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 enablers 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 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 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 present 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 FORMLA 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] metamodeling 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.](image)

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 property 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 \( L = (V, E, \lambda_V, \lambda_E) \), with a set of vertices \( V \), a set of directed edges \( E \subseteq V \times V \), and labeling functions 1) \( \lambda_V : V \mapsto \text{N} \) 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 \( \mathbb{N}^2 \cup \mathbb{N} \times \{\ast\} \).

For ease of presentation, we define the notation \( \forall e \in E, \forall v \in V: \)
- \( \text{src}(e) \in V \) denotes the source vertex of \( e \),
- \( \text{dst}(e) \in V \) denotes the destination vertex of \( e \),
- \( \text{ms}(e) \in I \) denotes the source multiplicity of \( e \),
- \( \text{md}(e) \in I \) 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:
- \( \forall e_1, e_2 \in E, \text{if } \text{src}(e_1) = \text{src}(e_2), \text{ms}(e_1) = \text{ms}(e_2), \text{and } \text{dst}(e_1) = \text{dst}(e_2) \text{ then } e_1 = e_2, \)
- \( \forall e_1, e_2 \in E, \text{if } \text{src}(e_1) = \text{src}(e_2), n(e_1) = n(e_2), \text{and } \text{dst}(e_1) = \text{dst}(e_2) \text{ then } e_1 = e_2 \).

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

**Definition 3.2.** *(Model graph)*. A model graph is a Labeled graph \( M = (V, E, \lambda_V, \lambda_E) \) 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 \( T = (L, M, \tau_V, \tau_E) \) where \( L \) and \( M \) are labeled and model graphs, respectively; \( \tau_V : V_M \cup V_L \mapsto 2^V, \tau_E : E_M \mapsto E_L \).

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 \( T = (L, M, \tau_V, \tau_E) \), 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:

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

where (i) represents logical formulae 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 \( T = (L, M, \tau_{v}, \tau_{e}) \), we derive the following conformance conditions:

1. \( \forall v_{M} \in V_{M}, \forall e_{L} \in E_{L} : \tau_{e}(v_{M}) = e_{L} \land \sigma_{e}(v_{M}) \subseteq \tau_{e}(v_{M}) \).

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

2. \( \forall e_{M} \in E_{M}, \forall e_{L} \in E_{L} : \tau_{e}(v_{M}) = e_{L} \land \sigma_{e}(v_{M}) \subseteq \tau_{e}(v_{M}) \).

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. \( \forall e_{M} \in E_{M}, \forall e_{L} \in E_{L} : \tau_{e}(v_{M}) = e_{L} \land \sigma_{e}(v_{M}) \subseteq \tau_{e}(v_{M}) \).

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. \( \forall v_{A} \in V_{M}, \forall e_{L} \in E_{L}, \forall V_{MS} \subseteq V_{M}, \exists v_{B} \in V_{M}, \forall v_{M} \in E_{M} : \sigma_{e}(v_{M}) \notin \tau_{v}(v_{A}) \lor \sigma_{e}(v_{M}) = v_{A} \land \sigma_{e}(v_{M}) = \nu_{A} \land \tau_{e}(v_{M}) \notin \nu_{B} \lor \nu_{B} \land \nu_{B} \notin \nu_{A} \land \nu_{A} \land \nu_{B} \land |V_{MS}| \in md(e_{L}). \)

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. \( \forall e_{L} \in E_{L}, \forall v_{M} \in V_{M}, \exists e_{M} \in E_{M} : \sigma_{e}(v_{M}) \notin \tau_{v}(v_{A}) \lor \sigma_{e}(v_{M}) \notin \nu_{A} \land \nu_{A} \land \nu_{B} \land |V_{MS}| \in md(e_{L}). \)

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 \( \sigma_{e}(v_{M}) = v_{M} \).

