A COMMON INTEGRATED FRAMEWORK FOR HETEROGENEOUS MODELING SERVICES

ANASTASIA MAVRIDOU, TAMAS KECSKES, QISHEN ZHANG, AND JANOS SZTIPANOVITS

VANDERBILT UNIVERSITY, NASHVILLE, TN, USA

WHY?

- WebGME – easy visual modeling with added features and integration capabilities for a smooth user experience
- Formula – strong formal base and concise form for expressing constraints
- Gremlin – strong querying capabilities and optimized performance due to graph based data representation

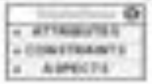
HOW? – COMMON LANGUAGE

- Typed graph $T = \langle L, M, \tau_v, \tau_e \rangle$
  - L is a labeled graph that represents the domain with its structural semantics
  - M is a model graph that implements the actual model that has to follow the rules of L
  - τ_v inheritance relationship among vertices of M and L
  - τ_e inheritance relationship among edges of M and L
- This type of graph description can be effectively modeled in all three representations

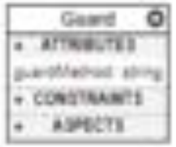
HOW? – FEATURES OF THE TYPED GRAPH SYSTEM

- Pros
  - Each representation understands this graph system
  - It is enough to define the well formedness rules once and then they can be applied to any DSML (not even tied to WebGME meta-modeling language)
- Cons
  - The system can be extended with special constraints if need be, though due to the lack of generic translation between Formula and Gremlin the rules need to be described in both in an equivalent way
  - Due to the nature of the typed graph system, the additional rules either has to be written on a domain agonstic abstraction level or they became cumbersome and complex

INTEGRATING

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);`

WELLFORMEDNESS RULES

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

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

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

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

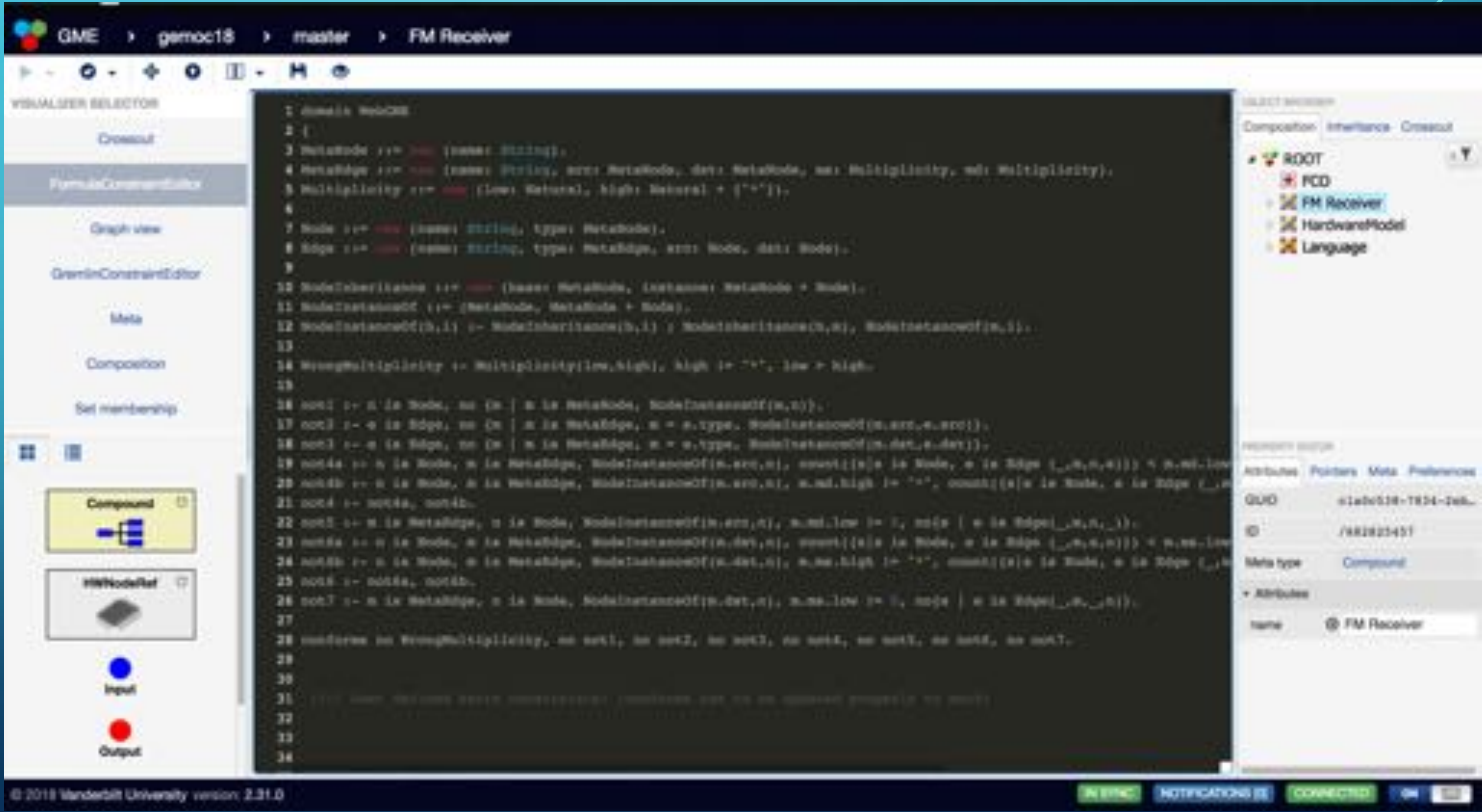
$$(5) \triangleq \forall e_L \in E_L, \forall v_M \in V_M, \exists e_M \in E_M : src(e_L) \notin \tau_v(v_M) \vee 0 \in md(e_L) \vee\ (src(e_M) = v_M \ \wedge\ e_L = \tau_e(e_M))\,.$$

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

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

The rules basically governs that every ‘model’ element should have a corresponding ‘language’ element and that every relationship in the ‘model’ has a definition that allows it. They also control that the number of relationships has to fit into the requirements of the cardinalities of the language defintions.

IMPLEMENTATION



FUTURE

- Creating a bi-directional translation among Formula rules and Gremlin queries so that the user would only need to use the more concise Formula language
- Allow the use of a domain specific formula depiction which would let users create their constraints on a more manageable abstraction level (the one they defined for their domain instead of the common typed graph one)

EXAMPLE

THANK YOU!

- Questions?