A Domain-Specific Visual Language For Domain Model Evolution

Jonathan Sprinkle, Gabor Karsai
Vanderbilt University

Abstract:

Domain-specific visual languages (DSVLs) are concise and useful tools that allow the rapid development of the behavior and/or structure of applications in well-defined domains. These languages are typically developed specifically for a domain, and have a strong cohesion to the domain concepts, which often appear as primitives in the language. The strong cohesion between DSVL language primitives and the domain is a benefit for development by domain experts, but can be a drawback when the domain evolves – even when that evolution appears insignificant. This paper presents a domain-specific visual language developed expressly for the evolution of domain-specific visual languages, and uses concepts from graph-rewriting to specify and carry out the transformation of the models built using the original DSVL.

1. Introduction

The concept of the domain-specific language has existed since the first computer languages were designed [1]. The expense and relative infancy of the computer required efficient code with a small footprint – thus many programs were fine-tuned in assembly code. When languages such as COBOL and FORTRAN emerged as a specific tools to build solutions to a particular type of problem (business, and mathematics, respectively), many programmers were skeptical that the assembly code generated from these abstract programs would be able to satisfy the speed and size requirements. However, the improved abstraction level of these languages provided outweighed the relative inefficiency of the generated code, since programs could be created and debugged much more efficiently.

Many years later, the idea of the domain-specific language remains nearly identical to that which emerged with the first languages: an improved abstraction of the problem allows for the rapid creation and maintenance of an application for a particular domain if the abstraction is sufficiently clear to a domain expert (i.e., someone who is familiar with the domain but not necessarily a programmer). The domain-specific language is then translated into some form more usable by the domain-specific runtime system, for example a configuration file, interface, or perhaps even code.

The domain-specific visual language (DSVL) is a special kind of domain-specific language that provides a visual programming interface. The advantages of a visual interface are plentiful, but in particular a DSVL allows someone who is a domain expert *and* also basically understands how to use a computer to use the visual language to create applications for the domain. As such, the DSVL is traditionally a restrictive language that consists only of domain concepts (an ontology), and also has well-formedness rules (constraints) that evaluate the domain models (or "programs") created in the DSVL. By restrictive, we mean that the unique ways that n types of visual objects may be associated with each other is limited to a relatively small number when compared to the total number of unique associations possible with n types, ($n(n-1)/2$).

The DSVL is restrictive for two main reasons. One, the smaller the possible constructs in the language, the more straightforward it is to learn the language and create a translator(s) that generates the domain artifacts. Two, the language uses the domain context and inherent semantics of domain primitives to define the semantics of language constructs. In this way, behavior that might be difficult or time consuming to express in a general purpose language could be concisely represented in one domain primitive. A restrictive language provides the user of the language with *only* those association and objects that are relevant to the domain, and permits their association *only* when that association has a defined semantic meaning.

As the early domain-specific languages evolved they retained much of their original syntax and semantics, and extended the ontology to enhance the language rather than modify the syntax or semantics of existing portions of the ontology. Consider languages such as COBOL and FORTRAN, which business and engineering professionals preferred for their relative ease in expressing problems in the business and mathematics domains (respectively). The domains did not change radically – the reasons for changes to the languages were driven by usage requirements such as a more convenient syntax, or additional features. Thus, the existing programs could (with a few exceptions for deprecated syntaxes) still work with the new language compilers.

Evolution of a DSVL is similar to a textual language when the ontology and semantics are extended rather than modified. Drawbacks to a DSVL with a restrictive syntax become apparent when the domain with which the DSVL is associated evolves. As opposed to "convenience" modifications which can retain the previous syntax and its semantics, domain-driven modifications can result in,

- changes to the semantics of an existing syntactical pattern,
- removal of a syntax pattern from the language, or
- replacement of a syntax pattern with another syntax pattern that will assume the existing semantics of the existing pattern.

Thus, the evolution of the DSVL immediately places into question the validity of any existing domain models (the "programs" of the DSVL). While many (or perhaps all) of the existing domain models may be syntactically correct, it is possible that their semantics in the new DSVL will not be what was intended when the models were originally built. This discrepancy between the original – or *intended* – meaning and the meaning after the DSVL is evolved is shown visually in Figure 1.

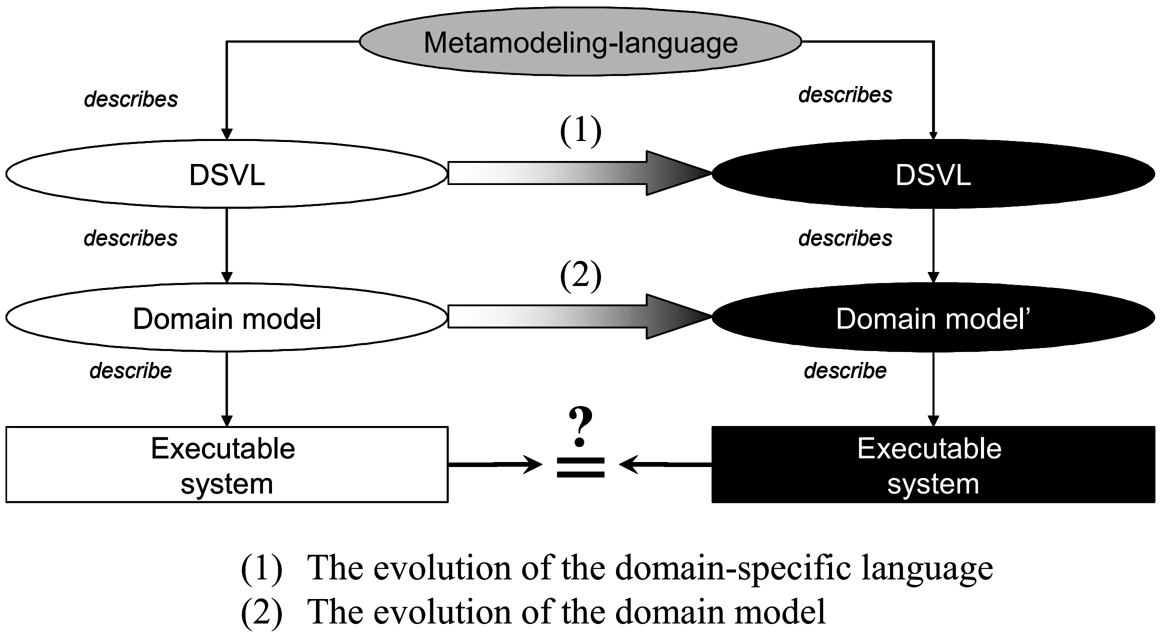


Figure 1. The meaning of the final executable system is in jeopardy when domain models are modified for use in the evolved DSVL.

