An Integrated Framework to Specify Domain-Specific Modeling Languages*

Bahram Zarrin and Hubert Baumeister
DTU Compute, Technical University of Denmark, Kgs Lyngby, 2800, Denmark

Keywords: Domain-Specific Modeling Languages, Formal Approach, Semantics Specification, DSL-Tools, FORMULA.

Abstract: In this paper, we propose an integrated framework that can be used by DSL designers to implement their desired graphical domain-specific languages. This framework relies on Microsoft DSL Tools, a meta-modeling framework to build graphical domain-specific languages, and an extension of ForSpec, a logic-based specification language. The drawback of MS DSL Tools is it does not provide a formal and rigorous approach for semantics specifications. In this framework, we use Microsoft DSL Tools to define the metamodel and graphical notations of DSLs, and an extended version of ForSpec as a formal language to define their semantics. Integrating these technologies under the umbrella of Microsoft Visual Studio IDE allows DSL designers to utilize a single development environment for developing their desired domain-specific languages.

1 INTRODUCTION

Domain-specific modeling languages (DSMLs) are specialized languages for a particular application area, which use the concepts and notations established in the field. Therefore, they allow domain experts, who are usually non-programmers, to directly employ their domain knowledge about what a system under development should do. Domain-specific modeling languages (DSMLs) not only increase the productivity of the domain experts to specify a problem domain in a manageable and analyzable way but also improve the readability and understandability of the problem specifications as well. Furthermore, they utilize the domain rules as constraints to disallow the specification of illegal or incorrect models in the problem domain.

Since the development of DSMLs is expensive in terms of time and cost, a proper framework can help to reduce the development time and cost. There are a number of language tools and frameworks to develop DSMLs such as EMF, MetaEdit+, MS DSL-tools, Onion, Rascal, Xtext, SugarJ, and MPS. Most of these tools and frameworks provide enough support for specifying the abstract and concrete syntax of DSMLs. Based on these specifications, they automatically generate graphical or textual editors with a set of IDE features, e.g. syntax coloring, synthetic compilation, for the DSML under the development.

Abstract syntax and concrete syntax do not provide any information about what the concepts in a language actually mean. Therefore, defining the semantics of a language is important in order to be clear about what the language describes and means. If the language semantics are not defined clearly and precisely, the language will be open to incorrect use and misinterpretation. Therefore it is essential to capture the semantics in a way that is precise and useful to the user of the language. Formal methods provide the required rigidity and precision for semantic specifications (Gargantini et al., 2009). They provide an unambiguous and precise specification of the language and aid reasoning about the properties of the language by utilizing the tools of mathematical logic. In addition, formal specifications can facilitate an automated generation of language editors, interpreters, compilers, debuggers and other related tools. Although most of the aforementioned tools provide generative approaches, e.g., model-to-model and model-to-text transformation, to specify the semantics of DSMLs, they do not provide a rigorous or formal approach for semantics specifications.

In this paper, we propose an integrated framework to formally specify the syntax and the semantics of DSMLs. We combine MS DSL-Tools and an extended version of ForSpec (Simko et al., 2013), a logic-based specification language which is an extension of

*This research has been partially sponsored by the IRMAR project funded by Danish Council for Strategic Research.

Zarrin, B. and Baumeister, H.
An Integrated Framework to Specify Domain-Specific Modeling Languages.
DOI: 10.5220/0006555800830094
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 83-94
ISBN: 978-989-758-283-7
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

FORMULA (Jackson and Sztipanovits, 2009) developed at Microsoft Research, under the Visual Studio umbrella. This work is motivated by the lack of supporting a formal approach by DSL Tools for specifying the semantics of the DSML proposed in an experience paper (Zarrin and Baumeister, 2014).

Our work in this paper is inspired by the approach presented at (Lindecker et al., 2015). The authors utilize FORMULA for specifying the semantics of DSMLs, and they provide a transformation tool to convert metamodels and models specified within Generic Modeling Environment (GME) (Lédeczi et al., 2001) to FORMULA for analyzing the semantics of the models. The drawback of their approach is switching between different programming tools and development environments. In this paper, we combine the aforementioned technologies under the umbrella of Microsoft Visual Studio IDE to facilitate the development of DSMLs within a single environment. Furthermore, we employ our extension of ForSpec instead of FORMULA that offers better support for semantic specifications. We also provide some language tools, as Visual Studio's extensions for ForSpec, such as code editor, command window, and LaTeXgenerator for pretty-printing specifications, which has been used to generate the specifications presented in this paper.

The remainder of this paper is organized as follows: in Sec. 2, we give a brief introduction to domain-specific modeling languages, we also discuss some of the approaches and existing formal languages for specifying the semantics of domain-specific languages. Afterwards, in Sec. 3 and Sec. 4, we introduce FORMULA and ForSpec. In Sec. 5, we present our extension of ForSpec. Finally in Sec. 6, we propose our integrated framework to design domain-specific modeling languages, illustrated via an example. We evaluate and conclude the paper in Sec. 7.

2 DOMAIN-SPECIFIC MODELING LANGUAGES

Modeling languages, regardless of their general or domain-specific nature, are formally defined (Clark et al., 2001) as a five-tuple: $L = \langle A, C, S, M_C, M_S \rangle$ where

- A is the abstract syntax of the language.
- C is the concrete syntax of the language.
- S is the semantic domain of the language.
- M_C is the mapping from concrete syntax to the abstract syntax.
- M_S is the mapping from abstract syntax to the semantic domain.

The abstract syntax (A) defines the language concepts, their relationships, and well-formlessness rules that state how the concepts may be legally combined. It is important to highlight that the abstract syntax of a language is independent of the concrete syntax (C) of the language. The syntax only defines the form and structure of concepts in a language without dealing with their presentation or meaning. The concrete syntax (C) defines the notations that are used for representing programs or models. There are two main types of concrete syntax; textual syntax and graphical syntax. A textual syntax presents a model or program in a structured textual form which typically consists of a mixture of declarations and expressions. A significant advantage to them is their ability to capture complex expressions. A graphical syntax presents a model or program as a diagram consisting of a number of graphical icons and arrows that represent the model elements and their relationships. The main benefit of these is their ability of expressing a large amount of details in an understandable and intuitive form.

