2009 14th IEEE International Conference on Engineering of Complex Computer Systems

The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering

Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, Fiona A.C. Polack
Dept. of Computer Science
University of York
Heslington, York, United Kingdom
{paige, dkolovos, louis, nikos, fiona}@cs.york.ac.uk

***Abstract*—Model management is the discipline of managing artefacts used in Model-Driven Engineering (MDE). A model management framework defines and implements the *operations* (such as transformation or code generation) required to manipulate MDE artefacts. Modern approaches to model management generally implement these operations via domain-specific languages (DSLs). This paper presents and compares the principles behind three approaches to implementing DSLs for model management and identifies some of the key differences between DSL engineering in general and for model management. It then shows how theory relates to practice by illustrating how DSL design and implementation approaches have been used in practice to build working languages from the Epsilon model management framework. A set of questions for guiding the development of new model management DSLs is summarised, and data on development costs for the different approaches is presented.**

I. Introduction

Model management frameworks provide languages and tools for managing the artefacts associated with Model-Driven Engineering. These frameworks ultimately provide the infrastructure necessary to implement task-specific operations, including model-to-model (M2M) transformation, model-to-text (M2T) transformation, model composition, model validation and consistency checking, refactoring, comparison, and general queries.

Several model management frameworks, languages and tools are now available. Perhaps most well-known is AMMA [2] which provides facilities to transform, weave, and generate text from models. Other well-known frameworks include Epsilon [22], KerMeta [44] and oAW [36]. There also exist languages and tools that support individual model management tasks, such as MOFscript [35] (for M2T transformation), Tefkat [30] (for M2M transformation), EMF Compare [13] (for model comparison). Generally, model management frameworks consist of one or more concrete *domain-specific languages (DSLs)* and supporting tools, e.g., syntax and type-checkers and execution engines, each of which provide the concepts and logic specific to tackling a model management task.

DSLs for model management have been designed and implemented in a number of ways; some have been built from scratch (e.g., MOFscript for M2T) to address a very specific problem, whereas others have been built atop or from other DSLs, exploiting existing features. This paper reports on our experience on the conceptual design and practical implementation of a large number of DSLs for model management, and attempts to distill lessons learned, particularly in relation to existing practice of general DSL development.

A. Contribution

The contribution of this paper is an analysis of technical approaches to designing and implementing model management DSLs, as well as guidance on how to select an approach for building a new model management DSL. This guidance is based on our substantial experience in developing a conceptual framework and technical infrastructure for model management DSLs in the Epsilon platform. As a result, we distill lessons learned from this experience. We systematically present three related approaches for implementing model management DSLs: by preprocessor, by language extension, and by language annotation. All three approaches are illustrated concretely with examples of DSLs from *Epsilon*, a model management platform that (i) provides a number of model management DSLs, and (ii) that has been designed to support the development of new model management DSLs. From this presentation we synthesise a set of simple questions that can guide engineers in choosing a justifiable approach for building a new model management DSL. We supplement these questions with empirical data that can help clarify the cost-flexibility tradeoff argument that is part of choosing a DSL design and implementation approach.

II. Related Work

This section reviews work related to the design and implementation of DSLs. The development of a DSL typically involves the following steps [1]:

- **Analysis:** the problem domain is identified, the relevant domain knowledge is gathered and an appropriate DSL is designed.
- **Implementation:** after the DSL is designed, the most suitable implementation approach should be chosen.

978-0-7695-3702-3/09 $25.00 © 2009 IEEE
DOI 10.1109/ICECCS.2009.14

- **Use:** programs are written in the new DSL, and if necessary feedback on the design and implementation are provided.

In this paper, we focus on the implementation phase of DSL development, and the approaches we can use therein. There are many implementation techniques for DSLs; some apply to the development of General Purpose Languages (GPL) as well, while others have no useful counterpart for GPLs. The purpose of this section is to discuss the main approaches to DSL implementation, as well as to present representative examples of their use.

A. Compiler/Interpreter Approaches

The direct approach to DSL (and GPL) implementation is the development of a compiler or interpreter for the DSL under consideration. There is a wide variety of tools and frameworks for the development of compilers and interpreters such as [1], [29]. This approach is used by many well known languages such as Ruby [6] or C. Building a compiler/interpreter has some clear advantages. The main advantage is that the implementation is completely tailored towards the DSL. In addition, error detection, static analysis, and optimizations can be performed at the domain level. The main disadvantage of this approach is the high cost of implementation, since the compiler or interpreter is built from scratch. Taking under consideration that DSL development is usually part of a larger project and that the development costs should be kept as modest as possible [7], more efficient implementation methods can possibly be used.

B. Embedded Languages

Another common approach to DSL implementation is the *embedding* of DSL constructs into a host GPL language. The term *Domain Specific Embedded Language* was coined by Hudak [37]. Embedding usually takes place in the form of a language library [4], while a more domain specific notation can be developed by defining new data types, operators and other constructs using the base language. The embedding of the DSL into the GPL is achieved by mapping the DSL code to library code, which provides the desired functionality. Embedding can also take place by using an *Attribute Grammar*, where the focus is on how distinct attributes can be realized in a modular way [11]. The main advantage of the embedding approach is that it can be less costly than compiler-based approaches, since the compiler/interpreter of the host language is reused. MetaBorg [32] is an approach that provides generic facilities and techniques for embedding languages. Representative examples of embedded DSLs include FPIC [41], which is a picture drawing language embedded in ML [39], Frob, a robot control language, and Fran, a DSL for reactive animations [19].

