

A SEMANTIC BACKPLANE FOR INCREMENTAL MODELING

By

Qishen Zhang

Dissertation

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

August 11, 2023

Nashville, Tennessee

Approved:

Janos Sztipanovits, Ph.D.

Daniel Balasubramanian, Ph.D.

Gabor Karsai, Ph.D.

Jeff Gray, Ph.D.

Ethan Jackson, Ph.D.

Copyright © 2023 Qishen Zhang
All Rights Reserved

Dedicated to my parents, grandparents, and other family members, whose unwavering love, support, and sacrifice made this achievement possible. Your belief in me never wavered, and I am forever grateful for everything you have done for me. This work is a tribute to your love and encouragement throughout my academic journey.

ACKNOWLEDGMENTS

First of all, I would like to thank Professor Janos Sztipanovits who saw my potential and accepted me as his Ph.D. student at Vanderbilt University. I'm truly grateful that he supports me over the years and believes in me even though at many points of my academic journey I have suffered from anxiety and doubt about myself. Without his support and encouragement, I will not be able to make it to the end and achieve so much.

I would also like to thank my Ph.D. Committee members, Daniel Balasubramanian, Gabor Karsai, Jeff Gray, and Ethan Jackson for their valuable feedback on my research work and dissertation along the way. I am also very thankful to the co-authors Daniel Balasubramanian, Anastasia Mavridou, and Tamas Kecskes for the collaboration and each of them provides insights and a tremendous help to my work. I also want to thank Ethan Jackson for sparing time from his tight schedule for our weekly meeting to explain the internals to us with great patience.

Finally, I would also like to thank all my family members, especially my mom they support me with love and encouragement, despite the difficulties of not being able to see each other in person for years and two of my grandparents will not have the chance to witness my achievements unfortunately. During the time of my graduate study, I also met lots of personal friends and colleagues who make my time in a foreign country more enjoyable and share a lot of memories. I want to express my deep appreciation to Yi Li, Fangzhou Sun, Yiyuan Zhao, Xingyu Zhou, Jian Lou, Saqib Hasan, Gergely Varga, Jingwei Fan, Brian Berstein, Lin Song, Hannah, Melisa, Bowen Jin, Zhiyu Wan.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
1 Introduction	1
1.1 Motivation	1
1.2 Background	5
1.2.1 WebGME	5
1.2.2 FORMULA Language	7
1.2.3 Semantic Backplane	11
1.2.4 Differential-Datalog and Differential-Dataflow	12
1.3 Contribution	13
1.4 Organization	13
2 Challenges in Semantic Integration for Modeling	14
2.1 General Research Challenges in Model-driven Engineering	14
2.2 Challenges in Graph-based Semantic Integration	15
2.2.1 Integration of Models, Semantics, Storage, and Execution	15
2.2.2 Unified Graph Representation in Modeling Framework	16
2.2.3 Scalability Issues on Large Models	17
2.3 Problems and Challenges in Incremental Modeling	18
3 Related Work	19
3.1 Generic Modeling Engineering Frameworks and Tools	19
3.2 Modeling Framework with Formal Semantics	20
3.3 Graph-based Modeling Frameworks	21
3.4 Bootstrapping Modeling Frameworks	22
3.5 Incremental Modeling	23
3.5.1 OCL-based Incremental Model Query	24
3.5.2 ATL-based Incremental Model Transformation	24
3.5.3 TGG-based Incremental Model Transformation	25
3.5.4 Graph-based Incremental Model Transformation	26
3.5.5 Other incremental approaches	26
4 Semantic Bridge from Models to Graphs	28
4.1 Introduction to Graph Database	28
4.2 Semantic Bridge Overview	29
4.2.1 Tools Comparison	30
4.2.2 Three Levels of Integration	31
4.3 Direct Integration of WebGME and FORMULA	33
4.4 Graph-based Integration of WebGME, FORMULA, and Graph Database	35
4.4.1 Semantic Foundation in Graph	35
4.4.2 Logic-based FORMULA Specification	37
4.4.3 Graph-based Gremlin Specification	39

4.5	FORMULA Model Operations Executed in Graph Database	41
4.5.1	Model and Metamodel Representation in Graph	42
4.5.2	Pre-processing of FORMULA Rule	44
4.5.3	Label Map for FORMULA Rule	45
4.5.4	Instance and Arguments	45
4.5.5	Handling Binding Label with Fragments	46
4.5.6	Constraints over Properties in Built-in Types	47
4.5.7	Negation and Set Comprehension Operators	47
4.5.8	Termination of Repeating Rule Execution	48
4.6	Benchmark and Performance Comparison	49
4.7	Contributions	50
5	Developing Differential-FORMULA in MDE Methodology	52
5.1	Differential-Dataflow Computation Model and Tool Suites	54
5.1.1	Timely Dataflow	54
5.1.2	Differential Dataflow	56
5.1.3	Differential Datalog	57
5.2	Introduction to Differential-FORMULA	60
5.3	Identification of Semantic Mismatch	61
5.4	Formal Specification of FORMULA Language and Differential-Datalog	63
5.4.1	Union Type Definition	64
5.4.2	Negation	66
5.4.3	Aggregation	68
5.5	Static Analysis on FORMULA Programs	69
5.5.1	Language Domain Modeling and Metamodels	70
5.5.2	Type Inference of Non-ground Term	74
5.5.2.1	Type Inference of Sub-terms	74
5.5.2.2	Term Unifiability Checking	75
5.5.3	Type Inference In the Context of FORMULA Rules	77
5.5.4	Nested Set Comprehension Validation	80
5.5.5	Rule Stratification Validation	82
5.6	<i>formula2datalog</i> Model Transformation	85
5.6.1	Positive Predicate Translation	85
5.6.2	Negated Predicate Translation	85
5.6.2.1	Negation as Set Difference	85
5.6.2.2	Negation as Set Comprehension	86
5.6.3	Set Comprehension Translation	86
5.6.3.1	Independent Set Comprehension	87
5.6.3.2	Dependent Set Comprehension	89
5.6.3.3	Nested Set Comprehension	90
5.6.4	<i>formula2datalog</i> Transformation Example	91
5.7	Benchmarks on Incremental Updates for Large Models	94
5.8	Contributions	101
6	Future works and Open Challenges	103
6.1	Future Works	103
6.1.1	Implementation and Optimization	103
6.2	Open Challenges	103
6.2.1	Model Synthesis	103
6.2.2	Formal Specification with Bootstrapping	104
	References	105

LIST OF TABLES

Table		Page
4.1	Execution times for FORMULA and Gremlin specifications. MN, ME, N, E stand for number of MetaNodes, MetaEdges, Nodes, Edges.	50

LIST OF FIGURES

Figure	Page
1.1 Semantic Backplane for Heterogeneous Modeling	3
1.2 WebGME Design Studio Workflow	6
1.3 ROSMOD Model Visualization in WebGME	7
2.1 Graph-based Integrated Modeling Framework	16
2.2 WebGME with FORMULA-based Semantic Backplane	17
4.1 Overview of WebGME, FORMULA, and Graph Database Integration	30
4.2 Translation of WebGME Meta Concepts into FORMULA	34
4.3 Patterns of translation into FORMULA and Gremlin of main WebGME meta-modeling language features	41
4.4 <i>MetaGraph</i> Domain and Model in FORMULA	43
4.5 FORMULA Model Represented in Graph	44
5.1 Differential-Datalog Internal Workflow	58
5.2 Generated Differential-Dataflow from a DDLLog Rule	59
5.3 Architecture of DDLLog-based Incremental Modeling Framework.	61
5.4 Metamodel of FORMULA Language	64
5.5 Metamodels of DDLLog and FORMULA Modeled Visually in WebGME	64
5.6 δF Incremental Running Time	93
5.7 Non-incremental Running Time: δF vs FORMULA	93
5.8 TTC2018 Social Network Metamodel in Ecore	98
5.9 Q1 Load Time	99
5.10 Q1 Batch Time	99
5.11 Q1 Updates Time	100
5.12 Q2 Load Time	100
5.13 Q2 Batch Time	101
5.14 Q2 Updates Time	101

CHAPTER 1

Introduction

1.1 Motivation

Modeling frameworks are the required infrastructure for model-based design. They provide necessary tools and services for system designers to facilitate model-based system development with domain-specific modeling languages and model transformations. Modeling frameworks are responsible for offering an intuitive engineering interface, often graphical, for model developers, as well as providing a range of services supporting safe model engineering practices, including composing, decomposing, visualizing, modifying, checking well-formedness, versioning, and storing large models.

There are six dominant activities in model-based engineering:

1. Specifying modeling domains using domain-specific modeling languages: design a metamodel in either textual or graphical syntax.
2. Building models: create models in either textual or graphical syntax.
3. Verifying with static model checking: verify the well-formedness of models during the modeling process using constraint checking.
4. Specifying model transformations: use a specification language for specifying mappings from one modeling domain to another.
5. Executing model transformations: implement or use an existing execution engine that ideally executes the transformation specifications directly.
6. Storing models: store and load models to and from persistent storage.

Model-driven engineering (MDE) has been applied extensively to Cyber-physical systems (CPS) in which functionality emerges from the interactions of computational and physical processes. The CPS design of autonomous vehicles, smart energy systems, public transportation systems, and the Internet of Things are dominantly model-based. OpenMETA (Sztipanovits et al., 2014) is one of the successful use cases of MDE that focused on integrating and testing an automated, model-based design for the power train and hull of the fast adaptable next-generation ground vehicle in DARPA Adaptive Vehicle Make (AVM) program. The toolchain improved design productivity with correct-by-construction component- and model-based design methods,

incorporated manufacturing-related constraints into the design flow, and support a web-based collaboration platform for crowd-sourced design for a meager cost.

MDE also has applications in robotic systems and embedded system such as mbeddr (Voelter et al., 2012) in MPS (Pech, 2021) that uses extensible DSLs, flexible notations, and integrated verification tools to generate verified C program from models to run reliably in embedded system. Furthermore, the MDE methodology is applied to pure software engineering and language engineering such as P language (Desai et al., 2013) to design a verifiable compiler from high-level language to low-level general-purpose programming languages.

The industrial use case such as Adaptive Vehicle Make mentioned above is a highly complicated application of model-driven engineering and prototyping of the conceptual design flow requires several categories of tools to be integrated before the execution and evaluation of automated workflow. The tools include 1) authoring tools to define component models and design space models; 2) model composition tools that transform models and derive inputs for domain-specific analysis tools; 3) domain-specific analysis test benches and tools for analysis and evaluation of the candidate system using models of progressively deepening fidelity; and 4) analytics and visualization tools for visualization of analysis results as well as for interactively conducting design trades.

A model integration platform is the foundation of cross-domain modeling and we developed a Model Integration Language named CyPhyML that has several sub-languages:

1. Component models (CMs) that incorporate: 1) several domain models representing various aspects of component properties, parameters, and behaviors; 2) a set of standard interfaces through which the components can interact; 3) the mapping between the component interfaces and the embedded domain models; and 4) constraints expressing cross-domain interactions
2. Design models (DMs) that describe system architectures using assemblies, components and their interconnections via the standard interfaces.
3. Design space models (DSMs) that define architectural and parametric variabilities of DMs as hierarchically layered alternatives for assemblies and components.
4. Analysis models (AMs) that specify data and control interfaces for analysis tools.
5. Testbench models (TMs) that 1) specify regions in the design space to be used for computing key performance parameters; 2) define analysis flows for computing key performance parameters linked to specific requirements; and 3) provide interfaces for visualization.

Well-known challenges of introducing model-based engineering to the end-to-end design of CPS product lines in the aerospace and automotive industries are *heterogeneity* and *scalability*. Engineering processes

require a large number of distinct tools throughout the design flow. In addition, the list of domain-specific modeling languages (DSMLs) used is not fixed, but rather changes and evolves according to the needs of the application domain. This is particularly true in cross-disciplinary fields like CPS, where the application of model-based engineering must be able to accommodate many different DSMLs. The accommodation of different DSMLs can be viewed as a *composition* problem in which multi-abstraction and multi-fidelity models must be captured and integrated. Composition of an end-to-end integrated toolchain from a heterogeneous collection of commercial-off-the-shelf (COTS), open-source, and proprietary tools is difficult because it is not simply a tool interoperability problem, but a major *semantic integration* problem.

Previous work at Vanderbilt addressed this need in the development of the OpenMETA design automation tool suite for DARPA’s Adaptive Vehicle Make program (Sztipanovits et al., 2014) and the overall workflow of OpenMETA and semantic backplane is shown in Figure 1.1.

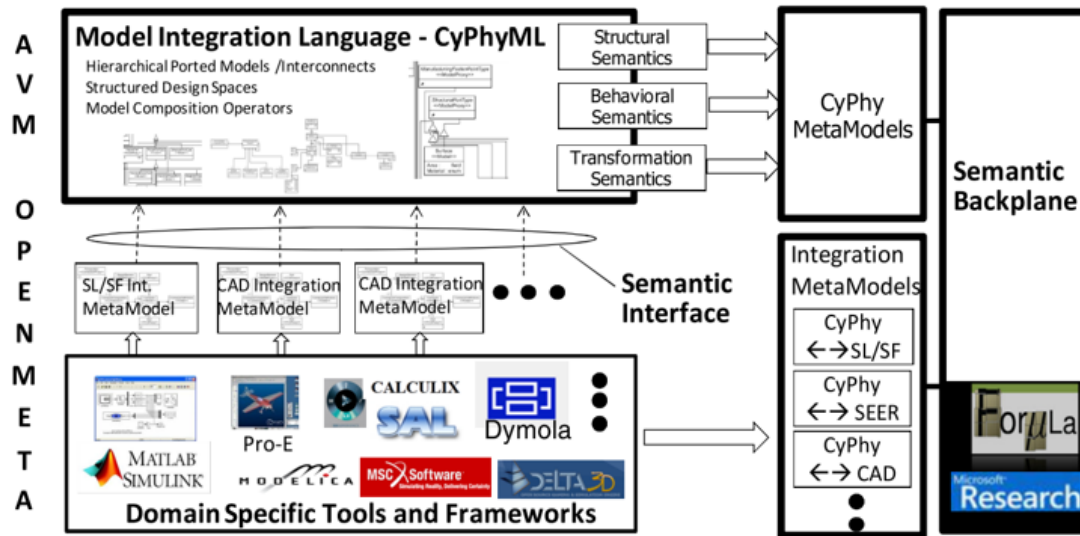


Figure 1.1: Semantic Backplane for Heterogeneous Modeling

Semantic integration in OpenMETA was achieved by the introduction and continuous evolution of the CyPhyML *model integration language* supporting the engineering process of ground vehicle design (Sztipanovits et al., 2014). Since the OpenMETA design flow extended to multiple physical and cyber domains, CyPhyML itself proved to be a complex DSML requiring the use of a meta-programmable modeling tool, GME. To satisfy the need for evolving CyPhyML without sacrificing semantic integrity, the project incorporated the OpenMETA Semantic Backplane (Simko et al., 2012), which provided a FORMULA-based formal specification of CyPhyML as well as the integration models across physical and cyber domains. Although this two-pronged approach of complementing a model engineering tool (GME) with a formal framework satisfied

the basic needs, the challenge of synchronizing the CyPhyML and FORMULA models and meta-models decreased the benefits of the Semantic Backplane.

Microsoft's FORMULA language (Jackson, 2013) supports formal metamodeling and the formal specification of model transformations in the Constraint Logic Programming (CLP) framework (Jackson et al., 2015). While the formalization of integration models and model transformations were crucial in managing complexity in OpenMeta (Sztipanovits et al., 2014), the loose connection between the model engineering tool suite and the FORMULA-based semantic backplane led to a divergence between the engineering models and their formal representation, which created consistency risks (Sztipanovits et al., 2014). The root cause of this divergence was that part of the FORMULA models (formal domain constraints and model transformations) still had to be modeled by hand after the continuously (and rapidly) changing production code written in JavaScript. This introduced a time delay in the formal modeling process and made its consistency with the production system error-prone.

The lack of a deeper semantic integration between the model engineering framework (GME and currently WebGME) and the FORMULA-based semantic backplane has been one of the major motivations for our research effort. The challenges of semantic integration include a fully automated mapping between WebGME-based models and metamodels and their FORMULA specification such that the consistency of structural semantics in the two representations is preserved.

Modeling tools like WebGME (Maróti et al., 2014), EMF (Steinberg et al., 2008), MPS (Bucchiarone et al., 2021) and MetaEdit+ (Kelly et al., 1996) provide services for graphical modeling, while specification languages like FORMULA (Jackson and Schulte, 2013), Alloy (Jackson, 2019) provide the definition of formal semantics and the execution engine for model operations. WebGME is the next generation of Vanderbilt's Generic Modeling Environment(GME) (Sztipanovits and Karsai, 1997) providing many newly designed features such as web-based deployment, version control, real-time collaborative editing, and prototypical inheritance to improve 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 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 querying, testing the well-formedness, visualizing, and analyzing very large models. In WebGME, these functions are implemented by manually designed scripts defined in JavaScript or Python language. while the approach is scalable, for model developers and engineers its use is cumbersome and lacks expressiveness and evolvability, which makes it hard to maintain. Elimination of this gap and moving toward a more declarative approach that scales well to very large models was another key motivation for the deeper integration between WebGME, and graph-based model stores that offer a partial solution for these problems.

The third motivation for the research has emerged from the significant mismatch between the engineering process of modeling and the FORMULA implementation. Engineering modeling is inherently incremental – the design models are subject to continuous updates – while the FORMULA execution engine is not incremental. The result of this mismatch is that after model updates, the entire design model had to be re-compiled into FORMULA code which made the process slow if the model sizes were large. There were two possible solutions for this challenge: to re-implement FORMULA core to be incremental or to delegate the execution of FORMULA models to another formal framework. The second approach requires that precise semantic mapping is established between FORMULA and the target language. Since the FORMULA core engine is highly complex, this latter route seemed to be the best solution. We selected Differential Datalog (DDLog) and Differential-Dataflow as the target, both of which are declarative, efficient, incremental, and even potentially could scale to multi-core execution platforms.

In summary, the key motivations of the proposed research are to address the conflicting requirements of heterogeneity and scaling in model-based design. Addressing heterogeneity demands the use of agile model integration languages without compromising semantic precision. This need can be satisfied by the integration of formal frameworks with model-engineering tools. However, real-life applications lead to very large model sizes that make scalability a primary concern. The need for scalability motivates two additional steps: (1) introducing formal frameworks that are incremental and (2) providing a bridge toward graph-based model stores that offer languages and tools for high-performance queries, analytics, and visualization.

1.2 Background

In this section, we briefly summarize the key concepts and introduce the related tools in our research work to get new readers familiar with the tools before diving into the integration of WebGME, FORMULA, graph database, and Differential-dataflow computation model.

1.2.1 WebGME

WebGME - Web Graphical Modeling Environment (Maróti et al., 2014) is an open-source, web-based platform for collaborative model-driven engineering, used to create and manipulate models of systems, software, and other complex artifacts. It allows users to create custom modeling languages, define models, and execute generated code or models in real time. Figure 1.2 depicts the common workflow of WebGME design studio and figure 1.3 is an example of how models from ROSMOD (Kumar et al., 2016) project are visualized in WebGME.

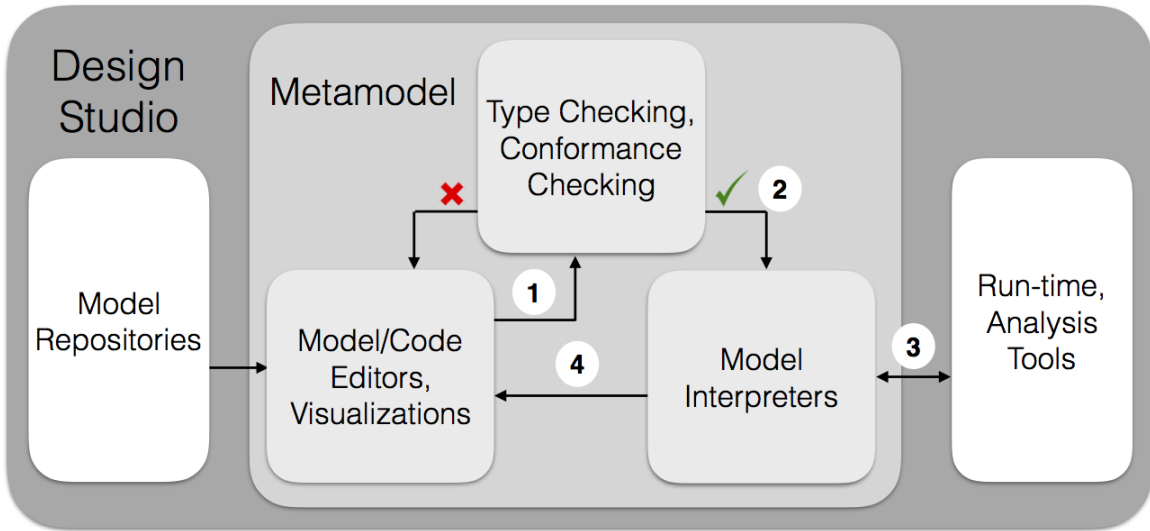


Figure 1.2: WebGME Design Studio Workflow

WebGME provides a graphical interface for creating and manipulating models using a drag-and-drop approach, allowing users to easily create and modify complex systems, and collaborate with others on the same project. It also includes tools for version control, automated testing, and project management. WebGME is built on top of Node.js and MongoDB and can be customized and extended using JavaScript. It is used in a variety of industries, including aerospace, automotive, and defense, to develop complex systems and software.

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. The end users have to manually implement a plugin in general-purpose programming languages such as JavaScript or Python to do more complex constraint checking beyond the simple built-in conformance checking in WebGME diagrams.

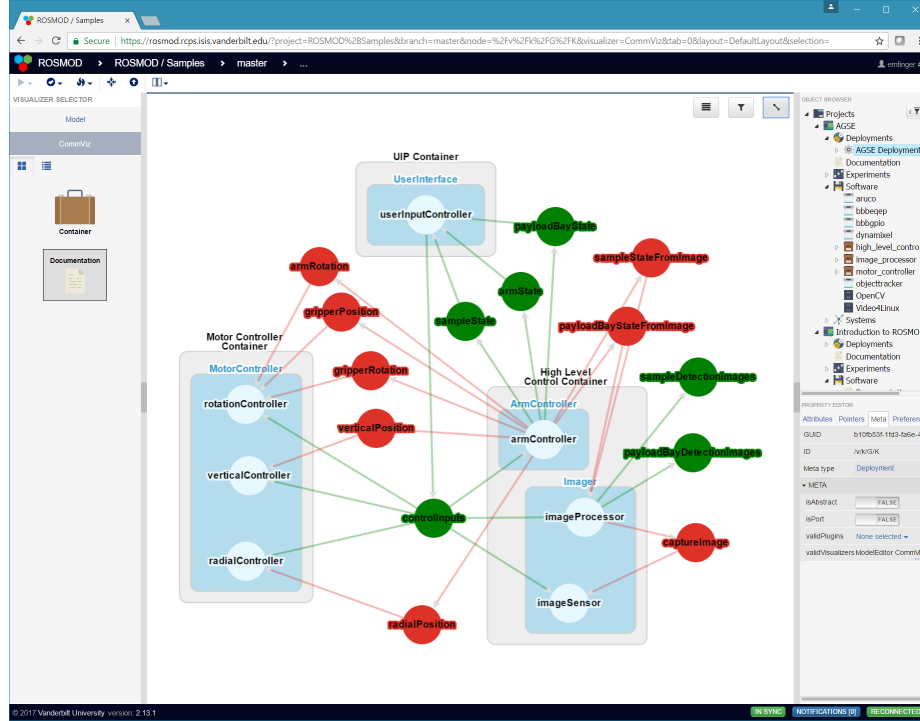


Figure 1.3: ROSMOD Model Visualization in WebGME

1.2.2 FORMULA Language

FORMULA (Jackson and Schulte, 2013) is a constraint logic programming language based on fixed-point logic over algebraic data types. FORMULA can deduce a set of final facts that is the least fixed-point solution given an initial set of facts specified in algebraic data types and a set of inference rules to execute,

In FORMULA language, the structure of models, validation rules, and transformation rules are specified using algebraic data types (ADTs) and Constraint Logic Programs (CLPs). We briefly introduce the features of FORMULA as a CLP-based modeling language for specifying DSMLs with the extended examples in (Sztipanovits et al., 2014),

The *Deployments* domain formalizes the following cross-domain problem: There are services and services that can be deployed to a node. Services are in conflict if deployed to the same node at the same time. The last line of code in *Deployments* domain is a rule with set comprehension to find out the number of nodes that have more than one service deployed to them.

FORMULA also has the concept of inheritance that any domain or even model can be inherited by another domain and even multiple times under different namespaces. For example, the *SessionedDeployments* domain can be constructed by gluing together three independent domains, in which two of them are from the same *SessionedDeployments* domain but under different scope or namespace, so $D1.Node(1)$ and $D2.Node(1)$ are

considered to be two different nodes from different sessions because they are under different scopes but they can still coexist and belong to the same newly defined type *Deploy* under *SessionedDeployments* in which $Deploy ::= D1.Deploy + D2.Deploy$ and the new *Deploy* type has a completely different meaning compared with *Deploy* under *Deployments* domain.

Additional validation rules and data types are added to detect across-session conflict, which aims to find conflict between services from two different sessions of deployments. Both type *Element* from *Deployments* and type *Deploy* from *SessionedDeployments* are algebraic union types that represent the union of multiple sets of terms, which is a unique feature that gives FORMULA more power of expression than other specification languages and basically any combination of more than one term can form a unique type in FORMULA.

domain Deployments

```
{
  Service ::= new (name: String).
  Node    ::= new (id: Natural).
  Element ::= Service + Node.
  Conflict ::= new (s1: Service, s2: Service).
  Deploy  ::= fun (s: Service => n: Node).
           conforms no { n | Deploy(s, n), Deploy(s', n), Conflict(s, s') }.
}
```

domain Session { Timestamp ::= new (time: String). }

domain SessionedDeployments extends Session,

```
  Deployments as D1, Deployments as D2
{
  Deploy ::= D1.Deploy + D2.Deploy.
  DeploymentSession ::= new (t: Timestamp, deployment: D).
  AcrossSessionConflict ::= new (s1: D1.Service, s2: D2.Service).
  AcrossSessionConflict(s11, s22) :- D1.Conflict(s11, s12), D2.Conflict(s21, s22),
    s12.name = s21.name.
}
```

Listing 1.1: FORMULA Domain Composition and Inheritance

Models are represented simply as sets of well-typed trees created from domain data types. Model modules hold the set of trees to represent the instances that conform to the data structures and rules defined in the domain modules. Partial models denoted by the keyword *partial* before the model name are partially closed

domains in that parts of the facts are not given and are subject to the FORMULA model synthesis to find solutions. For example, the *UnfulfilledDeployments* model requires two services to be deployed to at most 10 nodes and the total number of services and nodes must be less than 20. A domain can also be viewed as a partial model and both of them are Open Logic Programs (OLPs) that a model closes the domain or partial model with a set of facts. There could be an infinite set of models that solves it and the execution of model synthesis on OLPs returns the result of finding solutions in the search space with proofs.

```
model Undeployed of Deployments
```

```
{  
    sVoice is Service("In-car_voice_recognition").  
    sDB    is Service("Dashboard_UI").  
    n0     is Node(0).  
    n1     is Node(1).  
    Conflict(sVoice, sDB).  
}
```

```
model Good of Deployments extends Undeployed
```

```
{  
    Deploy(sVoice, n0).  
    Deploy(sDB, n1).  
}
```

```
model Bad of Deployments extends Undeployed
```

```
{  
    Deploy(sVoice, n0).  
    Deploy(sDB, n0).  
}
```

```
partial model UnfulfilledDeployments of Deployments
```

```
{  
    s1 is Service(_).  
    s2 is Service(_).  
    atleast 10 Node.  
    conforms count({ e | e is Element }) < 20.  
}
```

```
}
```

Listing 1.2: FORMULA Model Composition and Inheritance

Transforms are CLPs that transform models between domains. They are useful for formalizing changes in abstractions (such as compilers) and for projecting large integrated models into consistent submodels that can be fed to domain-specific tools. Below is a simple example that compiles Deployment models into Configuration models that can be directly translated into configuration files. A new domain *NodeConfigs* is defined to model the configuration files, in which each node location is associated with a list of services deployed on it. The transformation rule in the transform module *Compile* will gather all services deployed on the same node to a list and return the list together with the id of the node. Model transformation is essentially the concrete execution of model facts to derive new model facts that conform to the constraints in the target domain.

```
domain NodeConfigs
{
  Config ::= fun (loc: Natural -> list: any Services + { NIL }).
  Services ::= new (name: String , tail: any Services + { NIL }).
}

transform Compile (in :: Deployments) returns (out :: NodeConfigs)
{
  out.Config(n.id , list) :- n is in.Node ,
  list = toList(out.#Services , NIL , { s.name | in.Deploy(s , n) }).
}
```

Listing 1.3: FORMULA Model Transformation

The FORMULA example below is for the demonstration of some of the FORMULA advanced features that are not only unavailable in other modeling languages but also difficult to implement in a general-purpose programming language. *SortedDeployments* is the new domain that inherits *Deployments* domain with extensions by attaching a number to each deployment to indicate the priority. The first four rules regarding *Ordering* compute all orderings of the deployments by swapping two deployments in the list of 4 deployments and the ultimate goal is to find a sequence that cannot be sorted due to the existence of conflicts between deployments.