The syntactic mapping, $M_C : C \rightarrow A$, provides a realization of the abstract syntax, by mapping the elements of the concrete syntax to corresponding elements of the abstract syntax. The semantics of a language are defined by choosing a semantic domain S and defining a semantic mapping $M_S : A \rightarrow S$ which relates the concepts and terms of its abstract syntax to corresponding elements of the semantic domain.

In contrast to traditional programming languages, modeling languages are specified with the aid of metamodels, which describe graph structures, therefore instead of static semantics, they have *structural semantics* (Chen et al., 2005). Similar to static semantics, they define the well-formedness rules of the modeling language, which categorize the model instances into well-formed or ill-formed models. Their *behavioral semantics* declares the dynamic behavior of modeling languages as a mapping of the language's instances into a semantic domain that is rich enough for capturing essential aspects of the execution behavior of the modeling language (Chen et al., 2008). The semantic domain and the semantic mapping can be defined by different approaches. We will discuss some of them in detail in the following section.

2.1 Formal Approaches for Semantics Specifications of Modeling Languages

A number of formal approaches have been proposed for specifying the semantics of modeling languages. They can be classified into rewriting, weaving, and translational approaches.

In the rewriting approaches (Karsai et al., 2003; Agrawal et al., 2004; Balasubramanian et al., 2007; Wachsmuth, 2008), the behavior of a modeling language is specified by a set of rewriting rules, which define a mapping from the left-hand side of the rule to its right-hand side. Matching a specification phrase with the left-hand side of a rule triggers substituting it with the right-hand side of the rule. Substituting ends when there are no more applicable rules. The advantage of this approach is that the behavioral specification is directly defined in terms of the metamodel. This approach is more suitable for modeling languages where their behavior can be specified in the operational semantic style.

In the weaving approaches (Montages, 2007; Scheidgen and Fischer, 2007; Gargantini et al., 2009; Ducasse et al., 2009; Mayerhofer et al., 2013), inspired from the UML action semantics (Semantics, 2001), the behavioral semantics of a modeling language are specified directly in the metamodel of the underlying language by attaching operations to the meta-classes and employing a meta-language, e.g. xOCL (Montages, 2007), QVT (QVT, 2008), for the behavioral specification of these operations. These meta-languages are usually a set of primitive actions, e.g. assignment, declaration, conditions, loops, and object manipulations, to specify the behavior of the language. The advantage of this approach is that syntactical and semantical specifications of the language are encapsulated. Its main drawback is that some of these meta-languages are simplified versions of traditional programming languages. Therefore the semantic specifications written with these languages have the same complexity as the specification written in a conventional programming language (Gargantini et al., 2009).

In the translational approaches (Chen et al., 2005; Di Ruscio et al., 2006; Romero et al., 2007; Gargantini et al., 2009; Sadilek and Wachsmuth, 2009; Simko et al., 2012; Simko, 2014), the semantics of a modeling language are specified as a mapping from the metamodel of the underlying language to a metamodel of another language which already has a wellknown semantics, e.g. abstract state machines (Gurevich, 1995). The benefit of this approach is that the available tools of the target language can be used for performing formal analysis. Its disadvantage is that the DSL designer should have knowledge of the target language in order to specify and understand the semantics of the underlying language.

The approach utilized in this paper is the continuation of (Balasubramanian and Jackson, 2009; Simko et al., 2012; Simko, 2014), in which the metamodel of the underlying modeling language is translated into a formal specification language, e.g. FORMULA and ForSpec, and its behavioral semantics are specified using the constructs of the specification language. Therefore, in the following, we describe these specification languages in more details.

3 FORMULA

FORMULA (Jackson et al., 2010) is a specification language based on Constraint Logic Programming (Jaffar and Lassez, 1987; Frühwirth et al., 1992) which is the combination of the declarativity of logic programming and the efficiency of constraint solving.

Each program in FORMULA consists of several constructs called “module”s. Different kinds of modules are defined in this language, of which the most important ones are “domain”, “model”, and “transform”. The domain module is a blueprint for a set of models which are composed of type definitions, data constructors, rules, and queries. The other module called “model” is a model of a domain that consists of a set of facts that are defined through the data constructors of the domain. Accordingly, given a model and its related domain, FORMULA is able to deduce a set of final facts which provide the minimum fixed-point solution for the initial specifications in the model. FORMULA leveraging Microsoft’s Z3 (e.g. an SMT solver), therefore, for any given partial model i.e. a model with some underspecified facts and a set of constraints, it can search for a solution i.e. a completion of the model, where all the constraints are satisfied. If such a solution is not feasible, it returns “unsatisfiable” which indicates that the expected solution does not exist. FORMULA benefits from some important features; for example, bounded model checking is supported, model finding tools are provided, and finally FORMULA provides metamodels and straightforward representation of models. It has been used in several research works such as (Jackson and Sztipanovits, 2009; Jackson et al., 2009) to specify the structural semantics of DSMLs. In addition, some core components of Windows 8.0, e.g. the USB 3.0 stack, were built using DSLs specified with this language (Jackson, 2014). In the following we explain the notations of FORMULA, with some examples, that are used in this paper. A more detailed description of this language can be found in (Jackson et al., 2011; Jackson et al., 2010).

The “DirectedGraph” domain presented in the following example formalizes the metamodel of a simple directed graph language, and represents a model of this domain. Two data types called “Node” and “Edge” are used to specify the elements of the graph,

and also a union type called "Element" is used to refer to these elements.

```
domain DirectedGraph {
Node ::= new ( label :  String ).
Edge ::= new ( src :  Node, dst :  Node ).
Element ::= Node + Edge.
conforms no { n.label | n :  Node , m :
Node , n.label = m.label , m != n }.  }
model simple_graph of DirectedGraph {
node1 is Node ( "a" ).
node2 is Node ( "b" ).
edge1 is Edge ( node1 , node2 ).}
```

A query called "conforms" defined at the end of the domain definition guarantees the uniqueness of the node's labels in a graph. Queries are Boolean expressions that use the same constraint logic expressions as rules. Queries can also be defined as conjunctions, disjunctions, and negations of other queries. The "conforms" keyword denotes a special query that is used to distinguish between the well-formed models and ill-formed models of the domain.