C. Extension/Specialization of Languages

Language *extension* works by modifying the compiler or interpreter of a language to work with domain specific constructs. In [17], the *delegating compiler object* approach is proposed for constructing extensible compilers. As a result, a large compiler can be broken into smaller compilers that compile parts of the input specification. Bosch [17] uses this approach to develop the layered object model, an object oriented language that can be extended with new constructs. Multiple language inheritance is another technique for achieving language extension [33]. Another approach is offered by MontiCore [34], wherein new languages can be designed by extending existing ones and by composing language fragments to new DSLs. A DSL built in this way is Java-SWUL [42], which supports the development of Java Swing GUIs and extends Java. A special case of extension is *annotation*, for example, as is done by using stereotypes in UML; annotations are also used in DSLs, e.g., in the EuGENia toolset [8] that uses annotations to capture how models can be visually represented in GMF. A further special case of extension is *piggybacking*, in which the extension of a DSL is described by extracts of other languages. This is used in *Facile* language, a language for developing high performance processor simulations that augments C. Finally, a GPL can be reduced to fit the needs of a special domain. OWL-Lite [45], a subset of OWL, is implemented using this kind of specialization.

D. Preprocessor Approaches

In *preprocessing* approaches, the various DSL constructs are translated into constructs of a base language before compilation or interpretation takes place. While the simplicity of this approach makes it very attractive, static analysis and optimization is not performed at the domain level but is limited to that performed by the base language compiler or interpreter. Additionally, the user receives feedback on errors at the level of the base language. Under the preprocessing strategy, different approaches can be identified. *Macro preprocessing* [3] is the most common. In this approach, the meaning of new constructs is defined in terms of other constructs in the base language in the form of macros, e.g., as in C. Another approach is *source-to-source transformation*, where DSL source code is translated into a base language, e.g., using C++'s template facility. Bliz++ [43], a mathematical library for C++, was developed with this method. In the *lexical scanning* approach, only simple scanning is required, without complicated tree-based syntax analysis. Finally, the extensible compiler or interpreter approach integrates preprocessing in the compiler or in the interpreter. As a result, more type checking and better optimization is possible. An example of such an interpreter is the Tcl interpreter [18].

E. Model-Driven Approaches

MDE tools are posited to be a convenient approach to DSL implementation. In the context of MDE, a DSL is a set of coordinated models built in a modular way. A DSL can be described by the following models [16]:

- **Domain Definition Metamodel (DDMM):** a specification of the domains concepts, and defines the abstract syntax for a DSL.

- **Concrete Syntax:** the different concrete syntaxes of a DSL are defined by a transformation model that maps the DDMM onto a "display surface" metamodel.
- **Semantics:** the semantics definition of a DSL is also defined by a transformation model that maps the DDMM onto another DSL with a precise execution semantics (or even to a GPL).

Some specific MDE approaches to DSL implementation are described in the following.

1) Atlas Model Management Architecture: One of the most well-known approaches in the model-based DSL implementation is the Atlas Model Management Architecture (AMMA) [2] platform. It is a model engineering platform that can support the development of DSLs. AMMA consists of a set of DSLs, including KM3 [12], Textual Concrete Syntax (TCS) [20] and ATL [14] for the manipulation of the various models, and it offers the capability to build sets of new DSLs for a given domain or for a family of systems.

2) Open ArchitectureWare: Another well known model-driven DSL implementation approach is the openArchitectureWare platform (oAW) [36]. It consists of a set of tools that can be used for specifying the various aspects of a DSL. For example, the *Xpand* language is a statically typed template language for M2T transformations while *XText* enables text-to-model transformations. These are used to define the syntax of a DSL.

3) Other Model-Based DSL Approaches: KerMeta [44] is another framework for implementing DSLs. The metametamodel used is derived from EMOF and the platform contains an action language for specifying behaviour. As a result, it is possible to specify model semantics. KerMeta can be used directly for implementing metamodels for DSLs.

Ceteva's XMF-Mosaic [31] is a development environment for domain-specific languages. It supports the definition of grammars and the generation of parsers, as well as domain model design. XMF-Mosaic is not fully standards compliant. Its main metametamodel is XCore, which is similar, but not identical, to MOF 2.0. Other well known MDE approaches to DSL implementation are the MOFLON [5] and MetaEdit+ [21] frameworks. The metamodeling framework MOFLON combines MOF 2.0, OCL 2.0 and graph transformations to develop DSL implementations, while MetaEdit+ is an environment that enables the development of DSL tools and code generators.

III. Model Management and Epsilon

In this section we discuss a platform for model management in more detail, and use this platform as means to analyse and compare different approaches to DSL implementation for model management.

Epsilon [22] is both a platform of task-specific model management DSLs and a framework for implementing *new* model management DSLs by exploiting the existing DSLs. Epsilon is a component of the Eclipse Generative Modeling Technologies (GMT) research incubator project. More specifically, Epsilon provides a DSL for direct manipulation of models (EOL) [24], and further DSLs for model merging (EML) [23], model comparison (ECL) [27], model-to-model transformation (ETL) [28], model validation (EVL) [26] model-to-text transformation (EGL) [40], and unit testing of model management operations (EUnit).