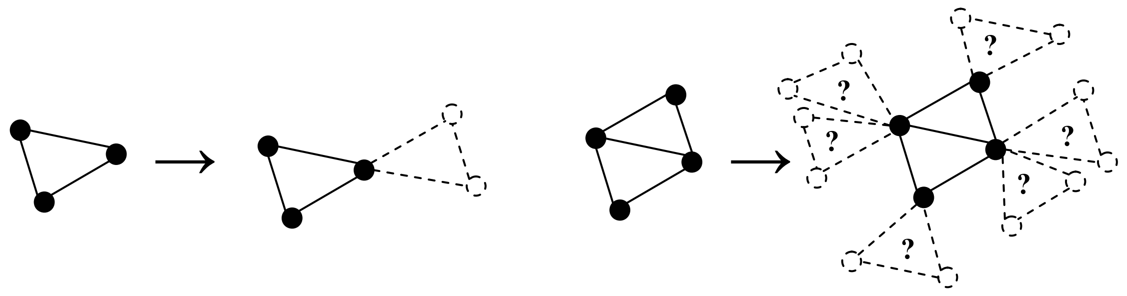
This paper describes a language that serves to mitigate the problems encountered when evolving a DSVL. Many of the techniques used to perform this evolution stem from research of graph transformations and graph rewriting. Some background on this research and some techniques used to perform graph transformations are provided in the following section. Once graph transformations have been introduced the formal definition of domain model evolution is defined. We then describe our domain evolution language in detail, including its model of computation, syntax structure, and fundamental properties. Once the language is described, we provide a canonical example for domain evolution that can be solved with our language. Finally, we present our conclusions.

2. Background

The models created using a DSVL – referred to throughout this paper as *domain models* – can be represented using a tree structure. The evolution of those models to conform to a new version of the DSVL can be seen as a rewriting of this tree to another tree as required by the new DSVL. For a better understanding of the basic theory of graph transformations, a brief review of the area is provided.

Graph rewriting is a useful tool for solving abstract problems that have a well-defined visualization method. Graph rewriting has been used to solve compiler theory problems [2][3], formalization of basic mathematical theory [4], not to mention graph theory and algorithm problems [5].

More recently, and to the subject of semantic mappings in high-level languages, graph rewriting (in the form of graph transformations) has been used as a sort of universal semantic description language that allows the specification for transformation from one syntax pattern to another. This approach appeared in [6] as a semantic extension for UML. Lately, graph grammars (the syntax used to express graph transformations) have been suggested as an implementation platform for the Model Driven Architecture (MDA) [7][8].



(a) (b)

Figure 2. (a) A pattern and its transformation rule. (b) The desired transformation is sometimes difficult to specify unambiguously.

Graph rewriting takes two forms: replacement – where sub-graphs are replaced with sub-graphs (and consequently, the rest of the graph remains unchanged), and recreation, where every node and line in the graph are created from scratch.

Unfortunately, rules defined to carry out graph transformations, while simply drawn up, can be quite difficult to implement. A simple rule, say to replace a triangle with a bowtie can be ambiguous in its desired implementation. Figure 2(a) shows a rule that might describe the rewriting of a graph element from triangle to bowtie. However, Figure 2(b) shows the six possible outcomes of the graph as presented. Should all of these outcomes be written? Only some of them? Only ones that meet certain criteria? A graph translation language should be specific enough that such ambiguities are not present upon translation.

Fortunately, much research exists in the area of graph transformation, and specifically in graph transformation languages. Well-formed arguments for the benefits of several types of graph rewriting approaches are found in [9][10][11][12]. There are also tools that claim to have devised a universal solution to the graph rewriting problem [13][14]. Most of these tools operate on a particular source graph (called the input graph) and produce an output (or target) graph. Nodes are traversed and matched based on syntax patterns entered into the graph-rewriting engine, and the output graph is produced based on the mapping between the syntax patterns and the desired output graph pattern.

In addition to such mature specifically graph oriented tools such as PROGRES [10][11][12], there are also tools that are not specifically graph oriented, but can perform rudimentary graph rewriting operations. One example tool is the Extensible Stylesheet Language (XSL) with its XSL Transformation script language [14].

3. Domain Evolution: Definition

Domain evolution as a problem is recognizable through the satisfaction of certain criteria. If any of these criteria are not met, then a solution such as the one presented in this paper is not necessary, as it will be needlessly complicated. It is important, therefore, to correctly identify the scope of the problem.

3.1. Scope

The following subsections define the five required aspects of a problem such that it is solvable through domain model evolution. If any of these portions is missing, then it is not practical to perform domain evolution with the DSVL in question; a low-level method should be sought.

3.1.1. Language syntax

There exists a well-defined syntax for the possible constructs in the DSVL.

3.1.2. Dynamic and static semantics

Without semantics, domain models are nothing more than illustrations. In order for system modeling to be effective there must exist a semantic definition that:

- ensures the correctness of domain models, and
- transforms the domain models into a semantic domain.

The correctness of domain models is governed by **static semantics**. These are essentially the well-formedness rules of the domain, although they may also be expressions of contextual differences in syntax (i.e., although certain model constructions may be permissible given the syntax, they are not allowed in certain contexts).

The style and form of the executable models is governed by the **dynamic semantics**, which are implemented in the DSVL's translator that performs the mapping from domain model (the language of the DSVL) to the semantic domain. The implementation of the dynamic semantics is known as the semantic mapping (see [16] for more information). Note that the executable form (i.e., domain artifact) need not necessarily be executable machine code, but can take the form of C++/Java files, SQL database constraints, XML schemas, etc. Basically, the models should be transformed into some artifact that is useful to the domain users.

3.1.3. Domain models

The domain models are produced using the DSVL, according to its language syntax. Since one of the advantages of a model-based approach to computing is that the models can be evolved throughout the life of a system (thus evolving the software that runs the system), it is advantageous that the domain models be available for modification by the DSVL.

However, any changes to the language syntax will change the DSVL. In this case, a new DSVL might not provide the correct interface to modify the domain models, thus preventing the evolution of those domain models to reflect the new system. It is important, then, to transform the domain models into some form that is usable by the new DSVL, for the purposes of model-reuse.

3.1.4. Updated language syntax

After a domain has been modified in some fundamental way, the metamodel that describes that domain must be changed to reflect this fundamental evolution. This results in the updated syntax.

3.1.5. Updated dynamic and static semantics

Because of the complexities of domain model storage, an updated language syntax means that domain models could be in danger of being invalid. By invalid, we mean that the existing types may no longer exist, or that the domain concepts may have changed since the original creation of the model, thus changing the semantics of the models when compared with the intent of the original modeler. Since the type of a domain model is encoded in it when it is stored, retrieving that domain model from storage could be difficult if its type no longer exists, or has been modified significantly.

The degree to which the language syntax was changed when it was updated is usually indicative of the difficulty in updating the domain models. Once the domain models are updated to reflect the new language syntax, then another obstacle emerges: the need to update the static semantics and translator.

Since the translator (the executor of the dynamic semantics) operates on domain models, it is usually defined to operate on types of models (i.e., through the metamodel). Any changes to the metamodel, therefore, require changes to the translator. If no changes are required for the translator, then the translator was not written correctly for the original language syntax.

This also holds true for the static semantics, with certain exceptions. Modifications to the metamodel may not require changes to the static semantics, if the modifications involved the addition or deletion of types that were/are not subject to context-specific syntax constraints. If any types subject to static semantics were added or deleted, then the static semantics of the DSVL should be updated to reflect the changes.