Note that this problem has negation, recursion, and even nested set comprehension that is part of the rule while the constraint inside the same set comprehension also has set comprehension itself. The execution of

rules on a concrete model is not hard but solving a partial model of this domain such as model *Unsorted* is an undecidable problem.

```

domain SortedDeployments extends Deployment
{
    PrioritizedDeployment ::= new (d: Deploy, priority: Integer).
    Initial :: new (d1: PrioritizedDeployment, d2: PrioritizedDeployment,
                  d3: PrioritizedDeployment, d4: PrioritizedDeployment).
    Ordering :: new (d1: PrioritizedDeployment, d2: PrioritizedDeployment,
                  d3: PrioritizedDeployment, d4: PrioritizedDeployment).

    Ordering(a, b, c, d) :- Initial(a, b, c, d).
    Ordering(b, a, c, d) :- Ordering(a, b, c, d), a.priority > b.priority.
    Ordering(a, c, b, d) :- Ordering(a, b, c, d), b.priority > c.priority.
    Ordering(a, b, d, c) :- Ordering(a, b, c, d), c.priority > d.priority.

    Unsortable(w, x, y, z) :- Initial(w, x, w, z),
        no { w, x, y, z | Ordering(w, x, y, z),
            no { n | Deploy(s, n), Deploy(s', n), Conflict(s, s') },
            x < y, y < w, w < z }
}

partial model Unsorted of SortedDeployments
{
    atleast 10 PrioritizedDeployment.
    Unsortable(w, x, y, z)
}

```

Listing 1.4: FORMULA Example for Advanced Usage

1.2.3 Semantic Backplane

A key challenge of OpenMETA (Sztipanovits et al., 2014) was the heterogeneity of model-based design flows for CPS. The META design flow introduces heterogeneity in multiple dimensions:

1. Heterogeneity caused by multiple physical domains (structural, mechanical, electrical, hydraulic, pneu-

matic, and others).

2. Abstraction heterogeneity across implementation layers (continuous/hybrid dynamics, logical time dynamics (automata models), discrete event dynamics).
3. Heterogeneity across behavioral abstractions developed for describing the same dynamic phenomenon (e.g. hybrid dynamics abstracted to concurrent state machines using precise relational abstractions).

As it is shown in the figure, the range of the modeling domains across these design dimensions is not closed: they continuously evolve driven by the needs of the products and the abstractions provided by the design tools. The introduction of a flexible model integration language connects the product-specific design models to the tool-specific analysis models. The role of the Semantic Backplane is to make this "semantic integration" sound by representing the semantics of the model integration language (CyPhyML), the semantic interfaces to the tool-specific semantic interfaces, and the required model transformations in a formal framework - FORMULA.

1.2.4 Differential-Datalog and Differential-Dataflow

Differential Datalog (DDLog) developed by Ryzhyk and Budiu (2019) is a programming language designed for incremental computations, where it is expected that the program input is continually changing. Programmers specify the input-output relationship using a syntax that resembles logic programming, and the DDlog compiler synthesizes an efficient, incremental implementation of this relationship.

DDLog operates on typed relations using a set of logic programming-style rules that are evaluated to incrementally perform complex computations defined in a DDlog program. The DDlog language has a strong type system that extends the traditional Datalog language with support for Haskell-style tagged union types, infinite precision primitive types, generic types, and built-in collection types, including maps, sets, arrays, and even functions as first-class objects.

Under the hood, a DDLog program is compiled to Differential Dataflow (DD), which is an incremental streaming data processing system that supports a wide range of relational operators including *map*, *filter*, *join*, *antijoin*, *recursion* (nested iteration with self-loops), *reduction* (aggregation), *union* (concatenation of collections) and *distinct*. Differential Dataflow uses a computation model called *differential computation* that allows states (snapshots of the results of computation) to vary according to a partial ordering of different versions and maintains an index of updates so the next state can be efficiently derived by combining partially ordered versions in different ways. The output of the compilation is a DD graph of differential dataflows in which each node is a DD operator connected to other operators with a highly optimized implementation for incremental computation.

1.3 Contribution

The contributions are the following:

1. Integration of WebGME, FORMULA, and graph database on three different levels. 1) Direct translation from WebGME to FORMULA 2) WebGME modeled as a graph in FORMULA with automatically derived conditions to reason over the models. 3) Integration of FORMULA and graph database with rule execution in graph queries.
2. Formal specification of Differential-Datalog and FORMULA languages with identification of syntax and semantics mismatch between FORMULA and DDLg. Support type inference and static analysis of the FORMULA program by constraint checking in logic programming.
3. The implementation of Differential-FORMULA, a tool based on FORMULA language that supports incremental evaluation, by the model transformation in bootstrapping style to automatically generate an equivalent DDLg program that executes rule incrementally.
4. Formal specification of various engineering domains with benchmark comparison between our integrated modeling framework and other high-performance state-of-art modeling frameworks.

The contribution does not address the symbolic execution option of FORMULA, which is an essential element for model synthesis with Z3. In this sense, Differential-FORMULA will be a subset of the FORMULA leaving out the partial model construct.

1.4 Organization

Chapter 2 summarizes the research challenges in dealing with the scalability and performance issues in large models. Chapter 3 lists all the related works about graph-based modeling frameworks and the latest research on incremental modeling tools. Chapter 4 introduces our integrated tools of WebGME, FORMULA, and graph database on three different levels. Chapter 5 introduces Differential-FORMULA and the details about how we use model transformation to generate an incremental version of the FORMULA program in DDLg that not only has better performance in model execution but also executes incrementally. Chapter 6 is about future work and open opportunities regarding the integration of modeling frameworks and scalability issues.

CHAPTER 2

Challenges in Semantic Integration for Modeling

2.1 General Research Challenges in Model-driven Engineering

Model-driven engineering (MDE) is an approach to software development that emphasizes the use of models to specify, design, and implement software systems. While MDE offers many potential benefits, such as increased productivity, quality, and maintainability, it also presents several challenges for researchers. Here are some of the general research challenges in model-driven engineering:

1. **Scalability:** As software systems become larger and more complex, models can become unwieldy and difficult to manage. Researchers need to develop techniques to manage the complexity of models and make them scalable.
2. **Verification and Validation:** Models need to be verified and validated to ensure that they accurately represent the desired system behavior. Researchers need to develop techniques and tools to automate these processes and to ensure the correctness of the models.
3. **Tool Integration:** MDE involves the use of multiple tools for modeling, analysis, and code generation. Researchers need to develop techniques to integrate these tools seamlessly and to ensure that they work together effectively.
4. **Language Design:** The design of modeling languages is critical to the success of MDE. Researchers need to develop techniques to design languages that are expressive, easy to use, and support a wide range of modeling tasks.
5. **Human Factors:** The success of MDE depends on the ability of developers to create and manipulate models effectively. Researchers need to understand the cognitive processes involved in modeling and develop techniques to support developers in these tasks.
6. **Evolution and Maintenance:** Software systems evolve over time, and models need to be updated to reflect these changes. Researchers need to develop techniques to manage the evolution and maintenance of models and to ensure that they remain valid and accurate.
7. **Domain-specific challenges:** MDE is often applied in specific domains, such as automotive or health-care, which present their own unique challenges. Researchers need to understand these domain-specific challenges and develop techniques to address them effectively.

In summary, model-driven engineering presents several research challenges that need to be addressed to realize its potential benefits fully. These challenges range from the design of modeling languages to the integration of tools and the management of model evolution and maintenance. Addressing these challenges will require interdisciplinary research efforts that combine expertise in computer science, software engineering, cognitive psychology, and other fields.

2.2 Challenges in Graph-based Semantic Integration

We aim to describe the semantics of a metamodeling language with an abstract, concise and high-level language that enables efficient reasoning over hierarchically complex data structures with high performance. Graphs provide such an abstract language and have been studied extensively for representing metamodels and models, but there are several problems that have to be addressed before applying graph-based reasoning and execution to our integrated modeling framework.

2.2.1 Integration of Models, Semantics, Storage, and Execution

Most modeling tools do not have a unified framework for representation, formally defined semantics, storage, and model execution all integrated seamlessly. The concept of this “Semantic Bridge” for integrated modeling framework is shown in Figure 2.1. The design process with graphical syntax is usually handled by MBSE tools such as WebGME on the left side with a specification language such as FORMULA to specify the formal semantics of DSMLs defined previously.

On the right-hand side, another bridge or link must be established to precisely map the formally defined semantics to a low-level language or representation for the purpose of fast execution and analytics. The challenge is to create a semantically sound link between our engineering-oriented modeling framework and graph-based modeling frameworks, such as TinkerPop (TinkerPop, 2020), not only for the mappings of models but also for the mappings of their query languages.

Before one language is translated into another language, the language should be formally defined no matter it is a general-purpose programming language, modeling language or a specification language like FORMULA used to specify other languages. FORMULA is a specification language to create DSMLs and verify the models of DSMLs, but the FORMULA language itself and its compiler are not formally defined. A formal specification of the FORMULA language will facilitate the translation to other tools and avoid semantic mismatches or errors.

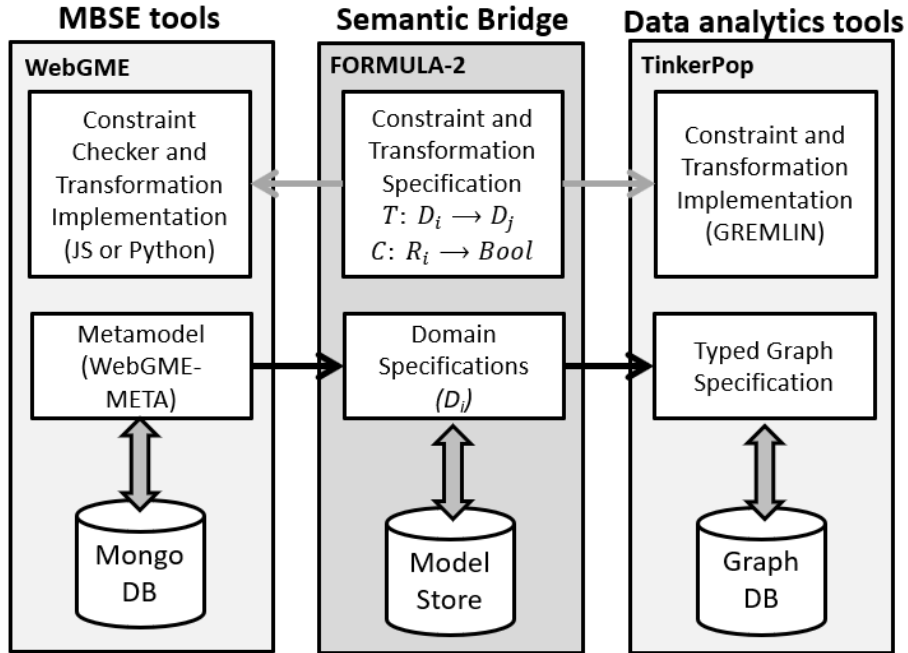


Figure 2.1: Graph-based Integrated Modeling Framework

2.2.2 Unified Graph Representation in Modeling Framework

Graph representation and graph database are currently underutilized for many modeling activities: because models and metamodels can be represented as typed graphs, they can take advantages of the efficient storage and fast queries offered by graph databases or even graph transformation using the existing highly optimized graph algorithms implementation. However, the graph representation does not exist in either the WebGME or FORMULA tools as a foundation to represent their unique metamodels and models. The data analytics tools on the right side of Figure 2.1 are only available with metamodels and models represented in graphs.

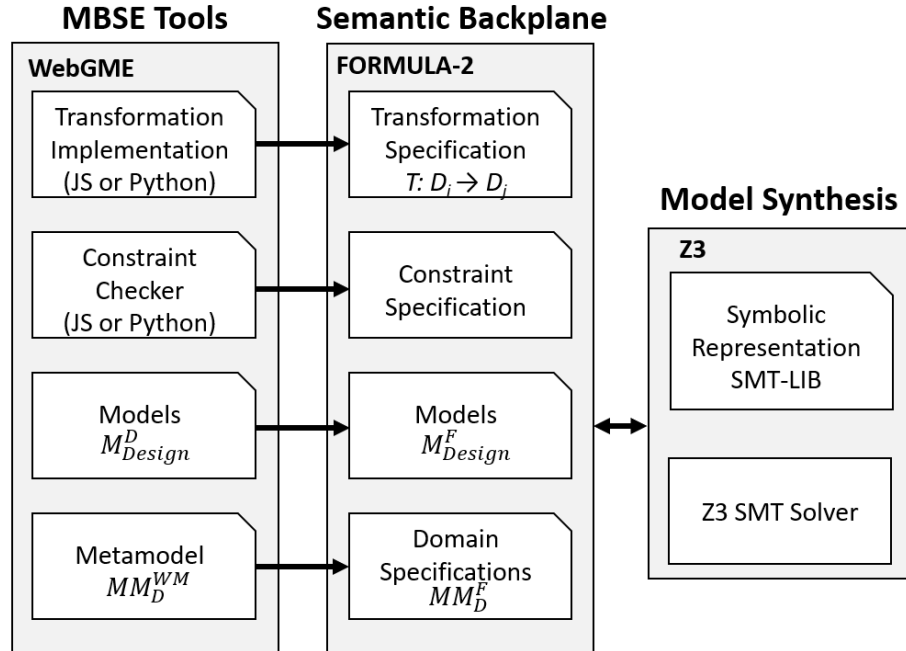


Figure 2.2: WebGME with FORMULA-based Semantic Backplane

2.2.3 Scalability Issues on Large Models

Scalability is the grand challenge for modeling frameworks to handle large models. WebGME and FORMULA integration does not scale well because separately they have their own performance issues and bottlenecks as shown below.

WebGME - has slow performance for model execution with manually written JavaScript plugins for constraint checking and may freeze the browser on large models because the web browser is not suitable for the rendering of huge amounts of hierarchical data structures and heavy computation on a single thread in JavaScript may block other tasks running in the browser. WebGME is only designed for the graphical representation of medium-size models and is not an ideal tool for direct model execution.

FORMULA - includes its own query execution engine, but does not scale to large models. The execution time grows exponentially very quickly due to the inefficient way to maintain the cache and intermediate results generated for traces and proofs. The FORMULA rules with more than two matching patterns in the body are transformed into a set of new intermediate rules, in which each has exactly two matching patterns, to index the intermediate results of pattern matching but the downside of this approach is that a large quantity of auto-generated intermediate cached results is maintained in the memory temporarily for each session of model execution. Based on the results of our experiment (Zhang et al., 2019), FORMULA struggles with large model execution and can run out of memory on large examples.

2.3 Problems and Challenges in Incremental Modeling

There was a significant mismatch between our modeling process and the semantics of the existing FORMULA implementation. Engineering modeling is inherently incremental – the design models are subject to continuous updates – while the FORMULA execution engine was not incremental. The result of the mismatch was that after each model update, the entire design model had to be reloaded into FORMULA, which made the process quite slow if the model sizes were large. There are three main incrementality-related issues that need to be addressed with our FORMULA-based integrated modeling framework to improve this process:

1. The inherent incremental nature of the modeling process requires model queries and transformations to be repeatedly performed on incrementally updated models. We want to perform only the required minimum computation and incrementally respond to the input change rather than re-run the entire execution.
2. The scalability issues regarding execution on large models in the real world hinder the adoption of FORMULA in large modeling projects especially in industrial use cases.
3. The identification of the delta model that represents the changed part in the model and the tracking of the propagation of changes since the very beginning of model input. We assume the model including the intermediate results in a complex scenario is not static but dynamically changes according to the rules or analytics strategies that have been executed over the model.

CHAPTER 3

Related Work

Model-driven engineering (MDE) is a software development approach that emphasizes the use of models to describe and specify the behavior of software systems. MDE has gained popularity in recent years due to its potential benefits, such as improved productivity, quality, and maintainability of software systems. However, MDE also presents several challenges that need to be addressed to realize these benefits fully. To address these challenges, researchers in the field have developed various techniques, tools, and languages for MDE.

In this section, we will review some of the related works in MDE. We will start by discussing the evolution of MDE and its key concepts, including metamodeling, model transformation, and code generation. We will then discuss some of the techniques and tools developed for MDE, including modeling languages, verification and validation techniques, and tool integration frameworks.

The related works will mainly focus on the frameworks and tools that apply graph-based solutions and various incremental approaches to address the scalability problems in MDE. We will discuss some of the application domains of MDE and how it has been applied in practice to develop software systems in various domains.

3.1 Generic Modeling Engineering Frameworks and Tools

UML and SysML are two commonly used modeling languages that adapt and evolve around one generic metamodel with many common concepts. While the generic modeling frameworks can be categorized into five different categories based on their definition of meta-metamodels.

1. ARIS - The Architecture of Integrated Information Systems (Scheer and Schneider, 1998) is an approach to enterprise modeling. The associated tool supports by default different modeling notations. Users can adapt already existing languages and the vendor can create completely new languages.
2. Ecore (Steinberg et al., 2008) is the meta-metamodel in the Eclipse Modeling Framework (EMF). The framework supports the development of (Eclipse) applications with a focus on Java programming language. Many model processing tools and extensions based on Ecore such as ATL for model transformation exist.
3. GME - The Generic Modeling Environment (Sztipanovits and Karsai, 1997) is primarily a tool for domain-specific modeling in the area of electrical engineering and cyber-physical system.

4. GOPPRR - Graph, Object, Port, Property, Role, Relationship (Kelly et al., 1996) is the metamodeling language in Meta Edit+. The tool supports typical DSM tasks such as (meta) modeling and code generation.
5. MS DSL Tools - The Microsoft Domain-Specific Language Tools (Kosar et al., 2007) enable the definition of languages and generators in Visual Studio.
6. MPS (Bucchiarone et al., 2021) is a projectional language workbench, meaning no grammar and parser are involved. Instead, an editor allows changing directly the underlying abstract syntax tree, which is projected in a way that looks like text. MPS supports mixed notations (such as textual, symbolic, tabular, and graphical) and a wide range of language composition features based on the *BaseLanguage*, which is the MPS's meta-metamodel. MPS users extend this BaseLanguage to define their own domain-specific modeling languages.

The comparison of meta-metamodel expressive power can be found in (Kern et al., 2011). GOPPRR and GME are the most powerfully expressive meta-metamodels and Visio has the least expressive power of all metamodeling languages. However, other criteria (e.g. usage, standardization, or tool features) play a crucial role in the selection of a certain metamodeling approach.

There are several modeling frameworks that are designed to handle scalability issues of model transformation and constraint checking on large, complex CPS-related models for industrial use cases. Most of them target frameworks based on Eclipse Modeling Framework (EMF) with customized extensions. The most significant frameworks include NeoEMF (Daniel et al., 2016b), EMF-IncQuery, and its successor Viatra framework (Varró and Balogh, 2007). The modeling frameworks in the EMF family are based on the traditional UML/OCL technical stack with metamodels defined in EMF's Ecore both graphically and textually but some of them extend the ECore and have their own query or transformation languages such as Viatra's *VQL* language and execution engine.

3.2 Modeling Framework with Formal Semantics

Frameworks with graphical modeling provide an intuitive model representation but most of them do not have formal semantics well defined for syntax in metamodels and models.

A modeling framework with formal semantics is a framework that provides a precise and unambiguous specification of the meaning of the models created within the framework. This allows for automated reasoning and analysis of the models, as well as verification and validation of the models against desired properties.

Event-B (Abrial, 2010) is a formal modeling language and method that provides formal semantics based on set theory and first-order predicate logic. It is useful for specifying and analyzing complex systems, and it

supports automated theorem proving.

TLA+ (Merz, 2008) is a formal specification language and toolset that provides formal semantics based on set theory and temporal logic. It is useful for specifying and verifying complex distributed systems, and it supports model checking and theorem proving.

Z (Spivey and Abrial, 1992) is a formal specification language and tool-set that provides formal semantics based on set theory and predicate logic. It is useful for specifying and analyzing complex systems, and it supports automated theorem proving.

Alloy (Jackson, 2019) is a well-known specification language that is based on first-order relational logic extended with some form of the Kleene closure operator. Alloy has integrated SAT solvers that allow proof generation and model synthesis. Nevertheless, Alloy does not have the expressiveness of logic programming, and as shown in (Jackson et al., 2010) and FORMULA can outperform Alloy in model generation tasks.

The K-framework (Roşu and Şerbănuţă, 2010) is a rewrite-based executable semantic framework in which programming languages, type systems, and formal analysis tools can be defined using configurations and rules. The K-framework specializes in defining the semantics of general-purpose programming languages but does not support incremental updates on large models or programs.

P language (Desai et al., 2013) is not a modeling framework but an application of our FORMULA specification language to specify a DSL and its compiler is also based on model transformation in the same manner as our Differential-FORMULA tool that compiles FORMULA program to DDLog program. The P compiler is one of the largest and most complex use cases of FORMULA in developing the original compiler for the *P Language*, a domain-specific language for specifying, testing, and implementing asynchronous distributed systems. That compiler translated a model in the P language to a corresponding implementation in the C programming language. Unfortunately, the performance of the FORMULA compiler and execution engine prevents FORMULA to be applied to larger industrial use cases. An incremental version of the transformation engine would have improved the original compiler's performance on code generation tasks.

3.3 Graph-based Modeling Frameworks

Gwendal Daniel and Gerson Sunye proposed and implement *UMLtoGraphDB* framework in (Daniel et al., 2016b) to map UML conceptual schemas to graph databases and also translate OCL schemas to an abstraction layer on top of various graph databases for constraint checking. While *UMLtoGraphDB* seamlessly maps conceptual schema to graph databases via an intermediate Graph metamodel, it still lacks a solid semantic foundation with formal specification. A similar work of graph-based semantics for UML class and Object diagrams was done by Anneke Kleppe and Arend Rensink in (Kleppe and Rensink, 2008) to give unambiguous, formal semantics to UML using the theory of graphs and formalize UML/OCL concepts in mathematical

arguments.

In the EMF world, Eclipse MDT OCL (Cabot and Gogolla, 2012) provides an execution environment to evaluate OCL invariants and queries over models. EMF-Query is a framework that provides an abstraction layer on top of the EMF API to query a model. It includes a set of tools to ease the definition of queries and manipulate results. These two solutions are strongly dependent on the EMF APIs, providing on the one hand an easy integration in existing EMF applications, but on the other hand, they are unable to benefit from all performance advantages of NoSQL and graph databases due to this API dependency.

Another EMF-based modeling framework that has decided to move to graph database as a backend in recent years is Emoflon (Weidmann and Anjorin, 2021). Emoflon::Neo rewrites a huge part of its backend engine for the integration with a graph database and even has a rule specification language eMSL, the eMoflon Specification Language, a family of modeling languages with a uniform textual and visual concrete syntax supported by an Xtext-based editor. The rule compiler will compile eMSL rules to Cypher graph queries based on TGG-based operations.

The work of *UMLtoGraphDB* work later evolved to a new framework named NeoEMF (Daniel et al., 2016a) that inherits EMF concepts but with an extension for the execution engine delegated to graph database and graph traversal language. NeoEMF framework has two components Mogwai and Gremlin-ATL that respectively translate OCL queries and ATL transformation rules to graph queries written in Gremlin Traversal Language. They both bypass the EMF APIs limitation to achieve huge performance gains on query execution with equivalent, optimized graph queries and less memory overhead to store intermediate results. However, NeoEMF does not support the translation of recursive calls and transitive closure operators specified in the latest OCL specification. ATL specification language for model transformation has only limited support for recursive patterns, while FORMULA allows horn clauses with recursion for model transformation.

EMF-IncQuery (Varró and Balogh, 2007) is an incremental pattern matching framework to query EMF models essentially based on graph transformation. It also bypasses the API limitations using a persistence-independent index mechanism to improve model access performance. It is based on an adaptation of the RETE algorithm, which was developed to efficiently apply many rules or patterns to many objects, or facts, in a knowledge base. The downside is that caches and indexes must be built for each query, implying a non-negligible memory overhead compared to other graph-based frameworks.

3.4 Bootstrapping Modeling Frameworks

The concept of meta-metamodel such as Ecore (Steinberg et al., 2008) for EMF is inherently bootstrapping in that it is used to explain its own classes and entities in a self-defining way. However, Ecore does not support modeling the operation logic or execution semantics. For constraint evaluation and transformations, we can

easily attach methods implemented in Java to the model and only the operation interface can be defined but not the concrete implementation.

The main purpose of modeling frameworks and tools is to design a Domain Specific Modeling Language (DSML) that is executable while the implementation of the execution engine itself is usually done by writing low-level code in a general-purpose programming language such as Java and C++. Modeling frameworks have their own specification languages or meta-languages to define DSL and as a matter of fact, the meta-languages are also just languages and therefore can be defined using themselves. The process is called bootstrapping and is often difficult to achieve. Although MDE methodologies are inherently self-descriptive and higher-order, very few of the actively developed MDE tools are bootstrapped. There are only a few MDE tools that are implemented in a bootstrapping way.

Emoflon (Leblebici et al., 2014) is one of the success stories that the developers developed Emoflon using Emoflon itself. Only some of the core components not all parts in eMoflon are implemented as both a unidirectional and bidirectional model transformation with eMoflon. To be more precise, A transformation language L_i is compiled to a lower-level language L_{i-1} with a compiler written in L_{i-1} . This corresponds to TGGs (Schürr and Klar, 2008) being compiled to SDMs (Zündorf et al., 1999) with SDMs in a similar way as in our Differential-FORMULA that FORMULA program being compiled to Differential-Datalog with Differential-Datalog.

MPS - Meta Programming System (Campagne, 2014) from JetBrains is also a bootstrapping meta-modeling language. The bootstrap of MPS includes a definition of almost complete Java called Base Language. It provides languages for collections, dates, closures, and regular expressions. All meta-languages of MPS are defined in MPS, with languages for structure, editor, constraints, type system, and generator. The new languages generated by MPS can be directly executed in Java without writing additional code.

DMLA - The Dynamic Multi-Layer Algebra (Mezei et al., 2019) is another bootstrapping metamodeling tool. DMLA is a multi-layer metamodeling formalism with a sound mathematical background. It is designed as a bootstrap of model entities that are both self-describing and self-validating by design. One of the unique features of DMLA is that it has a built-in operation language, which makes the underlying ASM functionality available for Bootstrap.

3.5 Incremental Modeling

Model query and model transformation are the two main parts of model-driven engineering. Most frameworks have two separate languages for query and transformation while others have a unified language for both model operations. We list several mature and well-maintained incremental modeling frameworks for model transformation that are used in both academia and industrial use cases: ATL (Jouault et al., 2008), VI-

ATRA (Varró and Balogh, 2007), eMoflon (Anjorin et al., 2011), YAMTL (Boronat, 2022) and NMF (Hinkel, 2016).