EOL is the core DSL in Epsilon, providing OCL-like model navigation and modification facilities; all the other DSLs of the platform build on EOL and its runtime environment in different ways. As a result, all the DSLs in Epsilon are highly interoperable. For example, an *operation* defined using EOL can be imported as-is by the M2M and the M2T transformation languages. Moreover, because all DSLs in Epsilon share a common runtime, modules of different DSLs can exchange variables with each other [25]. With regard to supported modelling technologies, the architecture of Epsilon allows users to manage models of different technologies such as MDR and EMF models and XML documents and even implement support for additional formats.

The justification for Epsilon's architecture comes from an argument that DSLs for model management are all conceptually related [24] – they all provide and require basic concepts and logic for navigating models (and many provide logic for modifying models). The existence of this conceptual common core is a key difference between DSL engineering in general and for model management. Another key difference is that DSLs for model management all operate on the same kinds of artefact – models with metamodels – whereas in general DSLs operate on different kinds of artefacts, e.g., programs, free-form text, or models.

The DSLs in Epsilon have been implemented in different ways, based on several of the approaches discussed in Section II. We now discuss these approaches in the context of several concrete examples.

IV. Implementing Model Management Languages in Epsilon

In this section, we describe techniques we have used in practice for implementing model management DSLs. As mentioned in Section III, DSLs for model management all share certain common features, such as the ability to navigate models, and all operate on the same kinds of artefacts (models with metamodels). As Section III also described, Epsilon provides an *extensible* platform for model management: it has been defined so that existing DSLs can be reused for building new model management DSLs. With the exception of EOL, the base language, all of the languages in Epsilon are built using other languages in the platform; as such, Epsilon exhibits a rich technical space of DSLs.

There are three basic styles of DSL implementation used in Epsilon:

- extension and specialisation, i.e., where the language and tools of one language are inherited and thereafter reused by a second language;
- annotation (a special form of extension), where a new language is formed by addition of lightweight annotations to an existing language;

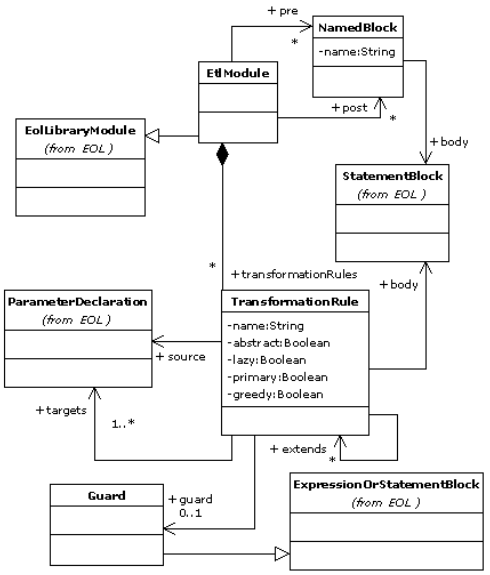

Fig. 1. The Abstract Syntax of ETL

- preprocessing, i.e., where a new language is implemented as a preprocessor, generating output in the form of another Epsilon language.

We now describe these three approaches to DSL engineering with concrete examples of DSLs for model management.

A. Extension and Inheritance

In this approach, a new language is developed as a syntactic and semantic extension of an existing language of the platform. In the majority of occasions in Epsilon, the core language of the platform (EOL) is used as the base language on which the new language is built, but there is also an example of second-level reuse where a new language (EML) [23] has been built not directly atop EOL, but atop ETL [28] which is itself built atop EOL.

The first part of the process of developing a new model management language under this approach is to implement the extension of the abstract syntax of the core language that contains the additional concepts of the new language. In Epsilon, this is achieved by defining a new metamodel which extends the metamodel of the core language and provides additional meta-classes that correspond to the task-specific constructs of the language. As an example, the metamodel of ETL, which is displayed in Figure 1, contains a number of new metaclasses (*EtlModule, TransformationRule, Guard* etc), and also imports core concepts from the base EOL metamodel (*StatementBlock, EolLibraryModule etc*).

The next step involves defining the concrete syntax of the new language as an extension to the concrete syntax of the core language; this is done so as to obtain the functionality of the core language to implement the bodies of ETL rules. In Epsilon this is achieved via the grammar reuse mechanism provided by the ANTLR 3.1 parser generator [38]. Listing 1 demonstrates a reduced version of the grammar of ETL. In lines 3,4 the ETL grammar imports the EOL grammar rules and in lines 11-26 it defines the concrete syntax of the additional constructs it provides.

Having defined both the abstract and concrete syntax of the language, the final step of the process is to define an execution engine to implement the execution semantics of the language and construct end-user tools (e.g., editors, launching facilities). This is achieved by largely reusing the respective infrastructure provided by the core language, and is outside the scope of this paper (though see [9]).

