Engineering Domain-Specific Languages with FORMULA 2.0

Ethan K. Jackson
Microsoft Research
One Microsoft Way
Redmond, WA 98052
ejackson@microsoft.com

1. INTRODUCTION

Domain-specific languages (DSLs) are useful for capturing and reusing engineering expertise. They can formalize industrial patterns and practices while increasing the scalability of verification, because input programs are written at a higher level of abstraction. However, engineering new DSLs with custom verification is a non-trivial task in its own right, and usually requires programming language, formal methods, and automated theorem proving expertise.

In this paper we present FORMULA 2.0, which is a formal framework for developing DSLs. FORMULA specifications are succinct descriptions of DSLs, and specifications can be immediately connected to state-of-the-art analysis engines without additional expertise. FORMULA provides: (1) succinct specifications of DSLs and compilers, (2) efficient compilation and execution of input programs, (3) program synthesis and compiler verification.

We take a unique approach to provide these features: Specifications are written as *strongly-typed* [4, 7] *open-world logic programs* [5], and a specialized module system supports modularity and reuse [3]. These specifications are highly declarative and can easily express rich *program synthesis* and *compiler verification* problems. Automated reasoning is enabled by efficient symbolic execution of logic programs into quantifier-free sub-problems, which are dispatched to the state-of-the-art SMT solver Z3 [1]. FORMULA has been applied within Microsoft to develop DSLs for verifiable device drivers and protocols [2]. It has been used by the automotive and embedded systems industries for software and hardware co-design [6] and design-space exploration [8] under hard resource allocation constraints. It is being used to develop semantic specifications for complex cyber-physical systems [9].

We now highlight of the features, design principles, and implementation approaches of FORMULA 2.0. It is released under an open-source license and can be found at `http://formula.codeplex.com`.

2. DOMAINS

The static semantics of DSLs are specified using *algebraic data types* (ADTs) and *open-world logic programs* (OLPs). ADTs are used to formalize DSL syntax and the shapes of judgments (e.g. type judgments). OLPs provide a semantic foundation for separating DSL axioms (e.g. the definitions of all well-typed expressions) from a specific program of the DSL. We shall refer to a specific program of a DSL as a *program instance*. We demonstrate these ideas with a small example.

Example 1 (Deployment DSL).

```
 1: domain Deployments
 2: {
 3:   Service  ::= new (name: String).
 4:   Node     ::= new (id: Natural).
 5:   Conflict ::= new (s1: Service, s2: Service).
 6:   Deploy   ::= fun (s: Service => n: Node).
 7:
 8:   conforms no { n | Deploy(s, n), Deploy(s', n),
 9:                     Conflict(s, s') }.
10: }
```

Domain modules contain ADTs for language syntax and judgments along with axioms written as logic programs. The *Deployments* domain formalizes a small DSL for mapping software services onto compute nodes: There are services, which can be in conflict, and nodes, which can run services. Services must be deployed to nodes such that no node executes conflicting services. Lines 3 - 6 introduce data types for the entities of the DSL. The *conformance rule* (lines 8 - 9) axiomatizes that conflicting tasks cannot run on the same node.

Notice that a domain does not contain any information about a specific program instance, but speaks generally about all possible deployments. Hence, the logic program within a domain is not a *closed-world* program; the sets of services, nodes, conflicts, and deployments have not been determined. A program instance closes the world of a domain by enumerating a set of ground facts, after which the composite has the usual closed-world semantics. Also, notice that constructing a conforming program instance for the *Deployments* domain is NP-complete. It is equivalent to coloring the conflict graph with nodes. In other words, writing statically correct programs is hard under many abstractions.

2.1 Design and Implementation

Algebraic Data Types. Every DSL must have syntax for programs and judgments. In practice, much energy is spent designing just these syntactic elements of a language. FORMULA supports this by first-class ADTs and is unapologetically strongly typed. For uniformity, *every user defined symbol is a data constructor* and there is only one implicit program relation K (*knows*). If the user wishes to define another program relation, she can represent it using the theory of ADTs plus constraints on K.

For example, the rule:

```
IsBigNode(id) :- Node(id), id > 100.
```

Should really be understood as:

```
K(IsBigNode(id)) :- K(Node(id)), id > 100.
```

IsBigNode() looks like a unary relation on integers, but it is really a data constructor. The previous rule axiomatizes those *IsBigNode* terms that can appear in K. The only technicality is that rule dependencies and stratification conditions are defined w.r.t. the precise shapes of values placed into K by each rule. FORMULA performs complex abstract interpretation on rules to determine the shapes of terms that can appear in the head of a rule.

Bottom-up fixpoints. It became clear that the logical semantics and operational semantics needed to coincide and be as intuitive as possible. This allows DSL specifications to be comprehensible and robust to change. Consequently, FORMULA uses bottom-up evaluation so that rule bodies and rules are not as sensitive to ordering. We require *stratified aggregates*, which simplifies the fixpoint semantics and is rather natural for our scenarios. After all, a given static analysis or compiler should produce a single result per input program. This places more burden on our LP engine to perform cross-rule optimizations and efficient indexing, but it allows engineers to think of their specifications as logical entities that happen to be executable.

