Differential-FORMULA: Towards a Semantic Backplane for Incremental Modeling

Qishen Zhang, Daniel Balasubramanian, Tamas Kecskes, Janos Sztipanovits
Vanderbilt University
Motivation

1. Evolving model engineering environment, which requires formal semantics for the model integration language.

2. For real world model-based engineering problems, three challenges need to be solved:
   (a) tight integration between engineering environment and formal semantics
   (b) scalability
   (c) incrementality.
Background: WebGME

1. Meta-programmable modeling environment (first released in 2015)
2. Web-based, client server architecture
3. Support version control and collaborative model editing

Web-based Generic Modeling Environment
https://www.webgme.org
Background: FORMULA

- Formal framework using algebraic data types and Constraint Logic Programming (CLP).
- Formal representation of metamodels, models and model transformations.
- Has execution semantics for reasoning and executing model transformation.
- Interfaced to state-of-the-art SMT solver Z3 for model synthesis.

Example 4.1 (Classes of graphs).

```
1: domain Digraphs
2: {
3: V ::= new (lbl: Integer).
4: E ::= new (src: V, dst: V).
5: }
6: domain DAGs extends Digraphs
7: {
8: path ::= (V, V).
9: path(u, w) :- E(u, w); E(u, v), path(v, w).
10: conforms no path(u, u).
11: }
12: domain Trees extends DAGs
13: {
14: conforms no { w | E(u, w), E(v, w), u != v }.
15: }
```

Try and execute FORMULA program in online editor https://rise4fun.com/Formula2
1. Differential Datalog (DDlog) is a programming language designed for incremental computations, where it is expected that the program input is continually changing.

2. 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.

3. Maintain an index of updates so the next state can be efficiently derived by combining partially ordered versions.

4. The output is a DD graph of dataflows in which each node is a DD operator connected to other operators with a highly optimized implementation for incremental computation.

Data-parallel operators can operate on differences:

\[
\text{Difference} : \{ ( \text{record, delta, version} ) \}
\]

**Important**: A version can be more than just an integer.

\[
\text{Reach}(s') \leftarrow \text{Reach}(s), \text{Trans}(s, \_ , s').
\]
Integrating WebGME, FORMULA-2 and DDLog

- **Tight integration is solved by:**
  - WebGME metamodels and models are translated into FORMULA domain/model specifications.
  - Instead of JavaScript or Python plugins, constraints are specified and evaluated in FORMULA-2.
  - Model transformation are specified and executed in FORMULA-2.

- **Scalability and incrementality is solved by:**
  - Translating FORMULA-2 into DDLog and executing FORMULA-2 programs over differential-dataflow.

- **Restriction:** the solution does not include the symbolic execution of FORMULA-2 programs by Z3 engine. (The *partial model* constructs of FORMULA-2 are not supported.)
Translating FORMULA-2 into DDLog

Metamodel of type definition of FORMULA language and DDLog language modeled in both textual and graphical syntax.

Main semantic difference:
- **DDLog**: The types are tagged union of at least one constructor. Each constructor contains multiple other types as its subtypes.
- **FORMULA**: The constructor is similar with DDLog constructor but the union type in FORMULA could be any combination of sub-types.

The more expressive FORMULA types are translated into datatypes of general-purpose logic programming language DDLog.

The complete metamodels of FORMULA and Differential-Datalog languages can be found in [tinyurl.com/formula2ddlog](http://tinyurl.com/formula2ddlog)
Metamodels, models and transformation (including all type definitions and rules) in a FORMULA program is modeled in FORMULA language domain.

After model transformation, we get models of DDLog language domain.

Code generation from DDLog language model to create an equivalent executable DDLog program.
Selected issues: Model Validation on Nested Set Comprehension

According to the specification of FORMULA language, only two adjacent set comprehension scopes can have shared variables. An example of nested set comprehension in which the condition of outer set comprehension to compute degrees contains another set comprehension to compute the number of nodes.

\[
\text{TotalDegree}(amt) := \text{amt} = \text{count}(\{ \text{src} \mid e \text{ is Edge}(\text{src}, \text{dst}) \}, \text{node amt} = \text{count}(\{n \mid n \text{ is Node}()\}), \text{node amt} > 1)
\]

If the inner set comprehension has another set comprehension inside it as part of the condition, that set comprehension cannot have variable `src`.

Reasoning over the models of a FORMULA program to detect invalid set comprehensions.

```prolog
// 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].

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

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

Error(r1, r2, "Conflict in nested set comprehension") :-
RuleContent[r1, r2],
not Constraint[SetcompreExpr(r1), r2],
VarInRule[variable, r1],
VarInRule[variable, r2].
```
Example of invalid rules:

Path(a, c) :- Path(a, b), Path(b, c).
NoCycle(u) :- no Path(u, u).
Path(u, u+1) :- NoCycle(u).

The FORMULA rules above cannot be stratified with valid semantics because of the cycling dependencies.
A Simple Domain for Benchmarks

A simple Graph domain modeled in FORMULA language in which the constraints are expressed in FORMULA rules with (a) negation, (b) recursion and (c) set comprehension.

```
domain Graph {
    Node ::= new(name:integer+String).
    Edge ::= new(src:Node, dst:Node).
    Path ::= new(src:Node, dst:Node).

    Nocycle ::= new(node:Node).
    OutdegreeByNode ::= new(node:Node, degrees:integer).
    Outdegree ::= new(degrees:integer).

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

    Nocycle(u) :- u is Node(_,), no Path(u, u).
    HasNoCycle :- no Path(u, u).

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

A recursive rule to find all paths in the graph

Use negation to prove the Absence of cycles

Set comprehension applied to compute the degrees of Each node
The new incremental differential-FORMULA achieves better performance compared with direct model execution on FORMULA-2.
Ongoing Work

- We significantly improved the scalability and enable incremental modeling for our integrated modeling framework.

- The future steps include:
  - Finishing up the translation of FORMULA model transformation to DDLog engine.
  - Change the specification of metamodels and transformation rules from DDLog to differential-FORMULA itself in a bootstrapping way.
  - Complete benchmarks of more domains and compare with other frameworks.
Questions?