Listing 1. Grammar fragment of ETL that reuses EOL

```
grammar Etl;

import EolLexerRules, EolParserRules,
  ErlParserRules, EtlParserRules;

tokens {
  ETLMODULE;
  TRANSFORM;
}

etlModule
  : importStatement* (etlModuleContent)*
  -> ^(ETLMODULE importStatement* etlModuleContent*)
  ;

etlModuleContent
  : pre | annotationBlock | transformationRule |
    operationDeclaration | post
  ;

transformationRule
  : r='rule'^ rule=NAME 'transform'! formalParameter
  'to'! formalParameterList
  extendz? '{'! guard? block '}'!
  {$r.setType(TRANSFORM);}
  ;
```

1) Advantages: The primary advantage of an extension-based approach in this context is that it produces a tailored syntax that directly matches the task-specific requirements of the new language. For example, as demonstrated in Listing 2 the syntax of ETL makes the structure of transformation rules clear. This particular example shows a transformation of an object-oriented class into a database table. The ETL rules (line 1) are first-class artefacts that clearly demonstrate the source (line 2) and target (line 3) elements as well as the guard (line 5) that limits the applicability of a rule to a particular subset of elements (those classes that are not abstract). The core complex behaviour of the transformation is in lines 8-9. The *columns* of the generated *Table* are set to the *equivalents* of the *Attribute*-typed *features* of the source class using the new ETL *::=* assignment operator. The ETL runtime is responsible for calculating the *equivalents* by delegating to other applicable rules or using the results of previous invocations and assigning the result to the target feature. Importantly, this DSL for model transformation gives a clear structure that is not available in the language being extended (as we illustrate shortly).

Listing 2. ETL Syntax Example

```
rule Class2Table
```

```
2   transform c : OO!Class
3   to t : DB!Table {
4
5   guard: not c.abstract
6
7   t.name := c.name;
8   t.columns ::=
9     c.features.select(f|f.isTypeOf(OO!Attribute));
10  }
```

To demonstrate the advantages of a using a DSL tailored to model transformation in this case, a program with the same functionality, expressed in the general-purpose EOL language, is shown in Listing 3. In the EOL program the source elements, the guard and the target elements are specified implicitly in lines 1, 2 and 3. Contrast how the core behaviour of the rule (mapping class attributes to columns of a database table) was specified in ETL in Listing 2 with how it is specified in lines 5-7 of Listing 3. The equivalents of the class attributes have to be calculated by explicitly invoking the *mapToColumn()* operation, thus making the transformation more verbose. Moreover, if the results of a transformation need to be cached, in EOL this functionality needs to be manually implemented (e.g., by adding @cached annotations to EOL operations).

Listing 3. Transforming classes to tables with EOL

```
1  for (c in OO!Class) {
2    if (not c.abstract) {
3      var t := new DB!Table;
4      t.name := c.name;
5      t.columns := c.features.
6        select(f|f.isTypeOf(OO!Attribute)).
7        collect(f|f.mapToColumn());
8    }
9  }
```

Overall, the advantages of a DSL in this situation come from a more clear structure, and a less verbose specification of behaviour.

2) Challenges: The main challenge of this approach to DSL implementation is the development effort it involves. Although the largest part of the functionality is derived by the base language (around 90% [9]), the developer still needs to specify the extensions to the abstract and concrete syntaxes of the new language. This task requires familiarity with parser generator tools (specifically ANTLR). As several languages have been already constructed using this aproach, it is anticipated that by inspecting existing solutions a developer could relatively easily identify and apply the same principles and tactics for the development of a new language.

B. Annotations

A new language can alternatively be constructed by assigning semantics to constructs of an existing language using task-specific annotations. Annotation-based approaches are related to DSL preprocessing approaches (Section IV-C): indeed, a language based entirely on annotated language constructs is in effect a lightweight preprocessor. Similarly, annotation-based approaches are related to extension-based approaches (Section IV-A): effectively, annotations extend an existing language in a very restricted way. From a design perspective, annotation-based approaches are a useful starting point to DSL development, followed by a more flexible (yet more expensive, in development effort terms) implementation in terms of a preprocessor or language extension.

Turning to annotation-based approaches for building DSLs for model management, in Epsilon's base language, EOL, annotations can be attached to user-defined operations. However, languages built on top of EOL support annotations in some task-specific constructs as well. For example, ETL enables attaching annotations to transformation rules, while EVL supports annotations on constraints. EOL supports two types of annotations: simple and executable. A simple annotation follows a $@ < name > (< value > (,< value >)*)?$ syntactic structure, and specifies a name and one or more values. For example, in Listing 4 the $fibonacci()$ operation is annotated as *cached*, which means that its body will only be executed once, and subsequent calls from the context of the same Integer will return the same result.

Listing 4. Example of simple annotations in EOL

```
1  @cached true
2  operation Integer fibonacci() : Integer {
3    if (self = 1 or self = 0) {
4      return 1;
5    }
6    else {
7      return (self-1).fibonacci()+(self-2).fibonacci();
8    }
9  }
```

An executable annotation follows the $\$ < name >< expression >$ form where $< expression >$ is a valid EOL expression. As opposed to simple annotations the value of which is specified statically, the value of an executable annotation is retrieved by executing the expression. As an example, EOL uses executable annotations to specify pre and post-conditions of user-defined operations (Listing 5). In line 1 the precondition for the add operation specifies that the argument i must be greater/equal than zero. In line 2 the postcondition specifies that the result of the operation (accessible using the $\_result$ variable) must be equal or greater than the Integer ($self$) on which it was invoked.