Domain composition is supported by the "extends" and "includes" keywords. Both denote the inheritance of all types (data constructors and rules). While "A extends B" ensures that all the well-formed models of A are well-formed models of B, "A includes B" may contain well-formed models in A, which are ill-formed models of B. The domains represented in the following example formalize the metamodels of a directed acyclic graph and a tree languages. These domains illustrate how domains can be extended and, particularly, how queries can be used in domains to specify the structural semantics of these simple languages.

```
domain DAG extends DirectedGraph {
Path ::= ( Node , Node ).
Path ( u , w ) :- Edge ( u , w ) ; Edge ( u
, v ) , Path ( v , w ).
conforms no Path ( u , u ).  }
domain Tree extends DAG {
conforms no { w | Edge ( u , w ) , Edge ( v
, w ) , u != v }.  }
```

FORMULA supports relational constraints, such as equality of ground terms, and arithmetic constraints over real and integer data types. A special data type called *Data* in FORMULA refers to all the data types defined in the domain. The special symbol "_" denotes an anonymous variable that cannot be referenced anywhere else.

FORMULA also supports model transformation using transform modules. This module consists of rules for deriving initial facts in an output model from initial and derived facts in an input model as well as input parameters. The rules are the same as they are in domains, except that the left-hand side contains facts in the output model and the right-hand side contains facts from the input and output models. The transform modules can also contain data constructors and type declarations for transform-local derived facts and union types. The following specifications transforms a given graph to a complete graph by adding all the possible edges between the given graph's nodes.

```
transform Complete ( GraphIn ::  DAG )
returns ( GraphOut ::  DAG ) {
GraphOut.Node ( x ) :- GraphIn.Node ( x ).
GraphOut.Edge ( x , y ) :- GraphIn.Node ( x
) , GraphIn.Node ( y ) , x != y.  }
```

Name-spaces are used for handling multiple definitions with the same name in different ancestor domains. For example, domain "GraphIn :: DAG" uses the name GraphIn for referring to elements of DAG. We can refer to the elements of GraphIn by inserting a dotted qualification "GraphIn." in front of the type identifiers defined in the domain.

4 ForSpec LANGUAGE

ForSpec is an extended version of FORMULA, proposed by Gabor Simko (Simko, 2014) to support the structural and behavioral semantics specifications of modeling languages for cyber-physical systems. ForSpec extends FORMULA with goal-driven and functional terms, semantic functions and semantic equations to provide support for operational, denotational and translational style specifications. ForSpec also has been used to specify the denotational semantics of a bond graph language, which is a physical modeling language (Simko et al., 2012), and to specify the structural and behavioral semantics of a cyber-physical system modeling language (Simko et al., 2013). In the following, we briefly introduce these extensions. For a more detailed description of the language see (Simko, 2014).

4.1 Functional Terms

Functions are very important in order to define computations within any system. Despite this, a set of function that mostly define the type of the relations between the data types (e.g. one-to-one, one-to-many, and etc.) defined within a domain, FORMULA does not provide means to specify a function, e.g. a function to sum up two numbers, as we know it in

other programming languages. The following specification is the conventional way to define a function called Add. In this example, two data types "Add" and "Add_triger" are defined. The first data type specifies the input parameters and the output parameter of the function as a three-tuple. The second data type specifies only the input arguments of the function. In addition, a rule is defined to specify the computation. This rule has a predicate to check if the function should be computed. These specifications will be more complicated if a function needs to use another function to do a part of its computation.

```
domain Equation {
 Add ::= (Integer, Integer, Integer).
 Add_trigger ::= (Integer, Integer).
 Add (x, y, z) :- z = x + y, Add_trigger (x,
y).  }
```

ForSpec addresses this deficiency by introducing functional terms. The function mentioned above can be defined in ForSpec as follows:

```
domain Equation {
 Add ::= [Integer, Integer ⇒ Integer].
 Add (x, y) ⇒ (z) :- z = x + y.  }
```

Given these specifications, ForSpec automatically generates the required data types and adds the required predicates to trigger the related rules. Therefore, the above specification will be converted into the following specifications by the ForSpec compiler (# is a reserved character in ForSpec which is used by its compiler).

```
 Add ::= (Integer, Integer, Integer).
#Add ::= (Integer, Integer).
 Add (x, y, z) :- z = x + y, #Add (x, y).
```

4.2 Semantic Functions

ForSpec introduces syntactic elements for defining semantic functions. The following example specifies a simple equation language and provides its denotational semantics by using a semantic function and a set of semantic equations.

```
 Exp ::= Plus + Mult + Minus + Real.
 Mult ::= new ( lhs :  Exp , rhs :  Exp ).
 Plus ::= new ( lhs :  Exp , rhs :  Exp ).
 Minus ::= new ( lhs :  Exp , rhs :  Exp ).
 ℰ :  Exp → Real.
 ℰ⟦Plus⟧ = addition where addition =
ℰ⟦Plus.lhs⟧ + ℰ⟦Plus.rhs⟧.
 ℰ⟦Mult⟧ = multiplication where
multiplication = ℰ⟦Mult.lhs⟧ * ℰ⟦Mult.rhs⟧.
 ℰ⟦Minus⟧ = subtraction where subtraction =
ℰ⟦Minus.lhs⟧ - ℰ⟦Minus.rhs⟧.
```

The semantic function first declares a data type of the same name. Secondly, it creates rules for extracting information from the semantic functions. For example, the semantic function "E : Exp − > Real" declares a data type equivalent to "E ::= [Exp => Real]", and the generated rules extract every possible instant of the Exp over which the function ranges in a concrete model (Simko, 2014).