3.5.1 OCL-based Incremental Model Query

There are several incremental OCL-based modeling frameworks available that can be used for developing software systems. Some of the popular ones are:

Cabot (Cabot and Teniente, 2009) presents an advanced three-step optimization algorithm for incremental runtime validation of OCL constraints that ensures that constraints are re-evaluated only if changes may induce their violation and only on elements that caused this violation. The approach uses promising optimizations, however, it works only on Boolean constraints, therefore the incrementality it achieves is very limited and it is less expressive than other techniques.

EMF-IncQuery (Ujhelyi et al., 2015) is a model query and transformation tool that can be used for incremental model-based software engineering. It is based on the Eclipse Modeling Framework (EMF) and provides a set of APIs and tools for querying and manipulating models using OCL.

Kermeta (Falleri et al., 2006) is a language and toolset for incremental model-based software engineering. It is based on the Eclipse Modeling Framework (EMF) and provides a set of APIs and tools for defining and manipulating models using OCL.

The Algebraic Model Weaver (AMW) (Marcos et al., 2005) is an incremental model-based software engineering tool that is based on algebraic specifications. It provides a set of APIs and tools for defining and manipulating models using OCL.

3.5.2 ATL-based Incremental Model Transformation

Many incremental frameworks resort to lazy computation that starts to re-compute only when it is necessary with certain conditions triggered. ATL is the most well-known model transformation language integrated into EMF but it is not incremental. ATL (Jouault et al., 2008)'s incremental execution mode enables the incremental forward propagation of model changes to target models.

The Atlas Transformation Language (ATL) is a model transformation language that can be used for incremental model-based software engineering. It is based on the Eclipse Modeling Framework (EMF) and provides a set of APIs and tools for transforming models. EMFTVM (EMF Transformation Virtual Machine) is the virtual machine that executes ATL transformations.

ReactiveATL (Martínez et al., 2017) is a new engine for executing ATL model transformation. ReactiveATL builds on the expression injection mechanism to detect which parts of an ATL transformation need to be executed and on lazy evaluation to defer computation in response to update notification of model elements.

However, ReactiveATL only supports a subset of ATL language skipping rule inheritance and multiple source elements. ATOL (Martínez et al., 2017) is a recent incremental compiler for ATL built upon AOF(Active Operation Framework) (Martínez et al., 2017) to implement incremental execution. ATOL also only supports a subset of ATL language with unique lazy rules.

YAMTL (Boronat, 2022) uses an internal domain-specific language of Xtend, like VIATRA, to define a declarative model transformation, widely inspired by the ATL constructs. The specification consists of a set of rules that are mainly composed of multiple input/output element declarations. Rules are declared with, at least, one matched input element from the source model.

YAMTL uses the design of a forward model change propagation procedure for executing model transformations in an incremental mode that can handle documented model changes, called change scenarios, i.e., documents representing a change to a given source model. Such documents are defined with the EMF Change Model, both conceptually and implementation-wise, guaranteeing interoperability with EMF-compliant tools.

3.5.3 TGG-based Incremental Model Transformation

Giese (Giese and Wagner, 2009) presents a triple graph grammar (TGG) based model synchronization approach, which incrementally updates reference (correspondence) nodes of TGG rules, based on notifications triggered by modified model elements. Their approach share similarities with the RETE-based algorithm for an expert system that maintain partially matched patterns. This approach is also similar to the original FORMULA execution engine that generates a lot of new intermediate rules and maintains a huge quantity of intermediate partially matched patterns, which may slow down the execution and consume more memory.

MoTE (model transformation engine) is based on TGGs that consists of a set of transformation rules specifying the simultaneous production of two models of a source and a target language, related by a third model composed of a set of correspondence nodes between model elements of the source and target domains (traceability model). MoTE uses an Operational Rules Generator to compile TGG rules into Story Diagrams (Zündorf et al., 1999) of the SDM (story-driven modeling) tool. Then, a TGG Engine transforms a source model into a target model by invoking the Story Diagram Interpreter in order to execute the appropriate story diagrams for the requested transformation (batch/synchronize or forward/mapping/backward)

eMoflon is another incremental TGG-based model transformation tool that provides a textual concrete syntax to define TGG rules and a set of "Prolog-like" attribute conditions to assign and constrain attributes. eMoflon::IBeX (Anjorin et al., 2011) is a TGG incremental bidirectional model transformation, that relies on an incremental graph pattern matcher. eMoflon generates a set of separate patterns from each TGG rule, representing the context to be matched, elements to be deleted, and elements to be created. When performing the transformation, a pattern invocation network is used to represent the patterns to be matched and structured

in a network (a graph with nodes as patterns and edges as pattern invocations). Then, an incremental pattern matcher produces match events, signaling when new matches appear (for created elements), and when old matches are violated (for deleted elements). All these matches are collected, and the transformation rules are applied.

3.5.4 Graph-based Incremental Model Transformation

Incrementality is supported on two graph-based modeling frameworks Viatra (Varró and Balogh, 2007) and IncA (Szabó et al., 2018), and both tools are based on graph transformation and sub-graph pattern matching in logic rules. The incrementality in those tools is achieved by using the classic *DRed* algorithm (Gupta et al., 1993) for view maintenance. In response to the updates on input, the *DRed* algorithm can overestimate the set of invalidated tuples and will, in the worst case, perform a large amount of work to “undo” the effects of a deleted tuple, only to conclude that the best approach is to start from scratch, especially for the more complicated recursive rules. The *DRed* algorithm does not maintain partially ordered versions for the recursive iterations as Differential-Dataflow does to facilitate the incremental updates even on iterative operators.

VIATRA makes use of the Xtend language (Bettini, 2016) to specify model transformations. The specification consists of model queries serving as preconditions to apply the transformation rules, which contain different model manipulation actions. The preconditions (patterns) are expressed using the graph pattern-based language VQL (VIATRA Query Language). VQL queries are stored in a separate file so that they can be reused easily. A pattern consists of a named query with parameters and a body of constraints. A matching set is the result of a graph pattern matching, which is a set of model objects fulfilling all constraints defined in the pattern. The rules are executed in an event-driven manner, meaning that the precondition is checked on every related element model change in an incremental way to determine the actions to be fired accordingly.

IncA (Szabó et al., 2016) is a domain-specific language for the definition of efficient incremental program analyses that update their result as the program changes. IncA compiles analyses into graph patterns and relies on existing incremental matching algorithms that are similar to the algorithms in VIATRA. To scale IncA analyses to large programs, IncA uses optimizations that reduce caching and prune change propagation. The application of IncA includes incremental control flow and points-to analysis for C, well-formedness checks for DSLs, and 10 FindBugs checks for Java.

3.5.5 Other incremental approaches

The other approach to achieve incrementality is to maintain a cache that includes all possible patterns used in the query and keep updating a small part of the cache by propagation. Tefkat (Lawley and Steel, 2005)

is a logic-based transformation language with data-driven evaluation, where models are encoded in a fact database. In Tefkat, the evaluation of declarative rule-based transformations is driven by a search for solutions to a Prolog-like goal (query) by relying on SLD resolution (Selective Linear Definite clause resolution). The algorithm constructs and preserves a Prolog-like resolution tree for patterns, which is incrementally maintained upon model changes and pattern (rule) changes as well.

The .NET modeling framework NMF has its own NMF Expressions that are presented as the implementation of an incremental computation system where the incrementalization of a model analysis function can be regarded as a functor. A dynamic dependency graph representing each executed instruction is built at run time. When a value in the DDG changes, dependent nodes in the expression tree are notified of the change, which is then propagated up to the root of the expression tree until the value of a subexpression does not change. NMF expressions are extensible with specific incrementalized algorithms, which take preference over the generic change propagation mechanism, in order to enable user-defined optimizations.

CHAPTER 4

Semantic Bridge from Models to Graphs

In this chapter, we describe the design and implementation of a semantic bridge that connects our engineering modeling framework with a graph representation-based modeling framework incorporating a graph database, formal semantics, and visualization tools for analysis.

4.1 Introduction to Graph Database

A graph database is a database that stores data in a graph format, where nodes represent entities and edges represent relationships between those entities. This makes it ideal for representing and querying complex relationships between entities, such as social networks, recommendation systems, and fraud detection.

Gremlin (Rodriguez, 2015) is an open-source graph database and traversal language that allows you to work with data as a graph. It provides a flexible, scalable, and efficient way to store, manage, and query complex relationships between entities. Gremlin supports a variety of graph database implementations, including Apache TinkerPop (TinkerPop, 2020), Amazon Neptune, and Microsoft Azure Cosmos DB. It also provides a powerful traversal language, which allows you to traverse and query graphs in a declarative manner. Gremlin's traversal language is similar to SQL, but instead of querying tables, it queries graphs. It provides a rich set of traversal steps, such as filtering, mapping, aggregating, and transforming data with limited recursion such as transitive closure. You can also chain multiple traversal steps together to form a more complex query.

Overall, Gremlin is a powerful and flexible graph database solution that can handle a wide range of use cases. Its support for different graph database implementations and its expressive traversal language make it a popular choice for building complex graph-based applications.

We chose a graph database as the alternative engine for model execution because models and meta-models can be expressed as graph structures. They allow retrieval of data with complex hierarchical structures in a simple and fast manner, in comparison with relational databases. Graph databases also greatly improve performance compared to traditional relational databases when data grows and the depth of relationships increases.

Graph algorithms can be used for various tasks in model execution, including model traversal, constraint satisfaction, and optimization. Here are some examples of how graph algorithms can be applied to model execution:

1. Graph traversal: Graph algorithms can be used to traverse models and extract information from them.

For example, depth-first search (DFS) and breadth-first search (BFS) can be used to traverse the nodes

of a model and identify paths and relationships between them.

2. Constraint satisfaction: Graph algorithms can be used to solve constraint satisfaction problems in models. For example, the minimum spanning tree algorithm can be used to identify the shortest path between two nodes in a model, or the Dijkstra algorithm can be used to find the optimal path between two nodes based on weighted edges.
3. Optimization: Graph algorithms can be used to optimize models based on certain criteria. For example, the traveling salesman problem can be solved using the branch-and-bound algorithm to identify the optimal route for visiting a set of locations.
4. Model transformation: Graph algorithms can be used to transform models into different representations. For example, the graph matching algorithm can be used to identify and align similar substructures in two models, which can be useful for model comparison and merging.

4.2 Semantic Bridge Overview

We define the formal semantics for WebGME models and metamodels as part of our integrated modeling framework with the semantic backplane. The models and metamodels are semantically mapped to graph representation that we either use FORMULA itself to execute model operations or even with a useful set of language features of FORMULA constraint checking rules translated into graph queries. In the second approach, the whole FORMULA model execution can be delegated to running graph queries in the graph database to significantly improve the performance. The semantic bridge in Figure 4.1 connects WebGME and graph database with formal semantics defined in FORMULA and we take advantage of the power of each tool to form the integration tools that greatly improve the usability, interoperability, and performance.

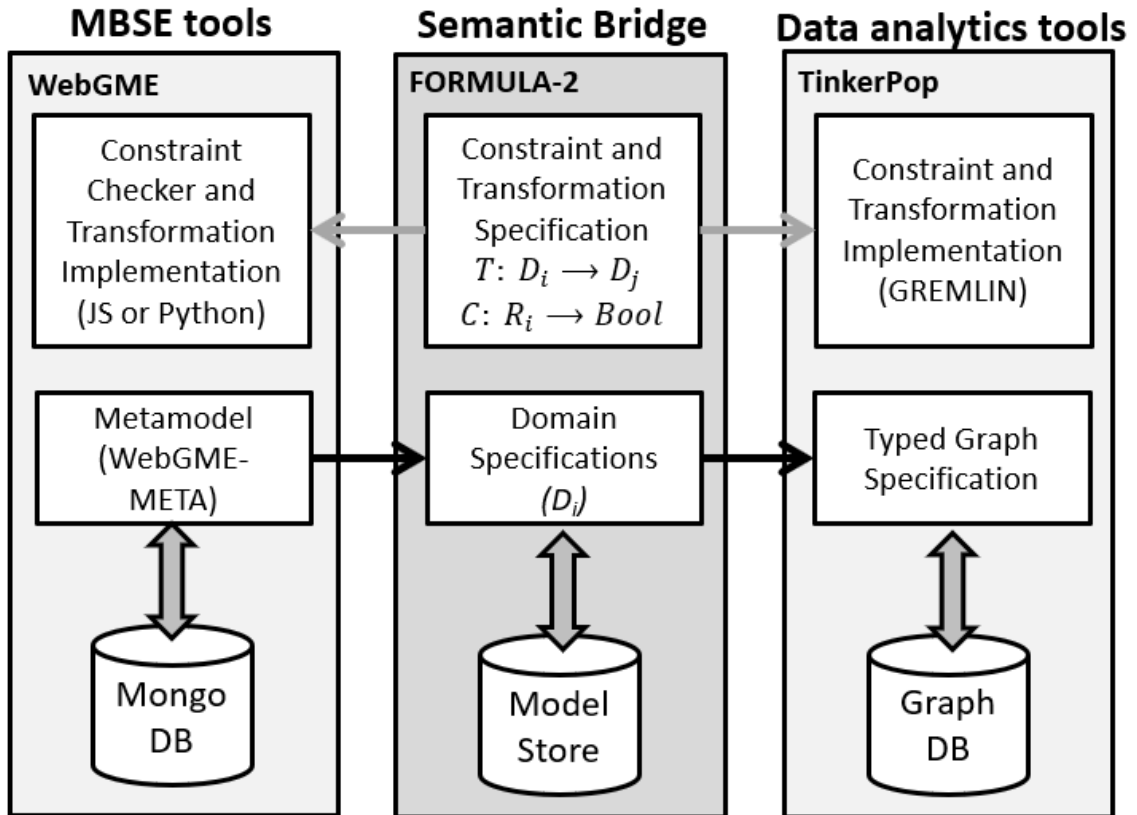


Figure 4.1: Overview of WebGME, FORMULA, and Graph Database Integration

4.2.1 Tools Comparison

The main features of both WebGME and FORMULA are listed below and we evaluate the pros and cons of both frameworks for the purpose of integrating WebGME with FORMULA as its semantic backplane. The semantic backplane and integration of WebGME and FORMULA lay the foundation for model operations to facilitate the translation to graph-based tools or other low-level representations.

WebGME (Maróti et al., 2014) services include:

- Meta-programmability with the prototypical inheritance that allows smooth language integration and evolution
- Graphical concrete syntax that is highly versatile and customizable
- Multiple well-defined APIs for model interpretation and tool integration
- Allow modeling with Git-style version control and branch support
- Collaborative and distributed modeling via web-interface

FORMULA (Jackson and Schulte, 2013) services include:

- Formal representation of structural semantics of modeling languages (Jackson and Schulte, 2013) as strongly-typed, open-world logic programs (OLP) (Jackson et al., 2013) 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 (De Moura and Bjørner, 2008),
- Modular reuse of DSMLs via the composition of OLPs in a strong category theoretic sense (Jackson et al., 2011b).

The logic-based modeling language of FORMULA is very concise in specifying constraints and model transformations with formal semantics, but this solution cannot effectively support the goal of large model analytics and fast model transformation. In order to provide such capabilities without sacrificing the expressiveness of FORMULA, we investigated the features of Graph Databases with the Gremlin Traversal Language (TinkerPop, 2020). Also, to connect all three representations in Figure 2.1, FORMULA can accurately specify the precise semantics and can even provide transformation among these representations functioning as a semantic link. Gremlin is a graph traversal language and virtual machine developed by Apache TinkerPop (TinkerPop, 2020). Graph Database implementations allow querying and retrieval of data from highly complex hierarchical structures as well as supporting highly optimized graph analytic algorithms. This solution, therefore, combines semantically precise modeling, model transformation, and effective querying through modern graph databases.

4.2.2 Three Levels of Integration

Our goal is to model the metamodels and models as typed graphs in both WebGME and FORMULA. Metamodels and models are translated into a typed graph in different ways and stored as graphs in the storage backend such as a graph database. Conceptually, metamodels and models represented in the typed graphs can generate automatically derived conditions. For example, the graph of metamodel and models cannot be a disconnected graph with dangling components. Additional constraints can be succinctly expressed in graph queries or translated into graph queries from native constraints of higher abstraction.

We have three different ways in our integrated framework to map metamodels and models into graph representations on different levels with graph-based constraint checking:

1. Translate WebGME meta-models and models directly into graphs stored in a graph database. Graph database has the capability to reason over graphs for constraint checking including the auto-derived conditions or even graph transformation using graph traversal query language like Gremlin with underlying built-in efficient graph-related algorithms. Only graph query is allowed to be executed on the models and we lose the expressiveness of FORMULA language since FORMULA is not involved in the mapping from WebGME to typed graph, which is later stored in a graph database.

2. Translate WebGME meta-models and models into FORMULA models that conform to metamodels defined in *Graph* domain in FORMULA (MetaNode, MetaEdge, Node, Edge, etc in the typed graph) with the capability to reason over graph (Solve graph related problem in logic programming languages but could be slower than graph algorithms implemented in imperative general purpose programming languages). We use FORMULA to formally specify the semantics of WebGME metamodels and models as the semantic backplane. Since the metamodels and models are all expressed as nodes and edges in the FORMULA graph domain. FORMULA constraint checking and graph transformation rules can be written succinctly in the FORMULA language. The downside of this approach is that the execution is still done in the FORMULA engine that does not scale to large models.

3. Maintain the same semantic link from WebGME to FORMULA above and create another semantic link to map FORMULA domain/model to the typed graph in a graph database. Translate a subset of FORMULA rules for constraint checking into equivalent graph queries assuming we already have the mapping of metamodels and models from WebGME to FORMULA and then from FORMULA domains and models to graph database. The FORMULA to graph database semantic link has nothing to do with WebGME but instead, a direct translation from FORMULA constraint checking rules to Gremlin graph queries in a graph database.

We have several different integrations implemented in the existing frameworks and tools for different purposes. Each implementation of the integration is given a unique name:

1. WebGME2DB: Integration of WebGME and MongoDB: Raw data storage and retrieval as default storage backend but direct model analysis over the storage of loosely structured raw data is not feasible.

2. WebGME2GraphDB: Integration of WebGME and graph database, where models and metamodels are translated directly into low-level nodes and edges in the graph database. Writing graph queries over the model requires a deep understanding of the low-level implementation details and the queries are not intuitive.

3. WebGME2FORMULA: Integration of WebGME and FORMULA on two different levels. 1) WebGME is directly translated into FORMULA where every component of a WebGME model such as properties, inheritance, and relationships are precisely translated into FORMULA. 2) WebGME models and metamodels are modeled as typed graphs specified in a FORMULA domain named *MetaGraph* and we use FORMULA

to reason over the graph or even data mining for large models. Rules are evaluated in the native FORMULA execution engine for data analytics on models.

4. FORMULA2GraphDB: Integration of FORMULA and graph database that targets the translation from generic FORMULA domain (metamodel) and model to graph database. Not only the concepts in FORMULA language are mapped to the graph database but constraint checking such as conformance rules are also programmatically translated into equivalent graph queries with limited support of recursive rules.

The following subsections explain the details about the mappings of models, metamodels, and high-level expressive constraint checking specifications to graph-based tools conceptually but also implemented in our integrated framework.

4.3 Direct Integration of WebGME and FORMULA

The integration of WebGME and FORMULA has the following advantages:

1. Solve the problem that WebGME models and meta-models have no formally defined semantics for their components and the relationship between components.

2. For model operations in WebGME such as constraint checking or model transformation, the user has to implement plugins on a lower-level generic programming language such as JavaScript or Python. With models and queries translated to FORMULA, models are executable directly with provable traces.

3. Allow more expressive constraint checking on FORMULA models and models are directly executable given the rewriting rules for conformance checking. More powerful constraints (e.g. absence of cycle in graph transitive closure) can be naturally and succinctly expressed in FORMULA language while in the WebGME domain users have to write a complicated and error-prone plugin implemented in a low-level language.

Our previous work (Kecskés et al., 2017) is the initial attempt to bridge WebGME and FORMULA with well-defined semantics for the first time. However, the execution of models on FORMULA does not scale well over even hundreds of terms depending on the specific problem and FORMULA rules, which may have recursion and take millions of iterations to converge and terminate the execution.

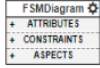


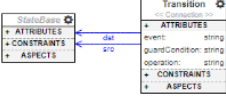
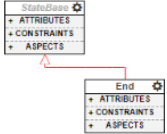
Concept description	WebGME (Meta) representation	Formula translation
Component		<code>FSMDiagram ::= new (id: String, parent: any {NULL}, attributes: any Attr__FSMDiagram, pointers: any Ptr__FSMDiagram).</code>
Containment		<code>StateBase ::= new (...parent: any FSMDiagramTYPE + {NULL},...).</code>
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 4.2: Translation of WebGME Meta Concepts into FORMULA

We implement 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 4.2, 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 translates them into instances in Formula.

The resulting FORMULA file is then processed with the help of the FORMULA engine to get the con-

straint evaluations and syntax checks. Being automated, the result is available after every change, so the user will be notified at the place of error. However, writing more complex FORMULA conformance rules for WebGME constraint checking still requires a low-level understanding of the translation details and loses some of the advantages of the high-level declarative definition of constraints in FORMULA language.

4.4 Graph-based Integration of WebGME, FORMULA, and Graph Database

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 (Kleppe and Rensink, 2008). 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.

4.4.1 Semantic Foundation in Graph

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

Definition 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 unique 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 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 (Ehrig et al., 2006; Jouault and Bézivin, 2006). We propose an extension of typed graphs to check the model conformance, which allows capturing inheritance defined between nodes of the metamodel.

Definition 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 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 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:

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

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

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.

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

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 .

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

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 $dst(e_M) = v_M$.

4.4.2 Logic-based FORMULA Specification

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.
```

To generate the conditions presented in Section 4.4.1 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.

4.4.3 Graph-based Gremlin Specification

Graph databases use graph structures to represent and store data. They allow retrieval of data of complex hierarchical structures in a simple and fast manner, in comparison with relational databases. We use the Gremlin traversal machine and language (Rodriguez, 2015) 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 edge 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);
```

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());
```

The detailed component-to-component translation in three ways can be found in Figure 4.3 from our work (Mavridou et al., 2018). The integration is designed in a way that WebGME metamodels and models are modeled in a graph with a *MetaGraph* domain defined in FORMULA that supports more sophisticated queries written in FORMULA language and model execution in FORMULA engine. On the other hand, the graph-based WebGME models can be naturally mapped to graphs with constraint checking mapped to and even executed in a graph database but the query has to be written manually with low-level optimization and is not as intuitive and powerful as the FORMULA conformance rules with rich semantics. Therefore, in the next subsection, we introduce the last integration in which we directly execute FORMULA rules in the graph database while we keep using FORMULA to specify the WebGME metamodels and models with formal semantics in a concise and unified language such as FORMULA.


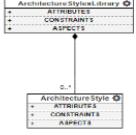


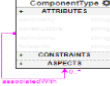

Concept description	WebGME (Meta) representation	Formula translation	Gremlin translation
Concept		StateBase is MetaNode('StateBase');	StateBase = graph.addVertex('class', 'MetaNode', 'name', 'StateBase');
Containment		ArchitectureStylesLibrary is MetaNode("ArchitectureStylesLibrary"). starMultiplicity is Multiplicity(0,"*"). exactlyOneMultiplicity is Multiplicity(1,1). MContainment1 is MetaEdge('MContainment1', ArchitectureStylesLibrary, ArchitectureStyle, exactlyOneMultiplicity, starMultiplicity).	ArchitectureStylesLibrary = graph.addVertex('class', 'MetaNode', 'name', 'ArchitectureStylesLibrary'). MContainment1 = graph.addVertex('class', 'MetaEdge', 'name', 'MContainment1'). MContainment1.addEdge('src', ArchitectureStylesLibrary, 'min', 1, 'max', 1). MContainment1.addEdge('dst', ArchitectureStyle, 'min', 0).
Attribute		Guard is MetaNode('Guard'). Guard_has_guardMethod is MetaEdge('guardMethod', Guard, String, exactlyOneMultiplicity, starMultiplicity).	Guard = graph.addVertex('class', 'MetaNode', 'name', 'Guard'); 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 (many to one association)		Connection is MetaNode('Connection'). ConnectorEnd is MetaNode('ConnectorEnd'). Connection_point_src_ConnectorEnd is MetaEdge('src', Connection, ConnectorEnd, staMultiplicity, exactlyOneMultiplicity).	Connection = graph.addVertex('class', 'MetaNode', 'name', 'Connection'); ConnectorEnd = graph.addVertex('class', 'MetaNode', 'name', 'ConnectorEnd'); 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 is MetaNode('ComponentType'). ComponentType_collects_ComponentType is MetaEdge('associatedWith', ComponentType, ComponentType, staMultiplicity, staMultiplicity).	ComponentType = graph.addVertex('class', 'MetaNode', 'name', 'ComponentType'); 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)		StateBase is MetaNode('StateBase'). State is MetaNode('State'). NodeInheritance(StateBase, State).	StateBase = graph.addVertex('class', 'MetaNode', 'name', 'StateBase'); State = graph.addVertex('class', 'MetaNode', 'name', 'State'); State.addEdge('type', StateBase).

Figure 4.3: Patterns of translation into FORMULA and Gremlin of main WebGME meta-modeling language features

4.5 FORMULA Model Operations Executed in Graph Database

Both of FORMULA constraint checking and graph queries are declarative languages to express computations over structured data and we build a semantic mapping from FORMULA constraint checking rules to Gremlin graph queries. We only map a subset of FORMULA language into graph query language because FORMULA language is more expressive and not all semantic gaps between FORMULA and Gremlin query language can be closed but this subset is still useful for the execution of large models. FORMULA has more rich semantics than Graph Query Language, which cannot do more complicated recursion other than transitive closure. We map pattern matching predicate, aggregation, and negation in the FORMULA rule to sub-graph pattern matching in graph query language as shown in the following subsections.

4.5.1 Model and Metamodel Representation in Graph

Before FORMULA constraint checking rules are mapped to equivalent graph queries, we need to form a solid connection from FORMULA domain/model to the graph database with a precise definition. Components and their interconnected relationships in the modeling design can be viewed as nodes and edges in the graph as a low-level representation but still preserve all information about model properties and relationships. We show the conceptual mapping from models and metamodels separately in the graphs below.

Domain (metamodel) Representation: FORMULA domain describes the "class of things" on the meta-model level and contains all type definitions with conformance rules. A FORMULA domain is defined in a graph database as a labeled graph $DG = \langle V_d, E_d, \lambda_{v_d}, \lambda_{e_d} \rangle$, where V_d is a set of vertices to represent all types in the current domain and $E_d \subseteq V_d^2$ is a set of directed edges to denote the relationship between a type and the types of its own arguments. The labeling function $\lambda_{v_d} : V_d \mapsto N$ maps all type vertices to N , which is a set of label names and $\lambda_{e_d} : E_d \mapsto S$, where $N \in S$ and S is a set of sets of names.

Model Representation: A labeled graph $MG = \langle V_m, E_m, \lambda_{v_m}, \lambda_{e_m}, \lambda_t \rangle$ is created for a FORMULA model instantiation, where V_m is a set of vertices to represent all models of different types in the current domain and $E_m \subseteq V_m^2$ is a set of directed edges between model and its components in arguments. The labeling function $\lambda_{v_m} : V_m \mapsto P$, where P is a set of sets of nested properties that contains key-value pairs, maps all model vertices to a set of properties. $\lambda_{e_m} : E_m \mapsto S$, where N and S have same meanings as in domain representation, such that $\lambda_t : V_m \mapsto V_d$ maps each model to its own type.

Models conform to their metamodels and this relationship is also enforced in graph representation by connecting each model node to its type node with edge label `type`. To highlight the above-described relationships, part of the graph representation is shown in Figure 4.5 given the FORMULA specification of *MetaGraph* in Figure 4.4.