Listing 5. Example of executable annotations in EOL

```
1  $pre i >= 0
2  $post _result >= self
3  operation Integer add(i : Integer) {
4    return self+i;
5  }
```

The EOL syntax does not limit applicable annotations; the execution engine is left to utilise those that are meaningful for it. An example of using an annotation-based approach for constructing a task-specific language is EUnit [10], a language tailored to unit-testing of model management operations. The

example of an EUnit test-case is in Listing 6. It consists of two tests (*testClasses* and *testAttributes*) to test the correctness of the ETL transformation in Listing 2. The two tests are valid context-free parameter-less EOL operations with the @test annotation. The default EOL runtime engine treats those operations as normal operations, i.e., they have to be explicitly invoked in order for their body to be executed. The EUnit runtime executes all @test-annotated context-less and parameter-less operations automatically. Moreover, in case an assertion fails under the default EOL runtime, a non-recoverable exception is raised and the execution is terminated. By contrast, the EUnit runtime allows assertions to fail and other exceptions to be raised within a test, records them and proceeds with the next test. A detailed discussion on EUnit is provided in [10].

Listing 6. EUnit example

```
@test
operation testClasses() {
  for (c in OO!Class.all) {
    if (not c.abstract)
      assert(DB!Table.all.exists(t|t.name = c.name));
  }
}

@test
operation testAttributes() {
  for (a in OO!Attribute.all) {
    if (not a.owner.abstract)
      assert(DB!Column.all.exists(c|c.name = a.name));
  }
}
```

1) Advantages: The main advantages of the annotation approach are that it provides a more task-oriented syntax for the language, does not require the developer to be familiar with specifying grammars, and does not involve development of new end-user tools (e.g. editors, outline viewers).

2) Challenges: This approach is essentially a trade-off between using an existing language as-is for the new task and implementing a new language using the inheritance-based approach discussed in Section IV-A. As a result, a program expressed in the annotation-based DSL is clearer and more brief than a program expressed directly using the base language, but more verbose than a program written in a DSL developed with the inheritance-based approach.

C. Preprocessing

Preprocessing is an approach where a new language is engineered by defining additional syntax for an existing (or *host*) language (as discussed in Section II). Each piece of new syntax is mapped onto host language constructs, using a *preprocessor*. In the context of model management and Epsilon, the M2T transformation language, the Epsilon Generation Language (EGL) [40], has been implemented as a preprocessor for the base language, EOL.

If we were to implement a new language by a preprocessor, we would start (as usual), by identifying new constructs (abstract syntax) and defining corresponding concrete syntax. For the purposes of M2T transformation, we identified three new constructs: dynamic tags, dynamic output tags, and end tags. The first and last constructs are typical of all M2T languages: they delimit dynamic text from static text. The second construct is a short-hand for dynamic sections that evaluate an expression and generate output. For EGL, we use the concrete syntax: (1) dynamic tags (`[%`), (2) dynamic output tags (`[%=`), and (3) end tags (`%]`).

Depending on the intended semantics of the new constructs, it may be possible to implement a preprocessor as a lexer that produces host language tokens when recognising new language constructs. CPP [15] is an example of this style of lightweight preprocessor. For EGL, this approach was not feasible as semantics needed to be attached to pairs of tags. Instead, a grammar was defined for EGL and used to construct a parser. The preprocessor for EGL generates EOL from the abstract syntax tree produced by this parser. The grammar for EGL is used to define *sections* in the source code. The `[%` and `%]` tags are used to delimit *dynamic sections*. Similarly, `[%=` and `%]` demarcate *dynamic output sections*. All other sections of text are *static sections*. Listing 7 illustrates the use of dynamic and static sections to form a basic EGL template.

Listing 7. A basic EGL template.

```
[% for (i in Sequence{1..5}) { %]
i is [%=i%]
[% } %]
```

Once syntax is established, a mapping between new and host language constructs must be specified. The preprocessor implementation conforms to this specification. The mapping between EGL and EOL can be informally summarised as follows:

- The contents of dynamic sections appear verbatim in the generated EOL.
- The contents of dynamic output sections are enclosed in a call to `out.print`.
- The contents of static sections are enclosed in a call to `out.print`, and wrapped in quotes (to denote a string literal).

Using these rules, the EGL program in Listing 7 translates to the EOL program shown in Listing 8.

Listing 8. Resulting EOL generated by the preprocessor.

```
for (i in Sequence{1..5}) {
  out.print('i is ');
  out.print(i);
  out.print('\r\n');
}
```

To simplify the execution of EGL programs, we have constructed a runtime engine that automatically invokes both the EGL preprocessor and then the EOL runtime engine. End-user tools have been developed using extension and inheritance as discussed in Section IV-A.

1) Advantages: Conceptually, the preprocessing approach provides a middle-ground between language extension (Sec-

tion IV-A) and annotation (Section IV-B). The latter is the least flexible, allowing only variants of existing syntactic constructs to be defined. Preprocessing facilitates the definition of new language constructs, with the restriction that a transformation to existing constructs in a host language can be specified. The former approach has no such restriction and hence exhibits the most flexibility.

2) Challenges: Traceability is a key concern for a preprocessing approach. To provide accurate error reports, the preprocessor must maintain a mapping between the source code and the preprocessed text. The approach used in EGL is to populate a look-up table of source lines indexed by preprocessed line. Errors reported by the EOL execution engine are then mapped to EGL error reports at the equivalent position in the source code.