3.2. Semantics versus syntax

Syntax and semantics are quite often distinguished as two different, yet related, aspects of any language (visual and modeling languages included). While it is not difficult to convince someone that these two concepts are not identical, their relationship to each other during the model evolution solution is not quite as clear-cut. In order to explore this further, let us examine the meaning of a semantic versus a syntactic evolution. The following definitions and formalizations will apply,

Let δ be a domain

Let B be the boolean result set

$$B \equiv \{true, false\}$$

Let O be a set of object types (an ontology)

Let Y be a set of syntax rules between objects $o_1, o_2, \ldots o_n \in O$

Let S_δ be a set of semantic mapping functions to a domain, δ,

$$S \equiv \{ s(o) \mid o \in O \}$$

$$s \equiv \{ f(o) : O \rightarrow \Omega_\delta \}$$

$$\Omega_\delta \equiv \{ \delta\text{-object, undefined, false} \}$$

Where a δ-object is an artifact in the δ domain.

Let C_δ be a set of static semantics (constraints) for the domain, δ,

$$C \equiv \{ c(o) \mid o \in O \}$$

$$c \equiv \{ f(o) : O \rightarrow B \}$$

Let α be a paradigm, a quadruple of $<O,Y,C,S>$

Let α' be the new paradigm

Let M be a model database (set of domain models)

Let M' be M evolved to conform to a new paradigm

Let m be any original domain model contained in the set M

Let m' be m evolve to conform to a new paradigm, contained in the set M'

Syntactic Transform: transforms a model, m, in the model database M into an evolved model, m', such that m' is syntactically correct according to the δ syntax rules of a new paradigm

$$Tr_{SYN} \equiv \{ f(m) : M \rightarrow M' \mid \text{transform } m \text{ into } m' \}$$

(Definition 1)

Syntax Evaluator: evaluates a model, m, in the model database M against the set of syntax rules, Y, that exist within paradigm α,

$$Syn \equiv \{ f(m,\alpha) : M \rightarrow \{ B \} \}$$

(Definition 2)

Semantic Evaluator: executes the set of semantic functions for a model, m, in the model database M, according to the domain semantics, S_δ, of a paradigm α

$$Sem \equiv \{ f(m,\alpha) : M \rightarrow \Omega_\delta \mid s(m) \}$$

(Definition 3)

Static Semantic Evaluator: evaluates the static semantics of a model, m, in the model database M, according to the static semantics C, of the paradigm α,

$$Sem_{STAT} \equiv \{ f(m,\alpha) : M \rightarrow \{ B \} \}$$

(Definition 4)

Semantic Transform: transforms a model, m, in the model database M into a model, m', such that m' is correct according to the [1.] δ

semantics, 2. δ static semantics, and 3. δ syntax rules of a new paradigm

```
      $Tr_{SEM} \equiv \{ f(m) : M \rightarrow M' \mid$ transform m into m' $\}$
                                                         (Definition 5)
```

3.2.1. Syntactic evolution

This is a crude method for solving domain evolution. Basically, this method will modify the existing domain models sufficiently (either through deletion, or a type of search-replace algorithm) such that the models obey the syntactic rules of the new language. When this technique is used, we say that a *complete syntactic evolution* has taken place. One drawback to a syntactic evolution (i.e., without taking semantics into account) is that the new data models will not necessarily reflect the intended semantics of the old domain. In order to model the system correctly, a domain user must modify the new data models (in the new language) using her domain knowledge; in essence, manually modeling the system all over again. A formalization of syntactic evolution is as follows:

```
Syntactic Model Evolution: transforms a model database, M, into an
      evolved model database, M', such that M' is syntactically
      correct in a new paradigm, α',
          Preconditions:
                $\exists m \in M$ s.t.       Syn( m, α' ) == false
                and
                $\forall m \in M$            Syn( m, α ) == true

          $MM_{SYN} \equiv \{$ f(M,α,α') : M → M'  | $\forall m \in M$
                                              if Syn(m,α') == true
                                                    m' = m
                                              else
                                                    m' = $Tr_{SYN}$(m)
                                              M'.insert(m')
                 }


          Postconditions:
                $\forall$ m' ∈ M'          Syn( m', α' ) = true
                                                         (Definition 6)
```

The main advantage to this technique is that if the required modifications are type-based (or if the system is in development stage, and the old domain models were only for internal testing, and are not an important artifact), then it is not necessary to perform a complete semantic evolution. Just as a trade-off analysis is required to measure the benefits of creating a DSVL versus the time required to develop one, it pays to perform a similar analysis to justify the creation of a syntactic evolution – based on the amount of changes in each model, and the number of models that need to be changed.

3.2.2. Semantic evolution

This is the most sophisticated method for solving model evolution. Semantic evolution requires that the meaning of the old domain models is preserved after the transformation, *and* that the new domain models conform to the entire set of static constraints required in the new DSVL. When these two needs are satisfied, we say that a *complete semantic evolution* has taken place. A formal notation of this requirement is as follows (see the definitions in 3.2.1):

```
Semantic Model Evolution: transforms a model database, M, into an
      evolved model database, M', such that M' is syntactically
      correct, and semantically correct in a new paradigm, $\alpha'$,
          Preconditions:
                $\exists m \in M$ s.t.       Sem( m, $\alpha$ ) != Sem( m, $\alpha'$ )
                                or
                                Sem( m, $\alpha'$ ) is undefined
                                or
                                $Sem_{STAT}$( m, $\alpha'$ ) == false
                and
                $\forall m \in M$              Syn( m, $\alpha$ ) = true

          $MM_{SEM} \equiv$ { f(M,$\alpha$,$\alpha'$) : M $\rightarrow$ M' | $\forall$ m $\in$ M
                                            if $S_\delta$(m,$\alpha'$)==$S_\delta$(m,$\alpha$)
                                                  m' = m
                                            else
                                                  m' = $Tr_{SEM}$(m)
                                            M'.insert(m')
                 }
          Postconditions:
                $\forall$ m' $\in$ M'           Syn( m', $\alpha'$ ) == true
                                and
```

$\mathrm{Sem}_{STAT}(m', \alpha') == true$
(Definition 7)

Satisfaction of all of the requirements for semantic evolution is not a trivial task. The domain knowledge of the human programmer creating the evolution algorithm must be extensive, as the meaning of some syntax in the old domain models must be accurately transformed into the appropriate syntax in the new domain models. This is in stark contrast to a syntactic evolution, where out-dated syntaxes can be deleted or restructured, with no thought given to the former or future semantics of the domain models. By closely examining Definitions 6 and 7, several things are apparent:

1. the postcondition for syntactic evolution is a postcondition for semantic evolution (i.e., semantic evolution occurs only if syntactic evolution occurs),
2. semantic evolution has additional constraints that speak of semantics of domain models,
3. syntactic evolution does not mention semantics at all,
4. no matching of semantics occurs in the postconditions of semantic evolution,
5. syntactic evolution includes preconditions not found for semantic evolution (namely, syntactic mismatch).