```

domain MetaGraph
{
  MetaNode ::= new (name: String).
  MetaEdge ::= new (name: String , src: MetaNode ,
    dst: MetaNode , ms: Multiplicity , md: Multiplicity).
  Node ::= new (name: String , type: MetaNode).
  Edge ::= new (name: String , type: MetaEdge ,
    src: Node , dst: Node).
  Multiplicity ::= new (low: Integer ,
    high: Integer + {"*"}).
  NodeInheritance ::= new (base: MetaNode ,
    instance: MetaNode + Node).
  NodeInstanceOf ::= (MetaNode , MetaNode + Node).
}
model example of MetaGraph
{
  exactlyOne is Multiplicity(1,1).
  atMostOne is Multiplicity(0,1).
  atLeastOne is Multiplicity(1,"*").
  anyNumber is Multiplicity(0,"*").
  mn1 is MetaNode("mn1").
  mn2 is MetaNode("mn2").
  me1 is MetaEdge("me1" , mn1 , mn2 , atLeastOne , atMostOne).
  n1 is Node("n1" , mn1).
  n2 is Node("n2" , mn2).
  e1 is Edge("e1" , me1 , n1 , n2).
}

```

Figure 4.4: *MetaGraph* Domain and Model in FORMULA

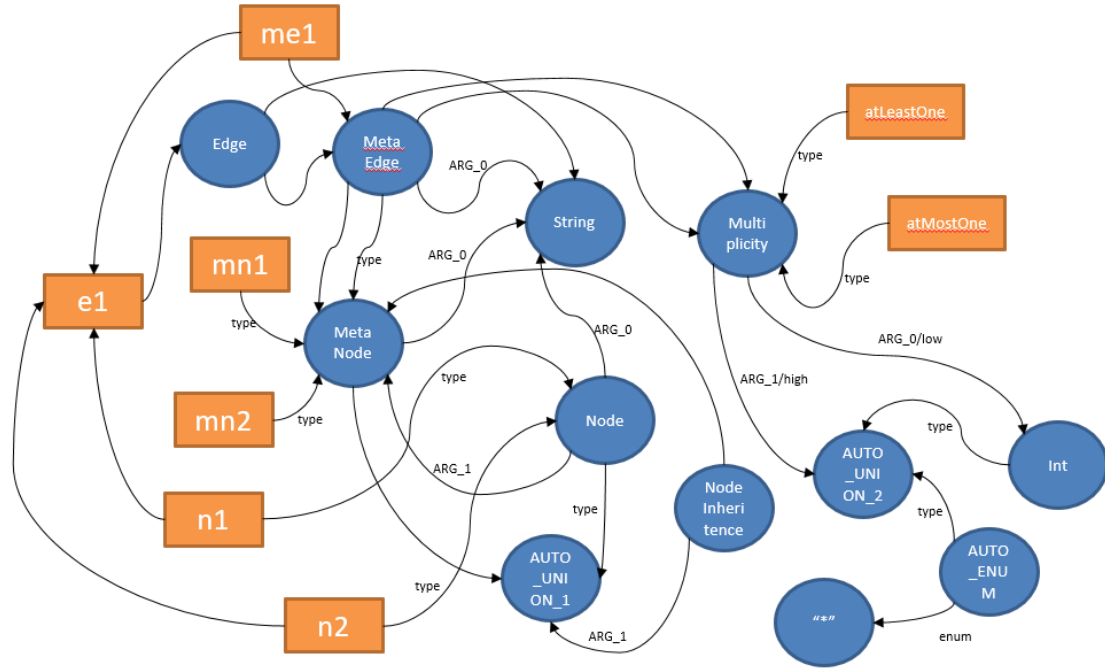


Figure 4.5: FORMULA Model Represented in Graph

4.5.2 Pre-processing of FORMULA Rule

The predicates in FORMULA rules are pre-processed in order to be further translated into sub-graph patterns or combination of several patterns. FORMULA rule is first parsed into the format of Abstract Syntax Tree (AST) with FORMULA Core APIs and then processed to create a label map that stores symbols and their interconnected relations to other symbols. The label map is a hash map that maps a label name to a set of label locations, where label location is a tuple $l \in L$ in a set of locations and $l = \langle type, index, instance \rangle$, $type$ is the name of the type represented by current label location, $index$ is the index of argument and $instance$ is a unique Identification number to distinguish different function terms such as $C(a, b)$, $C(b, c)$ that represent instances of the same type but share some same labels as their arguments. For example, label a in rule $TRUE :- C(a, b), C(d, a)$ is mapped to two locations $\langle C, 0, 0 \rangle$ and $\langle C, 1, 1 \rangle$.

Gremlin traversal language can be written in a combination of imperative and declarative styles to find matching sub-graphs by filtering out the whole graph step by step defined in the pipeline. Declarative style queries have the benefit to leverage different query planers and determine the best execution order based on historic statistics of previous patterns to filter the graph more efficiently. The following subsections describe the translation in detail with examples.

4.5.3 Label Map for FORMULA Rule

FORMULA rule is first parsed into the format of Abstract Syntax Tree (AST) with FORMULA Core APIs and then processed to create a label map that stores symbols and their interconnected relations to other symbols.

The label map is a hash map that maps a label name to a set of label locations, where label location is a tuple $l \in L$ in a set of locations and $l = \langle type, index, instance \rangle$, $type$ is the name of the type represented by current label location, $index$ is the index of argument and $instance$ is a unique Identification number to distinguish different function terms such as $C(a, b)$, $C(b, c)$ that represent instances of the same type but share some same labels as their arguments. For example, label a in rule `TRUE :- C(a, b), C(d, a)` is mapped to two locations $\langle C, 0, 0 \rangle$ and $\langle C, 1, 1 \rangle$.

4.5.4 Instance and Arguments

The relationship between each model instance and its arguments are represented as a graph-matching pattern in a declarative manner. In the graph representation of Models, each model is connected with its argument by an edge label that is automatically generated from the prefix "ARG_" the index of argument in the model instance. `..As("a").In("ARG_X").As("b")` is a declarative matching pattern to find models that two nodes are connected by an edge with label name "ARG_X" and node b points to node a . All tuples in the label map are translated into graph patterns in this way. An example is shown below to demonstrate the translation.

FORMULA rule:

```
C ::= new(x: Integer, y: Integer).
TRUE :- C(a, b), c1 is C(b, c).
```

Graph query in Gremlin:

```
g.V().Match(
  ..As("a").Has("type", "Integer").In("ARG_0")
  .Has("type", "C").As("instance_0_of_C"),

  ..As("b").Has("type", "Integer").In("ARG_1")
  .Has("type", "C").As("instance_0_of_C"),

  ..As("b").Has("type", "Integer").In("ARG_0")
  .Has("type", "C").As("c1"),

  ..As("c").Has("type", "Integer").In("ARG_1")
  .Has("type", "C").As("c1")
```

```
). Select("a", "b", "c", "c1");
```

4.5.5 Handling Binding Label with Fragments

A unique label may or may not be defined for every argument such as `A ::= new(id: Integer)` or `A ::= new(Integer)`. If the unique label is defined for the argument, the FORMULA user can write rules that have expressions like `a1.id` to represent the components of a model.

To deal with labels that extend a binding label to access its components like "c1.x.y", they are split into fragments as "c1", "c1.x" and "c1.x.y", then more constraints about relations on fragmented labels are added to existing graph pattern.

Model node is not only connected to each of its argument nodes by an edge with a label composed of the prefix "ARG_" and the index of the argument, but also another duplicate edge with a label that uses an argument label such as "id" in `A ::= new(id: Integer)` to name the label. The duplicate edge is used in an additional graph pattern to represent the relations between label fragments. If a label has chained fragments such as " $a_1.a_2.\dots a_{i-1}.a_i$ ", we simply extend the graph traversal to chain all label fragments one after one by name of the duplicate edge as follows:

$$As(a_1.a_2.\dots a_{i-1}.a_i).In(label_1).As(a_1.a_2.\dots a_{i-1}) \\ .In(label_2)\dots In(label_i).As(a_1)$$

FORMULA rule:

```
C ::= new(x: Integer, y: Integer).
TRUE :- c2 is C(a, c1.x), c1 is C.
```

Graph query in Gremlin:

```
g.V(). Match(
  . As("a"). Has("type", "Integer"). In("ARG_0")
  . Has("type", "C"). As("c2"),
  . As("c1.x"). Has("type", "Integer"). In("ARG_0")
  . Has("type", "C"). As("c2"),
  . As("c1.x"). In("x"). As("c1")
). Select("a", "c1", "c1.x");
```

4.5.6 Constraints over Properties in Built-in Types

FORMULA rules allow users to add constraints over the properties of built-in types like *Integer* or *String* type in models. Each tuple $t = \langle op, label1, label2, cnst \rangle$ is translated into Gremlin Traversal to add some constraints. The example below shows how this type of constraint is mapped to Gremlin traversal patterns.

FORMULA rule:

```
C ::= new(x: Integer, y: Integer).
TRUE :- C(a, c1.x), c1 is C(b, c), c1.x < c1.y, b != 2.
```

Graph query in Gremlin:

```
g.V().Match(
  ..As("a").Has("type", "Integer").In("ARG_0")
  .Has("type", "C").As("instance_0_of_C"),

  ..As("c1.x").Has("type", "Integer").In("ARG_1")
  .Has("type", "C").As("instance_0_of_C"),

  ..As("b").Has("type", "Integer").In("ARG_0")
  .Has("type", "C").As("c1"),
  ..As("c").Has("type", "Integer").In("ARG_1")
  .Has("type", "C").As("c1"),
  ..As("c1.x").In("x").As("c1"),
  ..As("c1.y").In("y").As("c1"),
  ..As("cc.x").Values("value").As("cc.x_value"),
  ..As("cc.y").Values("value").As("cc.y_value"),
  ..Where("cc.x_value", P.Lt("cc.y_value")),
  ..As("b").Values("value").Is(P.Neq(2))
).Select("a", "b", "c", "c1", "c1.x", "c1.y");
```

4.5.7 Negation and Set Comprehension Operators

Set comprehension with operators like `count` and `no` are commonly used in FORMULA language. Constraints with negation or set comprehension are not evaluated by calling graph queries, but evaluated programmatically by counting the number of matching results in C# code. For example, `count(s | s is C(m, n)) < 3` checks if the count of models of type C is less than 3, `no {d | d is D(c1, c2)}` checks if the count of models of type D is equal to 0. If all conditions are satisfied, the program will proceed with variable substitution in the head of the rule.

FORMULA rule:

```
C ::= new(x: Integer, y: Integer).
```

```
D ::= new(x: C, y: C).
```

```
TRUE :- count({s | s is C(m, n)}) < 3, no {d | d is D(c1, c2)}.
```

Graph query in Gremlin:

```
g.V().Match(  
  .As("m").Has("type", "Integer").In("ARG_0")  
  .Has("type", "C").As("s"),  
  .As("n").Has("type", "Integer").In("ARG_1")  
  .Has("type", "C").As("s"),  
  .As("c1").Has("type", "C").In("ARG_0")  
  .Has("type", "C").As("d"),  
  .As("c2").Has("type", "C").In("ARG_1")  
  .Has("type", "C").As("d"),  
) .Select("m", "n", "s", "c1", "c2", "d");
```

4.5.8 Termination of Repeating Rule Execution

Recursive calls can be applied in the FORMULA rule to describe concepts such as transitive closure that are beyond the scope of first-order logic. Instead of implementing concrete patterns matching algorithms such as Rete Algorithm and Magic Sets in a rule-based system, we repeatedly compute matches from RHS and synthesize new terms on LHS by executing graph queries under Gremlin APIs and counting if current execution reaches a fixed point that no more terms can be generated in the next round of rule execution.

FORMULA rule:

```
C ::= new(x: Integer, y: Integer).
```

```
C(a, c) :- C(a, b), C(b, c).
```

```
c1 is C(1, 2).
```

```
c2 is C(2, 3).
```

```
c3 is C(3, 4).
```

Rule execution:

```
Round 1: C(1, 2) C(2, 3) C(3, 4)
```

```
Round 2: C(1, 2) C(2, 3) C(3, 4) + C(1, 3) C(2, 4)
```

```
Round 3: C(1, 2) C(2, 3) C(3, 4) C(1, 3) C(2, 4) + C(1, 4)
```

```
Round 4: C(1, 2) C(2, 3) C(3, 4) C(1, 3) C(2, 4) C(1, 4)
```


4.6 Benchmark and Performance Comparison

We use the same *MetaGraph* domain, which is also used to represent WebGME models and metamodels as typed graphs, to benchmark the constraint-checking performance in FORMULA. The conformance rules in this domain are the automatically derived conditions in the typed graph representing the models and metamodels. The *MetaGraph* domain contains most of the features of FORMULA language such as pattern matching, negation, set comprehension, and recursion.

Models of different sizes are generated based on the *MetaGraph* domain but are not shown here due to space limitations. All experiments were run on an Intel(R) Core(TM) 3.40GHz i7-2600 CPU machine with 16GB RAM. The execution results are shown in table 4.1. Graph queries take less than 1 second for a large graph with 4082 nodes and 12670 edges in total, while FORMULA needs more than 100 seconds to finish. We also tried a larger graph with 121000 edges and 1100 nodes. In this case, FORMULA ran out of memory and was not able to finish the task of conformance checking. On the contrary, graph query execution was much faster and took less than 5 seconds. Model execution in graph database excels in concrete execution as shown in our benchmark with a trade-off between rich semantics and performance. but we only map a subset of the FORMULA language.

```
domain MetaGraph
{
  //Type definitions are defined in Figure 4b above.
  WrongMultiplicity :- Multiplicity(low,high),
  high != "*", low > high.
  not1 :- n is Node,
  no {m | m is MetaNode, NodeInstanceOf(m,n)}.
  not2 :- e is Edge, no {m | m is MetaEdge, m = e.type,
  NodeInstanceOf(m.src,e.src)}.
  not3 :- e is Edge, no {m | m is MetaEdge, m = e.type,
  NodeInstanceOf(m.dst,e.dst)}.
  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.
  not5 :- m is MetaEdge, n is Node, NodeInstanceOf(m.src,n),
  m.md.low != 0, no{e | e is Edge(-,m,n,-)}.
  not6a :- n is Node, m is MetaEdge, NodeInstanceOf(m.dst,n),
  count({s | s is Node, e is Edge (-,m,s,n)}) < m.ms.low.
  not6b :- n is Node, m is MetaEdge, NodeInstanceOf(m.dst,n),
  m.ms.high != "*",
  count({s | s is Node, e is Edge (-,m,s,n)}) > m.ms.high.
  not7 :- m is MetaEdge, n is Node, NodeInstanceOf(m.dst,n),
  m.ms.low != 0, no{e | e is Edge(-,m,-,n)}.
}
```

Listing 4.1: MetaGraph Domain Specified in FORMULA

Table 4.1: Execution times for FORMULA and Gremlin specifications. MN, ME, N, E stand for number of MetaNodes, MetaEdges, Nodes, Edges.

	MN	ME	N	E	Formula	Gremlin
1	49	39	347	805	6.34s	<1s
2	49	39	596	1596	12.58s	<1s
3	49	39	1094	3178	24.47s	<1s
4	49	39	2090	6342	56.95s	<1s
5	49	39	4082	12670	112.94s	<1s
6	49	39	1100	121000	-	4.28s

4.7 Contributions

In this chapter, we discussed the pioneering works of integrating modeling frameworks, the formal specification language FORMULA, and graph databases. They are on different levels of integration for different purposes to address the performance issue, the scalability issue, and the lack of a formal semantic foundation.

1. We translate WebGME meta-models and models into both relational database and graph database with automatically derived conditions in the form of graph query for efficient constraint checking over large models. The trade-off in this method is that the reasoning over models can only be done at a low level with only a vague connection to the semantics of models and meta-models.
2. We use FORMULA to formally specify the semantics of WebGME metamodels and models as the semantic backplane. FORMULA is flexible and powerful enough that we can either directly translate every component and relationship into FORMULA terms or use FORMULA to model both meta-models and models as nodes and edges in a FORMULA-defined typed graph domain. With the power of FORMULA, a lot of complex constraint checking such as reachability-related queries can be expressed concisely in the FORMULA language. The trade-off in this integration is that FORMULA lacks efficient performance in model execution compared with a graph-based solution but preserves the rich semantics of formally defined models and metamodels.
3. We keep the same semantic link between WebGME and FORMULA no matter whether it is a direct one-to-one rigorous translation for every WebGME component or have all models expressed in a graph domain but build a separate semantic bridge between WebGME and Formula. The purpose of the semantic bridge is to keep the rich semantics of FORMULA to facilitate the modeling of complex systems or queries that cannot be expressed concisely and accurately in SQL and other imperative languages like JavaScript or Python, which are low-level programming languages manually written

plugins for constraint checking. The translation only targets a subset of FORMULA language to graph database and graph query language due to the semantic mismatch between the two languages but is still powerful enough to solve a lot of complex constraints checking with fast execution.

CHAPTER 5

Developing Differential-FORMULA in MDE Methodology

WebGME and FORMULA both store structured model data in memory, text files, or databases. FORMULA currently lacks a built-in version control mechanism, and tracking changes to models is performed by taking a snapshot of each model (i.e., storing the entire content of the model). This approach has a lot of redundancy in the storage. Incremental updates are not feasible to be implemented if only snapshots of the whole model are stored.

With the goal of achieving high performance and incremental updates in mind, we naturally came across the Rust language (Klabnik and Nichols, 2019) and the family of differential computation. Timely-dataflow (Murray et al., 2013), differential-dataflow (Abadi et al., 2015), and Differential-Datalog (Ryzhyk and Budiú, 2019) are all related to each other in the context of data processing and analysis, particularly in the domain of stream processing and real-time data analytics.

Timely-dataflow is a computational model that provides a framework for building scalable and efficient distributed data processing systems. It allows for the creation of dataflow graphs, which are composed of operators that process data in parallel across multiple nodes in a distributed system. The Timely dataflow model enables the processing of both batch and streaming data and can be used for a wide range of applications such as network analytics, machine learning, and graph processing.

Differential-dataflow is an extension of Timely dataflow that allows for the efficient processing of incremental updates to data in real time. It enables the incremental computation of changes to a dataset, which can be more efficient than recomputing the entire dataset from scratch. This makes differential dataflow ideal for scenarios where data is constantly changing, such as in real-time analytics or online machine learning. Differential dataflow is particularly useful for processing large-scale datasets, as it can efficiently handle data with billions of records.

Differential-Datalog is a declarative programming language that extends Datalog, a logic-based query language used for deductive databases. Differential Datalog extends the traditional Datalog model by adding support for incremental updates, which makes it well-suited for real-time analytics and stream processing. Differential Datalog allows for the efficient computation of changes to a dataset by incrementally updating the logical rules that define the dataset, rather than recomputing the entire dataset from scratch.

In summary, Timely-dataflow provides a framework for building distributed data processing systems, while differential dataflow and differential Datalog extend Timely dataflow to enable the efficient processing of incremental updates to data in real-time. All three are important tools in the domain of real-time data

analytics and are used for a wide range of applications such as network analytics, machine learning, and graph processing.

We want to integrate the new Differential-Dataflow computation model into the FORMULA execution engine in order to enable incremental updates and handle large models. The integration with the Differential-Dataflow computation model has the following advantages:

1. FORMULA terms are translated into DDLLog data structures and later passed into the Differential-Dataflow dataflow graph implemented in the Rust language. The advantage here is that the Rust language compiles to code that is as fast as a native C++ implementation in general but with more memory safety guarantees.
2. Use Differential-Dataflow Arrangement to index models and maintain states in memory. The advantage here is better performance.
3. Efficiently update the output of each DD operator based on multiple partially ordered versions incrementally and compact all the versions before a certain timestamp while the users decide that they do not need to track the changes before this timestamp anymore.

Rather than re-implement FORMULA using a new computation model, which is a non-trivial task, we decided to semantically map FORMULA to a Datalog dialect named *Differential-Datalog*. The advantage of this approach is that Differential-Datalog already has an existing compiler that compiles a DDLLog program to a runtime in Rust that implements highly optimized dataflow pipelines based on Differential-Dataflow to incrementally derive new facts or even remove existing facts accordingly when the input changes. We proposed and are in the process of implementing a model-driven engineering framework for designing DSMLs, namely Differential-FORMULA, by adopting the same model-driven engineering methodology to develop our integrated modeling tools.

We are developing an addition that we call *Differential-FORMULA* (δF) that allows model queries and transformations to be performed incrementally. To avoid re-implementing the FORMULA execution engine from scratch, we leverage Differential-Datalog (DDLLog), which is a general-purpose logic-programming language. The idea is to translate FORMULA specifications into DDLLog specifications and use the incremental DDLLog execution engine to perform model transformations and query evaluations. The semantics of DDLLog are similar to FORMULA, but there are some semantic mismatches between the two languages that we identified and must be considered when translating a FORMULA specification into a DDLLog specification.

Our ultimate goal is to use FORMULA to formally specify both the FORMULA language itself and the DDLLog language, namely the metamodels of the two languages. Model transformation can then be applied

to transform models of FORMULA programs into models of DDLLog programs. Naturally, the models of the language domains must conform to the predefined language metamodels.

As shown in Figure 5.3, we formally specify the metamodels of the two languages so a transformation can be specified in the same way. Instead of manually writing an equivalent DDLLog program to execute FORMULA models, we first model the language features based on their grammar and semantics in both FORMULA and DDLLog languages as shown in the lower part of Figure 5.3. In the next step, the metamodels, models, and transformation from a user-defined domain are all treated as models of the FORMULA language domain and after performing the transformation, we derive an equivalent DDLLog program. This DDLLog program can execute models from the previous user-defined domain and incrementally update the results when the input model changes.

5.1 Differential-Dataflow Computation Model and Tool Suites

The efficient incremental model execution engine in Differential-FORMULA is achieved by applying the new Differential-Dataflow computation model. However, the implementation is built by translating to the incremental logic programming language Differential-Datalog (an incremental version of the Datalog dialect) that has already implemented the compilation to Differential-Dataflow pipelines. Differential-Dataflow provides the basic building blocks for Differential-Datalog that the Differential-FORMULA will be translated to by executing a formally defined model transformation. Differential-Dataflow builds upon Timely-Dataflow, which is a computation model for low-latency, parallel and iterative computation with a logical timestamp. In general, high-level programming models are built upon or mapped to low-level primitives in each layer as shown in the following subsections, and on the highest level we have Differential-FORMULA that allows users to create models with constraints and also has the capability to execute models incrementally for the modeling purpose.

5.1.1 Timely Dataflow

Model execution with incremental updates in a distributed system is a hot research topic for model-driven engineering and could be a suitable target to tease out the potential of Timely-dataflow framework to maximize the performance gain. Timely-dataflow is a new data stream computation model for executing data parallel, cyclic dataflow programs. This model enriches dataflow computation with timestamps that represent logical points in the computation and provide the basis for an efficient, lightweight coordination mechanism. Timely-dataflow supports asynchronous and fine-grained synchronous execution with a logical timestamp and distributed progress tracking protocol. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform iterative computations.

Its key contribution is a new coordination mechanism that allows low-latency asynchronous message processing while efficiently tracking global progress and synchronizing only where necessary to enforce consistency. The implementation of Naiad (Murray et al., 2013) demonstrates that a timely-dataflow system can achieve performance that matches and in many cases exceeds many specialized systems.

```
extern crate timely;
use timely::dataflow::InputHandle;
use timely::dataflow::operators::{Input, Exchange, Inspect, Probe};
fn main() {
    // initializes and runs a timely dataflow.
    timely::execute_from_args(std::env::args(), |worker| {

        let index = worker.index();
        let mut input = InputHandle::new();

        // create a new input, exchange data, and inspect its output
        let probe = worker.dataflow(|scope|
            scope.input_from(&mut input)
                .exchange(|x| *x)
                .inspect(move |x| println!("worker {}: \thello {}", index, x))
                .probe()
        );

        // introduce data and watch!
        for round in 0..10 {
            if worker.index() == 0 {
                input.send(round);
            }
            input.advance_to(round + 1);
            worker.step_while(|| probe.less_than(input.time()));
        }
    }).unwrap();
}
```

```
}
```

Listing 5.1: Timely-dataflow Example

The example above in Figure 5.1 from the Timely-dataflow official tutorial is a relatively intuitive and concise example to demonstrate how timely-dataflow works to coordinate and track programs. The dataflow described in Rust language can take different configurations and have multiple workers coordinated to finish one task. In the example, only the first worker receives the input or keeps the data to itself. If there are only two workers, one worker processes the even number and the other one processes the odd number. The data is passed to its peer workers based on the index of the worker. The most important part is in `worker.step_while()` that the worker passively receives the notification with both data and a digital timestamp. The `probe` variable, and the use of a probe to determine how long we should step the worker before introducing more data. The data are only processed when new data with a bigger timestamp enters the scope and without the coordination, the output of the dataflow will be in random order that the occurrence of a bigger integer could happen before a small integer. Thus, the Timely-dataflow can be applied to coordinate the computation of a more complex problem such as finding the shortest path in a graph with multiple workers in threads or even across machines.

5.1.2 Differential Dataflow

Timely-Dataflow alone is still far away from having primitive building blocks that are robust and expressive enough to implement an incremental framework. Therefore, we introduce another framework named Differential-Dataflow, which is built upon Timely-dataflow and has its only way to maintain the snapshots of the computation state for incremental updates. The novelty of differential computation is twofold: first, the state of the computation varies according to a partially ordered set of versions rather than a totally ordered sequence of versions as is standard for incremental computation; and second, the set of updates required to re-construct the state at any given version is retained in an indexed data-structure, whereas incremental systems typically consolidate each update in sequence into the “current” version of the state and then discard the update.

Differential-Dataflow is an ideal computation model to re-implement the core engine of FORMULA because Differential-Dataflow natively supports the iterative operator and aggregation operators besides the relational operators. The execution of FORMULA logic programming rules can be expressed by the construction of a dataflow graph with those primitive operators.

5.1.3 Differential Datalog

The main use case for Datalog is to take a database of facts and iteratively infer additional interesting facts via given rules from the current knowledge base. Datalog and related programming languages are commonly called *logic programming*. Differential-Datalog is a general-purpose logic programming language extending the traditional Datalog language and is built upon Differential-dataflow. The output of the DDLog compiler is a dataflow graph, which may contain cycles (introduced by recursion). The nodes of the graph represent relations; the relations are computed by dataflow relational operators. Edges connect each operator to its input and output relations. Differential-dataflow natively implements the following operators: `map`, `filter`, `distinct`, `join`, `antijoin`, `groupby`, `union`, `aggregation`, and `flatMap` with a highly optimized implementation in Rust.