Aggregates. We found aggregates to be indispensable. Engineers use them in complex nested patterns. To support this, we developed a syntax that mimics quantifiers with lexical scope. Our aggregation primitive is:

```
{ head | body }
```

This represents the set of all terms that would be produced by the rule *head* :- *body*. Such aggregates can only be used as arguments to special operators, such as *count* or *no*. Negation (*no*) is equivalent to $count(\{head|body\}) = 0$. Variables introduced within an aggregation are local to that aggregation and variables introduced in a parent lexical scope are bound w.r.t. to child aggregations.

For example, we wish the subset $K(Deploy(Service(x), n))$ to behave like a relation over values $K(Service(x))$. The axiom we desire is:

$$\forall x, n.\ K(Deploy(Service(x), n)) \Rightarrow K(Service(x)).$$

As with many LP systems, this axiom becomes an integrity constraint using negation to achieve universal quantification.

$$\neg\exists x, n.\ K(Deploy(Service(x), n)) \wedge \neg K(Service(x)).$$

The variable x is bound by the existential quantifier. In FORMULA, this integrity constraint would be written as a *conforms* clause with the body:

```
no { Deploy(s, n) | Deploy(s, n), no Service(s.name) }
```

The variable s is bound in the outer aggregate, and so $s.name$ is fixed for each evaluation of *no* $Service(s.name)$. With this lexical scoping rule, the order in which constraints are written is irrelevant. For example, the constraint below is equivalent the one above:

```
no { Deploy(s, n) | no Service(s.name), Deploy(s, n) }
```

Note that the constraint *no* $\{f(\ldots)|f(\ldots)\}$ can be written as simply *no* $f(\ldots)$.

3. MODELS

Program instances are represented as sets of well-typed ground facts w.r.t. some domain. *Model modules* hold these ground facts.

Example 2 (Several deployments).

```
model Undeployed of Deployments
{
   sVoice is Service("Voice Recognition").
   sDB    is Service("Big Database").
   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).
}
```

Formally, a domain D is an OLP. A model M *closes* D with a set of facts, written $D[M]$. The properties of a model M are those properties provable by the closed logic program $D[M]$. For example:

- $Deployments[Undeployed] \not\models conforms$, because services are not deployed to nodes.
- $Deployments[Good] \models conforms$, because all services are deployed and all conflicts are respected.
- $Deployments[Bad] \not\models conforms$, because its deployments violate conflicts.

3.1 Design and Implementation

Module system. Models cleanly separate program instances from DSL axioms. Because models themselves encode (possibly very large) programs, the module system for models focuses on composing and sharing large sets of large terms. The aliasing construct:

```
x is f(...)
```

simultaneously introduces the fact $K(f(\ldots))$ and the definition $x = f(\ldots)$. The scope of x is global to model module, so x can be used wherever $f(\ldots)$ would be written. Theoretically, aliasing can allow models to be exponentially more succinct. Practically, aliases help to factorize models into reusable parts. Cyclic aliases cause a compile-time error.

Example 2 also shows that models can be composed through the *extends* operator. In this case, the contents of a model are unioned with the models listed to the right of *extends*. Aliases from imported models are visible to importer. The full module system is described in [3].

Requires and ensures clauses. Because models are complex, it is often useful to make specific statements about them. For this purpose, we provide the *requires* and *ensures* constructs found in other languages:

```
requires body. ensures body.
```

A model satisfies requirements if every body of the form *requires body* is provable in the model. If all requirements are provable then it should follow that every clause of the form *ensures body* is provable. The *requires* clauses state what we expect to be true about the model. The *ensures* clauses state what we believe to follow from these expectations. Implicitly, every model is expected to conform to its domain.

```
requires conforms.
```

These clauses can speak about specific model elements via aliases. However, model aliases are prefixed by % in rule bodies to distinguish them from variables. For example:

```
requires %n0.id = 0, count({id | Node(id)}) = 2.
```

This clause requires the model element *n0* to specifically have the id zero, and the total number of nodes in the model to be two. The % prefix emphasizes that model aliases have a different quantifier scope from body variables, and prevents users from confusing one for the other.

4. PARTIAL MODELS

Partial models partially close domains. A partial model P is *solved* by a model M if M concretizes all the facts of P such that all requirements of P are satisfied in $D[M]$. In this way, partial models describe problem instances. The partial model below describes a specific deployment problem.