Out of these four, the last two items are the most interesting. The first of these – semantics between the domains do not necessarily match – seems somewhat counter-intuitive. After all, why would one perform a semantic model evolution if the semantics were not to be preserved? The answer is that it is not always necessary (or even desirable) to preserve the same semantics, but the resulting semantics should always be *correct* in the new domain. Thinking back to the reasons for domain environment evolution, they revolve around evolution of the DSVL versus evolution of the domain. When the DSVL evolves separately from the domain it is generally preferred to preserve exact semantics of the output models *m'*. However, when the domain evolves along with the DSVL, then the semantics of the output models should match the new domain, and should be preserved only as much as the TR_{SEM} deems appropriate.

For the last point, the fact that a syntactic mismatch need not be in place for a semantic evolution to take place is, perhaps, perplexing at first. More thought, however, reveals that just because data types match does not mean that data semantics do. Consider, for instance, the well-publicized mistake in the Mars spacecraft that failed based on misunderstanding of the units of given data (meters versus feet). The semantics of the data were important, but there was not a problem in the syntax (e.g., reading a floating point value).

Since so many factors must be considered, and a great number of syntax patterns might need to be examined, semantic evolution requires a more significant time and training investment to carry through correctly. However, for significant amounts of data models, the reward of developing a transform of old data into useful new data might be worth the cost of creating the semantic evolution algorithm.

3.2.3. The link between syntax and semantics in evolution

Although syntax and semantics are frequently distanced both in definition and discussion, their cohesion in the domain evolution process is tighter than in most other software applications, because the syntax and semantics are inputs to the domain evolution process. The reason for this link is that a semantic evolution cannot take place unless a syntactic evolution has occurred.

This claim is best justified by a consideration of its negation (i.e., it is possible to perform a complete semantic evolution without a complete syntactic evolution). If this is true, then there could exist a domain model that, once evolved, is a semantically accurate model in the domain. However, if syntactic evolution has not occurred, then there would exist syntactic problems that would prevent the new domain model from being valid in the new DSVL. Thus, it is not possible to perform a semantic evolution without performing a complete syntactic evolution (i.e., a semantic evolution subsumes a syntactic evolution).

The link between syntax and semantics is further strengthened by the following claim: that if a complete semantic evolution is performed, then the process by which the semantic evolution is specified must be defined as a syntactic evolution. While this might seem extraordinary, consider how the semantics of a language are defined – in terms of the syntax of that language. Since the old and new DSVLs are two languages, then the domain model evolution may be cast as a pattern-based transformation, where syntactical patterns in the old language are transformed into a semantically accurate new syntax.

4. The Domain Evolution Language

The domain evolution language is a domain-specific visual language for the domain of domain evolution. The *domain evolution* domain requires that the preconditions found in Definitions 6 or 7 are met, and that there exists a DSVL that meets the criteria set forth in Section 3.1. The overall use of this language, including its basic syntax and semantics, model of computation, and the objective of its interface are described in the following sub-sections.

4.1.   Objective

The objective of the domain evolution language is very simply to provide an interface that is specialized for describing an algorithm to transform domain models from one DSVL to another. Once the user understands the process of evolving the domain models from the DSVL to its evolved self (i.e., the algorithm for domain evolution) then the user should be able to enter that algorithm with a language friendly to the concepts of domain model evolution. The way this is achieved in the language is to provide the user with the ability to create mappings from pieces of the existing DSVL's metamodel (syntax patterns) to the evolved metamodel. The advantages to this approach are that,

- knowledge of the tools is more important than knowledge of the transformation language,
- mappings are described with elements directly from the domain,
- the solution is metamodel driven, and
- abstracts the format of the data storage from the problem.

The intention, then, is that if a user is familiar with metamodeling concepts, and understands the algorithm to transform from one DSVL to its evolved self, then with very little guidance that user can create a domain evolution transformation that will evolve the domain models for user in the evolved DSVL.

4.2. Ontology

The domain evolution language allows the domain-specific abstraction of algorithms for the translation of one graph to another. However, a precept of the language is that the two DSVLs that are the focus of the evolution are more similar than they are alike. Therefore the language itself is geared toward describing how different things are equivalent, or how things should be removed or created in certain circumstances. The overall definition for the evolution is created in a Transformation (as shown in Figure 3).

A Transformation contains an OldClassDiagram and a NewClassDiagram (the existing and evolved DSVL metamodels), which serve as the libraries from which patterns are extracted to define the semantic mapping. Most of the ClassDiagrams used during domain evolution are in UML form, and this is currently supported in our domain evolution language. However, the language is extensible to other metamodeling approaches. As the Transformation is an abstraction of an algorithm, it must also contain a sequence of actions, as well as flow control; these are found in the set of sequenced Transforms contained in the Transformation.

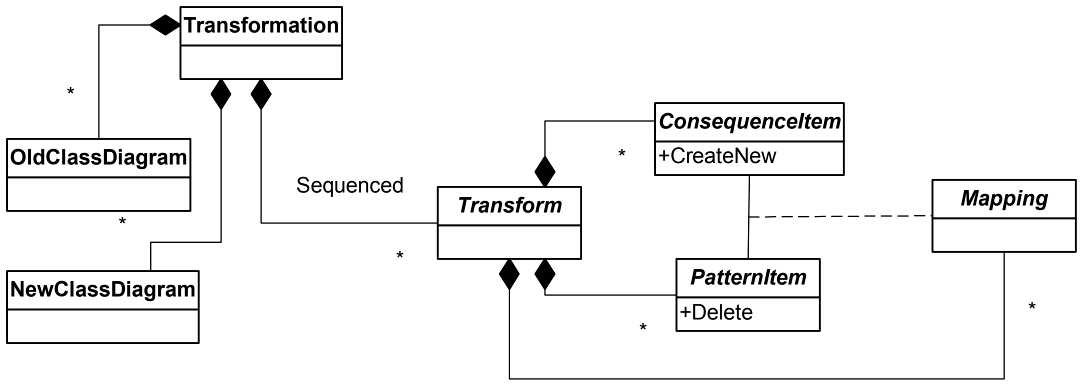


Figure 3. Simplified metamodel of the domain evolution language.

Each Transform is made up of PatternItems (items from the existing metamodel) and ConsequenceItems (items in the evolved DSVL metamodel). The PatternItems specify a syntax pattern that – if matched – would be transformed according to the Mapping between the Pattern and Consequence items.

4.3. Semantics

The meaning of the sequence of Transforms and of the mapping between Patterns and Consequences is required before domain evolution algorithms can be described. The next section (4.4) describes the evaluation of the Transforms, while the semantics of the patterns and mappings is given in the current section.

4.3.1. Describing patterns

The most complex – and powerful – portion of the language is the constructs that describe the patterns and mappings of the DSVL evolution. Based on the syntax of a metamodeling language, the patterns are created as object diagrams using as objects the classes in the metamodel. An example construct is shown in Figure 4.

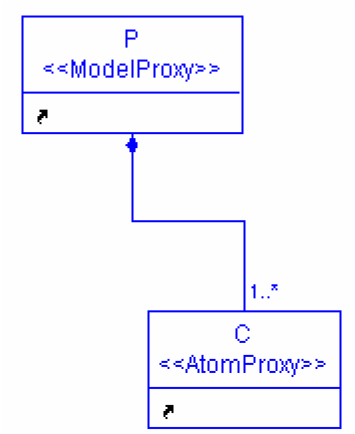


Figure 4. A pattern specification within a Transform. Patterns from the existing DSVL metamodel are colored blue, whereas Consequences (from the evolved DSVL) are shown in red.