```
typedef NID = StrId {nid: string} | NumId {nnid: u32} | Constant1 | Constant2
input relation Node(id: NID)
input relation Edge(src: Node, dst: Node)
output relation Path(src: Node, dst: Node)
output relation NoCycle(node: Node)
```

```
typedef NodeListNxt = Nxt {list: Ref<NodeList>} | NULL
output relation NodeList(item: Node, next: NodeListNxt)
```

```
NodeList(node, nxt) :- Node[node], var tail = NodeList{node, NULL},
    var nxt = Nxt{ref_new(tail)}.
```

```
Path(a, c) :- Path(a, b), Path(b, c).
```

```
HasCycle(HasCycleConstant) :- Path(u1, u2), u1 == u2, var g = u1.group_by(()),
    var count = g.group_count(), count == 0.
```

```
Outdegree(Node{src}, sum) :- Edge(Node{src}, Node{dst}),
    var sum = dst.group_by(src).group_count().
```

Listing 5.2: Introduction to Differential-Datalog with Examples

In Listing 5.2 we give a short example to introduce the major language features of DDLog and the meaning of its syntax. The keyword `typedef` denotes the type definition of tagged union type that contains at least one constructor. For example, The type `NID` can be either a string ID or a numeric ID. `Constant1`,

Constant2 and NULL are implicit constructors that take zero argument. Differential-Datalog also supports advanced types such as generic types `List<T>`, `Set<T>` and `Ref<T>` in the same flavor as Rust language itself even though the Differential-Datalog has a different imperative programming language for writing external function. For example, `Nxt {list:Ref<NodeList>}` is a constructor that takes the reference of `NodeList` as the only one argument without keeping another copy of the same data.

Relations denoted by keywords `input` and `output` is the entry and exit points in the dataflow generated by the DDLog program that `output relation Path(src:Node, dst:Node)` means a data container named `Path` that has a set of tuples. The input relations only receive changes from data input while the output relations incrementally reflect the changes propagating in the dataflow graph from the inputs all the way to the final outputs.

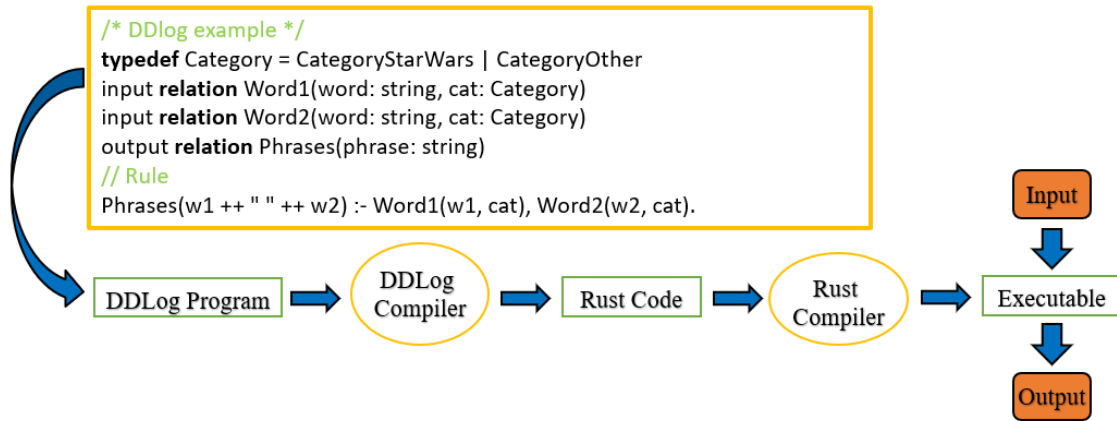
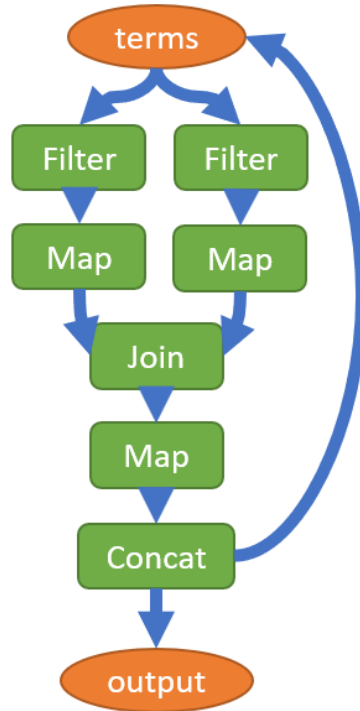


Figure 5.1: Differential-Datalog Internal Workflow



$$Reach(s') \leftarrow Reach(s), Trans(s, _, s').$$

Figure 5.2: Generated Differential-Dataflow from a DDLog Rule

The dataflow graph and pipelines are described by the rules defined in DDLog program that each rule is translated into operators in Differential-Dataflow. For example, the pattern-matching predicates in the rule are mapped to the `join` operator that takes two incoming streams of data and incrementally updates the results. DDLog also has the special `group_by` operator that groups the incoming data stream based on the key. For example, the last rule in Listing 5.2 groups all the edges `Edge(src, dst)` in the graph by the key `src` in order to compute the outdegree of each node. DDLog program may have recursion in its rules that repeatedly feed the updates from the output back to the input until the fixed point is reached that the dataflow sub-graph with self-cycle cannot derive new facts anymore.

In a nutshell, Differential-Datalog is a declarative language to define the versatile data structures to be passed around in the dataflow and describe how timestamped streams of data should be computed on a higher level abstraction. Differential-Datalog also has a compiler that compiles the description of dataflow into a real runtime implemented in Rust to do the incremental computation. The users either add changes to the input in the command line or call the runtime APIs in Rust.

Before we dive into the implementation of an incremental version of FORMULA language for metamodelling, we also did some research on each tool and the potential of tool integration as summarized below

1. Introduce several frameworks based on differential computation models and analyze the relationship between Timely-dataflow, Differential-dataflow, and Differential-datalog.
2. We investigate the possibility of integrating the differential computation model into our integrated modeling framework mainly FORMULA to achieve better performance and incremental computation.
3. Dissect and compare the computation model of FORMULA and Differential-Dataflow to have a one-to-one mapping from FORMULA semantic to the dataflow operators. The data structures of FORMULA terms and data types in Differential-dataflow and DDLog are also compared for the implementation of a translator from concrete FORMULA terms to the tagged union in DDLog.

In this chapter, we describe in detail the novel idea of generating an Incremental version of logic style modeling language FORMULA by modeling the language domains, identifying the semantic mismatches, and doing a model transformation for code generation. We use a general logic programming language named Differential-Datalog to do the model transformation incrementally because DDLog is also an incremental version of Datalog with extensions and more features. The whole process of model transformation and extraction from models to generate executable code can be viewed as incrementally generating an equivalent DDLog program using the exact same incremental DDLog language by reasoning and rule execution.

5.2 Introduction to Differential-FORMULA

Differential-FORMULA is initially planned to be a re-implementation of FORMULA in differential-dataflow or at least follows the same computation model of Differential-Datalog but later is transitioned to using the same model-driven engineering methodology to develop a tool that is exactly created for the model-driven engineering. The actual implementation is a translation from FORMULA to DDLog with precise semantic matching and model transformation.

To address this issue, we have developed an incremental version of FORMULA that can perform efficient model queries and transformations in the face of continual model updates. In a nutshell, we use the same Model-driven engineering methodology by applying a model transformation to another modeling framework FORMULA and generating a faster incremental version of the model execution engine. The end result and the performance of model execution turn out to be successful and is one of the leading high-performance incremental modeling frameworks based on the result of TTC18 (Boronat, 2018) model transformation benchmark.

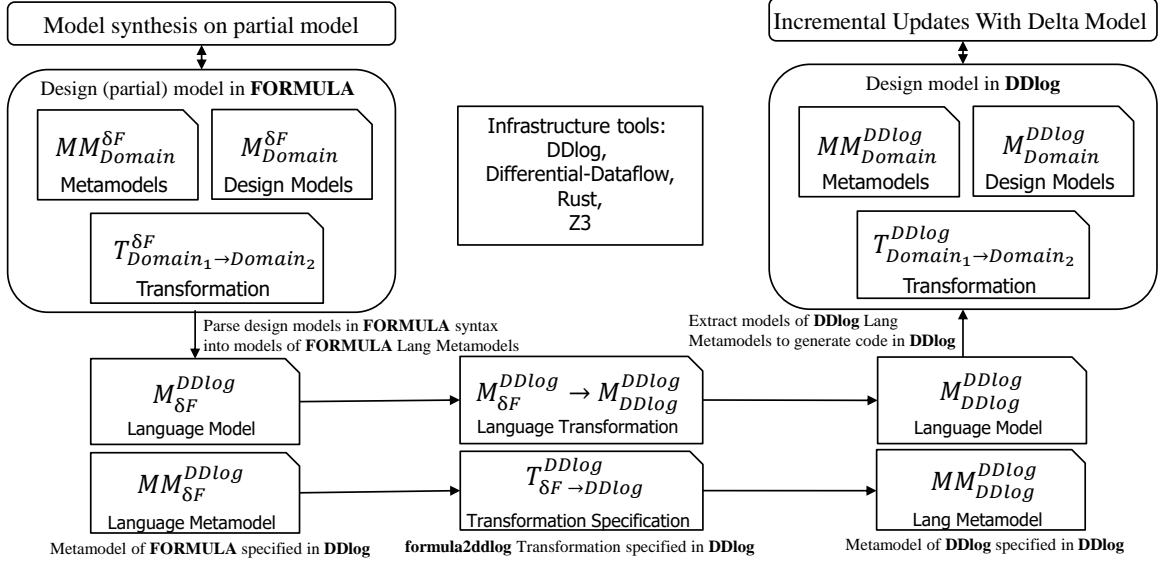


Figure 5.3: Architecture of DDLog-based Incremental Modeling Framework.

5.3 Identification of Semantic Mismatch

The FORMULA language is used for domain-specific modeling, and its syntax provides high-level abstractions for this purpose. DDLog, on the other hand, can be viewed as a more general-purpose logic programming language that extends the traditional Datalog language with additional features. Below is a partial list of the most important semantic differences between FORMULA and DDLog, along with an explanation of selected core language features with examples.

1. DDLog has a larger set of primitive types than FORMULA; this includes both finite-precision and infinite-precision numeric types, reference types, internment types (automatically de-duplicated data with pre-computed hash and atomic reference counting), tuple types, generic types, and collection types (e.g. vector, map, group, set). The set of base types in FORMULA includes integers, floating-point numbers, strings, and finite enumerations. FORMULA *terms* represent the elements in a domain and are built using either base-type constants or user-defined function symbols applied, possibly recursively, to a set of arguments, which are themselves terms. FORMULA has a *behavioral* type system, meaning that FORMULA only cares about the set of values that a type represents, not how that set is written. In other words, the same type can often be expressed multiple ways. This type system has been described in previous works (Jackson et al., 2011a).
2. DDLog uses a Haskell-style *tagged union type* or *sum type* which creates a new type holding a value that can take on several different, but fixed, values. A tag is added to the value to explicitly indicate which type is in use, and the same value used in different tagged unions is viewed as a different type.

In contrast, FORMULA's *union type* represents the mathematical union of the contained types, and the same value used in different unions is viewed as the same type. The type systems of FORMULA and DDLog are modeled in DDLog as part of them are shown in the following metamodels in both textual and graphical syntax. The graphical syntax used in WebGME (Maróti et al., 2014) is similar to UML syntax in which the red arrows denote the inheritance relationship and the black arrows denote the containment relationship with cardinality constraints in the annotations.

3. A DDLog *relation* is a structure to organize strongly typed data, similar to the concept of tables in databases. Input relations denoted by the keyword `input` receive streams of changes (insertion or deletions) and the corresponding changes after the execution of rules is reflected in the output relations denoted by the keyword `output`. The output relation cannot take updates from external input streams. The definition of a relation in DDLog is `[input|output]relation Relname[T]` where `Relname` is the name of the relation and `T` is the type of the data stored in the relation.

In FORMULA, each type has a relation implicitly associated with itself, and relational constraints are written as $Pred(\vec{t})$. DDLog requires the relation name to be written explicitly in relational constraints, such as `Relname[pattern]` unless both the relation and the type have the same name.

4. DDLog negation is exclusively for set difference (antijoin), whereas the meaning of negation in FORMULA depends on the context in which it is used (either set difference or set comprehension).
5. Set comprehension and aggregation are unique features of FORMULA. They are used to reduce a set of provable terms derived from a rule's execution to a single value using aggregation functions such as *count*, *sum*, *max*, and *min*. Below is an example showing the power of this feature:

$$S = \{H_1, H_2, \dots, H_n \mid C_1, C_2, \dots, C_n\} \quad (5.1)$$

$$aggregation = SetcompreOp(S, default) \quad (5.2)$$

$$aggregation \in expr(C_n) \quad (5.3)$$

The set S contains terms derived from the head H_n in the rule $H_1, H_2, \dots, H_n : - C_1, C_2, \dots, C_n$ where C_n is the constraint that needs to be satisfied for head terms in the head to be derived. After the other parts of the rule have been executed, the semantics of the set comprehension group all the facts derived from the rule into one set. In the metamodel of the FORMULA language domain, we model set comprehension as a rule with an additional aggregation function and default value.

On the other hand, aggregation in DDL_{og} is essentially the `groupby` operator over the variables in the constraints in the body of a rule; however, there is support for customization on the aggregation functions in DDL_{og} but in the translation, from FORMULA to DDL_{og} we only have to use a small fixed subset of the aggregation semantics in DDL_{og}.

6. FORMULA allows nested set comprehensions; that is, an outer set comprehension can contain another inner set comprehension as part of the expression in its constraint of the outer set comprehension as shown in $aggregation \in expr(C_n)$ in the definition of set comprehension. An example of nested set comprehension is given in Listing 5.20.
7. FORMULA allows instances (singletons) of a constant type written as a Boolean variable to be used directly as a constraint in a rule. This is typically applied to show the result of a conformance rule, where the singleton of constant type is derived as a fact only if the constraints in the body are all satisfied. For example, in the rule *hasCycle - no Path*(u, u), the variable in the head *hasCycle* (without syntactic sugar as *hasCycle*(\cdot)) is a singleton composite term in disguise that has no arguments to its constructor and can only be derived if the absence of cycles is proved in the graph.

After the identification of semantic mismatches, we create metamodels for both FORMULA and DDL_{og} language domains, and FORMULA programs are parsed into models of the FORMULA language domain that have to conform to the metamodel we defined. In Figure 5.5, the metamodel of type definition syntax of both FORMULA and DDL_{og} are modeled in WebGME's graphical syntax to illustrate the subtle semantic difference between the seemingly similar syntax to define a type with their type systems. The complete metamodels of languages can be found from the link in (Zhang et al., 2021).

With the help of formally defined metamodels of the FORMULA language, we also write conformance rules based on the metamodels for static analysis of the FORMULA program to check the validation of nested set comprehension and rules stratification. This method is more elegant and rigorous than a manually written FORMULA compiler and execution engine in csharp or C++ to validate those properties. A well-formed FORMULA program or a valid model of the FORMULA language domain is then passed to the next step for model transformation.

5.4 Formal Specification of FORMULA Language and Differential-Datalog

The model transformation from models of the FORMULA Language Domain to the DDL_{og} Language Domain cannot be achieved without the formal definition of the syntax and semantics of both languages. We elaborate on the difference of language semantics on similar syntax in both languages and even evaluate the validity of programs with formal reasoning of language models.

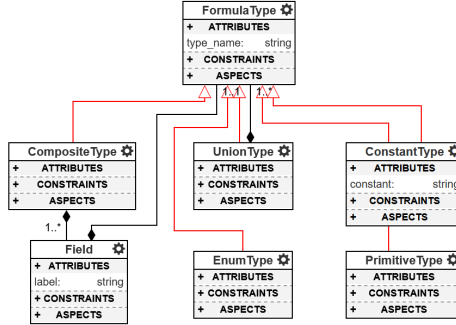


Figure 5.4: Metamodel of FORMULA Language

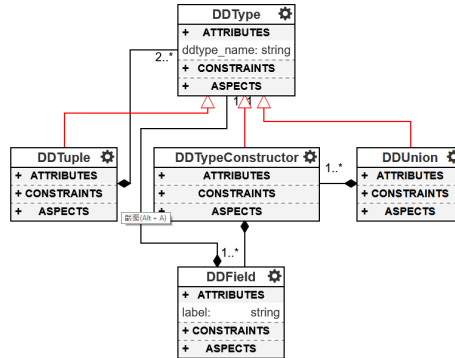


Figure 5.5: Metamodels of DDLLog and FORMULA Modeled Visually in WebGME

5.4.1 Union Type Definition

The union type in FORMULA $U ::= T_1 + T_2 + \dots + T_n$ has different semantics compared with the tagged union in DDLLog as `typedef U = Cons1{t1: T1} | Cons2{t2: T2} | ... | Consn{tn: Tn}` in which each subtype in the union type has a separate constructor to wrap a record of a subtype. However, FORMULA terms of a union type keep the original form and are added to the set of terms represented by the union type.

For type definitions such as $A ::= B + C$, $D ::= E + F$ and $G ::= A + D$, base type B , C , E , F are constructors and type A and D are not constructors. If we don't use `Any` type, the definition of G in DDLLog will be awkwardly cumbersome and overly complicated to be expressed as $G ::= A\{inner : A\} | G :: D\{inner : D\}$ and the collection of type G has to be downgraded twice if we want to do a join with a collection of type B , which is a subtype of the union type G .

The solution is to use the concise built-in `Any` type in DDLLog rather than the traditional tagged union in DDLLog to represent FORMULA union types because the `Any` type includes any combination of multiple constructors such as $A + B + C + D + E + F$ in our previous example of nested union type definition. Therefore, the current implementation of *differential-FORMULA* adopts `Any` type because dealing with the translation of FORMULA nested union type into DDLLog is nontrivial if we choose to translate nested union type in FORMULA into nested tagged union types in DDLLog. This solution elegantly solves the type seman-

tic mismatch between the two languages is to convert all explicit union types in FORMULA into `Any` type in DDL_{og}.

However, the disadvantage of this approach is that there are some additional overheads in the type conversion of each term that belongs to a union type. All join operations involving union types are executed on an over-estimated set of terms because `Any` type includes all existing terms in the set, which means the final computation dataflows have to process more unnecessary data.

In DDL_{og}, `Any` type is mapped to a whole collection or data stream that includes all term facts. The collection of other types is always the subset of this collection. Other subtype data collections can be obtained by passing the `Any` type data stream through some filtering operators in DDL_{og}. There will be type conversion overheads that slow down the overall performance of rule execution.

The term collection of a union type or even nested union type is always the union of term collections of its subtypes with the basic constructor. There are no other nested union types in any of the subtypes of a union type because the nested union types are all flattened to the ground level with only constructor-based types or built-in basic types. In the previous example, $G ::= B + C + E + F$ is the end result after type reduction rather than $G ::= A + D$ in which both A and D are union types themselves that can be further reduced to more basic types because $A ::= B + C$ and $D ::= E + F$.

The following three parts are the intermediate rules, types, and relations that are automatically generated in the translation of Union Type from FORMULA to DDL_{og}

- If the type of an argument is a union type in FORMULA, we replace it with `Any` type in DDL_{og}, which is an overestimated set of terms. If U_1 is a union type and a constructor $U_2 ::= \text{new}(\text{inner} : U_1)$ uses U_1 as the type of one of its arguments, the type representation in DDL_{og} is $U_2 ::= \text{new}(\text{inner} : \text{Any})$
- A separate DDL_{og} relation specifically for this union type U to hold all terms from several different normalized sub-types that represent subsets of the terms represented by this union type U .
- Generate new rules to derive the terms for the union type relation from the other relations of the normalized subtypes of the current union type. $U[x] : -U_i[x]$.

For example, in FORMULA both union type $C ::= A + B$ and $D ::= A + \text{String}$ contain the term $A(0)$ because each FORMULA type represents a set of dynamically typed items but in DDL_{og} the record $A(0)$ is wrapped by two constructors as $\text{CA}\{A(0)\}$ and $\text{DA}\{A(0)\}$ respectively. Additionally, the FORMULA constant is natively translated to DDL_{og} zero-argument constructor with the same semantics.

An implicit union type is implicated by terms of various types in H_n in the head of the set comprehension such as in Listing 5.3 where there are not only terms of type `Node` in the set but also atom term

of type `String` and terms of type `Edge` that could be added into the set. A hidden implicit union type `SetcompreHeadUnion ::= String + Edge + Node` is created behind the scene to represent the type of all possible terms that could exist in the set of terms derived from set comprehension.

```
SomeAggregation(amt) :- amt =
    count({ "hello", src, Edge(dst, src), Edge(src, dst) |
        e is Edge(src, dst)
    }).
// An union type generated implicitly in set comprehension
SetcompreHeadUnion ::= String + Node + Edge.
```

Listing 5.3: Implicit Union Type in Set Comprehension

The type inference for the implicit union types in the set comprehension is handled by two rules in the model transformation from FORMULA to DDL_g. In Listing 5.4, the first rule infers the type for each term in the head of the set comprehension and the second rule aggregates the types into a single union type. Therefore, the aggregation later will be applied to the values of this union type rather than values of various types.

```
TypeInHead(type, inner_rule) :-
    Constraint[SetcompreExpr(inner_rule), rule],
    Head[subterm, inner_rule],
    Pred[term, inner_rule],
    SubtermType[subterm, term, type].

Type[union_type] :- TypeInHead(type, inner_rule),
    var type_group = (type).group_by(()),
    var union_type = to_union_type(type_group).
```

Listing 5.4: Union Type Generation

5.4.2 Negation

In general, negation in a logic programming language is denoted by an additional negation prefix before a predicate constraint in the rule. It means the absence of the terms of certain patterns can prove the existence of some facts in the head of the rule. Below is the definition of negation in FORMULA, where each t_n in the predicate is a term (an atom, a variable, or a constructor combining terms recursively)

$$Head : - P_1(\vec{t}_1), P_2(\vec{t}_2), \dots, P_m(\vec{t}_m), no P_n(\vec{t}_n)$$

The meaning of negation in FORMULA differs depending on its context: it either means (1) the absence of facts of a certain pattern defined in the predicate constraint $P(\vec{t})$, or (2) the set difference between two collections of records of different patterns. However, the semantics of negation in DDLog is strictly set difference and the negation translates to the *antijoin* operator in differential-dataflow.

```
HasNoCycle :- no Path(u, u).
NoCycle(u) :- u is Node(u), no Path(u, u).
```

Listing 5.5: Rules with negated predicates

For example, the rule $HasNoCycle : - no Path(u, u)$ derives the term $HasNoCycle$ if there is no path of the self cycle with the pattern expressed as $Path(u, u)$ found in the graph. FORMULA will aggregate all existing self-cycle paths $set = \{p \mid p \text{ is } Path(u, u)\}$ into a set and apply set comprehension to check the size of the set. The negated predicate is evaluated to be true only when the size of the set is zero.

However, the following rule with negation under a different context $NoCycle(u) : - u \text{ is } Node(-), no Path(u, u)$ has a completely different semantics because the variable u outside the negated predicate constraints also exist in the negated predicate itself. The semantics in this example are altered to mean, “find all nodes in the graph that do not exist in the path of a cycle.” In this case, both FORMULA and DDLog treat the negation as the difference between two sets over one variable u : the set of all nodes as $u \text{ is } Node(-)$ and the set of all nodes that occur in the path of a cycle as $Path(u, u)$.

The context of negative predicate constraints decides if the negation should be interpreted as set difference or set comprehension. In our `formula2ddlog` transformation, the following rules are executed to derive new facts about each negated predicate under different contexts. One of the transformation rules even uses negated predicates operating on the models of all negated predicates in the FORMULA program that do not have the set difference semantics.

```
NegPredAsSetdiff(pred, rule) :- NegPred(pred, rule),
    PosVar(variable, rule),
    NegVar(variable, rule),
    VarOfTerm(variable, pred).
```

```
NegPredAsSetcompre(pred, rule) :- NegPred(pred, rule),
```

not NegPredAsSetdiff(pred, rule).

Listing 5.6: Reasoning over negated predicates

NegPred, NegPredAsSetdiff, NegPredAsSetcompre are relations with two columns to represent different relations between negated predicate constraints and the rule that is associated with it. The first rule in Listing 5.6 adds new facts into the relation NegPredAsSetdiff for each negated predicate if a variable in the negated predicate also exists in one of the positive predicate constraints in the same rule, otherwise, the new facts are added into the relation NegPredAsSetcompre by the second rule in Listing 5.6. The negated predicate is then translated to an equivalent expression in DDLLog based on the new information derived from the rules. The final form of equivalent DDLLog expression is either the same negated pattern expression *not Rel[pattern]* for set difference semantics, or *Rel[pattern], var count = (\vec{v}_n).group_by(\vec{v}_p), count = 0* for set comprehension semantics.

5.4.3 Aggregation

The aggregation or set comprehension in FORMULA is converted into three parts in order to be mapped into an equivalent DDLLog program.

1. A implicit union type is inferred by combining every type in the head of a set comprehension, even though the type of each term in the head may not be explicitly specified in the rule.
2. A new rule is added into DDLLog to derive new facts wrapped in the form of the new union type because DDLLog only accepts strongly typed data in every relation while FORMULA accepts dynamically typed data (this is inferred and checked by the FORMULA compiler at load time).

$$U ::= UT_1 + UT_2 + \dots + UT_n \quad (5.4)$$

$$UT_1(h_1), UT_2(h_2), \dots, UT_n(h_n) :- C_1, C_2, \dots, C_n \quad (5.5)$$

3. The set of items of union type is further aggregated into a single numeric value stored in a new relation *Aggregation* and a new relational constraint is appended to the rule that uses the result of the set comprehension as shown in (7).

$$Aggregation(result) :- result = Op(\{u \mid U(u)\}). \quad (5.6)$$

$$H_1, H_2, \dots, H_n :- C_1, C_2, \dots, C_n, Aggregation(result). \quad (5.7)$$

4. The new set comprehension *result = Op(\{u \mid U(u)\})* on a set of items of only one type (union type)

is consistent with the semantics of the *groupby* operator in DDLLog and is translated into the form $group = (u).group_by((\vec{v}))$. \vec{v} could be empty if the set comprehension is a closure that does not share variables with the outer scope, similar to how the meaning of negation is determined (see Section 5.4.2).

The nested set comprehension or aggregation is handled with exactly the same rules used for single set comprehension with no additional rules because the declarative transformation rules recursively translate each set comprehension and replace it with the corresponding DDLLog expressions. This example shows one advantage our approach to translation using a formal specification has over the alternative of manually writing imperative code with recursion to translate from one language to another.

5.5 Static Analysis on FORMULA Programs

FORMULA is considered the guard for the specification and execution of DSMLs but the FORMULA language itself including the implementation of its own compiler and the translator to other languages like DDLLog is not guarded by any guard. Therefore, we have to formally specify the FORMULA language itself, detect the semantic errors at an early stage and write transformation rules to eventually transform a FORMULA program into a DDLLog program. For future work, we are planning to implement this as a FORMULA transformation, with both FORMULA and DDLLog language domains specified in the FORMULA language. This approach will give confidence to the translation between the two languages through the use of a formal specification.

Static analysis is necessary in our use case to check the validity of the FORMULA program semantically after the program is parsed by our new FORMULA language parser implemented in Rust language. A trace of error is provided as the result of execution of conformance rules in the same way as in the old version of P language compiler (Desai et al., 2013) to prove the correctness of FORMULA program formally with traceable proofs. The static analysis of the FORMULA program is a crucial step to gain knowledge of the structure of the program and for the type inference in the FORMULA program and transformation in FORMULA because type conversion may exist under certain circumstances in the model transformation from FORMULA to DDLLog.

Below is a FORMULA program example to demonstrate the core concepts such as algebraic data type, union type, and other language features of FORMULA language. We use this toy example to explain the semantic difference, modeling of language domain, model transformation, and code generation in the following sections.

```
domain Graph {  
    Node ::= new(name: Integer).}
```

```

Edge ::= new(src: Node, dst: Node).
Triangle ::= new(one: Node, two: Node, three: Node).
Item ::= Node + Edge.
Union ::= Triangle + Node.

//NestedUnion is a union of two union types but it can be decomposed
//into the union of three base types:
NestedUnion = Triangle + Node + Edge.
NestedUnion ::= Union + Item.

//Both are similar with constructor Edge but its arguments are more inclusive.
BigEdge ::= new(src: NestedUnion, dst: NestedUnion).
GigaEdge ::= new(src: NestedUnion, dst: Node).
}

```

Listing 5.7: Example of Nested Set Comprehension

This FORMULA graph domain not only models the conventional graph with nodes and edges but also models a hierarchical graph in which we expand the definition of the node to be a union type as either an integer number, an edge, or even a triangle as part of the graph. The new `Node` type with the expanded node definition is named `NestedUnion` and is the type of some arguments in constructors such as `BigEdge` and `GigaEdge`.

5.5.1 Language Domain Modeling and Metamodels

Both static analysis of the FORMULA program and model transformation to generate an equivalent DDLLog program relies on a complete definition of the metamodels of FORMULA and DDLLog language. Both languages belong to the category of logic programming language.

The FORMULA language is modeled in DDLLog and the core language is simple with main concepts such as `Term`, `TypeSpec`, `Constraint`, `Expr`, `Rule`, `Model` and `Domain`. Some of the building blocks such as `Term` may contain other terms of the type `Term` but self-reference is not allowed to avoid endless recursion. The other main feature of FORMULA is that it allows nested set comprehension in which the constraints in the set comprehension may have another inner set comprehension. Below is part of the meta-models of FORMULA language that is used to specify other DSLs.