4.3 Union Type Extension

Union types are supported well in ForSpec. This allows extending the existing union type declarations with additional data types. This is essential in the modular development of modeling languages, since it facilitates the language composition (Jackson and Sztipanovits, 2009). The following example specifies a simple language for defining arithmetic equations:

```
domain Equations {
 Exp ::= Real + Operation.
 Operation ::= BinOp + UniOp.
 UniOp ::= Neg.
 Neg ::= new (any Exp).
 BinOp ::= Plus + Minus + Mult.
 Plus ::= new (any Exp, any Exp).
 Minus ::= new (any Exp, any Exp).
 Mult ::= new (any Exp, any Exp).  }
```

If we need to reuse this language and extend it to support relational expressions, the extended domain can be specified in ForSpec as follows:

```
 domain AdvancedEquations extends Equations {
 Exp += Boolean.
 BinOp + = LT + LET + GT + GET + EQ + NotEQ.
 LT ::= new (any Exp, any Exp).
 LET ::= new (any Exp, any Exp).
 GT ::= new (any Exp, any Exp).
 GET ::= new (any Exp, any Exp).
 EQ ::= new (any Exp, any Exp).
 NotEQ ::= new (any Exp, any Exp).
 UniOp += Not.
 Not ::= new (any Exp).  }
```

As presented in this example, we can extend *Exp*, *BinOp*, and *UniOp* data types in the base domain with the new data types introduced in the extended domain. This is a useful feature in ForSpec which can help avoid code duplication by using union type extension.

5 EXTENDING ForSpec

ForSpec provides essential means for the structural and behavioral specifications of DSMLs. However, it has some limitations for specifying the semantics of the DSML proposed in the experience paper (Zarrin and Baumeister, 2014). In this section, we address these limitations and extend ForSpec to support these deficiencies.

5.1 List Data Type

ForSpec does not support *List* data types explicitly. The following example shows how a list can be defined at the moment in ForSpec. It specifies a domain which includes a type called *Substance* and a list called *SubstanceList*. The elements of the list are the type of *Substance*. A model is defined as an instance of this domain, and it describes three substances and one list that includes them.

```
domain Material {
 Substance ::= new (name:String, value:Real).
 SubstanceList ::= new ( hd :  Substance ,
tail :  {SubstanceList + Nil}).  }
model material of Material {
 CH4 is Substance ("CH4", 0.23).
 H2 is Substance ("H2", 2.44).
 SubstanceList(H2, SubstanceList(CH4, Nil)).}
```

Although we are able to define a list structure implicitly as above, this will require more effort when we need to apply operations on the lists, e.g. add elements, or remove elements. To this end, we extend ForSpec syntax to support *List* as a primitive data type. The following example presents the same specification as the example given above but using our explicit list construct.

```
domain Material {
 Substance ::= new (name:String, value:Real).
 SubstanceList ::= list < Substance >.  }
model material of Material {
 CO is Substance ("CO", 0.23).
 H2 is Substance ("H2", 2.44).
 CH4 is Substance ("CH4", 1.03).
 SubstanceList <CO, H2, CH4> }
```

The extension provides a syntactic sugar to explicitly define a list data type in ForSpec, and it transfers the list data type in the first specifications to the equivalent specification in the second specifications. Therefore, head and tail of a list can be obtained by accessing the structural fields of the list, e.g. “hd” and “tail”. We also extend the built-in functions of ForSpec with some list operation functions as follows:

- **append (list1, list2):** this function appends list1 and list2 and returns the result as a list.
- **count (list):** this function counts the number of elements in the list.
- **isin (element , list):** this function indicates whether or not the list contains the given element. It can also provide information on whether or not the list contains an item with the value of a particular field of the item matches with the given element. The syntax to do this is **isin**(element, list[field_name]). The following specifications count the number of SubstanceList that contains a Substance named “CO” in the model.

```
NoSubstanceListForCO ::= new ( Integer ).
NoSubstanceListForCO ( n ) :-
 n = count ( { sl | sl is SubstanceList ,
isin ("CO", sl[name]) } ).
```

5.2 Set Comprehensions

Set comprehensions in ForSpec are defined as $\{head \mid body\}$, and they are used by built-in functions *count* and *toList*. *Count* computes the number of elements in the set and *toList* stores the items in the set in a list data structure as presented earlier. We extend the *toList* function to accept the data type of the *list* as the arguments. The extended function stores the elements of the set comprehension in an instance of the given list, and it uses constant value *Nil* to represents the end of the list. For example, the following example specifies a list of all the substances which have positive values.

```
PositiveSubstance ::= new ( SubstanceList ).
PositiveSubstance (sl) :-
 sl = toList ( SubstanceList , { s | s is
Substance , s.value > 0 } ).
```

Furthermore, we also extend ForSpec by providing syntax to iterate the list elements within set comprehensions. This extension allows ForSpec to support some list operations, e.g. *filter()*, *map()*, and *reduce()*, which are quite useful in functional programming. The following specifications defines a function that rescales a *SubstanceList*. It maps each element of a given *SubstanceList* to the rescaled element in the list, called result. The list iterator is defined as $e \leftarrow sl$ where *sl* is the given list, and *e* is a variable representing an element of the list.

```
RescaleSubstanceList ::= [ SubstanceList ,
Integer ⇒ SubstanceList ].
RescaleSubstanceList (sl, n) ⇒ (result) :-
result = toList (SubstanceList,
{Substance(e.name, e.value * n) | e ← sl}).
```

Given the above specifications, the extension automatically generates a data type called *#iterator* and the required rules to iterate the list elements. Therefore, the following specifications present the ForSpec translation of the given example:

```
RescaleSubstanceList ::= [ SubstanceList ,
Integer ⇒ SubstanceList ].
RescaleSubstanceList (sl , n) ⇒ (result):-
result = toList(SubstanceList,
{Substance(e.name, e.value * n) |
#iterator_0x0 (e , _)}).

#iterator_0x0 ::= new ( value : Substance ,
seq : SubstanceList ).
#iterator_0x0 ( sl.hd , sl.tail ) :-
#iterator_0x0 ( _ , sl ) , sl != Nil.
#iterator_0x0 ( sl.hd , sl.tail) :-
#RescaleSubstanceList ( sl , _).
```

We also provide support for *union* and *intersection* operators for the set comprehensions. The following example specifies a function that merges two substance lists.

```
MergeSubstanceList ::= [SubstanceList ,
SubstanceList ⇒ SubstanceList + {Nil}].
MergeSubstanceList ( l1 , l2 ) ⇒ ( l3 ) :-
 l3 = toList ( SubstanceList ,
 { Substance ( e1.name , e1.value + e2.value
) | e1 ← l1 , e2 ← l2 , e1.name = e2.name}
 union {e1 | e1 ← l1 , e1.name ∉ l2[name]}
 union {e2 | e2 ← l2 , e2.name ∉ l1[name]}).
```

5.3 Reduce Function

In functional programming, *reduce* refers to a function that operates on a list or any recursive data structure to collapse or accumulate its elements into a single element or value by applying the same computation to each element. In this work, we extend ForSpec to be able to specify the *reduce* function.