This particular pattern describes a parent-child relationship between the parent P (of stereotype ModelProxy) and the child C (of stereotype AtomProxy). As a sidenote, all objects used to describe Patterns or Consequences will have the stereotype [Type]Proxy, as they are proxy representations of the actual objects in the metamodel; the Type of the object in the metamodel is what is used to create the pattern meaning. This particular pattern can be semantically interpreted in two equivalent ways, as shown below.

```
1. There exists an P:Model with 1..* childen C:Atom
2. There exists a C:Atom with parent P:Model
```

It should be noted that the containment connection between the parent and child is navigable in the domain evolution language. While not all DSVLs support this, the domain evolution language operates regardless of these concerns, because of its ability to use the domain models as data, rather than domain representations, and it operates outside the normal bounds of the created objects.

4.3.2. Describing mappings

The mappings, which provide the low-level transform from one type to another, follow a "*pattern implies consequence*" form. The mappings are associations between patterns and consequences, or attributes of a pattern or consequence, and are formed from a fundamental set of operations, in similar fashion to the fundamental set of string replacement operations (i.e., insert, concatenate, delete, replace). The determination of the fundamental set of replacement operators for models is a significant problem in its own right. Textual replacement has only one "association" – position – while object replacement in a graph can have as many associations as are permitted for that type of object by the metamodel. For the purposes of this paper, that set of operators is limited to `Create`, `CreateWithin`, `Becomes`, and `Delete` (analogous to insert, concatenate, replace and delete for textual manipulation).

Create – a unary association of a Consequence object. This specifies that if the Pattern is matched, then this object is created. There must exist a `CreateWithin` association to determine the parent of this object.

CreateWithin – a binary association between a Consequence object (child) and another object (parent), which details the parent of the Consequence object that is to be created. The parent object need not be a Consequence object. It is an error to specify a circular dependency with CreateWithin associations between Consequence objects.

Becomes – a binary association between a Pattern object (source) and a Consequence object (destination). The semantics is that the source will be removed from the tree, and replaced with the destination. Neither the source or destination objects may participate in a Create or Delete association.

Delete – a unary association of a Pattern object. If this object is matched, it will be deleted. Any children of this object will not necessarily be deleted, but will be promoted up one level of ancestry in the tree unless specified otherwise.

4.4. Model of Computation

The model of computation provides an exact knowledge of the behavior of a Transformation to the user.

4.4.1. Sequences of Transforms

The domain evolution algorithms must be deterministic, and therefore the model of computation for sequenced Transforms operates on the input domain models with the first Transform and continues on through until there are no further Transforms to apply. After each Transform, a complete set of domain models exists, and is passed on to the next Transform. Thus, the exact semantics of the Transform sequence shown in Figure 5 is,

```
Let a Transform be a function, T(d), such that, d' = T(d)
Let A be a Transform
Let B be a Transform
Let M be the input, then

     $M'_1$ = A(M)
     $M'_2$ = B($M'_1$)
     M' = $M'_2$
          Or, more simply,
     M' = B(A(M))
```

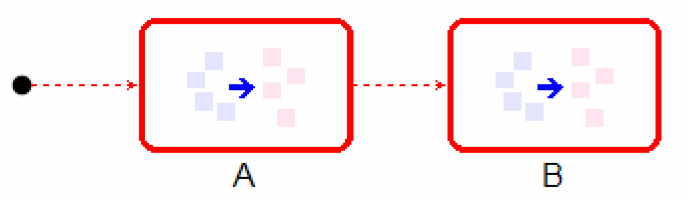


Figure 5. A sequence of Transforms, A and B, with an initial transition.

4.4.2. Matched Patterns

The mapping from one object to another operates *en masse* rather than hierarchically. That is, the morphing of the objects from Pattern to Consequence may be considered an atomic action that occurs simultaneously on all patterns in the file. Thus, matches made that result in the deletion of an object will not affect matches where that object is used for context, and not deleted. If any features that require the state of the input objects to change are needed, then Transform sequences should be used to guarantee it.

5. Example: Port Specialization

A canonical example of the requirement for domain evolution is found in the specialization of a domain concept into two or more derived concepts. For example, consider the two DSVL metamodels shown in Figure 6. In the domain of signal processing, there are individual signal processors that may be joined together through their Ports via Connection associations. However, the domain language evolved by requiring that there be exactly two different types of Port, Input and Output, and furthermore, that Input and Output ports may be connected only in accordance to certain static semantics, namely,

- `Any Port may not be the source of a connection where the destination of the connection is itself (previously existing constraint)`
- `An Input may not be the source of any connection in which the destination of that connection is an Output`
- `An Output may be the destination of any connection in which the source is an Input`

- `An Input may be the source of a connection in which the destination is an Input provided that the parent ProcessSignal of the destination is contained by the parent ProcessSignal of the source`
- `An Output may be the destination of a connection in which the source is an Output provided that the parent ProcessSignal of the source is contained by the parent ProcessSignal of the destination`