Of significant importance for preprocessing is choosing a suitable host language. To specify new constructs, it must be possible to map onto host constructs. For example, it is not possible to provide a looping construct by providing a preprocessor for a language that does not already define a mechanism for looping or jumping. However, extensible languages simplify preprocessing in this regard. For example, an extensible type system can be used in conjunction with preprocessing to provide new functionality. This approach was used in EGL to provide the `Template` type, which enables operations to be performed on EGL templates.

In general, selecting an appropriate host language for preprocessing is as important as selecting the language to annotate. In the case of Epsilon, the base/host language available for all model management DSLs is EOL – a language that was specifically designed to be extended, annotated, or generated [24].

From a development effort perspective, we can make only typical observations as to how preprocessing approaches compare with others. The extension approach usually requires more development effort than preprocessing. For the extension approach, a lexer, parser and execution engine must be implemented to enable program execution, whereas for preprocessing, semantics are specified by the mapping between new and host language constructs (though normally a lexer and parser must be written). Also, annotation-based approaches usually require less effort to implement than preprocessing, requiring only a simple lexer/parser (for the annotations) and an execution engine to interpret the semantics introduced by new language constructs. These are typical observations, but it is difficult to say precisely whether implementing a *set* of annotations will always require less development effort than implementing a preprocessor.

When using a preprocessor, re-use is achieved by mapping new to host language constructs, and re-use is not required at the grammar specification level. For example, when defining a new language for Epsilon using the inheritance and extension approach, ANTLR must be used to specify a grammar, but this is not required for the DSL built using pre-processing. We found it much more straightforward to specify the EGL parser in Java rather than in ANTLR (as it uses back-tracking, and processes whitespace differently depending on the current context). As such, preprocessing was more suitable than inheritance and extension for the specification of EGL.

V. Synthesis

The previous section has presented the advantages and challenges of three approaches to model management DSL implementation. In this section we summarise these and present a lightweight process that aims to help engineers decide which approach to use for a particular language engineering problem. We supplement this process with empirical data in terms of the development cost of each approach, in terms of lines of code needed to implement a working solution.

A. Basic process

The basic process for engineering a new model management DSL is similar to that for building other DSLs, e.g., for programming tasks. For simplicity of presentation, we describe the process in terms of engineering a new language using Epsilon.

1) **New task identification:** an engineer identifies a new model management task that they wish to support, for which no existing DSL satisfies the engineer's requirements.
2) **Application of existing language:** the engineer expresses the task in an existing Epsilon language. For example, the engineer might choose to use a GPL such as EOL for carrying out the task.
3) **Pattern identification:** while implementing the task in the existing language, the engineer observes recurring syntactic and semantic patterns in the EOL code. The engineer has to decide whether or not the patterns occur often enough to justify the expense of designing and implementing a new task-specific language.
   For example, before Epsilon introduced the Epsilon Comparison Language (ECL), pure EOL was used to compare models. Two patterns were detected in the EOL code: explicit variables were defined to record matching elements, and loops were used to check all elements of a specific type. The existence of these patterns triggered the development of ECL.
4) **Define syntax:** once the engineer is convinced of the need for a new language, abstract constructs, derived from the patterns, are elicited. Abstract and concrete syntax for these constructs are defined, with connection points to existing syntaxes.
   For example, with ECL, based on the patterns mentioned above, the decision was made to abstract away from both variables recording matches, and loops to carry out matching, and to produce a rule-based comparison mechanism, which generated matches as side-effects. It is typically not the case that recurring patterns are implemented directly; engineering judgement must be used to find the most appropriate abstraction to use to abstract away from and encode the patterns.

5) **Implement syntax and semantics:** at this stage, the engineer must choose an approach to use for implementing the new language. The three approaches presented above in Epsilon (and in other approaches to DSL design and implementation) are: inheritance/extension, annotation, and preprocessing. Based on the challenges and advantages presented in Section IV, we identify the following questions that the engineer can use to help make their decision. The questions can be broken down, sequentially, into three categories: syntax, semantics, and cost, as summarised in the activity diagram (Figure 2).
   The selection process commences by asking whether it is necessary and useful to have a tailored concrete syntax for the new model management task. If yes, either an inheritance/extension or preprocessing approach may be appropriate. If not, the engineer should clarify whether the identified new constructs are syntactic variants of constructs in an existing language (e.g., EOL). If the constructs are just variants, then an annotation approach may be the most approriate. If they are not, then the engineer should reconsider the first question (the arc back to the initial action state in Figure 2): because there is uncertainty in the language's requirements, and the engineer may benefit from considering the questions again (though of course they should ask whether they have sufficient information to answer the questions). The engineer may also benefit from considering cost and programming effort.
   If a tailored syntax has been determined to be useful, the engineer must next ask whether a *tailored semantics* is required (e.g., to support traceability, rule scheduling). If so, the extension or inheritance approach may be appropriate; otherwise, a preprocessor approach is worth considering.
6) **Deployment and maintenance:** the language is deployed. As the language is used, and feedback obtained from engineers on the suitability of the syntax, semantics, and tool support, it may be desirable to revise the implementation approach (e.g., to switch from annotations to extension) at a future date.