```

// There are some restrictions on the field name for each variant that they
// have to be unique

```

```

typedef Term =
  AtomStr { i0: string } | AtomInt { i1: signed<64> } |
  AtomPosInt { i2: signed<64> } | AtomNegInt { i3: signed<64> } |
  AtomFloat { i4: float } | AtomBool { i5: bool } |
  Variable { root: string , fragments: Vec<string> } |
  // Constant can be viewed as a Composite term without arguments
  Composite { name: string , arguments: List<Ref<Term>> }

typedef Field = Field { field_name: string , type_spec: TypeSpec }
typedef TypeSpec = Integer | Boolean | FloatNum | String |
  CompositeType { name: string , arguments: Vec<Field> } |
  UnionType { name: string , subtypes: Vec<TypeSpec> } |
  EnumType {name: string , items: Vec<Term>} |
  // ConstantType is a CompositeType without arguments
  ConstantType {constant: string}

// The head terms and conditions are the same as head and body in a rule so the set
// comprehension can be viewed as a rule and set comprehension is applied to the set
// of derived terms
typedef Setcompre = Setcompre { rule: Ref<Rule>, sop: SetOp, default: Term }
typedef Expr = BaseExpr { term: Term } |
  SetcompreExpr { sc: Ref<Setcompre> } |
  UnaryExpr { expr: Ref<Expr>, uop: UnaryOp } |
  ArithExpr { left: Ref<Expr>, right: Ref<Expr>, aop: ArithOp }

// Constraints are either predicates or expressions that may contain
// set comprehension 'ExprCons' could be
// 1) a binary constraint over two expressions.
// 2) An assignment that holds the result of the evaluation of right-hand side expression.
// 3) Type constraint in the form of 'var: Type'
typedef Constraint = PredCons { negated: bool , term: Term, alias: Option<string> } |
  BinaryCons { left: Expr , right: Expr , bop: BinOp } |
  AssignCons { variable: Term , expr: Expr } |
  TypeCons { variable: Term , var_type: TypeSpec }

```

```

typedef AnyTypeEnum = ToAny | FromAny | NoConversion

// A rule is composed of a linked list of head terms and a linked list of constraints
// Rule has at least one constraint and at least one head term.
// The term in the head could be a ground term and only count once in the set
typedef Rule = Rule { id: string , head: NonNullList<Term>,
body: NonNullList<Constraint> }
typedef Domain = Domain { typedefs: List<TypeSpec>, rules: List<Rule> }
typedef Model = Model { terms: List<Term>}

```

Listing 5.8: Metamodels of FORMULA Language Domain

DDLog language is modeled in DDLog itself and we put a prefix in front of the name of some language concepts such as `DDTerm` in order to distinguish from similar language concepts in FORMULA. DDLog as an incremental logic programming language is very similar to FORMULA with many similar concepts such as term, constructor, and rule but fundamentally different in its semantics. DDLog has its unique concepts of `Relation` that represent a collection of records of certain types while FORMULA does not have the same restriction and any set of limited or unlimited terms can form a unique type in FORMULA. The DDLog language has more flexibility for customization in that it supports adding external functions and embedding imperative code programs to DDLog programs but the execution of such programs could be nondeterministic and harder to analyze depending on the external functions.

```

typedef DDField = DDField { field_name: string , type_spec: Ref<DDTypeSpec> }
typedef DDTypeCons = DDTypeCons { cons_name: string , fields: List<DDField> }
typedef DDTypeSpec = BigInt | Bool | Str | Bitvector | Int | Double | Float | Any |
// (T1, T2, ..., Tn) and the tuple cannot be empty
DDTupleTypeSpec { name: string , tuple: NonNullList<Ref<DDTypeSpec>> } |
// T = T1 {...} | T2 {...} | ... | Tn {...} as tagged union and cannot be empty
// Tagged union in DDLog is a sum type of more than one constructors
// FORMULA composite type can be represented as a sum type of exactly
// one constructor as typedef T = T {...}
DDUnionTypeSpec { name: string , types: NonNullList<DDTypeCons> }

typedef DDTerm = DDBigInt { i0: bigint } | DDBool { i1: bool } | DDStr { i2: string } |
DDBitvector { i3: bit<64> } | DDInt { i4: signed<64> } |
DDDdouble { i5: double } | DDFloat { i6: float } |
// A variable term in ddlog represented by a string

```



```

DDVar { name: string } |
// var xxx = .., add keyword in the front to denote it is only for declaration
DDVarDecl { name: string } |
// It has type name and a list of DDLog expressions
DDCons { cons_name: string, arguments: List<DDEExpr> }

// DDRelation represents a collection of records of certain type
typedef DDRelation = DDRelation { is_input: bool, name: string, type_spec: DDTypeSpec }

// There are more expression in DDLog referene and we only pick a few we need
typedef DDEExpr = DDTermExpr { term: Ref<DDTerm> } |
// A tuple of DDEExpr as (e1, e2, ..., en)
DDTupleExpr { exprs: Vec<Ref<DDEExpr>> } |
// Function call as 'func(arguments)'
DDFunctionCallExpr { func_name: string, arguments: Vec<Ref<DDEExpr>> } |
// Function call on an object as 'a.call(arguments)'
DDDottedFunctionCallExpr { obj: Ref<DDEExpr>, func_name: string,
arguments: Vec<Ref<DDEExpr>> } |
// Access struct field by identifier like 'list.node'
DDTermFieldExpr { expr: Ref<DDEExpr>, field: string } |
// Unary operator over expression
DDUnaryExpr { expr: Ref<DDEExpr>, uop: UnaryOp } |
// Arithmetic operator over two expressions
DDArithExpr { left: Ref<DDEExpr>, right: Ref<DDEExpr>, aop: ArithOp } |
// Binary operator over two expressions
DDBinExpr { left: Ref<DDEExpr>, right: Ref<DDEExpr>, bop: BinOp }

// The complete form of DDLog predicate is 'u in RelName[Term]' to
// represent predicate constraint
typedef DDAtom = DDAtom { var_name: Option<string>, rel: Ref<DDRRelation>,
expr: DDEExpr }
typedef DDRhs = DDRhsAtom { negated: bool, atom: DDAtom } |
DDRhsCondition { expr: DDEExpr } |
DDRhsAssignment { to: DDEExpr, from: DDEExpr } |
// if the default set comprehension operators are not flexible enough

```

```

// 'DDGroup' could be viewed as a special case of 'DDRhsAssignment'
DDGroup { var_name: string, group: DDEExpr, by: DDEExpr }

typedef DDRule = DDRule { head: Vec<DDAtom>, body: Vec<DDRhs> }

```

Listing 5.9: Metamodels of Differential-Datalog Language Domain

5.5.2 Type Inference of Non-ground Term

The non-ground term is a term that has variables in it and it represents a set of ground terms. The inferred type information in each variable can be used to gain a more comprehensive understanding of the meaning of a FORMULA program for semantic error detection and assist the transformation from FORMULA to DDLog in later steps to derive a DDLog program.

5.5.2.1 Type Inference of Sub-terms

Every non-ground term has a certain structure that is defined by its type and this type definition contains the type information of the arguments of a type. The type of a variable in a non-ground term can be traced down by locating the position of the variable in that term and recursively finding the types of its arguments in a type until reaching the same position. For example, *be1* is $BigEdge(Edge(n1, Node(0)), n3)$. is the definition of a non-ground term, and the type of variables *n1* and *n3* are not specified but given the type definition of $BigEdge ::= new(src : NestedUnion, dst : NestedUnion)$. The type of variables in the term can be inferred directly from their positions that the type of *n3* is `NestedUnion` and the type of *n1* is `Node`.

```

Edge ::= new(src : Node, dst : Node).
e1 is Edge(Edge(n1, Node(0))).
BigEdge ::= new(src : NestedUnion, dst : NestedUnion).
be1 is BigEdge(Edge(n1, Node(0)), n3).
be2 is BigEdge(Edge(n, Node(0)), n).

```

Listing 5.10: Subterm Type Inference Example

In some special cases when a non-ground term has more than one variable with the same name. For example, *be2* is $BigEdge(Edge(n, Node(0)), n)$. is a variant of non-ground term *be1* with two *n* in two places and each one has a different type that the first *n* is of type `Node` and the second *n* is of type `NestedUnion`. The type of variable *n* here is the overlap of two sets of terms represented by both type `Node` and type `NestedUnion`. Since `Node` is a subset of `NestedUnion`, the type of *n* is still `Node`. In general, the type of a variable *n* with multiple occurrences is $T = T_1 \cap T_2 \cap \dots \cap T_{n-1} \cap T_n$, and if $T = \emptyset$ it means that there is type conflict in the type definition or in other words that the non-ground term is invalid.

```

output relation SubtermTypeSpec [(Term, Term, TypeSpec)]
// A term is a sub-term of itself and the SubtermTypeSpec relation is a tuple
// of (argument, term, type of the argument)
SubtermTypeSpec[(c, c, t)] :- c in Term[Composite{cons_name, arg_list}],
    t in TypeSpec[CompositeType{cons_name, _}].

// Recursively infer the type of the direct children of a term until
// a sub-term of basic type such as integer is reached.
SubtermTypeSpec[(arg, ancestor, arg_type_spec)] :-
    SubtermTypeSpec[(term, ancestor, CompositeType{_, arg_types})],
    ArgOfTerm[(index, arg, term)],
    var arg_type_spec = arg_types.nth(index).option_unwrap_or_default().type_spec.

```

Listing 5.11: Type Inference Rules in DDLog

5.5.2.2 Term Unifiability Checking

One of the applications of type inference is to compare the types of two terms for term unifiability checking. Term unifiability checking is crucial for the reasoning of semantically correct FORMULA program and also the execution of the FORMULA program because the order of the execution of FORMULA rules has to be determined based on the dependency relationships between rules.

The rules have to be stratified into strata in a way that the execution of rules from a higher stratum cannot produce new terms that can be the input of rules of a lower stratum.

```

R1: Edge(Node(1), n) :- Node(n).
R2: Edge(n, n) :- Edge(Node(2), n)

```

Listing 5.12: Unifiability Checking in FORMULA Rules

Rule *R2* is a recursive rule that keeps generating new derived terms until a fixed point is reached and no more new terms can be added to the term set. In some cases, rule *R2* may not terminate depending on the input and the rule itself. Given the example above with only two FORMULA rules, both rules may derive new terms of type `Edge`, and rule *R2* takes the term of type `Edge` as input too. In other logic programming languages like Datalog or Prolog, both rule *R1* and rule *R2* have to be put in the same stratum because the input of rule *R2* depends on the output of rule *R1* and rule *R2* itself.

However, every non-ground term in the FORMULA rule represents a unique set of terms that can be determined by type inference and affect the stratification of FORMULA rules. In FORMULA the output of one rule may not be the input of another rule even though terms in both rules have the same data constructor

such as `Edge` in our example. $Edge(Node(1),n)$ and $Edge(Node(2),n)$ represent two different sets of terms and they are disjoint so any term derived from $Edge(Node(1),n)$ in rule $R1$ will not contribute the input of rule $R2$. Therefore, rule $R1$ and rule $R2$ are independent of each other so they can be put in two separate strata and executed independently.

```
// Variables are not ground terms.
NonGroundTerm[term] :- term in Term[Variable {}].

// A composite term is a non-ground term if any of its subterms are non-ground terms.
NonGroundTerm[term] :- term in Term[Composite {}],
    ArgOfTerm[(index, arg, term)], NonGroundTerm[arg].

// Use the set difference to find ground terms.
GroundTerm[term] :- Term[term], not NonGroundTerm[term].

// Subset, overlap, and disjoint?
// 1. Two different ground terms are obviously in the DisjointSetRel relation.
DisjointSetRel[(term1, term2)] :- GroundTerm[term1], GroundTerm[term2], term1 != term2.

// 2. Term A and Term B are in the DisjointSetRel relation if they belong to the same
// composite type but at least one of
// Their arguments at the same position are in the DisjointSetRel relation too.
DisjointSetRel[(term1, term2)] :- term1 in NonGroundTerm[Composite{name, args1}],
    term2 in NonGroundTerm[Composite{name, args2}],
    term1 != term2,
    ArgOfTerm[(index, arg1, term1)], ArgOfTerm[(index, arg2, term2)],
    DisjointSetRel[(arg1, arg2)].

// Only apply to two composite terms of the same type.
NonDisjointSetRel[(term1, term2)] :- term1 in NonGroundTerm[Composite{name, args1}],
    term2 in NonGroundTerm[Composite{name, args2}],
    not DisjointSetRel[(term1, term2)].
```

Listing 5.13: Rules for Type Unifiability Checking

The first two rules above recursively check if variables exist in a term to derive all non-ground terms in relation `NonGroundTerm`. The third rule uses negation in the rule to derive all ground terms in the relation

GroundTerm by filtering out all terms that are not non-ground terms. The next step is to check if two non-ground terms can be unified or not since each non-ground term represents a unique type or a set of ground terms. Here we use the same tactic to find unifiable pairs of terms by finding the non-unifiable pairs of terms first. The relation `DisjointSetRel` holds all pairs of terms that cannot be unified because two ground terms are obviously not the same term or two non-ground terms have different ground terms inside themselves at the same position. Once `DisjointSetRel` is derived, the relation `NonDisjointSetRel` can be easily derived by negation in the exact same way `NonGroundTerm` is derived. Both `DisjointSetRel` and `NonDisjointSetRel` is important to the reasoning over FORMULA programs because FORMULA has its own unique type semantics and the order of rule execution or rule stratification changes based on the type semantics of terms.

5.5.3 Type Inference In the Context of FORMULA Rules

The FORMULA rule above does not have to distinguish the type of variable b in the predicate $GigaEdge(b, c)$, which is a union type, and the variable b in the predicate $GigaEdge(b, c)$, which is a *Node* type according to the type definition of *GigaEdge*. FORMULA internally joins two sets of terms without checking the specific type of variables that are going to be joined. Every variable in FORMULA simply represents a set of terms and FORMULA does not require that the variables in the join operation have to be the same type as DDLLog does otherwise the join operation will return an unexpected empty set of terms.

```
BigEdge ::= new(src: NestedUnion, dst: NestedUnion).
GigaEdge ::= new(src: NestedUnion, dst: Node).
Nested Union ::= Triangle + Node + Edge.
GigaEdge(b, a) :- GigaEdge(a, b), GigaEdge(b, c), GigaEdge(c, a).
```

Listing 5.14: FORMULA Rule

On the other hand, DDLLog also has a similar concept of FORMULA union type named *Any* type in DDLLog, which is a wrapper that converts all terms of different constructors into the same *Any* type. This is the solution to resolve the semantic mismatch when it comes to joining operation over variables of union type. The final DDLLog rule we should have after the translation is below and it has used `to_any()` and `from_any()` to specify if the terms in one variable have to be converted into terms of *Any* type or the other way around.

```
GigaEdge(to_any(b), from_any(a).unwrap()) :- GigaEdge(a, b),
GigaEdge(to_any(b), c), GigaEdge(to_any(c), from_any(a).unwrap()).
```

Listing 5.15: DDLLog Rule Translated from FORMULA Rule

The first predicate $GigaEdge(a,b)$ is matched first in the rule execution with variable a matched to a collection of terms of type `Any` and with variable b matched to a collection of terms of type `Node`. The second predicate $GigaEdge(b,c)$ has to use the $to_any()$ function to convert every term of `Node` type from variable b into a term of `Any` type because in DDLLog only two terms of the same type can be compared otherwise they are treated as two different terms even though one term has the exact same meaning as the other one.

The type of the variable is determined by the position of its first occurrence because DDLLog rule execution strictly starts from left to right to build a dataflow graph. For example, the variable a is set to `Any` type in the first occurrence in $GigaEdge(a,b)$ so the type of variable a is firstly fixed to `Any` type but in later predicate, it has to be converted back into `Node` type in predicate $GigaEdge(c,a)$ with additional function $from_any()$ wrapping around variable a . The same conversion applies to the variables in the predicates in the head of the rule in order to derive the terms with the correct type.

In order to translate the FORMULA rule into an equivalent DDLLog rule, we create a new relation named *ConsVarConversion* to track the position of every variable in a predicate of a rule and an enumeration type to decide if keyword $to_any()$ or $from_any()$ has to be added for the type conversion in DDLLog's dataflow construction.

Since DDLLog always starts from the left side, every variable in the leftmost predicate does not need to be converted and type `Conversion` is an enumeration type that decides how the term should be converted such as *FromAny*, *ToAny* or *NoConversion*. The following rule finds out the first predicate represented as $ConstraintInRule[(0, PredCons false, pos_term, _, rule)]$ in all existing rules and derives some new facts saying no conversion is needed for the first occurrence of a variable in the first predicate.

```
ConsVarConversion[(0, indices, variable, pos_term, rule, NoConversion)] :-
    DescendentVarOfTerm[(indices, variable, pos_term)],
    ConstraintInRule[(0, PredCons {false, pos_term, _}, rule)].
```

Listing 5.16: No Conversion for the Leftmost Predicate Expressed in DDLLog Rule

Since the first occurrence of a variable in the rule (anchor variable) may not occur in the first predicate in a rule, we have to find which predicate in the rule has the first occurrence of a certain variable, and that variable never occur in any of the previous predicates on the left side in a FORMULA rule $head : - pred_1, pred_2, pred_m(v), \dots, pred_n(v)$ in which the first variable v occurs in $pred_m(v)$. The following DDLLog rule accumulates all variables that occur before the current predicate in predicates $pred_1, pred_2, \dots, pred_{m-1}$ and negation is applied to find all matches in which no variable v in the accumulated variable set. The newly derived facts in relation `ConsVarConversion` have the last argument set to *NoConversion* because the

first variable that occurs in a rule is an anchor variable that determines the type of that variable in the rule and other variables are converted based on the type of the anchor variable.

```

ConsVarConversion[(index2, indices2, variable2, pos_term2, rule2,
NoConversion)] :-
    // Locate a variable in a FORMULA rule
    ConstraintInRule[(index, PredCons {false, pos_term, _}, rule)],
    DescendentVarOfTerm[(indices, variable, pos_term)],

    // Find all predicates and variables before a certain predicate
    ConstraintInRule[(pre_index, PredCons {false, pre_pos_term, _}, rule)],
    pre_index < index,
    DescendentVarOfTerm[(pre_indices, pre_variable, pre_pos_term)],

    // Find all variables in the previous predicates before the current predicate
    var pre_var_group = (pre_variable).group-by((index, indices, pos_term,
variable, rule)),
    (var index2, var indices2, var pos_term2, var variable2, var rule2) =
        pre_var_group.group-key(),
    var variable_set = pre_var_group.to_hashset(),
    not hashset_contains(variable_set, variable2).

```

Listing 5.17: DDLLog Rule for Finding Anchor Variables

The places to add `to_any()` conversion are where a variable v of a union type appears in $pred_n(v)$ but appears not to be a union type in $pred_m(v)$. $pred_m(v)$ must be prior to $pred_n(v)$ and this is one of the situations in that we need to add `to_any()` keyword to denote the conversion when translating from FORMULA rules to DDLLog rules. The following DDLLog rule has two parts the first part enumerates all variables of union type and the second part enumerates all possible predicates that contain the same variable but do not have to be converted or in other words the anchor variable.

```

ConsVarConversion[(index, indices, variable, pos_term, rule, ToAny)] :-
    // Find the variable that is of union type.
    ConstraintInRule[(index, PredCons {false, pos_term, _}, rule)],
    DescendentVarOfTerm[(indices, variable, pos_term)],
    DescendentTermTypeSpec[(indices, variable, pos_term, UnionType{ })],

    // Find another predicate term prior to the current predicate

```

```

// that has the same variable in it but not union type.
DescendentTermTypeSpec[(pre_indices , variable , pre_pos_term , non_union_type)],
not is_union_type(non_union_type),
ConsVarConversion[(pre_index , pre_indices , variable , pre_pos_term , rule ,
NoConversion)],
pre_index < index .

```

Listing 5.18: DDLLog Rule for Locating *ToAny* Conversion Places

The first occurrence of variable v is of union type in $pred_m(v)$ but the next one is not a union type in $pred_n(v)$ and we need to convert from union type to regular type with the keyword *from_any*().

```

ConsVarConversion[(index , indices , variable , pos_term , rule , FromAny)] :-
// The first occurrence of variable that is born to be union type.
ConsVarConversion[(pre_index , pre_indices , variable , pre_pos_term , rule ,
NoConversion)],
DescendentTermTypeSpec[(pre_indices , variable , pre_pos_term , UnionType{ })],
// In the current predicate , the variable is not union type .
ConstraintInRule [(index , PredCons {false , pos_term , _}, rule)],
pre_index < index ,
DescendentVarOfTerm [(indices , variable , pos_term)],
DescendentTermTypeSpec [(indices , variable , pos_term , non_union_type)],
not is_union_type(non_union_type).

```

Listing 5.19: DDLLog Rule for Locating *ToAny* Conversion Places

5.5.4 Nested Set Comprehension Validation

This subsection describes additional validation rules, implemented as static checks in our translator, that ensure the FORMULA programs being translated can be executed by the DDLLog engine. The FORMULA compiler contains similar semantic checks that catch these inconsistencies.

Additionally, we are working towards a complete model of the FORMULA language domain with well-defined metamodels and models in the FORMULA language itself to ensure the semantic correctness of FORMULA programs with proofs. The DDLLog language metamodels will be defined, as well, to create a formally specified transformation between FORMULA and DDLLog.

An example of a nested set comprehension is Listing 5.20 where one of the constraints inside a set comprehension contains another set comprehension to check the number of existing nodes.

```

TotalDegree(amt) :- amt = count({src | e is Edge(src , dst),

```



```

node_amt = count({n | n is Node(x)}),
node_amt > 1
}).

```

Listing 5.20: Example of Nested Set Comprehension

The basic definition of set comprehension and its semantics are described in 5.3 but the semantics of set comprehension is slightly altered when variables outside the set comprehension exist in the constraints of set comprehension. The first rule in Listing 5.21 depicts an example, where each possible concrete value in variable *src* creates a separate set comprehension with the variable *src* substituted in the constraints of the set comprehension with the concrete value as in the second rule in Listing 5.21.

```

(1) OutdegreeByNode(src, amt) :- src is Node(_),
    amt = count({dst | e is Edge(src, dst)}).

(2) OutdegreeByNode(Node(1), amt) :- src is Node(1),
    amt = count({dst | e is Edge(Node(1), dst)}).

```

Listing 5.21: Example of Nested Set Comprehension

Each constraint C_n could have a nested set comprehension expression inside itself to express more complex constraints. The example in Listing 5.20 has a set comprehension inside another set comprehension, which means the outer set comprehension to compute the total degrees of the graph has nodes in the set $\{src \mid e \text{ is } Edge(src, dst)\}$ only if the number of nodes computed in the inner set comprehension as $node_amt = count(\{n \mid n \text{ is } Node(x)\})$ is larger than one. Semantic problems arise if one more set comprehension containing variables from the outer rule such as the variable *src* is added to the constraints of the inner set comprehension. A valid FORMULA program only allows two adjacent layers of nested set comprehensions to share variables.

The detection of such conflicting variables in nested set comprehensions in a FORMULA program is succinctly expressed in the following rules in our DDLLog translator. These rules check the validity of nested aggregations with more than two layers and pinpoint the exact cause of the conflict with error messages expressed as newly derived facts; the error message displayed appears as, `Error(r1, r2, "Variable conflict in nested set comprehension")`.

```

// Derive constraints from a rule
Constraint[constraint] :- Rule[rule],
    constraint = FlatMap(rule.constraints).

```

```

// Derive the inner rule from a constraint that reduces
// a set into a single value.
// The set s = {head | body} is a rule in disguise
Rule[inner_rule] :- Constraint[SetcompreExpr(inner_rule), rule].

RuleContainment[r1, r2] :- Rule[r1], Rule[r2],
    Constraint[SetcompreExpr(r1), r2].

RuleContainment[r1, r3] :- RuleContainment[r1, r2],
    RuleContainment[r2, r3].

Error(r1, r2, "Conflict_in_nested_set_comprehension") :-
    RuleContainment[r1, r2],
    not Constraint[SetcompreExpr(r1), r2],
    VarInRule[variable, r1],
    VarInRule[variable, r2].

```

Listing 5.22: Nested Set Comprehension Validation Rules

5.5.5 Rule Stratification Validation

Rules in the FORMULA program are *stratified* into two dependency graphs to decide the order of rules for execution. Recursion on positive predicates is allowed with valid semantics while recursion involving negation does not have valid semantics such as $HasCycle(u) :- u \text{ is } Node(_), no \text{ HasCycle}(u)$ because the absence of a predicate cannot be used to derive new facts of the same type recursively.

Rule $R1$ depends on rule $R2$ positively if a term in the body of rule $R1$ is unifiable with a term in the head of rule $R2$ because the new facts derived from rule $R2$ trigger the execution of rule $R1$. A rule could positively depend on itself with recursion that the newly derived facts will be taken as the input of the same rule. FORMULA has its special term semantics to decide if two non-ground terms in the rules are dependent on each other or not by checking if two non-ground terms are unifiable.

```

(R1) Edge(Node(1), n) :- Edge(n, n+1).
(R2) Edge(m, m) :- Edge(Node(2), m).

```

Listing 5.23: FORMULA Rules on Different Stratum

The two rules in 5.23 are usually dependent on each other in other logic programming languages because the input term and output term are of the same type `Edge`. However, in FORMULA each non-ground term has

its own unique type of semantic that can be represented as a unique set of terms. In the example above, the output of rule $R1$ is $Edge(Node(1),n)$ and the input of rule $R2$ is $Edge(Node(2),m)$, therefore the output of rule $R1$ does not contribute to the input of rule $R2$ because $Edge(Node(1),n)$ and $Edge(Node(2),m)$ are not unifiable. The same goes for the other way around that $Edge(n,n+1)$ and $Edge(m,m)$ are not unifiable so the output of rule $R2$ does not contribute to the input of rule $R1$. The conclusion is that rules $R1$ and rules $R2$ are independent of each other and puts into different strata after rule stratification.

```

PosDependency [( rule1 , rule2 )] :-
    // term1 from the body of one rule depends on term2 from
    // the head of another rule .
    ConstraintInRule [(index , PredCons {false , term1 , _} , rule1 )] ,
    HeadInRule [(term2 , rule2 )] ,
    rule1 != rule2 ,
    NonDisjointSetRel [(term1 , term2 )] .

// A single rule with recursion in it .
PosDependency [( rule , rule )] :-
    ConstraintInRule [(index , PredCons {false , term1 , _} , rule )] ,
    HeadInRule [(term2 , rule )] ,
    NonDisjointSetRel [(term1 , term2 )] .

PosDependency [(r1 , r3 )] :- PosDependency [(r1 , r2 )] , PosDependency [(r2 , r3 )] .

```

Listing 5.24: Stratification Validation Rules for Positive Dependency

The relation `PosDependency` tracks the dependency relationship between rules and forms a dependency graph for rule stratification. The first rule in 5.24 enumerates all pairs of two rules and checks if non-ground term t_1 as a predicate in the body of the rule $R1$ is unifiable with non-ground term t_2 as a predicate in the head of rule $R2$. If a match is found, a new fact is added into relation `PosDependency`, which means rule $R1$ depends on rule $R2$. The second rule in 5.24 is similar to the first rule but only applies to one single rule that has recursion or self-dependency in itself because recursion is allowed in the positive dependency graph for rule stratification. The last rule in 5.24 computes the transitive closure in a positive dependency graph for rules that are not directly dependent on each other.

A negative dependency occurs when a non-ground term t_1 in a negated predicate in rule $R1$ is unifiable with another non-ground term t_2 in the head of rule $R2$. Recursion is not allowed because the pattern of a negated predicate in the body of rule $R1$ requires that all derived facts of a certain type are saturated before

the evaluation of the absence of such pattern in another rule. Rules in later stratum are forbidden to derive new facts that fit into the same pattern of a negated predicate in the rule from a previous stratum.

```

NegDependency[(rule1, rule2)] :-
    // term1 from the body of one rule depends on term2 from the head of another rule.
    ConstraintInRule[(index, PredCons {true, term1, _}, rule1)],
    // term2 should not occur in the body of rule1 or rules that
    // directly or indirectly depend on the derived facts of rule1.
    HeadInRule[(term2, rule2)],
    rule1 != rule2,
    NonDisjointSetRel[(term1, term2)].

NegDependency[(rule, rule)] :-
    // term1 from the body of one rule depends on term2 from the head of another rule.
    ConstraintInRule[(index, PredCons {true, term1, _}, rule)],
    // term2 should not occur in the body of rule1 or rules that
    // directly or indirectly depend on the derived facts of rule1.
    HeadInRule[(term2, rule)],
    NonDisjointSetRel[(term1, term2)].