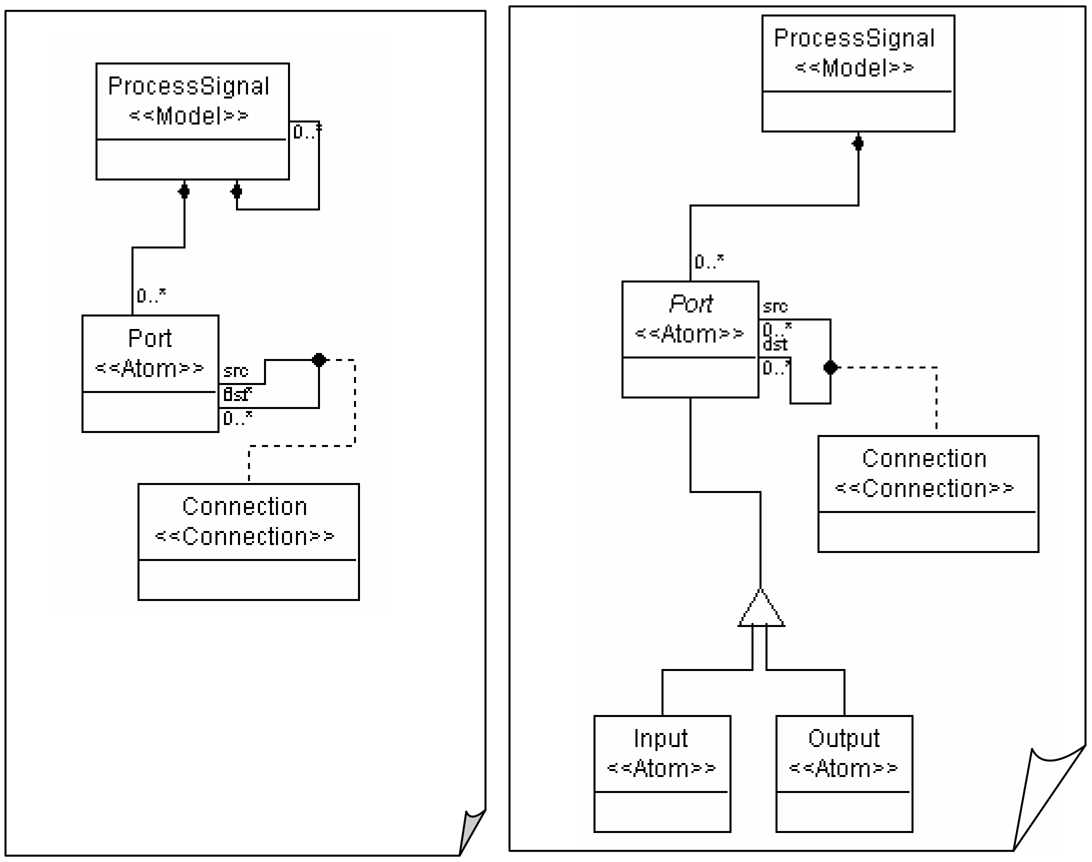


Figure 6. The original DSVL metamodel (left) and the evolved DSVL metamodel (right).

Simply said, Input and Output ports may be used to pass along the input or output signals of parent or child processes, but should never connect the same type if the ProcessSignals exist on the same level of hierarchy.

5.1. Algorithm

In this case, we wish to maintain the exact semantics of the domain models when they have been evolved, so we assume that the original modeler intended that the Port objects behave as if they were the Input or Output ports in the evolved DSVL. Given the existing constraints, and the evolved constraints, it is possible to write an algorithm to convert any input domain model into its equivalent evolved domain model, while maintaining the intention of the original modeler. The algorithm is as follows:

1. Transform all Ports that will become Input ports connected to other Input ports
2. Transform all Ports that will become Output ports connected to other Output ports
3. Transform all remaining Ports that will become Input ports
4. Transform all remaining Ports that will become Output ports

This algorithm is fairly simple, but must be divided into at least two portions, based on the fact that any port may be either an Input or an Output if it is the source of a connection, due to the ability of ports to pass along signals from ProcessSignal models on another level of hierarchy. Since this sort of control flow exists, the algorithm is divided into four sequenced Transforms, as shown in Figure 7.

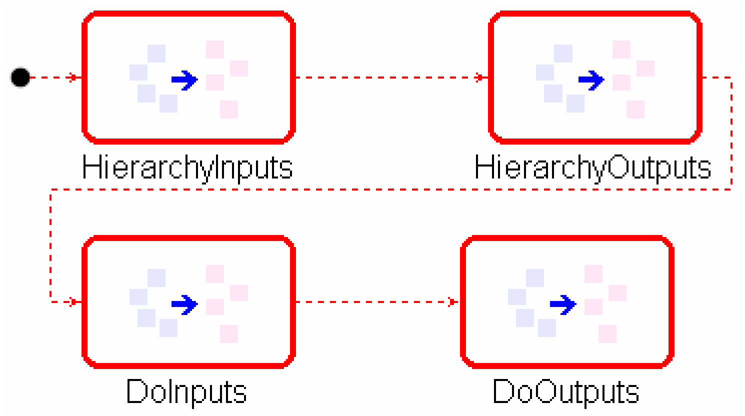


Figure 7. The sequence of Transforms to evolve the domain models.

The domain evolution language provides interfaces to generate any type of graph rewriting language that is convenient to the end user. In this case, the domain models are all stored in an XML format, which tailors nicely to the generation of XSL stylesheets to perform the evolution. Each Transform in the sequence above will generate an XSL document, which will be called in sequential order on the output artifact of the previous

Transform output, until the final document – suitable for use in the new DSVL – is produced.

5.2. Transforms and Output XSL

Only two Transform contents need be shown, since the other two will be the dual (i.e., for Outputs instead of Inputs). The first of these is the contents of the HierarchyInputs Transform, which is shown in Figure 8. This represents the algorithm step number 1. Notice that the parent of the source Port is the grandparent of the destination Port. The output XSL is given in Figure 9.