The following example defines a *reduce* function called *MergeSubstanceLists* to aggregate the substances of a list of *SubstanceList*:

```
SubstanceLists ::= list < SubstanceList >.
MergeSubstanceLists ::= [SubstanceLists >>
MergeSubstanceList >> SubstanceList].
```

The following example defines a *reduce* function called *Sum* to calculate the sum of the numbers in the given list:

```
NumberList ::= list <Real>.
Add ::= [ Real , Real ⇒ Real ].
Add ( x , y ) ⇒ ( z ) :- z = x + y.
Sum ::= [ NumberList >> Add >> Real].
```

5.4 Typed Union Type

FORMULA and ForSpec allow us to freely combine different types e.g. built-in types, composite types, and union types. The current limitation is that they do not provide any mechanism to define constraints on the components of a union type. This is essential when a union type should be extended from other domain modules by using union type extension, since any arbitrary types can be added to the components of the union type. For instance, in the *Equations* example the *BinOp* union type can be extended in the extended domain by any type, which may not be a binary operation anymore. Therefore, we need to specify that any binary operator should have at least a *left* and *right* fields of type *Exp*. To support this, we introduce a new built-in type to ForSpec that allows defining a type for the union types. This data type can be considered as the equivalent of interfaces in object oriented paradigm.

The syntax to define a typed union is the same as the syntax to define a composite type. The main difference is using "::=" instead of "::=". This helps us to distinguish between the definition of these two built-in types. A typed union is a particular union type which can declare the *new* keyword and a set of named fields. It enforces a set of type checking rules to ensure that all of the components of the typed union match this common declaration. The rules of the type checking is as follows:

- If the typed union has the *new* keyword in its definition, then all of its components should have the *new* keyword in their declaration.
- All the fields defined in the typed union definition should have a unique name.
- For each field specified in the typed union declaration, all the components of the typed union should have the same field with the same type (not necessarily the same order) in their declaration.

Union type extension "+=" can be used to add a type component to a typed union type. It can be used both within the same domain that contains the declaration or within other domains modules extending the domain.

A typed union type cannot be used to generate a fact either via rules or constructors within a model of a domain. But its signature can be used as a composite type to match the facts of the type of its components in the knowledge base of the ForSpec interpreter. There is a difference between matching a composite type and a typed union type. For the composite type, the matching is done by comparing the arguments of the fact and matching terms in the sequence,

and comparing the type of the fact with the type of matching term. While in the typed union the argument matching is done according to the name of the fields and not their position in the constructor (same fields should have the same value and the type matching is done according to the type of component.

The following example utilizes a typed union type to define an interface for a type of *Component*. This type has been extended with a type of *Network* using a union type extension operator in another domain called "Network". The typed union type is also used to match the facts of the type of *Component* to check if they have ports with duplicated name.

```
domain Core {
InPort ::= new (id:  String, type:  String).
OutPort ::= new (id:  String, type:  String).
Port ::= InPort + OutPort.
PortList ::= list < Port >.
Component :;= new (id:  String, ports:
PortList).
...
InvalidComponent ::= ( String ).
InvalidComponent ( id ) :-
 Component ( id , ports ),
 no { x | x ← ports , y ← ports , x != y ,
x.id = y.id }.  }
domain Network extends Core {
Component += Network.
Network ::= new ( id :  String , ports :
PortList, ...  ).
...  }
```

6 INTEGRATION WITH MICROSOFT DSL TOOLS

The Visualization and Modeling Software Development Kit (VMSDK) is a meta-modeling framework to build powerful graphical domain-specific languages that can be integrated into Microsoft Visual Studio. The core of Visualization and Modeling Software Development Kit (VMSDK) is a DSL definition diagram for specifying both the metamodel and the graphical notations of a domain-specific language. The definition diagram is used by the framework to generate a graphic editor for the VMSDK, so that modelers can edit and view the whole or parts of the model, serialization objects which store the models in XML format, model transformation commands, mechanisms for generating code or other artifacts from the model by using text templates, customized Visual Studio Shell, and APIs to interact with the shell (Microsoft, 2014).

In this paper, we use a simple example to explain

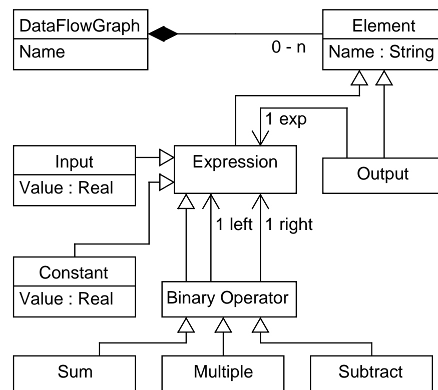


Figure 1: Metamodel of a simple data flow language.