// Rule 'r2' has negation that depends on some derived facts in rule 'r3'.
NegDependency[(r1, r3)] :- PosDependency[(r1, r2)], NegDependency[(r2, r3)].
NegDependency[(r1, r3)] :- NegDependency[(r1, r2)], NegDependency[(r2, r3)].

// No cycle is allowed in the negative dependency graph.
ErrorNegDep[rule] :- NegDependency[(rule1, rule2)], rule1 == rule2, var rule = rule1.

```

Listing 5.25: Stratification Validation Rules with Negative Dependency

Two dependency graphs (both positive and negative) are succinctly expressed in the rules above to detect semantic conflicts in the recursion with negated predicates involved. Dependency cycles are allowed in the positive dependency graph because a recursive rule or mutually recursive rules may exist and do not violate the valid semantics of a FORMULA program. However, the negative dependency graph must be cycle-free otherwise a set of FORMULA rules cannot be stratified due to the semantic conflicts. The valid positive and negative dependency graph is followed by a topological sort that splits rules into multiple stratum and the execution of rules in each stratum is independent of other stratum but still follows the topological order.

5.6 *formula2ddlog* Model Transformation

Roughly 50 transformation rules are written in *formula2ddlog* (Zhang et al., 2021) transformation to transform models of FORMULA language domain into models of DDLLog language domain, which are later converted into an equivalent textual DDLLog program that can take constantly changing structured data as input and output the incremental updates at every point in the dataflow graph. We describe some selected key features of the FORMULA language and how they are formally translated into DDLLog via model transformation and validation.

5.6.1 Positive Predicate Translation

Positive predicates in FORMULA are translated directly into their counterparts in DDLLog except for the keywords *from_any()* and *to_any()* type are added to variables for conversion based on the type inference and the position of each variable.

5.6.2 Negated Predicate Translation

The negated predicate has different meanings in a different context. The negation in FORMULA can be either interpreted as set difference or set comprehension depending on if the variables in the negated expression occur in other parts of the same rule. Several DDLLog rules are written to infer the relationship between negated predicates and other expressions from the outer scope in the same rule.

```
// Check if any of the variables in the negated term
// occurs in the positive predicate term in the same rule
NegPredAsSetdiff[(neg_pred, r)] :- NegPredInRule[(neg_pred, r)],
                                   PosVarInRule[(variable, r)],
                                   NegVarInRule[(variable, r)],
                                   VarOfTerm[(variable, neg_pred)].

// Use set difference to find negated expressions that have set comprehension semantics
NegPredAsSetcompre[(neg_pred, r)] :- NegPredInRule[(neg_pred, r)],
                                   not NegPredAsSetdiff[(neg_pred, r)].
```

Listing 5.26: Rules for Reasoning over Negation

5.6.2.1 Negation as Set Difference

$F_1(\vec{t}_1), F_2(\vec{t}_2), \dots, F_n(\vec{t}_n), \text{no } F_{neg}(t_{neg})$ is a rule body with one negated predicate in it and $\vec{t}_s = \text{vars}(\vec{t}_1), \text{vars}(\vec{t}_2), \dots, \text{vars}(\vec{t}_n) - \text{vars}(t_{neg})$ is the shared variables between positive predicates and negated

predicate. The negation in Formula and DDLLog under this context means a set difference between two matched sets. The set difference is a left join (return rows in the left table if no matches are found in the right table). The first set contains the matches from the result of joins of all positive predicate constraints $F_n(\vec{t}_n)$ and the second set contains the matches from negative predicate $F_{neg}(\vec{t}_{neg})$.

For example, u is $Node(_)$, not $Path(u, u)$ means finding the set difference between nodes and self-cycle paths, which is the set of nodes that exist in relation $Node[u]$ but not as both arguments in the set of records of self-cycle $Path(u, u)$.

5.6.2.2 Negation as Set Comprehension

In Formula and DDLLog, when the variables of predicate $F(\vec{t})$ are not matched with any variables in the current scope, no $F(\vec{t})$ means the absence of records of a certain pattern in the term database and can be reduced to set comprehension as $count(x \mid x \text{ is } F(t_1, t_2, \dots, t_n)) = 0$ and the semantic translation of set comprehension will be explained in the next subsection.

For example, `noCycle :- no Path(u, u)` with only one negated predicate in Formula can be translated into `noCycle :- Path(u, u), var count = u.group_by(()).group_count(), count == 0.` in DDLLog. The rule means we group the matches from $Path(u, u)$ by an empty key, which means we put all matches into one group and check if the size of the group is zero. The set difference between them is the set of nodes that exist in $Node[u]$ but not as both arguments of any record in the set of $Path(u, u)$.

5.6.3 Set Comprehension Translation

Set comprehension is a unique feature in FORMULA as a logic programming language that does reasoning over any subset of terms matched in the constraints and even supports nested comprehension in which the constraints in set comprehension also contain set comprehension inside. Therefore, we have to distinguish between the inner scope and outer scope in the context of set comprehension that the inner scope is a separate environment that contains all the variables, constraints, and rules inside the set comprehension expression, while the outer scope is everything else outside the inner scope and has to wait for the completion of aggregation from set comprehension as part of its constraints before starting to evaluate the rest of constraints that do not have set comprehension in it. A formal representation of set comprehension is the following: $head : - CO_1, CO_2, CO_3, \dots, val = Op(H_1, H_2, \dots, H_n \mid CI_1, CI_2, \dots, CI_n), \dots, f(val)$, in which CO is a constraint in the outer scope while CI is a constraint in the inner scope.

If CI and CO do not share any variables at all, the evaluation of constraints CI in the set comprehension can be executed independently. In the opposite situation where the inner scope and outer scope share

some variables, the constraints in the outer scope have to be involved in the evaluation of set comprehension from the inner scope, thus the semantics of set comprehension could be drastically different based on the relationship between inner scope and outer scope.

5.6.3.1 Independent Set Comprehension

In the situation where the constraints in set comprehension do not have shared variables with any constraint from the outer scope, we call it independent set comprehension. $head : - C_{n+1}, C_{n+2}, C_{n+3}, \dots, val = Op(H_1, H_2, \dots, H_n \mid C_1, C_2, \dots, C_n), \dots, f(val)$ where C_n is a constraint in the body and term pattern H_n represents usually a non-ground term in the head that will be instantiated and put into the set of newly derived terms. Note that H_n could also be a ground term too. v_1, v_2, \dots, v_n is a list of variables in the constraints of set comprehension.

The semantic of independent set comprehension is very simple we first find all matches for the constraints in the body of set comprehension, derive new terms in the head part of the set comprehension and the final step is to put all newly derived terms into a single set of terms that do not have duplicates before aggregation. $head : - C_1, C_2, \dots, C_n, var\ g = (v_1, v_2, \dots, v_n).group_by(()), var\ r = g.my_aggregation_func()$ is the pseudo-DDLog imperative code to group all matches in the form of $[v_1, v_2, \dots, v_n]$ into one big group or set.

If we do aggregation directly on matches, $my_aggregation_func()$ needs to handle a lot of things. First, it has to derive new terms and remove duplicates. Second, it has to reduce a set of newly derived terms in the head into a single value based on the semantics of the aggregation operator Op .

In order to avoid generating a heavily customized aggregation function with manual optimization, we decompose set comprehension into multiple parts step by step to achieve the same semantics.

- In FORMULA we conceptually create a union type to include all types in the head of set comprehension as $U ::= T_1 + T_2 + \dots + T_n + CONST_1, CONST_2, \dots, CONST_n$ in which T_i is a non-ground type term and constant C_i is a ground term but we will not use FORMULA for the reasoning or rule execution.
- Create a new constructor in DDLog that wraps a tagged union type in DDLog as $T ::= (inner : U)$ and $U ::= H_1 \mid H_2 \mid \dots \mid H_n \mid CONST_1 \mid CONST_2 \mid \dots \mid CONST_n$. in FORMULA representation.
- $T(U :: H_1(\dots)), T(U :: H_2(\dots)), \dots, T(U :: H_n(\dots)), T(U :: CONST_1()), \dots, T(U :: CONST_n()) : - C_1, C_2, \dots, C_n$. is the pseudo-FORMULA code for the additionally generated rule that derives new terms from set comprehension into a new collection of data. The tagged union type can also be simplified as Any type in DDLog later.
- The next step is to generate a new rule in DDLog to derive new terms as the result of rule execution: $SCHeadUnion[T_1\{H_1\{\vec{v}_1\}\}], SCHeadUnion[T_n\{H_n\{\vec{v}_n\}\}] : - SCPred_1(\vec{v}_1), SCPred_n(\vec{v}_n)$., in

which *SCHeadUnion* is the name of a new collection in DDLLog to hold all terms derived from rule execution and h_n may contain the variables in predicate $SCPred_n(\vec{v}_n)$ as one of the constraints in the set comprehension.

- The new set comprehension is $r = Op(u | T(u))$ to reduce the set into a single value r and then translated into DDLLog expression as `var g = u.group_by()` and return a set of terms in a single group g .

The following rule in DDLLog is part of the model transformation that reads FORMULA language models and outputs the DDLLog language models that can be assembled into a real DDLLog rule. The first rule creates a new DDLLog rule to collect all terms derived in the head of set comprehension with a new tagged union type to represent the FORMULA union type, which is a union of all types of head terms. The second part of the transformation rule is to replace the set comprehension expression in the original rule with a new expression for the aggregated result.

```

DDRule[DDRule {
  [ sc_result_head_atom ],
  [ dd_head_union_atom_rhs , group_assignment , result_assignment ] }],
DDRhsInRule[( dd_head_union_atom_rhs_with_def , rule )] :-
  IndependentSetcompre [( setcompre , rule )],
  ConstraintInRule [( _, AssignCons{ def_term , SetcompreExpr{ setcompre_ref } }, rule )],
  setcompre == deref( setcompre_ref ),
  var dd_def_term = to_dd_term( def_term ),
  var rule_id = rule.id ,
  var sc_id = setcompre.rule.id ,
  var union_type_name = "SCHeadUnion" ++ "R" ++ rule_id ++ "SC" ++ sc_id ,
  var sc_result_name = "SCResult" ++ "R" ++ rule_id ++ "SC" ++ sc_id ,
  dd_sc_head_union_relation in DDRelation[DDRelation { _, union_type_name , _ }],
  dd_sc_result_relation in DDRelation[DDRelation { _, sc_result_name , _ }],
  // Add a new Rhs SCHeadUnion[t] to the new rule
  var dd_head_union_atom_rhs = ddterm_to_ddrhs(
    None ,
    ref_new( dd_sc_head_union_relation ),
    DDDVar{ "t" },
    false ),
  var dd_head_union_atom_rhs_with_def = ddterm_to_ddrhs(

```



```

one ,
ref_new( dd_sc_result_relation ),
DDCons{
  sc_result_name ++ "_usize" ,
  from_singleton_to_list(DDTermExpr{ref_new(DDVar{ to_string( dd_def_term) })})
},
false ),
var t_ref = ref_new(DDVar { "t" } ),
var t_expr_ref = ref_new(DDTermExpr { t_ref } ),
var t_vec = [t_expr_ref],
var group_assignment = DDGroup { "g", DDTupleExpr{t_vec}, DDTupleExpr{vec_empty()} },
var result_ref = ref_new(DDVarDecl { "result" } ),
var result_expr = DDTermExpr { result_ref },
var g_ref = ref_new(DDVar { "g" } ),
var g_expr_ref = ref_new(DDTermExpr { g_ref } ),

var aggregation_expr = DDDotFunctionCallExpr { g_expr_ref, "group_count", vec_empty() },
var result_assignment = DDRhsAssignment { result_expr, aggregation_expr },
var sc_result_head_term = DDCons {
  sc_result_name ++ "_usize" ,
  from_singleton_to_list(DDTermExpr{ref_new(DDVar{ "result" })})
},
var sc_result_head_atom = DDAtom {
  None ,
  ref_new( dd_sc_result_relation ),
  DDTermExpr { ref_new( sc_result_head_term ) }
}.

```

Listing 5.27: Rules for Reasoning over Independent Set Comprehension

5.6.3.2 Dependent Set Comprehension

Group matches by variables in the outer scope before the aggregation of each group indexed by outer scope variables. The constraints in the outer scope share variables with conditions in the set comprehension of the inner scope. $head : - CO_1, CO_2, \dots, CO_n, val = Op(H_1, H_2, \dots, H_n \mid C_1, C_2, \dots, C_n), \dots, f(val)$ where $\vec{v}_s = vars(\vec{C}) \cap vars(\vec{CO})$ is the set of shared variables from the inner and outer scope and the set must not be

empty. $\vec{v}_c = \text{vars}(\vec{C}) - \text{vars}(\vec{CO})$ and $\vec{v}_{co} = \text{vars}(\vec{CO}) - \text{vars}(\vec{C})$ are both set difference of two variable sets.

We combine constraints from both inner and outer scope into $\text{head} : - C_1, C_2, \dots, C_n, CO_1, CO_2, \dots, CO_n$, and after the rule execution we get a collection of matches in which each shared variable in $\vec{v}s$ is bonded to some concrete value. We group the matches by shared variables $\vec{v}s$ so for each key we put the matches with the same key (shared variables) into the same group.

- Create the union type for the head of the set comprehension in the same way as independent set comprehension above $U ::= T_1 + T_2 + \dots + T_n + \text{CONST}_1, \text{CONST}_2, \dots, \text{CONST}_n$.
- Create a new constructor to hold the new union type or a new relation in DDLLog together with outer scope variables as $\text{PreAggrContainer} ::= (\text{vco}_1 : T_1, \text{vco}_2 : T_2, \dots, \text{vcon} : T_n, \text{inner} : U)$ and

$$\text{PreAggrContainer}(\text{vco}, \dots, U :: H_1(\dots)),$$

$$\text{PreAggrContainer}(\text{vco}, \dots, U :: H_2(\dots)), \dots, \text{PreAggrContainer}(\text{vco}, \dots, U :: H_n(\dots)),$$

$$\text{PreAggrContainer}(\text{vco}, \dots, U :: \text{CONST}_1()), \dots, \text{PreAggrContainer}(\text{vco}, \dots, U :: \text{CONST}_n())$$

$$:- \text{CO}_1, \text{CO}_2, \dots, \text{CO}_n, C_1, C_2, \dots, C_n.$$

- Group by all outer scope variables \vec{v}_{co} in another rule

$$\text{PostAggrContainer}(\text{vco}_1, \text{vco}_2, \dots, \text{vco}_n, \text{val}) :- \text{PreAggrContainer}(\text{vco}_1, \text{vco}_2, \dots, \text{vco}_n, u),$$

$$u.\text{group_by}((\text{vco}_1, \text{vco}_2, \dots, \text{vco}_n)), \text{var } \text{val} = g.\text{some_aggr_func}().$$

For example, $\text{Outdegree}(\text{src}, \text{sum}) :- \text{src is Node}(_), \text{sum} = \text{count}(e \mid e \text{ is Edge}(\text{src}, \text{dst}))$. is translated into $\text{PreAggrContainer}(\text{src}, U :: E(e)) :- \text{src is Node}(_), e \text{ is Edge}(\text{src}, \text{dst})$. and

$$\text{PostAggrContainer}(\text{src}, \text{aggr_value}) :- \text{PreAggrContainer}(\text{src}, u), \text{var } g = u.\text{group_by}((\text{src})),$$

$$\text{var } \text{aggr_value} = g.\text{some_aggr_func}().$$

5.6.3.3 Nested Set Comprehension

A nested set comprehension is a unique feature in FORMULA language that constraint in both a rule and a set comprehension can have another set comprehension for the purpose of aggregation over a set of selected terms. The following example is a nested set comprehension in which we calculate the outdegree of a node only if it has a self-cycle.

$$\text{SelfCycleNodeOutdegree}(x, n) :- x \text{ is Node}, n = \text{count}(\{y \mid \text{Edge}(x, y),$$

$$\text{count}(\{x \mid \text{Edge}(x, x)\}) > 0$$

})

Listing 5.28: Nested Set Comprehension Examples

The nested set comprehension does not have to be handled separately because the transformation rules for both independent and dependent FORMULA set comprehension in the previous two subsections recursively flatten the nested set comprehension into multiple rules that do not have set comprehension inside.

5.6.4 *formula2ddlog* Transformation Example

We use a small example below to demonstrate how a FORMULA program is transformed into an equivalent DDLLog program with model transformation. The generated DDLLog program can be used to run model execution incrementally and even the model transformation itself is incremental meaning it has the capability to update the generated DDLLog program incrementally, even though the size of models of language domains are actually small compared with large models in CPS domain and the whole execution of model transformation only takes a fraction of millisecond.

```
domain Graph {
  Node ::= new(name: Integer).
  Edge ::= new(src: Node, dst: Node).
  Triangle ::= new(one: Node, two: Node, three: Node).

  Item ::= Node + Edge.
  Union ::= Triangle + Node.

  NestedUnion ::= Union + Item.
  Nested Union ::= Triangle + Node + Edge.
  BigEdge ::= new(src: NestedUnion, dst: NestedUnion).

  Edge(a, c) :- Edge(a, b), BigEdge(b, c).
}
```

Listing 5.29: Graph Domain in FORMULA

```
typedef Edge = Edge{src: Node, dst: Node}
typedef Node = Node{name: usize}
typedef Triangle = Triangle{two: Node, one: Node, three: Node}
typedef BigEdge = BigEdge{src: Any, dst: Any}
typedef BoolConstantR0N0 = BoolConstantR0N0{}
```

```

input relation BigEdgeInput [ BigEdge ]
input relation EdgeInput [ Edge ]
input relation NodeInput [ Node ]
input relation TriangleInput [ Triangle ]

output relation BigEdge [ BigEdge ]
output relation Edge [ Edge ]
output relation Item [ Any ]
output relation NestedUnion [ Any ]
output relation Node [ Node ]
output relation Triangle [ Triangle ]
output relation Union [ Any ]

Node[p] :- NodeInput[p].
BigEdge[p] :- BigEdgeInput[p].
BigEdge2[p] :- BigEdge2Input[p].
Item[to_any(p)] :- Edge[p].
Item[to_any(p)] :- Node[p].
NestedUnion[to_any(p)] :- Edge[p].
NestedUnion[to_any(p)] :- Node[p].
NestedUnion[to_any(p)] :- Triangle[p].
Triangle[p] :- TriangleInput[p].
Union[to_any(p)] :- Node[p].
Union[to_any(p)] :- Triangle[p].

Edge[Edge{a, from_any(c).unwrap_or_default()}] :- Edge[Edge{a, b}],
    BigEdge[BigEdge{to_any(b), c}].

```

Listing 5.30: Generated Graph Domain translated in DDL_{log}

We have done benchmarks over the graph domain that has some of the FORMULA advanced features such as recursion, negation, and set comprehension to test out the performance on a relatively computationally hard problem. The result shows that Differential-FORMULA uses much less time than the original FORMULA and theoretically it takes exponential time to run the specific problem we pick to compute the transitive closure in a graph for the benchmark.

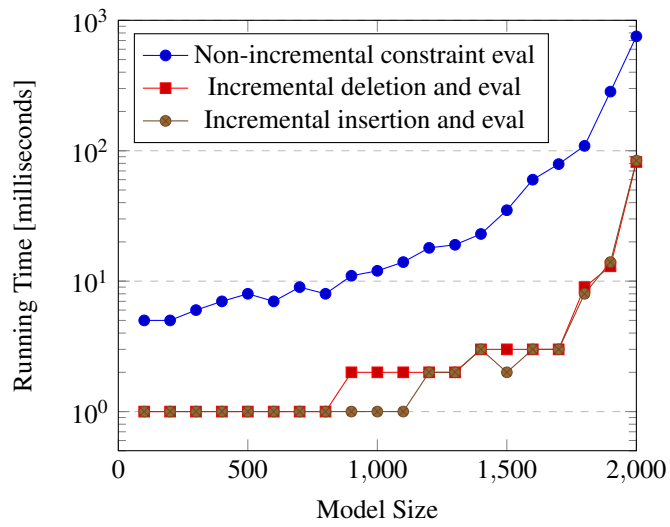


Figure 5.6: δF Incremental Running Time

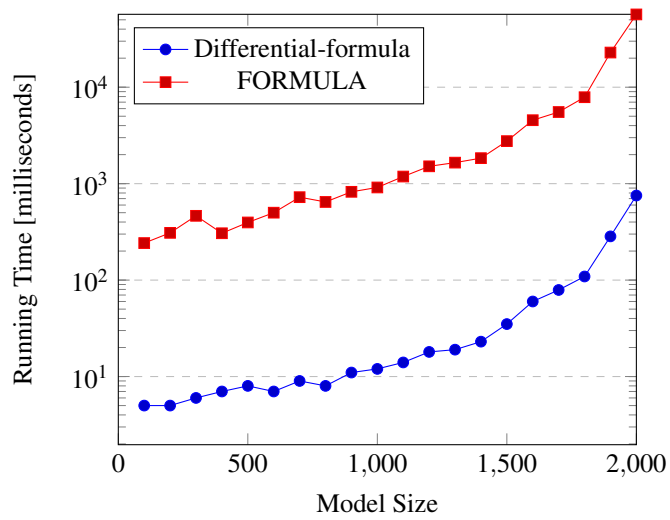


Figure 5.7: Non-incremental Running Time: δF vs FORMULA

5.7 Benchmarks on Incremental Updates for Large Models

Our Differential-FORMULA tool has to be tested out on its performance in handling large models and compare the results with other state-of-art modeling frameworks, especially the ones that support incremental updates. Model Transformation Tool Contest Benchmark is a good candidate for benchmark evaluation except for the Social Network domain from the TTC2018 does not include some major features in FORMULA language such as negation to check the absence of cycle in friendship graph or the nested set comprehension because the domain is fairly simple to model. On the other hand, FORMULA language may not natively support the features such as sorting or finding the top 3 scores in the social network benchmark but there are workarounds to express it in Differential-FORMULA by embedding some native differential-dataflow operators.

The competition can be found here: <https://github.com/TransformationToolContest/ttc2018liveContest> and this is so far the model transformation contest that includes the most modeling frameworks and participants. We may consider adding a Model Transformation contest from a different year into our benchmark but much fewer tools are available for comparison unless we want to implement the solutions for other tools.

However, benchmark evaluation only using the Model Transformation Contest social network benchmark gives us an estimate of the performance of a medium-size use case but not enough to evaluate the worst-case scenario, so we need to extend the graph domain benchmark in (Zhang et al., 2021) on constraint checking to separately test out negation, recursion, and set comprehension in order to find the bottlenecks, which leave clues to optimize the implementation.

The Metamodel of the social network domain is described in Ecore in Figure 5.8. The social network consists of users, posts, and comments. Users may have friends and the posts or comments have a score based on the interactions between users. The social network benchmark has two queries to be evaluated and both of them are computationally hard that can be reduced to problems similar to finding strongly connected components incrementally when the input changes.

In the Listing 5.31, we show the essential rules from a DDLLog program that is fine-tuned from the output of Differential-FORMULA to demonstrate the usage of logic programming rules to succinctly express the complex queries in a social network domain. Please note that part of the query in the social network domain is beyond what the FORMULA language can express. For example, the top three score needs an external function in DDLLog to handle

```
// Q1: The top 3 posts with the highest score based the number of comments
// and likes and must be sorted by timestamp.
CommentAncestor(commentid, parentid) :- Comments(commentid, -, -, -, parentid).
```

```

CommentAncestor(commentid, grandparentid) :- CommentAncestor(commentid, parentid),
    CommentAncestor(parentid, grandparentid).

// Find the root post id for each comment since the direct parent of a comment
// could be a comment.
CommentFromPost(commentid, ancestorid) :- CommentAncestor(commentid, ancestorid),
    Posts(ancestorid, -, -, -).

// Each comment gives its post 10 points.
PostCommentScore(pid, score) :- CommentFromPost(cid, pid),
    var score = (cid).group-by((pid)).group-count-distinct() * 10.

// Each like on the comment adds one point.
CommentLikeScore(cid, score) :- Likes(user, cid),
    var score = (user).group-by(cid).group-count-distinct().

// How many likes a post receives on all of its comments including the comments on comments.
PostLikeScore(pid, score) :- CommentFromPost(cid, pid),
    CommentLikeScore(cid, val), var score = (val).group-by((pid)).sum-of(|val| val).

// Combine two scores together based on the post id.
PostTotalScore(pid, score) :- PostCommentScore(pid, s1), PostLikeScore(pid, s2),
    var score = s1 + s2.

// For situation when a post has no likes, then just return the comment score.
PostTotalScore(pid, score) :- PostCommentScore(pid, score), not PostLikeScore(pid, _).

// Sort by score and then by timestamp in the order of its arguments.
PostTotalScoreTimestamp(score, timestamp, post) :- PostTotalScore(post, score),
    Posts(post, timestamp, -, -).

// Find the top 3 posts with highest scores first by score and then by timestamp.
// The most recent timestamp takes precedence if the scores are the same.
Top3PostScore(first.postId, second.postId, third.postId) :-
    pt in PostTotalScoreTimestamp(score, timestamp, post),
    var group = (pt).group-by(()),

```

```
(var first, var second, var third) = group.top_three()
```

Listing 5.31: Fine-tuned DDLLog Program for Q1

Most controversial posts - We consider a post controversial if it starts a debate through its comments. For this, we assign a score of 10 for each comment that belongs to a post. Hereby, we consider a comment belonging to a post, if it comments on either the post itself or another comment that already belongs to the post. In addition, we also value if users liked comments, so we additionally assign a score of 1 for each user that has liked a comment. The score of a post then is the sum of 10 plus the number of users that liked a comment and overall comments that belong to the given post. The goal of the query is to find the three posts with the highest score. Ties are broken by timestamps, i.e. more recent posts should take precedence over an older post.

Listing 5.32 shows the rules to concisely handle the complex query Q2 in the social network benchmark. Please note that we also replace the part that computes strongly connected components with manually optimized differential-dataflow implementation named *graph:SCC* with external functions to execute the rules more efficiently and specifically for computing the strongly connected components incrementally.

```
UserLikesNode(u, cid) :- Likes(u, cid).

// Add self-loop to every node in the graph.
UserLikesEdge(n, n) :- n in UserLikesNode(u, comment).
UserLikesEdge(n1, n2) :- n1 in UserLikesNode(u1, comment),
    n2 in UserLikesNode(u2, comment), Friend(u1, u2).

function convert_like1(ule: UserLikesEdge): UserLikesNode { ule.src }
function convert_like2(ule: UserLikesEdge): UserLikesNode { ule.dst }

// The result is a tuple of node Id and the lowest node Id in the group as an anchor.
output relation SCCLabel[(UserLikesNode, UserLikesNode)]
apply graph::SCC(UserLikesEdge, convert_like1, convert_like2) -> (SCCLabel)

// Each component is represented by an anchor followed by the size of component.
SCC(anchor, size) :- SCCLabel[(likes, likes_lowest)],
    var group = (likes).group_by(likes_lowest),
```



```

var anchor = group.group_key(),
var size = group.group_count_distinct().

SCCScore(sum, comment_key) :- SCC(anchor, size),
var comment = anchor.comment,
var user = anchor.user,
var group = (user, size).group_by(comment),
var comment_key = group.group_key(),
var sum = group.sum_of(|tuple| tuple.1 * tuple.1).

// For comments that do not have likes, we give it a score of 0.
SCCScore(0, cid) :- Comments(cid, -, -, -, -), not Likes(_, cid).

SCCScoreTimestamp(score, timestamp, comment) :- SCCScore(score, comment),
Comments(comment, timestamp, -, -, -).