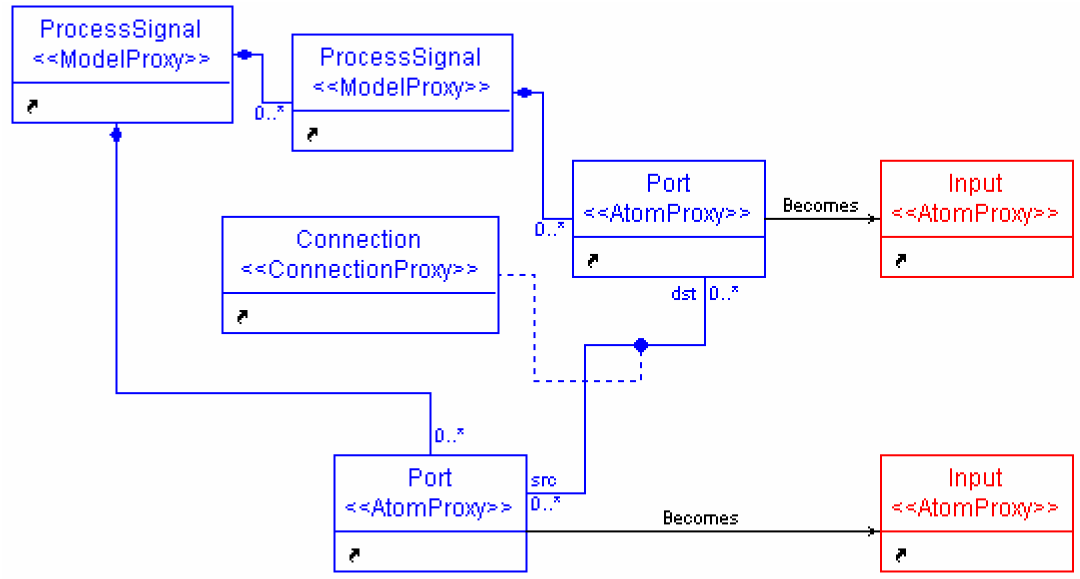


Figure 8. Contents of the HierarchyInputs Transform.

```
<xsl:template match="atom[@kind='Port']">
    <xsl:variable name="Becomes62081992_focus_Port62075208" select="current()"/>
    <xsl:variable name="Becomes62081992_ProcessSignal62073296"
        select="current()/parent::model[@kind='ProcessSignal']"/>
    <xsl:variable name="Becomes62081992_Port62074080"
        select="current()/parent::model[@kind='ProcessSignal']/child::model[@kind='ProcessSignal']/child::ato
        m[@kind='Port']"/>
    <xsl:variable name="Becomes62081432_Port62075208"
        select="current()/parent::model[@kind='ProcessSignal']/parent::model[@kind='ProcessSignal']/child::at
        om[@kind='Port']"/>
    <xsl:variable name="Becomes62081432_ProcessSignal62073296"
        select="current()/parent::model[@kind='ProcessSignal']/parent::model[@kind='ProcessSignal']"/>
    <xsl:variable name="Becomes62081432_ProcessSignal62075856"
        select="current()/parent::model[@kind='ProcessSignal']"/>
    <xsl:variable name="Becomes62081992_ProcessSignal62075856"
        select="current()/parent::model[@kind='ProcessSignal']/child::model[@kind='ProcessSignal']"/>
    <xsl:variable name="Becomes62081432_focus_Port62074080" select="current()"/>
```

```
        <xsl:choose>
            <xsl:when
                    test="current()[@id=//connection[child::connpoint[@role='dst']/@target=$Becomes62081992_P
                    ort62074080/@id]/child::connpoint[@role='src']/@target][$Becomes62081992_focus_Port6207
                    5208][$Becomes62081992_ProcessSignal62073296][$Becomes62081992_ProcessSignal6207
                    5856][$Becomes62081992_Port62074080]">
                <xsl:call-template name="PortBecomesInput_62081992"/>
            </xsl:when>
                    <xsl:when
                    test="current()[@id=//connection[child::connpoint[@role='src']/@target=$Becomes62081432_P
                    ort62075208/@id]/child::connpoint[@role='dst']/@target][$Becomes62081432_focus_Port6207
                    4080][$Becomes62081432_ProcessSignal62075856][$Becomes62081432_ProcessSignal6207
                    3296][$Becomes62081432_Port62075208]">
                <xsl:call-template name="PortBecomesInput_62081432"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:copy>
                    <xsl:apply-templates select="@*|node()"/>
                </xsl:copy>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
    <xsl:template name="PortBecomesInput_62081432">
        <xsl:element name="atom">
            <xsl:apply-templates select="@*"/>
            <xsl:attribute name="kind">Input</xsl:attribute>
            <xsl:attribute name="role">Input</xsl:attribute>
            <xsl:comment>transformed using template 'PortBecomesInput_62081432'</xsl:comment>
            <xsl:apply-templates select="node()"/>
        </xsl:element>
    </xsl:template>
    <xsl:template name="PortBecomesInput_62081992">
        <xsl:element name="atom">
            <xsl:apply-templates select="@*"/>
            <xsl:attribute name="kind">Input</xsl:attribute>
            <xsl:attribute name="role">Input</xsl:attribute>
            <xsl:comment>transformed using template 'PortBecomesInput_62081992'</xsl:comment>
            <xsl:apply-templates select="node()"/>
        </xsl:element>
    </xsl:template>
```

Figure 9. XSL Output for the HierarchyInputs Transform.

Each Pattern object is given its own context in relation to the objects being transformed through the Becomes association, and this context is shown in the xsl:variable that is created for each member of the Pattern context. When each node is parsed in the input file, it is passed through the set of filters to determine whether it is matched. If so, it is passed on to become an Input; if not, it is copied as-is.

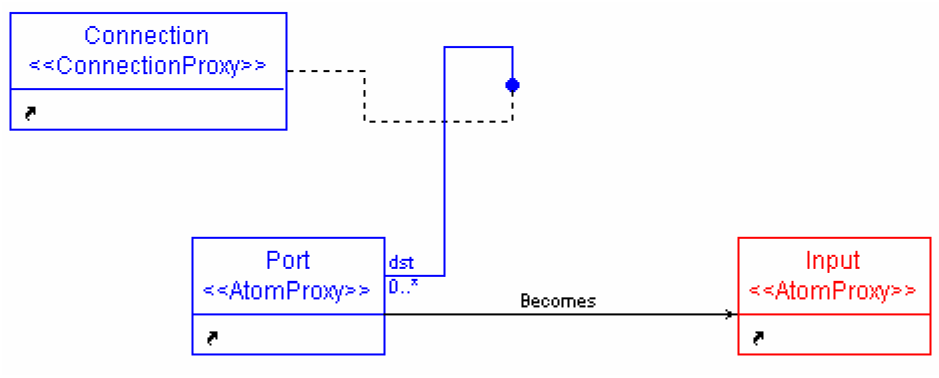
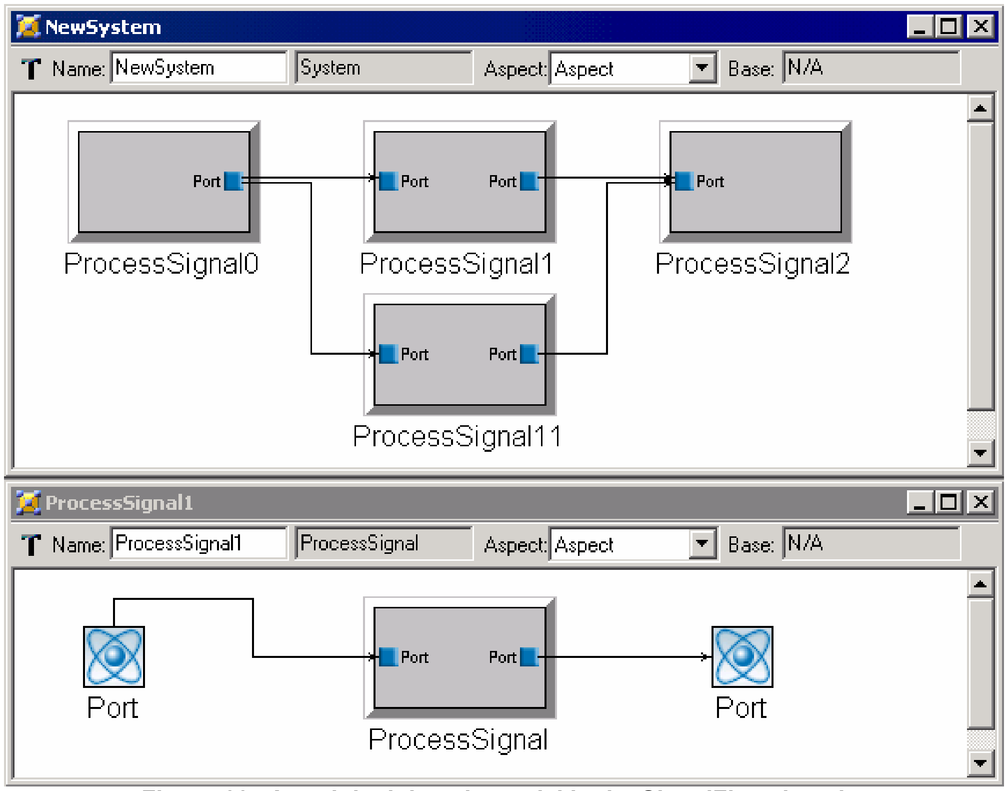


Figure 10. Contents of the DoInputs Transform.

The other Transform contents that bear noting are those of the DoInputs Transform, shown in Figure 10. In this portion of the algorithm (step 3) the destination of the Connection and the containment context of the Port are irrelevant: only the type of the destination matters. Therefore, no further context is required for the Transform pattern. The XSL output for the DoInputs Transform is much simpler, but omitted for brevity.