The process above focuses specifically on the *flexibility* required in the new language; the questions ignore issues of development cost. In parallel to the activities in Figure 2, the engineer may also choose to assess the development costs associated with using a specific DSL implementation approach. Ideally, the engineer balances requirements for flexibility against the costs for development[1]. While it is generally quite clear what the advantages and challenges are for each approach in terms of flexibility, the situation is somewhat more complicated in terms of development costs.

[1]While development cost can be measured in many ways, we use a very simple measurement here: the number of lines of new code that must be hand-written.

B. Observations

Having implemented a number of languages, we have made several observations on the cognitive and technical effort needed to implement a new language using the three approaches presented in Section IV.

First, we observe that the extension approach requires most technical effort and skills, as the developer needs to be familiar with the inner workings of the platform for implementing the semantics of the new constructs, as well as with lexer/parser generators for specifying the grammar that describes the concrete syntax of the language.

By contrast, the annotation approach requires no effort in terms of implementing the syntax of the new language as the syntax of the base language is reused as-is. The effort needed to implement the semantics of the language is also typically smaller compared to the extension approach as annotation-based languages are augmenting or modifying existing semantics rather than defining from scratch.

Finally, the preprocessor approach requires little or no expertise of the inner workings of the platform, as the base language's run time engine is reused. However, it may involve significant effort to construct the actual preprocessor which is responsible for performing the translation into the base language. For example, in the case of EGL, a fairly complicated parser had to be written so that the language could encompass both dynamic and static regions.

We also observe that all approaches benefit substantially from reuse of the EOL runtime engine: as displayed in Table I, the size of the runtime (in Java LOC) for languages developed with the extension approach is approximately 10% that of the base language. As well, we confirm our observation that the annotation-based approach is the most lightweight and the less effort-demanding: EUnit required only 147 lines of code to specify its additional semantics.

Approach (Language)	Semantics	Syntax	Developer Tools
Preprocessor (EGL)	2609	0	921
Extension (ETL)	721	95	439
Extension (EML)	737	64	325
Extension (EVL)	816	119	457
Extension (ECL)	749	109	374
Annotation (EUnit)	147	0	0
EOL	7984	1005	2990

TABLE I
LINES OF CODE REQUIRED FOR LANGUAGE SEMANTICS, SYNTAX, AND DEVELOPMENT TOOLS

The amount of code required for the development tools is roughly consistent across all types of language engineering [9]. The development tools for EGL (implemented as a preprocessor) required slightly more code, because EGL's syntax is more complicated, and also because additional tools were needed for handling file management and content preservation – factors that had nothing to do with use of the preprocessing approach.

This demonstrates that although the effort required to im-

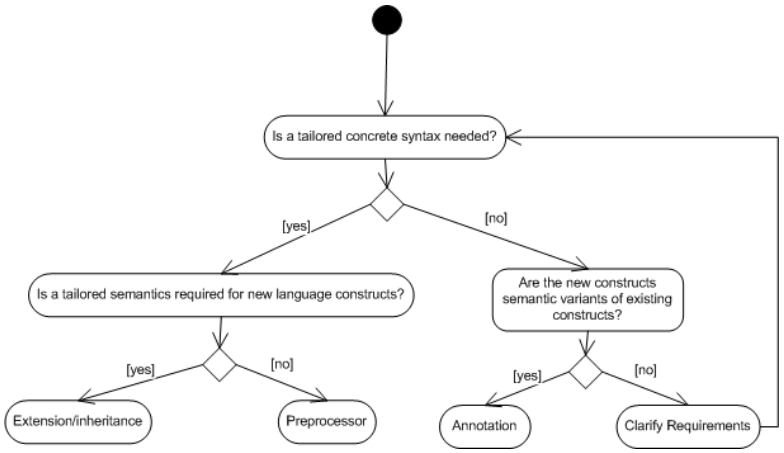

Fig. 2. Selecting a DSL implementation approach

plement a language is significantly affected by the chosen implementation approach, it is also affected by the complexity of the language itself. For example, one can easily envisage a situation where a complex annotation-based language can exceed in size a conceptually simpler language built with the extension approach.

As a general remark, regardless of the complexity of the DSL that needs to be implemented, a definite conclusion we make from our work on Epsilon is that having a suitable base, host, or GPL language to build on for DSL implementation is critical. EOL was a suitable language for this domain, as it was designed to provide the basic constructs that are critical for all model management tasks. If we had chosen a different language – e.g., OCL – for this, the results may not have been as favourable in terms of the benefits of reuse.

VI. Conclusion and Further Work

We have described the technical process, challenges, and advantages of three approaches to DSL implementation for model management, in the context of the Epsilon platform. The approaches – extension, annotation, and preprocessing – allow DSL engineers to trade off language and implementation flexibility for development cost. We have illustrated how the three approaches have been used in implementing languages in the Epsilon platform, and have provided some data illustrating the respective costs (in terms of amount of hand-written code needed) of using the approaches.

Conceptually, there is a continuum of flexibility in implementation approaches from the most flexible approach (language extension) to the least flexible (annotation).

The picture is not so clear in terms of the development cost associated with each approach. While the annotation approach appears to be the least costly to use for implementation (in terms of new lines of code to write), the more annotations that are implemented for a DSL for model management, the closer the language approximates one implemented using preprocessing. As such, there is a point where it becomes more cost-effective to switch to a preprocessing implementation from an annotation implementation. The point at which this switch makes sense in terms of effort will likely vary among model management tasks.