Top3CommentScore(first.comment, second.comment, third.comment) :-
st in SCCScoreTimestamp(score, timestamp, comment),
var group = (st).group_by(()),
(var first, var second, var third) = group.top_three().

```

Listing 5.32: Fine-tuned DDLLog Program for Q2

Most influential comment - In this query, we aim to find comments that are commented on by groups of users. We identify groups through friendship relations. Hereby, users that liked a specific comment form a graph where two users are connected if they are friends (but still, only users who have liked the comment are considered). The goal of the second query is to find strongly connected components in that graph. We assign a score to each comment which is the sum of the squared component sizes. Similar to the previous query, we aim to find the three comments with the highest score. Ties are broken by timestamps.

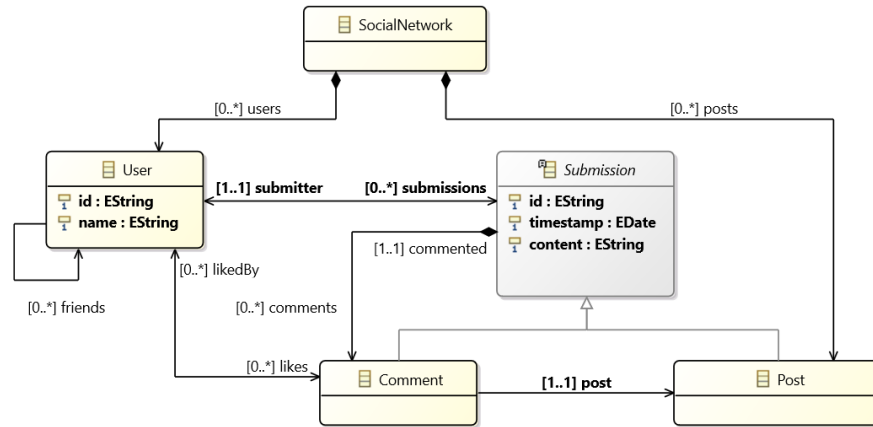


Figure 5.8: TTC2018 Social Network Metamodel in Ecore

We provide several models and change sequences of different sizes. In each change sequence, new posts are added, new comments to existing posts are added, new users are added, and users become friends or users like comments. The changes are always incremental, i.e. friendship relations do not break. Posts, comments, or likes are also not withdrawn.

The benchmark consists of the following phases:

1. Initialization: In this phase, solutions may load metamodels and another infrastructure independent of the used models as required. Because time measurements are very hard to measure for this phase, time measurement is optional.
2. Loading: The initial model instances are loaded.
3. Initial: A initial view is created after the first round of execution of models.
4. Updates: A sequence of change sequences is applied to the model. Each change sequence consists of several atomic change operations. After each change sequence, the view must be consistent with the changed source models, either by entirely creating the view from scratch or by propagating changes to the view result

For this benchmark, we have two solutions for differential-FORMULA that one is a DDLog program generated after model transformation and fine-tuned to take advantage of the native differential-dataflow implementation of strongly connected components dataflow, while the other one is the direct implementation in Differential-dataflow by the creator of Differential-dataflow. As we can see from the benchmarks, the native implementation of Differential-dataflow dominates in every aspect while our Differential-FORMULA solution is not but still the top three. One of the major reasons is that the data structure passing through

the dataflow in DDLog has more overheads and rules execution in a logic programming language is not as efficient as a manually written low-level implementation in Rust with Differential-Dataflow library.

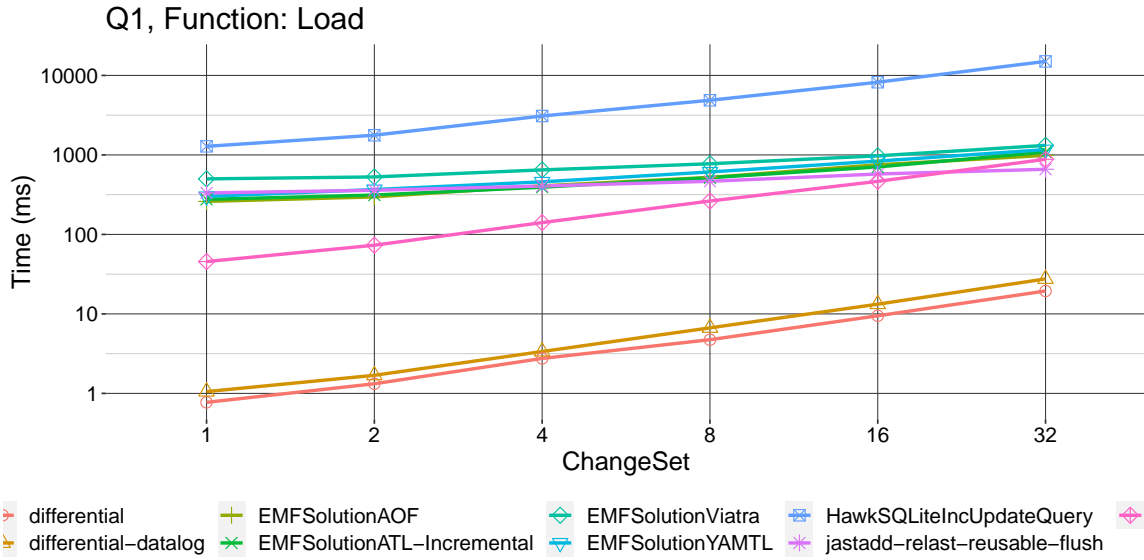


Figure 5.9: Q1 Load Time

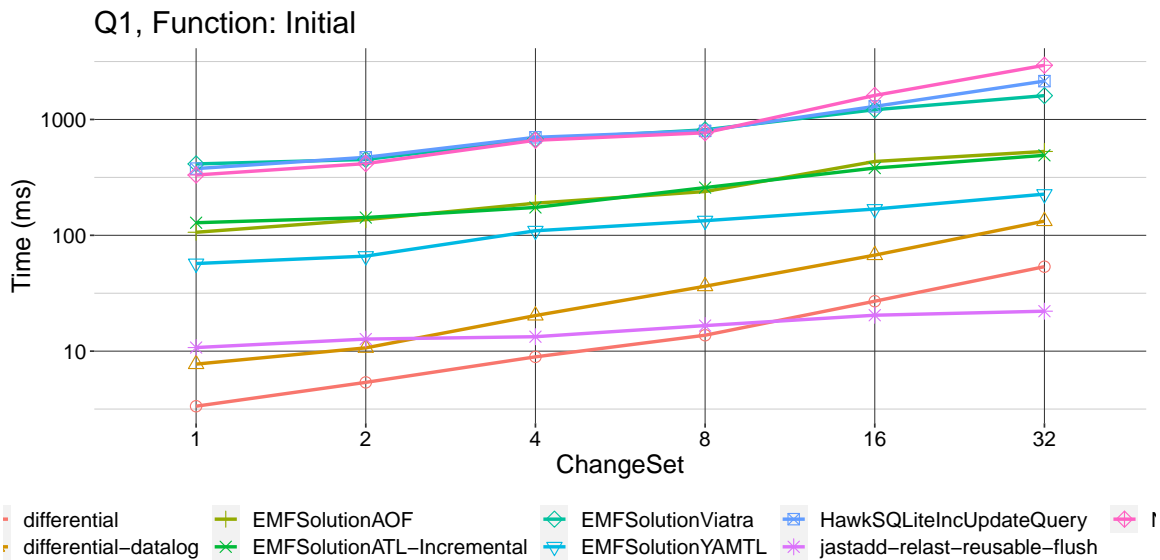


Figure 5.10: Q1 Batch Time

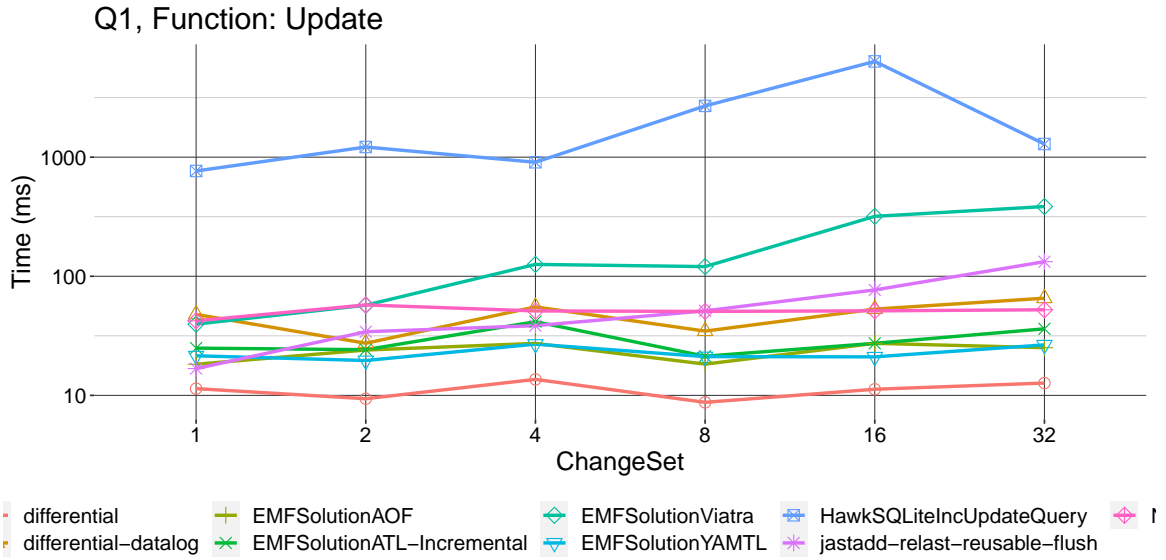


Figure 5.11: Q1 Updates Time

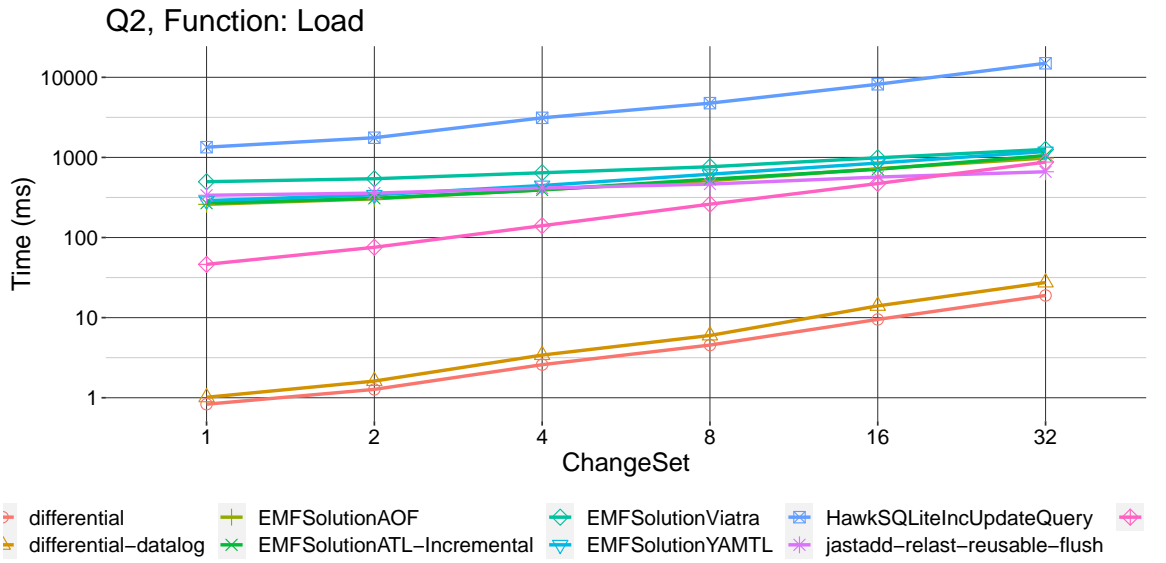


Figure 5.12: Q2 Load Time

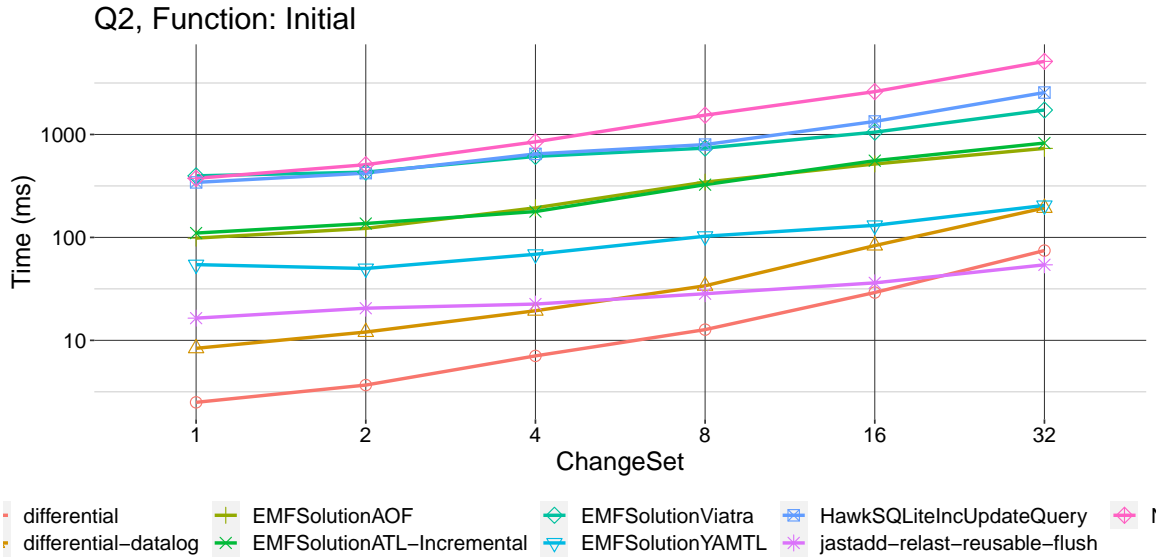


Figure 5.13: Q2 Batch Time

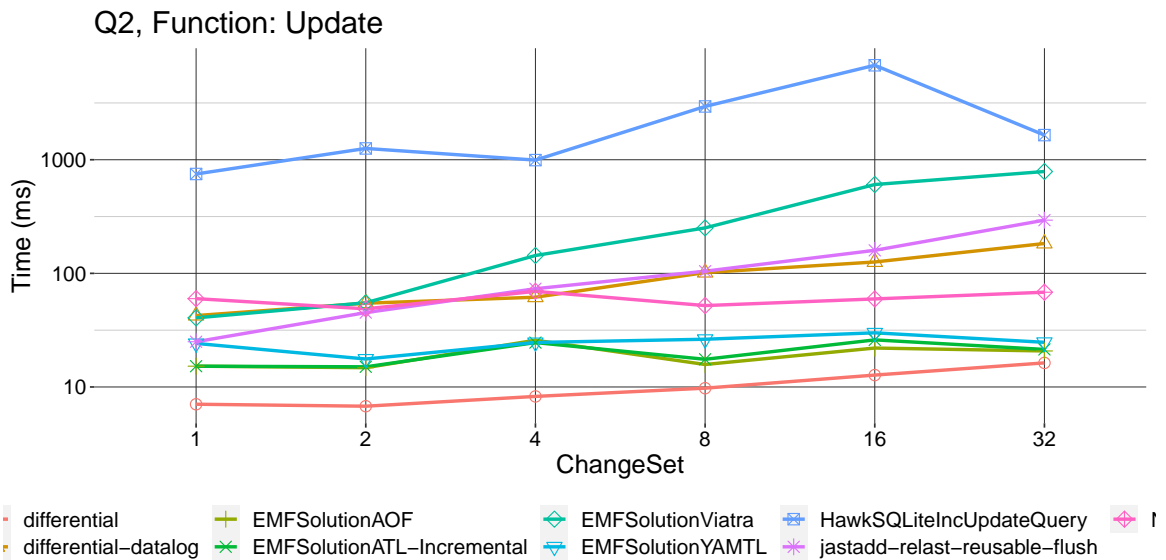


Figure 5.14: Q2 Updates Time

5.8 Contributions

This chapter has the most significant work and contributions so far to solve the performance and scalability issues regarding large models and computationally intensive model execution. Unlike the graph-based solution that only works for a subset of the FORMULA language, it still preserves the whole set of rich semantics and the unique language features of the original FORMULA language.

The innovation in this work is that we adopt the exact same Model-driven engineering methodology we

promote over the years to develop not only faster but also an incremental Model-driven engineering tool. The results from several benchmarks have proved that it has one of the leading performances compared with other state-of-art modeling frameworks on executing very large models and can now solve the problems that the original FORMULA will fail to even load the intermediate results into the memory without overflow.

So far as we know, our tool is the first modeling framework that is not manually implemented by software engineers or researchers using general-purpose programming language but completely specified and transformed into another program using the specification language itself in a bootstrapping way.

Our main contributions related to the concrete implementation of *formula2ddlog* transformation are the following:

1. We have the complete language definition of both DDL_{og} and FORMULA specified in DDL_{og} language as language models. Even though both languages belong to the logic programming language family, the semantics of some same concepts such as negation could be drastically different and the meaning changes under different contexts. The semantic difference has been identified either directly from the specification or from the result of rule execution.
2. One of the major breakthrough in overcoming semantic differences is that static analysis of the FORMULA program has been enabled for the first time to reason over the models of the FORMULA program. Type inference is crucial to the later type conversion in the DDL_{og} rule to emulate the execution of the FORMULA language.
3. Model transformation from models of FORMULA to models of DDL_{og} using a hand-crafted translator that is implemented as hundreds of rules in DDL_{og} to elegantly handle some complex language features such as nested set comprehension and dependent/independent set comprehension,

CHAPTER 6

Future works and Open Challenges

6.1 Future Works

The future works include the complete implementation of Differential-FORMULA and applying our integrated tool to some large industrial use cases with benchmark comparison with other leading modeling frameworks. Investigation of the potential optimization is encouraged to maximize the performance in Differential-Dataflow.

6.1.1 Implementation and Optimization

The full implementation is not finished yet especially for the model and domain inheritance and more optimization is needed. The current version of Differential-FORMULA only supports constraint checking but the full model transformation that generates new facts and converts them back into FORMULA terms. The original FORMULA also supports the inheritance of domains (metamodels) and models in order to concisely extend the reusable modules but this feature is not supported in the current version of Differential-FORMULA.

6.2 Open Challenges

There are mainly two open opportunities to extend the Differential-FORMULA and our integrated modeling framework.

6.2.1 Model Synthesis

Model finding or model synthesis in model-driven engineering finds feasible solutions to complete partial models that satisfy all the constraints and it notoriously does not scale well for even small models depending on the problems. Overall it is an undecidable problem and needs heuristics to guide the unrolling of rules wisely to speed up the search space exploration.

There are many tools that support model-finding features such as Alloy (Jackson, 2019) and Viatra Solver (Varró and Balogh, 2007) but the semantics they can express is more limited than FORMULA language. For example, the Viatra solver supports recursion but is limited to only transitive closure in the graph. The model synthesis in FORMULA is not complete and this task is nontrivial. FORMULA has richer semantics than other modeling tools and the mapping from partial models and symbolic states with its unique semantics to Z3 is complicated. e.g. Recursion, nested aggregation, and embedding of union types.

Some SMT solvers support incrementality, but a deeper integration with differential-FORMULA is needed

for the iterative mapping and execution from FORMULA to extend the capability of incremental execution in existing SMT solvers for open-world logic programs.

6.2.2 Formal Specification with Bootstrapping

FORMULA can be considered as the guard for the specification and execution of DSMLs but the FORMULA language itself (the implementation of its own compiler and the translator to other languages like DDlog) is not guarded by any guard.

The current rewriting rules that transform a FORMULA program into a DDlog program are specified in DDlog. For future work, we are planning to implement this as a FORMULA transformation, with both FORMULA and DDlog specified as domains in the FORMULA language itself. This approach will give confidence in the translation between the two languages through the use of a formal specification.

References

- Abadi, M., McSherry, F., and Plotkin, G. D. (2015). Foundations of differential dataflow. Technical report.
- Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press.
- Anjorin, A., Lauder, M., Patzina, S., and Schürr, A. (2011). Emoflon: leveraging emf and professional case tools. In *GI-Jahrestagung*, page 281.
- Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.
- Boronat, A. (2018). Yamtl solution to the ttc 2018 social media case.
- Boronat, A. (2022). Incremental execution of relational transformation specifications in yamtl: a case with laboratory workflows.
- Bucchiarone, A., Cicchetti, A., Ciccozzi, F., and Pierantonio, A. (2021). Domain-specific languages in practice: With jetbrains mps.
- Cabot, J. and Gogolla, M. (2012). Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems*, pages 58–90. Springer.
- Cabot, J. and Teniente, E. (2009). Incremental integrity checking of uml/ocl conceptual schemas. *Journal of Systems and Software*, 82(9):1459–1478.
- Campagne, F. (2014). *The MPS language workbench: volume I*, volume 1. Fabien Campagne.
- Daniel, G., Sunyé, G., and Cabot, J. (2016a). Mogwai: a framework to handle complex queries on large models. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE.
- Daniel, G., Sunyé, G., and Cabot, J. (2016b). Umltographdb: Mapping conceptual schemas to graph databases. In *International Conference on Conceptual Modeling*, pages 430–444. Springer.
- De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.
- Desai, A., Gupta, V., Jackson, E. K., Qadeer, S., Rajamani, S. K., and Zufferey, D. (2013). P: safe asynchronous event-driven programming. In Boehm, H. and Flanagan, C., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM.
- Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006). Fundamentals of algebraic graph transformation. EATCS Series.
- Falleri, J.-R., Huchard, M., and Nebut, C. (2006). Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40. Sintef ICT, Norway.
- Giese, H. and Wagner, R. (2009). From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43.
- Gupta, A., Mumick, I. S., and Subrahmanian, V. S. (1993). Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166.
- Hinkel, G. (2016). *NMF: A Modeling Framework for the NET Platform*. KIT.
- Jackson, D. (2019). Alloy: a language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76.

- Jackson, E. K. (2013). Engineering domain-specific languages with formula 2.0. In Boleng, J. and Taft, S. T., editors, *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, HILT 2013, Pittsburgh, Pennsylvania, USA, November 10-14, 2013*, pages 3–4. ACM.
- Jackson, E. K., Bjørner, N., and Schulte, W. (2011a). Canonical regular types. In Gallagher, J. P. and Gelfond, M., editors, *Technical Communications of the 27th International Conference on Logic Programming, ICLP 2011, July 6-10, 2011, Lexington, Kentucky, USA*, volume 11 of *LIPICs*, pages 73–83. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- Jackson, E. K., Bjorner, N., and Schulte, W. (2013). Open-world logic programs: A new foundation for formal specifications. Technical report, Microsoft technical report MSR-TR-2013-55. See <http://research.microsoft.com>
- Jackson, E. K., Kang, E., Dahlweid, M., Seifert, D., and Santen, T. (2010). Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 39–48.
- Jackson, E. K., Levendovszky, T., and Balasubramanian, D. (2011b). Reasoning about metamodeling with formal specifications and automatic proofs. In *International Conference on Model Driven Engineering Languages and Systems*, pages 653–667. Springer.
- Jackson, E. K., Levendovszky, T., and Balasubramanian, D. (2015). Automatically reasoning about metamodeling. *Softw. Syst. Model.*, 14(1):271–285.
- Jackson, E. K. and Schulte, W. (2013). FORMULA 2.0: A language for formal specifications. Technical report.
- Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39.
- Jouault, F. and Bézivin, J. (2006). Km3: a dsl for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer.
- Kecskés, T., Zhang, Q., and Sztipanovits, J. (2017). Bridging engineering and formal modeling: Webgme and formula integration. In *MODELS (Satellite Events)*, pages 280–285.
- Kelly, S., Lyytinen, K., and Rossi, M. (1996). Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *International Conference on Advanced Information Systems Engineering*, pages 1–21. Springer.
- Kern, H., Hummel, A., and Kühne, S. (2011). 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.
- Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.
- Kleppe, A. and Rensink, A. (2008). On a graph-based semantics for uml class and object diagrams. *Electronic Communications of the EASST*, 10:1–16.
- Kosar, T., Mernik, M., and Lopez, P. E. M. (2007). Experiences on dsl tools for visual studio. In *2007 29th International Conference on Information Technology Interfaces*, pages 753–758. IEEE.
- Kumar, P. S., Emfinger, W., Karsai, G., Watkins, D., Gasser, B., and Anilkumar, A. (2016). Rosmod: a tool-suite for modeling, generating, deploying, and managing distributed real-time component-based software using ros. *Electronics*, 5(3):53.
- Lawley, M. and Steel, J. (2005). Practical declarative model transformation with tefkat. In *International Conference on Model Driven Engineering Languages and Systems*, pages 139–150. Springer.

- Leblebici, E., Anjorin, A., and Schürr, A. (2014). Developing emoflon with emoflon. In *Theory and Practice of Model Transformations: 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings 7*, pages 138–145. Springer.
- Marcos, D. D. F., Jean, B., Frédéric, J., Erwan, B., and Guillaume, G. (2005). Amw: A generic model weaver. *Proc. of the 1eres Journées sur l'Ingénierie Dirigée par les Modeles*, 200.
- Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., and Lédeczi, Á. (2014). Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure. In *Proceedings of the 8th Workshop on Multi-Paradigm Modeling*, volume 1237 of *CEUR Workshop Proceedings*, pages 41–60. CEUR-WS.org.
- Martínez, S., Tisi, M., and Douence, R. (2017). Reactive model transformation with atl. *Science of Computer Programming*, 136:1–16.
- Mavridou, A., Kecskés, T., Zhang, Q., and Sztipanovits, J. (2018). A common integrated framework for heterogeneous modeling services. In *MODELS Workshops*, pages 416–422.
- Merz, S. (2008). The specification language tla+. *Logics of specification languages*, pages 401–451.
- Mezei, G., Theisz, Z., Urbán, D., Bácsi, S., Somogyi, F. A., and Palatinszky, D. (2019). A bootstrap for self-describing, self-validating multi-layer metamodeling. In *Proceedings of the Automation and Applied Computer Science Workshop*, pages 28–38.
- Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455.
- Pech, V. (2021). JetBrains mps: Why modern language workbenches matter. In *Domain-Specific Languages in Practice: with JetBrains MPS*, pages 1–22. Springer.
- Rodriguez, M. A. (2015). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM.
- Roşu, G. and Şerbănuţă, T. F. (2010). An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434.
- Ryzhyk, L. and Budiu, M. (2019). Differential datalog. *Datalog*, 2:4–5.
- Scheer, A.-W. and Schneider, K. (1998). Aris—architecture of integrated information systems. In *Handbook on architectures of information systems*, pages 605–623. Springer.
- Schürr, A. and Klar, F. (2008). 15 years of triple graph grammars. In *Icgt*, pages 411–425. Springer.
- Simko, G., Levendovszky, T., Neema, S., Jackson, E. K., bapty, T., Joe, P., and Sztipanovits, J. (2012). Foundation for model integration: Semantic backplane. In *Proceedings of the ASME 2012 IDETC/CIE 2012*, pages 1–8. ASME.
- Spivey, J. M. and Abrial, J. (1992). *The Z notation*, volume 29. Prentice Hall Hemel Hempstead.
- Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.
- Szabó, T., Bergmann, G., Erdweg, S., and Voelter, M. (2018). Incrementalizing lattice-based program analyses in datalog. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29.
- Szabó, T., Erdweg, S., and Voelter, M. (2016). Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331.

- Sztipanovits, J., Bapty, T., Neema, S., Howard, L., and Jackson, E. K. (2014). Openmeta: A model- and component-based design tool chain for cyber-physical systems. In Bensalem, S., Lakhnech, Y., and Legay, A., editors, *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop*, volume 8415 of *Lecture Notes in Computer Science*, pages 235–248. Springer.
- Sztipanovits, J. and Karsai, G. (1997). Model-integrated computing. *Computer*, 30(4):110–111.
- TinkerPop, A. (2020). The gremlin graph traversal machine and language.
- Ujhelyi, Z., Bergmann, G., Hegedüs, Á., Horváth, Á., Izsó, B., Ráth, I., Szatmári, Z., and Varró, D. (2015). Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99.
- Varró, D. and Balogh, A. (2007). The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234.
- Voelter, M., Ratiu, D., Schaetz, B., and Kolb, B. (2012). mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140.
- Weidmann, N. and Anjorin, A. (2021). emoflon:: Neo-consistency and model management with graph databases. In *STAF Workshops*, pages 54–64.
- Zhang, Q., Balasubramanian, D., Kecskes, T., and Sztipanovits, J. (2021). Differential-formula: towards a semantic backplane for incremental modeling. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling*, pages 51–60.
- Zhang, Q., Kecskes, T., Mathe, J., and Sztipanovits, J. (2019). Towards bridging the gap between model- and data-driven tool suites for cyber-physical systems. In *2019 IEEE/ACM 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, pages 7–13. IEEE.
- Zündorf, A., Schürr, A., and Winter, A. J. (1999). *Story driven modeling*. Univ.-Gesamthochsch. Paderborn, Fachbereich Mathematik-Informatik.