to get the information.

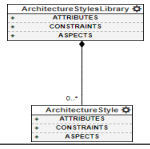
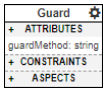
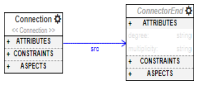
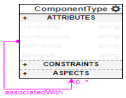
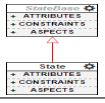
Objects	WebGME meta	FORMULA translation	Gremlin translation
Concept		`StateBase is MetaNode('StateBase');`	`StateBase = graph.addVertex( 'class','MetaNode','name','StateBase);`
Containment		`metaContainment1 is MetaEdge('metaContainment1', ArchitectureStylesLibrary, ArchitectureStyle, exactlyOneMultiplicity, starMultiplicity);`	`metaContainment1 = graph.addVertex( 'class', 'MetaEdge', 'name', 'metaContainment1'); MContainment1.addEdge('src', ArchitectureStylesLibrary,'min',1,'max',1). MContainment1.addEdge('dst', ArchitectureStyle,'min',0);`
Attribute		`Guard_has_guardMethod is MetaEdge( 'guardMethod',Guard,String, exactlyOneMultiplicity,starMultiplicity);`	`Guard_has_guardMethod = graph.addVertex( 'class','MetaEdge','name','guardMethod'); Guard_has_guardMethod.addEdge('src',Guard, 'min',0); TransitionHasGuard.addEdge( 'dst',String,'min',1,'max',1);`
Pointer (one to one association)		`Connection_point_src_ConnectorEnd is MetaEdge( 'src',Connection, ConnectorEnd, exactlyOneMultiplicity, exactlyOneMultiplicity);`	`Connection_point_src_ConnectorEnd = graph .addVertex('class','MetaEdge','name','src'); Connection_point_src_ConnectorEnd.addEdge('src', Connection,'min',0); Connection_point_src_ConnectorEnd.addEdge('dst', Connection,'min',1,'max',1);`
Set (many to many association)		`ComponentType_collects_ComponentType is MetaEdge( 'associatedWith', ComponentType, ComponentType, exactlyOneMultiplicity, starMultiplicity);`	`ComponentType_collects_ComponentType = graph .addVertex('class','MetaEdge', 'name','associatedWith'); ComponentType_collects_ComponentType.addEdge( 'src',ComponentType,'min',0); ComponentType_collects_ComponentType.addEdge( 'dst',ComponentType,'min',0);`
Inheritance (identical to Mixin)		`NodeInheritance(StateBase, State);`	`State.addEdge('type',StateBase);`

Figure 2: Patterns of translation into FORMULA and Gremlin of main WebGME meta-modeling language features.

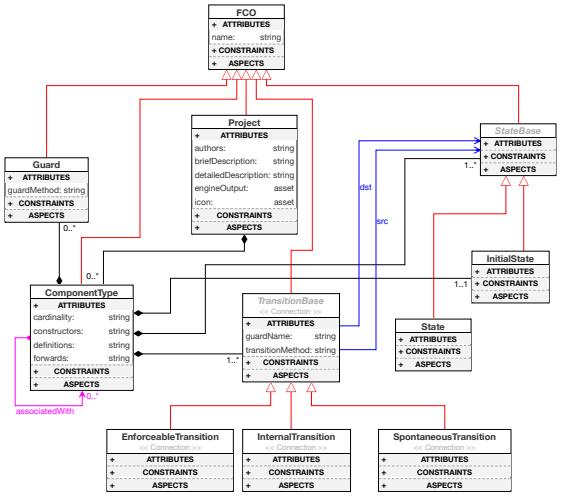


Figure 3: Part of the DesignBIP metamodel in WebGME.