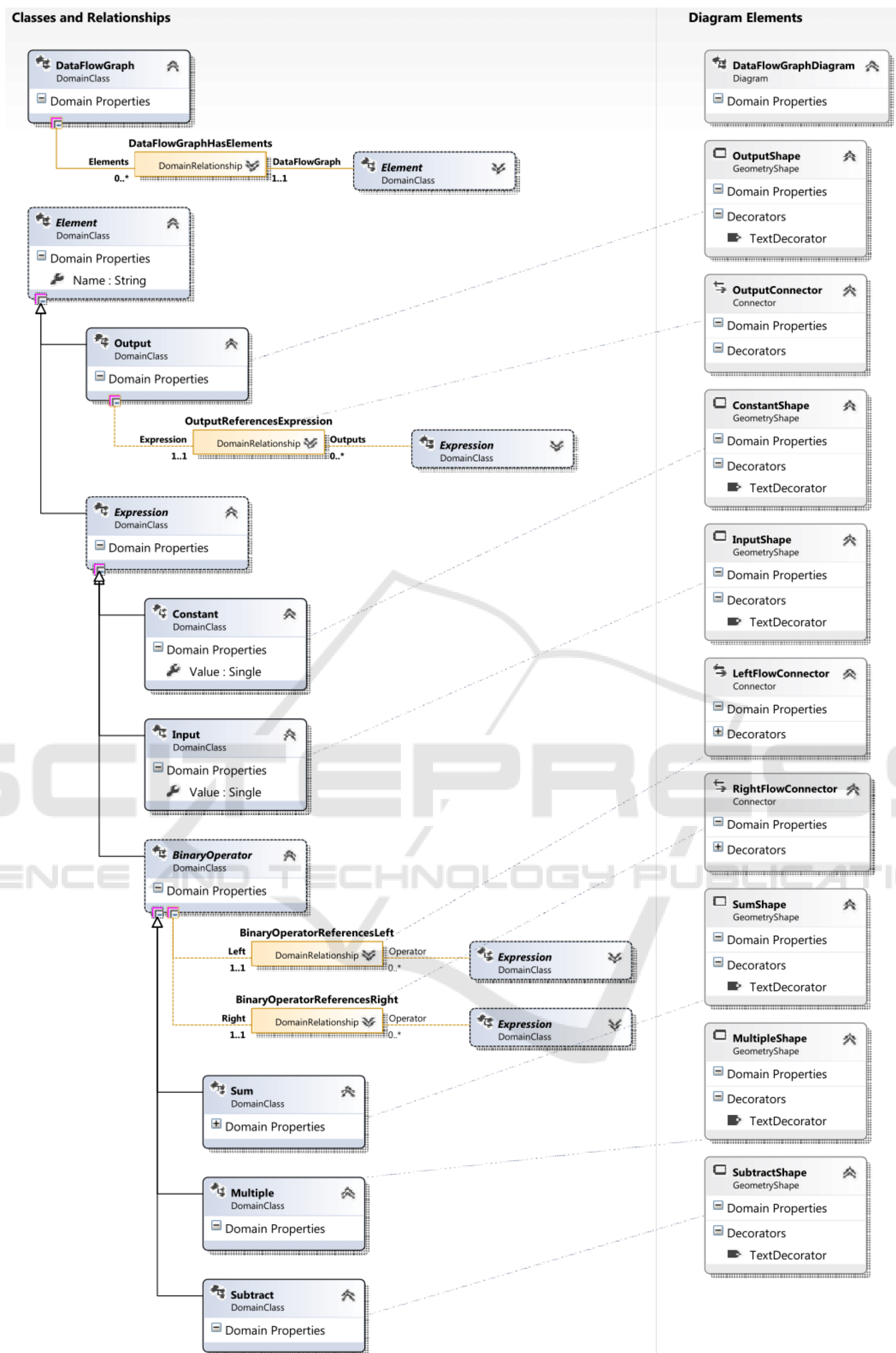
the integration of these tools. More complex DSLs, e.g., domain specific language for modeling waste management systems, developed within this framework can be found in (Zarrin, 2016).

Figure 1 presents the metamodel of a simple dataflow language. This language is composed of input, output, constant, and three binary operators including sum, multiply, and subtract. By this simple language, a modeler can specify the computation of a simple arithmetic expression, e.g. $Z = 5 * X + 6 * Y$. In the following, we briefly introduce MS DSL Tools and specify the metamodel and concrete syntax of this language. Afterwards, we explain our approach for transferring the metamodel and the models of this language to ForSpec specification. At the end, we provide an operational semantics of this language.

6.1 Defining the Proposed DSL

The DSL definition diagram for the proposed DSL in Visual Studio is illustrated in Figure 2. The diagram has two swim lanes, one of which is used to show the domain classes and their relationships (abstract syntax) and the other of which is used to show the diagram notations (concrete syntax). The domain classes are used to define the model elements and the domain relationships are used to define the relationships between the elements. The appearance of the model elements in the diagram is defined by using shape classes and connectors.

Domain classes can be defined as abstract or nonabstract classes, and they can be inherited from each other to define the model elements. Model elements can be linked to each other by using relationships. These links are binary and they precisely connect two elements of the model, while each element of the model can be linked to multiple elements. There are two different kinds of domain relationships; embedding relationships and reference relationships.



SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Figure 2: DSL definition diagram for a simple data flow language.

As presented in Figure 2, a domain class called *DataFlowGraph* is used as the root element of the diagram representing the language. An embedded relationship, representing the composite relationship in the metamodel, is defined from the root element to an abstract domain class called *Element*, which is the base class of all of the model elements. According to the metamodel presented in Figure 1, the other domain classes are defined and inherited from the *Element* class. The *left* and *right* relationship of the

binary operators and the *exp* relationship of the output elements are defined as reference relationships, which provide connectivity between the model elements. Finally, for all of the non-abstract domain classes, which should appear as an element in the DSL diagram, a shape class is defined to describe the concrete syntax of the element in the model diagram.

6.2 Generating ForSpec Specifications

VMSDK generates code for domain classes, connectors, shapes, diagram editor, model explorer, validations, and other artifacts based on the DSL definition file (.dsl). The code generation is done by using a set of text templates files (.tt) located in a folder, within a DSL project, called *Generated Code*.