Example 3 (A deployment problem).

```
1:  partial model SpecificProblem of Deployments
2:  {
3:     requires Deployments.conforms.
4:
5:     sVoice is Service("Voice Recognition").
6:     sDB    is Service("Big Database").
7:     n0     is Node(0).
8:     n1     is Node(1).
9:     Conflict(sVoice, sDB).
10: }
```

The assertions in lines 5 - 9 must hold in a solution. Line 3 requires a solution to conform to the Deployments domain. (Actually, this requirement is implicitly present.) The *Good* model is a solution (Example 2, lines 9 - 13) to this partial model. Its facts contain all the facts of Example 3 plus the missing deployment facts required to satisfy *Deployments.conforms.*

4.1 Design and Implementation

Solving. Finding a solution to a partial model is clearly non-trivial (undecidable). It necessitates finding a finite set of ground facts that close an OLP such that K satisfies a property under this closure. FORMULA uses a number of state-of-the-art techniques to solve the problem. First, it performs cardinality arguments on P to decide the number and shapes of facts that might be missing from the solution. Second, it symbolically closes P by possibly introducing more facts whose arguments are not concretized. Third, it performs symbolic execution of the symbolically closed logic program to create a set of quantifier-free constraints. The solutions to these constraints correspond to solutions to the original problem. Constraints are solved using the *satisfiability modulo theories* (SMT) solver *Z3*. The generated constraints can be quite complex involving many mathematical theories (e.g. ADTs, lists, bit vectors, linear arithmetic, etc...). All built-in FORMULA operators have been carefully axiomatized into Z3. If these constraints are found to be unsatisfiable, then the process repeats by increasing the size of the symbolic closure.

Illustration. We now give an illustration of this process on Example 3. First, *Deploy* is declared to behave like a total function from services in K to nodes in K. The *SpecificProblem* introduces two services into K, so by cardinality arguments at least two *Deploy* values are missing. Two symbolic *Deploy* values are introduced with symbolic constants as arguments:

```
Deploy(a0, b0). Deploy(a1, b1).
```

Next, the program $D[P]$ is *symbolically executed* allowing it to make progress when encountering symbolic constants. In this execution mode, rules produce values conditioned on quantifier-free constraints involving only symbolic constants. In this example, the derivation of *conforms* is conditioned on the following constraints:

conforms `is provable if`
`Deploy is functional:`
$(a0 = sVoice \vee a0 = sDB) \wedge$
$(a1 = sVoice \vee a1 = sDB) \wedge$
$(b0 = n0 \vee b0 = n1) \wedge$
$(b1 = n0 \vee b1 = n1) \wedge$
$(a0 \neq a1)$
`Deploy avoids conflicts:`
$\neg(a0 = sVoice \wedge a1 = sDB \wedge b0 = b1) \wedge$
$\neg(a1 = sVoice \wedge a0 = sDB \wedge b0 = b1)$

This constraint can be repeatedly solved to generate solutions to P. Unlike *answer set programming* (ASP), there can be many non-minimal solutions to P. Theoretically, these can be eliminated by placing stronger requirements on P. Practically, FORMULA produces smaller solutions first, because they are easier to solver.

5. TRANSFORMS

Transforms are OLPs that transform models between domains. They are useful for formalizing operational semantics, compilers, and product constructions.

Example 4 (A compiler).

```
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) }).
}
domain NodeConfigs
{
   Config ::=
      fun (loc: Natural ->
           list: any Services + { NIL }).
   Services ::=
      new (name: String,
16:            tail: any Services + { NIL }).
17: }
```

Models of the *NodeConfigs* domain contain node configuration files (lines 9 - 17). Each file lists the services that run on a node. These are modeled using recursive ADTs. The *Compile* transform takes a *Deployments* model called *in* and produces a *NodeConfigs* model called *out*. This is accomplished by the *rule* in lines 4 - 7. This rule converts every node into a configuration file containing a list of services.

Transforms utilize all of the features previously discussed. Our type system ensures transforms cannot produce malformed terms. Our module system allows them to be composed into more complex operations. Our *requires* and *ensures* clauses allows users to write additional functional properties. Our OLP semantics allows transforms to be verified by searching for inputs that satisfy requirements but violate ensures.

6. CONCLUSION

We have demonstrated a few of the key concepts provided by FORMULA 2.0 for defining and reasoning about DSLs. Additionally, domains, models, and transforms can be composed to build complex specifications. Transforms can be verified using the same model finding techniques for solving partial models. Rules can contain rich constraints, such as arithmetic, string, and list constraints. More information can be found at `http://formula.codeplex.com/formula`.

7. REFERENCES

[1] L. M. de Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[2] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[3] E. K. Jackson. A module system for domain-specific languages. *TPLP*, 14(4-5):771–785, 2014.

[4] E. K. Jackson, N. Bjørner, and W. Schulte. Canonical regular types. In *ICLP (Technical Communications)*, pages 73–83, 2011.

[5] E. K. Jackson, N. Bjørner, and W. Schulte. Open-world logic programs: A new foundation for formal specifications. Technical Report MSR-TR-2013-55, Microsoft Research, 2013. `http://research.microsoft.com/pubs/192963/MSR-TR-2013-55.pdf`.

[6] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, Platforms and Possibilities: Towards Generic Automation for MDA. In *EMSOFT*, pages 39–48, 2010.

[7] E. K. Jackson, W. Schulte, and N. Bjørner. Detecting specification errors in declarative languages with constraints. In *MoDELS*, pages 399–414, 2012.

[8] E. K. Jackson, G. Simko, and J. Sztipanovits. Diversely Enumerating System-Level Architectrues. In *EMSOFT*, 2013.

[9] G. Simko, D. Lindecker, T. Levendovszky, S. Neema, and J. Sztipanovits. Formal semantics specification of cyber-physical components integration and composition. In *MoDELS*, 2013.