6.2 DesignBIP Model Example

Let us now consider the DesignBIP FSM model shown in Figure 4. The FSM has four states `off`, `on`, `done`, `wait`. The states are connected with transitions that are of three types: `EnforceableTransition`, `SpontaneousTransition`, and `InternalTransition` (Figure 3). Transitions `on`, `off`, and `finished`, illustrated by continuous lines, are of type `EnforceableTransition`. Transition `end`, illustrated by a dashed line, is of type `SpontaneousTransition`. The last transition that does not have a name is of type `InternalTransition`. The square-brackets labels `[finished]` and `[!finished]` denote guards, i.e., predicates, on the end and internal transitions.

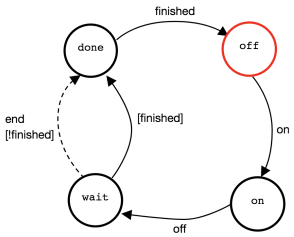


Figure 4: Part of a DesignBIP model.

6.2.1 DesignBIP Model in FORMULA. We present a subset of the automatically-generated FORMULA rules for the model shown in Figure 4. Model nodes are instances of the `Node` type, e.g.,:

```
State2on is Node("on", State).
```

The evaluated attributes of the model are specified as nodes, e.g., the attribute `guardName` of the `end` transition is specified as follows:

```
modelAttr1 is Node("!finished", String).
```

The containment relation between the `!finished` attribute and the `end` transition is specified through an Edge as follows:

```
modelContain1 is Edge("modelAttr1", metaAttr1,
    SpontaneousTransition4end, modelAttr1)
```

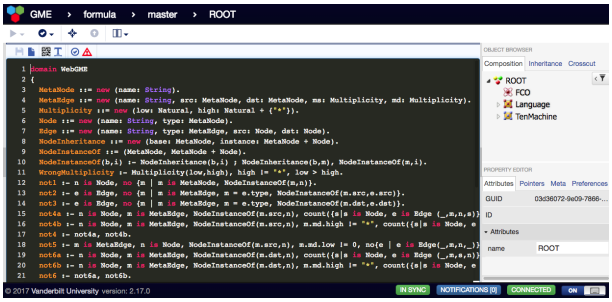


Figure 5: Formula code editor.

Similarly, edge nodes for all relations except for inheritance are generated as instances of the Edge type (Section 4), e.g.,:

```
modelPointer1 is Edge ("modelPointer1", metaPointer1src,
EnforceableTransition3on, InitialState1off ).
```

Inheritance relations between the nodes and metanodes are generated as instances of `NodeInheritance` type, e.g.,:

```
modelInheritance1 is NodeInheritance(State2on, State).
```

6.2.2 DesignBIP Model in Gremlin. Next, we present portions of the generated Gremlin specification for the model of Figure 4. For instance, the *on* state is specified in Gremlin as follows.

```
State2on = graph.addVertex('class','Node', 'name','s2').
State2on.addEdge('type',State).
State2on.addEdge('type',StateBase).
State2on.addEdge('type',FCO).
```

The translation take cares of realizing the transitive closure of the FORMULA definition of `NodeInheritance`. Similarly, the specification of the `guardname` attribute in Gremlin is as follows:

```
modelAttr1.addEdge('type',String).
modelAttr1HasGuard = graph.addVertex('class','Edge',
 'guardName','_t4').
modelAttr1HasGuard.addEdge('src',Transition4end).
modelAttr1HasGuard.addEdge('dst',modelAttr1).
modelAttr1HasGuard.addEdge('type',metaAttr1).
```

The specification of relations, for example the pointer relation between state `off` and enforceable transition `on`, is as follows:

```
modelPointer1=graph.addVertex('class','Edge','name','p1').
modelPointer1.addEdge('type',metaPointer1src).
metaPointer1.addEdge('src',EnforceableTransition3on).
metaPointer1.addEdge('dst',InitialState1off).
```

6.3 Implementation

To extend and allow the integration of these different representations, we developed two dedicated code editors (the FORMULA one can be seen in Figure 5) to the WebGME. Both editors have two types of content: 1) read-only content that is generated from the visually created meta-model and model and 2) a user-defined portion where the user can add any additional constraint. The read-only part can only be altered by using the graphical model or meta-model editor of the WebGME. The code-editors also provide 'check-buttons' to run an on-demand evaluation of the given model. This is done through dedicated plugins that communicate with the integrated FORMULA and Gremlin console engines to check the well-formedness of the model. Though we have not completed any extensive performance testing between the two representations, our first tests show that conformance checking in Gremlin is more scalable and efficient than a similar task in FORMULA. With the BIP domain (having 49 MetaNodes and 39 MetaEdges) and a relatively large model (4082 Nodes and 12670 Edges), the FORMULA execution took 113s while the Gremlin remained under 1s.