We use the same method to generate ForSpec specifications for the metamodel specified within the DSL definition file and its corresponding models. To this end, we extend the DSL project template to include two additional text template files in the *Generated Code* folder, for whenever DSL developers create a new DSL project. These text template files, along with the other templates located in the folder, regenerate the required code automatically whenever the DSL definition file is changed. One of these files is used to generate the ForSpec specification for the metamodel of the DSL. This template directly generates a ForSpec file (.4sp) which contains a domain module equivalent to the metamodel. Therefore, the specifications are available at the design time of the DSL, and the DSL developer can extend these specifications with the DSL semantics. The following ForSpec specification is generated for the metamodel of the language defined in the DSL definition diagram presented in Figure 2.

```
domain SimpleLanguage {
 DataFlowGraph::= new (Elements:ElementList).
 Element :;= new (name:  String).
 Element += Output + Expression.
 Input ::= new (value:Single, name:String).
 Output ::= new (name:  String, Expression:
Expression).
 Constant::= new (value:Single, name:String).
 Expression :;= new (name:  String).
 Expression += Input + Constant +
BinaryOperator.
 BinaryOperator :;= new (name:  String, Left:
Expression, Right:  Expression).
 BinaryOperator += Sum + Multiple + Subtract.
 Sum ::= new (name:  String, Left:
Expression, Right:  Expression).
 Multiple ::= new (name:  String, Left:
Expression, Right:  Expression).
 Subtract ::= new (name:  String, Left:
Expression, Right:  Expression).
 ElementList ::= list < Element >.  }
```

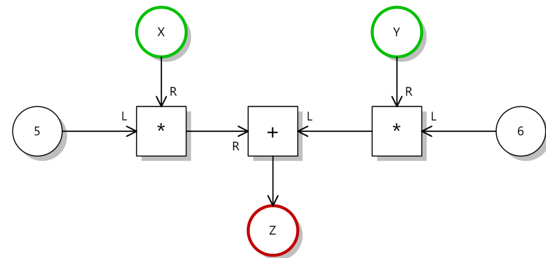


Figure 3: A a model to compute $Z = 6 * Y + 5 * X$.

The ForSpec specification is generated according to the following procedures: For each non-abstract domain class (e.g. *Sum*), a data type with the same name will be generated. The parameters of the data type include all the domain properties explicitly defined for the domain class and its base classes (e.g. *name* are defined in the *Element* domain class). In addition, for each domain relationship with which their source is associated, the domain class or its base classes (e.g. the *Left* and *Right* domain relationship associated to the *BinaryOperator*), a parameter will be generated. The type of parameter is defined according to the multiplicity of the relationship. For one to one relationships, the type is the same as the type of the domain class associated with the target of the relationship. For the one to many relationships, the type is defined as a list of the type of the domain class targeting the relationship. For each abstract class (e.g. *Element*), depending on its domain properties, a union type or a typed union type is generated. The first is generated if the class does not have any domain property. The later is generated if the class or its base class has one or more domain properties. In both cases, the classes inherited from the abstract class are added to the union definition.

The other template is used for generating ForSpec specifications for the model instances of the DSL. To this end, it generates, e.g. C#, VB, code for an adapter that can transfer the model instances of the DSL to ForSpec. This adapter can be used by applications or within another text template file in Visual Studio IDE. Figure 3 illustrates a concrete model of the language, designed with the diagram editor of the language, to model $Z = 6 * Y + 5 * X$. The following specifications are produced for this model by the adapter generated for the language.

```
model simple_exp of SimpleLanguage {
 Constant1 is Constant (5, "Constant1").
 Constant2 is Constant (6, "Constant2").
```

```
 X is Input (5, "X").
 Y is Input (2, "Y").
 Multiple1 is Multiple ("Multiple1",
Constant1 , X ).
 Multiple2 is Multiple ("Multiple2",
Constant2 , Y).
 Sum1 is Sum ("Sum1", Multiple2, Multiple1).
 Z is Output ("Z", Sum1).
 DataFlowGraph (ElementList<Constant1,
Constant2, Y, Multiple1, X, Sum1, Multiple2,
Z>).  }
```

To make this language executable, we extend the domain with the semantic specifications. For this example, we first define a semantic domain for the language, which has a data type and the required functions over this data type. Then, we use denotational semantics to specify the mapping from the syntactic elements to the semantic elements. Although we could use *integer* as the semantic domain here, we choose to use a simple semantic domain to show the general approach.

```
domain ExecutableSimpleLanguage extends
SimpleLanguage {
 // Semantic domain :
 SD ::= Mult + Add + Sub + Val.
 Val ::= new (Integer).
 Mult ::=[ Val , Val ⇒ Val ].
 Mult ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c
) ) :- c = a * b.
 Add ::=[ Val , Val ⇒ Val ].
 Add ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c )
) :- c = a + b.
 Sub ::= [ Val , Val ⇒ Val ].
 Sub ( Val ( a ) , Val ( b ) ) ⇒ ( Val ( c )
) :- c = a - b.
 // Semantics functions :
 𝒢 :  DataFlowGraph → SD.
 ℰ :  Element → SD.
 𝒢⟦DataFlowGraph⟧ = value where
 e ← DataFlowGraph.Elements , e :  Output
, ℰ⟦e⟧ = value.
 ℰ⟦Constant⟧ = Val (Constant.value).
 ℰ⟦Input⟧ = Val (Input.value).
 ℰ⟦Output⟧ = value where
 ℰ⟦Output.Expression⟧ = value.
 ℰ⟦Sum⟧ = value where
 ℰ⟦Sum.Left⟧ = l , ℰ⟦Sum.Right⟧ = r , Add (
l , r ) ⇒ (value).
 ℰ⟦Subtract⟧ = value where
 ℰ⟦Subtract.Left⟧ = l , ℰ⟦Subtract.Right⟧ =
r , Sub ( l , r ) ⇒ (value).
 ℰ⟦Multiple⟧ = value where
 ℰ⟦Multiple.Left⟧ = l , ℰ⟦Multiple.Right⟧ =
r , Mult ( l , r ) ⇒ (value).  }
```

According to the given semantics and the metamodel, we can execute the given model in ForSpec. The result of the expression specified in the model is computed by the semantic function $\mathcal{G}$ which is evaluated to 37 for the expression.

7 CONCLUSIONS

The integrated framework presented in this paper can be used to specify graphical domain-specific languages. In this framework, DSL Tools are used to formally define the concrete syntax and ForSpec is used to specify the semantics of domain-specific languages. We extended ForSpec with list datatypes, union operators, iterators, map and reduce functions, and typed union datatype which helps write more complicated specifications within the language. We combined ForSpec with Microsoft DSL Tools under the umbrella of Microsoft Visual Studio IDE. This allows DSL designers to utilize a single development environment to develop their desired domain-specific languages. Since we developed this framework based on existing technologies and languages, the users do not need to learn new programming languages or tools to develop DSLs.