We are currently experimenting with implementing model management DSLs that *combine* two or more implementation approaches. Such combinations of approaches appear to be needed for abstract languages that hide complex operational semantics – e.g., workflows – from the user, and for languages that have substantial graphical user interface requirements as well.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

[1] A. Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[2] Atlas and LINA Project Team. AMMA Platform. http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/, 2008.

[3] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40. ACM, 2002.

[4] Martin Bravenboer and Eelco Visser. Designing Syntax Embeddings and Assimilations for Language Libraries. In *4th International Workshop on Software Language Engineering (ATEM),*, 2007.

[5] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proc. Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066, pages 361–375. LNCS, Springer, 2006.

[6] Ruby Community. Ruby official website. http://www.ruby-lang.org/en/, 2008.

[7] D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *The Journal of Systems and Software*, 56:91–99, 2001.

[8] Dimitrios Kolovos, Louis Rose, Richard F. Paige, Fiona Polack. Raising the Level of Abstraction in the Development of GMF-based Graphical Model Editors. In *Proc. 3rd Workshop on Modeling in Software Engineering (MISE)*. ACM/IEEE, 2009.

[9] Dimitrios S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, Department of Computer Science, The University of York, York, United Kingdom, June 2008.

[10] Dimitrios S. Kolovos, Richard F. Paige, Louis M. Rose, Fiona A.C. Polack. Unit Testing Model Management Operations. In *Proc. 5th Workshop on Model Driven Engineering Verification and Validation (MoDeVVa), IEEE ICST*, April 2008.

[11] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *11th International Conference on Compiler Construction*. LNCS, Springer, April 2002.

[12] F. Jouault and J. Bèzivin. KM3: A DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer Berlin / Heidelberg, 2006.

[13] The Eclipse Foundation. EMF Compare. http://wiki.eclipse.org/index.php/EMF_Compare, 2008.

[14] ATLAS Group. Atlas Transformation Language Project Website. http://www.eclipse.org/m2m/atl/, 2007.

[15] Free Software Foundation Inc. The C Preprocessor [online]. [Accessed 12 July 2008] Available from World Wide Web: http://gcc.gnu.org/onlinedocs/cpp/, 2008.

[16] I. Kurtev J. Bézivin, F. Jouault and P. Valduriez. Model-based DSL Frameworks. In *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, October 2006.

[17] J. Bosch. Delegating compiler objects: modularity and reusability in language engineering. *Nordic Journal of Computing*, 4(1):66–92, 1997.

[18] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 1998.

[19] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages (PADL 99)*. Springer-Verlag, 1999.

[20] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes. In *Proc. GPCE '06:*, pages 249–254. ACM, 2006.

[21] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling*. Wiley, 2008.

[22] Dimitrios S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website. http://www.eclipse.org/gmt/epsilon, 2007.

[23] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In *MoDELS*, volume 4199 of *LNCS*, pages 215–229. Springer, 2006.

[24] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *ECMDA-FA*, LNCS 4066, pages 128–142. Springer, 2006.

[25] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. A framework for composing modular and interoperable model management tasks. In *Model-Driven Tool and Process Integration Workshop*, pages 79–90, 2008.

[26] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In *Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis*. LNCS 5115, Springer-Verlag, 2008.

[27] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proc. GaMMa '06*, pages 13–20. ACM Press, 2006.

[28] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In *Proc. 1st International Conference on Model Transformation, ICMT*, Zurich, Switzerland, July 2008. LNCS, Springer-Verlag.

[29] L. Nakatani and M. Jones. Jargons and infocentrism. In *1st Acm SIGPLAN Workshop on Domain-Specific Languages*, 1997.

[30] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *MoDELS Satellite Events*, pages 139–150, 2005.

[31] Xactium Limited. XMF-Mosaic. http://www.xactium.com/, 2007.

[32] M. Bravenboer, R. de Groot, and E. Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, Braga, Portugal,, 2005.

[33] M. Mernik and V. Zumer and M. Lenic and E. Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool LISA. *ACM SIGPLAN Notices*, 34(6):68–75, jun 1999.

[34] MontiCore. Project Website. http://www.sse-tubs.de/monticore/, 2007.

[35] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Øyvind Aagedal, and Arne-Jørgen Berre. Toward standardised model to text transformations. In *ECMDA-FA*, pages 239–253, 2005.

[36] openArchitectureWare. openArchitectureWare Project Website. http://www.eclipse.org/gmt/oaw/, 2008.

[37] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[38] Terence Parr. ANTLR Parser Generator. http://www.antlr.org/, 2008.

[39] Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[40] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. The Epsilon Generation Language. In *Proc. ECMDA '08*. LNCS, Springer-Verlag, 2008.

[41] S. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In *USENIX Conference on Domain-Specific Languages*, 1997.

[42] Stratego/XT. Java-Swul. http://www.program-transformation.org/Stratego/Java-Swul, 2005.

[43] T. L. Veldhuizen. Blitz++ User's Guide. Version 1.2. http://www.oonumerics.org/blitz/manual/blitz.ps., 2001.

[44] Triskell Team. KerMeta Platform. http://www.kermeta.org, 2008.

[45] W3C. OWL-Lite - Language Synopsis. http://www.w3.org/TR/2002/WD-owl-features-20020729/, 2002.