7 CONCLUSION

This paper discusses the integration of WebGME with 1) FORMULA a formal framework for specifying DSMLs and 2) Gremlin an graph-based query language. The purpose of the integration has been the construction of an advanced modeling tool that provides heterogeneous services. In particular, our framework provides: 1) extensive model engineering services, such as graphical modeling interfaces, scalable model repositories, a web-based architecture that allows collaborative modeling which is backed-up by version control services; 2) rigorous formal foundations through the integration of the FORMULA language that can be used for specifying language semantics, model transformation rules, automated consistency checking, and model synthesis; 3) efficient Gremlin queries that can be used for checking model conformance, clustering, and data analysis of large and complex systems. In the future, we will use FORMULA for model synthesis and proof generation.

8 ACKNOWLEDGMENT

This research was supported by the National Science Foundation under award CNS-1521617.

REFERENCES

[1] Renzo Angles and Claudio Gutierrez. Querying rdf data from a graph database perspective. In *European Semantic Web Conference*, pages 346–360, 2005.

[2] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI*, volume 178, pages 631–635, 2008.

[3] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[4] H Ehrig, K Ehrig, U Prange, and G Taentzer. Fundamentals of algebraic graph transformation. EATCS Series, 2006.

[5] Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000.

[6] Ethan K Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 39–48. ACM, 2010.

[7] Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[8] Tamas Kecskes, Qishen Zhang, and Janos Sztipanovits. Bridging engineering and formal modeling: Webgme and formula integration. In *Proceedings of the fifth International Workshop on The Globalization of Modeling Languages*.

[9] Heiko Kern, Axel Hummel, and Stefan Kühne. Towards a comparative analysis of meta-metamodels. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 7–12. ACM, 2011.

[10] AG Kleppe and Arend Rensink. On a graph-based semantics for UML class and object diagrams. In *Graph Transformation and Visual Modelling Techniques*. EASST, 2008.

[11] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamer Levendovszky, and Ákos Lédeczi. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS*, 1237:41–60, 2014.

[12] Anastasia Mavridou, Emmanouela Stachtiari, Simon Bliudze, Anton Ivanov, Panagiotis Katsaros, and Joseph Sifakis. Architecture-based design: A satellite on-board software case study. In *International Conference on Formal Aspects of*

Component Software, pages 260–279. Springer, 2016.

[13] Anastasia Mavridou, Joseph Sifakis, and Janos Sztipanovits. DesignBIP: A design studio for modeling and generating systems with BIP. In *Proc. of 1st International Workshop on Methods and Tools for Rigorous System Design (MeTRiD)*, June 2018.

[14] Object Management Group (OMG). Unified Modeling Language, Version 2.5. https://www.omg.org/spec/UML/2.5/About-UML/, 2015. [Accessed 1-May-2018].

[15] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In *International Conference on Conceptual Modeling*, pages 449–464. Springer, 1998.

[16] Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.

[17] Janos Sztipanovits, Ted Bapty, Xenofon Koutsoukos, Zsolt Lattmann, Sandeep Neema, and Ethan Jackson. Model and tool integration platforms for cyber-physical system design, 2018. URL dx.doi.org/10.1109/jproc.2018.2838530.

[18] Bahram Zarrin and Hubert Baumeister. An integrated framework to specify domain-specific modeling languages. In *6th International Conference on Model-Driven Engineering and Software Development*, pages 83–94, 2018.