One of the drawbacks of function calls in ForSpec is that it executes functions through a set of rules, which means that for each call it adds the facts generated during the execution to the knowledge-base. These facts will stay there forever, even after the function returns. This makes the execution slower and slower after each function call, due to increasing the number of facts in the knowledge-base. This could be addressed by proposing a mechanism which allows to remove facts from the ForSpec's knowledge-base.

The integrated framework presented in this paper is not suitable for developing none graphical DSLs. Combining this framework with other language development tools for developing textual languages such as Microsoft Irony is our future work.

REFERENCES

Agrawal, A., Simon, G., and Karsai, G. (2004). Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*.

Balasubramanian, D. and Jackson, E. K. (2009). Lost in translation: Forgetful semantic anchoring. In *Proceedings of the 2009 IEEE/ACM International Confe-*

rence on Automated Software Engineering, ASE '09. IEEE Computer Society.

Balasubramanian, D., Narayanan, A., van Buskirk, C., and Karsai, G. (2007). The graph rewriting and transformation language: Great. *Electronic Communications of the EASST*.

Chen, K., Sztipanovits, J., Abdelwalhed, S., and Jackson, E. (2005). Semantic anchoring with model transformations. In *Model Driven Architecture–Foundations and Applications*. Springer.

Chen, K., Sztipanovits, J., and Neema, S. (2008). Compositional specification of behavioral semantics. In *Design, Automation, and Test in Europe*, pages 253–265. Springer Netherlands.

Clark, T., Evans, A., Kent, S., and Sammut, P. (2001). The MMF Approach to Engineering Object-Oriented Design Languages. In *Proceedings of the First Workshop on Language Descriptions, Tools and Applications*, LDTA 2001. Middlesex University.

Di Ruscio, D., Jouault, F., Kurtev, I., Bézivin, J., and Pierantonio, A. (2006). Extending amma for supporting dynamic semantics specifications of dsls. Technical report, LINA Research Report.

Ducasse, S., Girba, T., Kuhn, A., and Renggli, L. (2009). Meta-environment and executable meta-language using smalltalk: an experience report. *Software & Systems Modeling*, 8(1):5–19.

Frühwirth, T., Herold, A., Küchenhoff, V., Le Provost, T., Lim, P., Monfroy, E., and Wallace, M. (1992). *Constraint logic programming*. Springer.

Gargantini, A., Riccobene, E., and Scandurra, P. (2009). A semantic framework for metamodel-based languages. *Automated Software Engineering*, 16(3-4):415–454.

Gurevich, Y. (1995). Evolving algebras 1993: Lipari guide. *Specification and validation methods*, pages 9–36.

Jackson, E., Porter, J., and Sztipanovits, J. (2009). Semantics of domain specific modeling languages. *Model-Based Design of Heterogeneous Embedded Systems*.

Jackson, E. and Sztipanovits, J. (2009). Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 8(4).

Jackson, E. K. (2014). A module system for domain-specific languages. *Theory and Practice of Logic Programming*, 14(4-5).

Jackson, E. K., Bjørner, N., and Schulte, W. (2011). Canonical regular types. *ICLP (Technical Communications)*.

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*. ACM.

Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119. ACM.

Karsai, G., Agrawal, A., Shi, F., and Sprinkle, J. (2003). On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11).

Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., and Karsai, G. (2001). Composing domain-specific design environments. *Computer*, 34(11):44–51.

Lindecker, D., Simko, G., Levendovszky, T., Madari, I., and Sztipanovits, J. (2015). Validating transformations for semantic anchoring. *Journal of Object Technology*.

Mayerhofer, T., Langer, P., Wimmer, M., and Kappel, G. (2013). *xMOF: Executable DSMLs Based on fUML*. Springer International Publishing.

Microsoft (2014). Visualization and Modeling SDK — Domain-Specific Languages. http://msdn.microsoft.com/en-us/library/bb126259.aspx.

Montages (2007). xocl: executable ocl.

QVT (2008). Omg. mof 2.0 query/view/transformation (qvt). OMG Document - formal/08-04-03.

Romero, J. R., Rivera, J. E., Durán, F., and Vallecillo, A. (2007). Formal and tool support for model driven engineering with maude. *Journal of Object Technology*.

Sadilek, D. and Wachsmuth, G. (2009). Using grammarware languages to define operational semantics of modelled languages. In *International Conference on Objects, Components, Models and Patterns*, pages 348–356. Springer.

Scheidgen, M. and Fischer, J. (2007). Human comprehensible and machine processable specifications of operational semantics. In *Proceedings of the 3rd European Conference on Model Driven Architecture Foundations and Applications*. Springer-Verlag.

Semantics, A. (2001). The action semantics consortium for the uml. OMG Document - formal/2001-03-01.

Simko, G. (2014). *Formal Semantic Specification of Domain-Specific Modeling Languages for Cyber-Physical Systems*. PhD thesis, Vanderbilt University.

Simko, G., Levendovszky, T., Neema, S., Jackson, E., Bapty, T., Porter, J., and Sztipanovits, J. (2012). Foundation for model integration: Semantic backplane. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers.

Simko, G., Lindecker, D., Levendovszky, T., Neema, S., and Sztipanovits, J. (2013). Specification of cyber-physical components with formal semantics–integration and composition. In *Model-Driven Engineering Languages and Systems*. Springer.

Wachsmuth, G. (2008). Modelling the operational semantics of domain-specific modelling languages. In *Generative and Transformational Techniques in Software Engineering II*. Springer.

Zarrin, B. (2016). *Domain Specific Language for Modeling Waste Management Systems*. PhD thesis, Technical University of Denmark.

Zarrin, B. and Baumeister, H. (2014). Design of a domain-specific language for material flow analysis using microsoft dsl tools: An experience paper. In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, DSM '14, pages 23–28. ACM.