5.3. Domain Models – Before and After

Consider the set of models in Figure 11. The upper portion of the figure shows the high level of the models, and the lower portion is the contents of ProcessSignal1, which contains another level of ProcessSignal. Note that the contents of ProcessSignal1 satisfy the hierarchy portion of the algorithm, where Input and Output ports may be connected to others of the same type. Also, note that the name “Port” on the objects is an instance name, not a type name (type in this domain is denoted by icon).

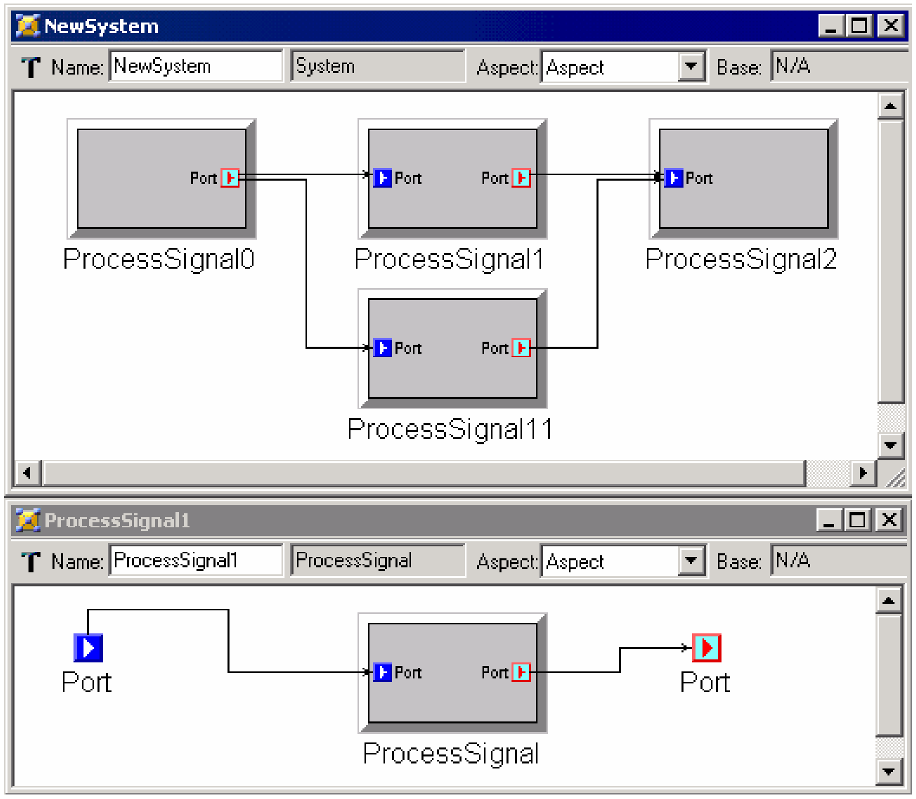

NewSystem

Name: NewSystem | System | Aspect: Aspect | Base: N/A

ProcessSignal1

Name: ProcessSignal1 | ProcessSignal | Aspect: Aspect | Base: N/A

Figure 11. An original domain model in the SignalFlow domain.


NewSystem

Name: NewSystem | System | Aspect: Aspect | Base: N/A

ProcessSignal1

Name: ProcessSignal1 | ProcessSignal | Aspect: Aspect | Base: N/A

Figure 12. The evolved domain model in the evolved SignalFlow domain.

After processing the models in Figure 11 with the domain evolution algorithm created in the earlier part of the section, the output models are ready for use in the evolved domain. Figure 12 shows the evolved domain models. Once again, it is important to note that the icon with blue filling and white arrow represents Input type, and white with red arrow and border represents Output type (contains a border). The example evolved perfectly, and satisfies all of the syntactic and semantic constraints of the evolved domain.

5.4. Other Approaches

Most of the other approaches to domain evolution are general purpose graph rewriting tools such as [6][8][13]. Note that all of these approaches provide a particular method to transform from one domain to another. However, this approach is more appropriate in several ways. First, it is aimed at transforming between domains that are nearly identical. Thus, no 1-1 mapping is required, as in [6]. Second, the interface is focused on the metamodel concepts rather than abstractions that are mapped by the domain evolution modeler onto domain concepts, as in [13]. Once again, the domain-specific approach of the solution presented in this paper decreases the amount of domain-independent specification required to do the evolution.

6. Conclusions

The problem of domain evolution is encountered whenever changes are made to any domain specific language – including visual languages. Given that languages are easier to create – and thus modify – it is important to provide a language designed specifically to aid in the task of the evolution of models from one DSVL to its evolved self. The language presented in this paper provides such an interface, and is convenient to use from the perspective of a metamodeler, while powerful enough to create syntax patterns in any form, since the pattern language is derived from the metamodeling language. Furthermore, it is important to note that the language does not provide any magic for creating the transformation algorithm, but instead provides an interface that is most like the algorithms created in such an instance.

7. References

[1] S. Lohr, *The Programmers who Created the Software Revolution – Go To*, Basic Books, 2001.

[2] S. Johnson, "A Portable Compiler: Theory and Practice", *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pp. 97-104, January 1978.

[3] M. Ganapathi, C. Fischer, J. Hennessy, "Retargetable Compiler Code Generation", *ACM Computing Surveys*, Vol. 14, No. 4, pp. 573-592, December 1982.

[4] F. W. Lawvere, S. Schanuel, *Conceptual Mathematics: A First Introduction to Categories*, Cambridge University Press, Cambridge, UK, 1997.

[5] D. West, *Introduction to Graph Theory*, 2nd edition. pp. 289-90. Prentice Hall, Upper Saddle River, NJ. 2001.

[6] D. H. Akehurst, "Model Translation: A UML-based specification technique and active implementation approach". Ph. D. Thesis. University of Kent at Canterbury, United Kingdom. December 2000.

[7] J. Bézivin, N. Ploquin, "Tooling the MDA framework: a new software maintenance and evolution scheme proposal", *OOPSLA, 2001: Workshop on Engineering Complex Object-Oriented Systems for Evolution*, Tampa Bay, FL, 2001.

[8] A. Agrawal, T. Levendovszky, J. Sprinkle, F. Shi, G. Karsai, "Generative Programming via Graph Transformations in the Model-Driven Architecture", *OOPSLA - Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November 2002.

[9] D. Blostein, A. Schürr, "Computing with Graphs and Graph Rewriting", *Software – Practice and Experience, 6th Proceedings in Informatics*, pp. 1-21, 1997.

[10] A. Schürr, "Specification of Graph Translators with Triple Graph Grammars". University of Aachen, AIB 94-12.

[11] A. Schürr, "Adding Graph Transformation Concepts to UML's Constraint Language OCL", *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2000.

[12] G. Engels, R. Heckel, "From Trees to Graphs: Defining the Semantics of Diagram Languages with Graph Transformation", *ICALP Satellite Workshops, Proceedings in Informatics*, pp. 373-382, 2000.

[13] A. Schürr, "Programmed Graph Replacement Systems". In *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*, pp. 479-546. World Scientific, Singapore, 1997.

[14] The Extensible Stylesheet Language, http://www.w3.org/Style/XSL/.

[15] "Turing Machine Markup Language", http://www.xml.com/pub/r/1036

[16] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics, and All That Stuff. Part I: The Basic Stuff", Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Rehovot, Israel.