6. \( \forall v_{A} \in V_{M}, \forall e_{L} \in E_{L}, \forall V_{MS} \subseteq V_{M}, \exists v_{B} \in V_{M}, \forall v_{M} \in E_{M} : \sigma_{e}(v_{M}) \notin \tau_{v}(v_{A}) \lor \sigma_{e}(v_{M}) = v_{A} \land \nu_{A} \land \nu_{B} \land \nu_{B} \notin \nu_{A} \land \nu_{A} \land \nu_{B} \land |V_{MS}| \in md(e_{L}). \)

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. \( \forall e_{L} \in E_{L}, \forall v_{M} \in V_{M}, \exists e_{M} \in E_{M} : \sigma_{e}(v_{M}) \notin \tau_{v}(v_{M}) \lor \sigma_{e}(v_{M}) = v_{M} \land \nu_{A} \land \nu_{B} \land |V_{MS}| \in md(e_{L}). \)

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 \( \tau_{e}(v_{M}) = \tau_{e}(v_{M}). \)

### 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:

```formula
MetaNode := new (name : String).
Multiplicity := new (low : Natural, high : Natural + (*"\cdot")),
```

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:

```formula
Node := new (name : String, type : MetaNode).
```

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:

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

```formula
WrongMultiplicity(low, high) := low < "\cdot" \land high > "\cdot".
```

### 4.1 Automatically-derived Conditions

To generate the conditions presented in Section 3.1 in FORMULA, for a typed graph \( T = (L, M, \tau_{v}, \tau_{e}) \), 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:

```formula
not2 := e is Edge, no (n | n is MetaEdge, m = e.type, NodeInstanceOf(m, src.e, src)).
```

The negation of (4) is translated to FORMULA as the conjunction of \((\neg 4a)\) and \((\neg 4b)\), which are defined as follows:

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

\[
\text{not4b} := n \text{ is Node}, m \text{ is MetaEdge},
\text{NodeInstanceOf}(m, n), \text{m.md.high} = "s",
\text{count}((s|s \text{ is Node}, e \text{ is Edge (_,m,n,s)}) > m.md.high).
\]

The underscores denote “dont care” variables.

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.:

\[
\text{starMultiplicity is Multiplicity}(0,"*"),
\text{exactlyOneMultiplicity is Multiplicity}(1, 1).
\]

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:

\[
\text{nodeOrMetaNode.addEdge('type', metaNode);}
\text{edge.addEdge('type', metaNode);}
\]

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):

\[
\text{not2 = not (match(\_ \_ . as('s')) . out('type') . out('src') . as('a'),
\_ \_ . as('s') . out('src') . out('type') . as('b'))
\_ \_ . eq('a')) . hasNext());}
\]

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:

\[
\text{not4a = g.V().has('class','MetaEdge').match(\_ \_ . as('m'))
\_ \_ \_ \_ . groupCount().by(out('src')).
\_ \_ \_ \_ . order(local).by(values, incr).select(values).
\_ \_ \_ \_ . limit(local,1).as('actual'),
\_ \_ \_ \_ . outE('src').properties('min').value(),
\_ \_ \_ \_ . as('allowed').where('allowed', gt('actual')).hasNext();}
\]

\[
\text{not4b = g.V().has('class','MetaEdge').where(outE('src')
\_ \_ \_ \_ . has('max')).match(\_ \_ \_ \_ as('m')).int('type').}
\]
The FSM has four states: "off", "on", "done", and "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.

### 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", and "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.

### 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.:

```plaintext
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:

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

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

```plaintext
modelContains1 is Edge("modelAttr1", modelAttr1, SpontaneousTransition4end, modelAttr1)
```
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.

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

The translation takes care of realizing the transitive closure of the FORMULA definition of NodeInheritance type. Similarly, the specification of the guardname attribute in Gremlin is as follows:

```java
modelAttr1 = graph.addVertex('class','Edge',
   'guardName','_t4').
modelAttrHasGuard = graph.addEdge('src',TransitionEnd).
modelAttrHasGuard.addEdge('dst',modelAttr1).
modelAttrHasGuard.addEdge('type',modelAttr1).
```

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

```java
modelPointer1 = Edge('modelPointer1','metaPointer1src,
   EnforceableTransition3on, InitialStateoff').
```

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

```java
modelInheritance1 = NodeInheritance(State2on, State).
```

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 visually created meta-model and model and 2) a user-defined portion where the user can add any additional constraint. 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 DSLs 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


