An Examination of DSLs for Concisely Representing Model Traversals and Transformations

Jeff Gray
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL 35294-1170
gray@cis.uab.edu

Gábor Karsai
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37203
gabor@vuse.vanderbilt.edu

Abstract

A key advantage for the use of a Domain-Specific Language (DSL) is the leverage that can be captured from a concise representation of a programmer's intention. This paper reports on three different DSLs that were developed for two different projects. Two of the DSLs assisted in the specification of various modeling tool ontologies, and the integration of models across these tools. On another project, a different DSL has been applied as a language to assist in aspect-oriented modeling. Each of these three languages was converted to C++ using different code generators. These DSLs were concerned with issues of traversing a model and performing transformations. The paper also provides quantitative data on the relative sizes of the intention (as expressed in the DSL) and the generated C++ code. Observations are made regarding the nature of the benefits and the manner in which the conciseness of the DSL is best leveraged.

1. Introduction

An important step in solving a problem is to choose the notation. It should be done carefully. The time we spend now on choosing the notation may be well repaid by the time we save later avoiding hesitation and confusion. Moreover, choosing the notation carefully, we have to think sharply of the elements of the problem which must be denoted. Thus, choosing a suitable notation may contribute essentially to understanding the problem. [20]

A Domain-Specific Language (DSL) is a "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain" [25]. DSLs assist in the creation of programs that are more concise than an equivalent program written in a traditional programming language. In fact, DSLs are often called "little languages" [1, 4, 24].

An upward shift in abstraction often leads to a boost in productivity. It has been observed that a few lines of code written in a DSL can generate a hundred lines of code in a traditional programming language [11]. A key advantage is that a DSL is perspicuous to the domain expert using the language. A DSL is typically more concise because the notations and abstractions characterizing the intention of the domain are built into the generator that synthesizes a program written in a DSL. This is a key benefit of the approach that has become known as generative programming [6]. Another common characteristic of DSLs is the declarative nature of these languages. A DSL can be declarative because the domain semantics are clearly defined, and thus the declarations have a precise interpretation. DSLs can also offer benefits to individuals who possess detailed knowledge about a particular domain, yet lack the technical programming skills needed to implement a computerized solution. In such cases, "A DSL allows a computationally naïve user to describe problems using natural terms and concepts of a domain with informality, imprecision, and omission of details" [2].

A DSL can assist in separating programmers from lower-level details, such as making the decisions about specific data structures to be used in an implementation. DSL's capture the variability of a domain: the user is allowed to express his/her constructs in terms of this variability, while the invariants of the domain appear as "primitives" in the language. By using a DSL, a programmer uses idioms that are closer to the abstractions found in the problem domain. This has several advantages:

- The tedious and mundane parts of writing a program are automated in the translation from the DSL to a traditional programming language.
- Repetitive code sequences are generated automatically instead of the error-prone manual cut-and-paste method. The generation of such tedious

code also has advantages in the maintenance phase of a project's lifecycle. Programs written in a DSL are usually easier to understand and modify because the intention of the program is closer to the domain.

- Solutions can be constructed quickly because the programmer can more easily focus on the key abstractions. A DSL hides the underlying details of the solution space as implemented in a traditional programming language.

This paper describes several advantages that were realized in using three different DSLs on two separate projects. In section 2, a tool integration project is described [12]. This project utilized a DSL to describe the ontologies of fault-analysis modeling tools in the avionics vehicle health management domain. Another language was used to specify the method for transforming a model from one tool into the format used by a different tool. These DSLs assisted in isolating the programmer from the underlying CORBA data structures and service calls that are needed to perform the model integration. In section 3, a different project is described. This effort is focused on the idea of bringing the concept of aspect-oriented programming [13] to domain modeling [9]. In this project, a benefit was achieved by using a DSL to specify navigation within the domain models while performing transformations. This language shielded the programmer from the details of the core XML Document Object Model (DOM) API calls. The paper also contains a section on general observations, as well as a conclusion.

2. Tool integration

The ability to specify the modeling semantics of new tools, and to integrate them with a set of previously defined tools, can be very useful. Often, however, researchers independently develop similar tools to perform a specific function (e.g., some type of analysis) within a particular domain. Each isolated effort defines a different semantic model and uses diverse persistent storage mechanisms (e.g., a database, or a set of comma separated files, etc.). Unfortunately, this poses a problem when it comes to the important issue of integration – the result is an inability to provide a seamless exchange of model representations between tools. This is a serious problem in bioinformatics [7, 22] and other domains that foster environments demanding rich toolsets to support various forms of analysis. The solution presented in this section describes DSLs to support integration among a set of engineering tools [12].

2.1 Tool Integration Framework (TIF)

Our tool integration framework provides an architectural solution to the semantic integration problem. In our approach, an Integrated Model Server (IMS) is created for each distinct tool domain. Built into each IMS instance is a *single* domain-specific schema that is capable of representing all of the principal entities/relations of all tools in a given domain. The IMS also contains the unique definitions of each tool that is to be integrated, as well as semantic translators that describe the mapping between each unique tool and the single tool domain schema (note that mappings must be described in both directions – from the tool to the integrated schema, and from the integrated schema to the tool). As can be seen in Figure 1, the integrated schema provides a semantic mapping between similar concepts in different tools. Using this technique, each new tool, in a sense, becomes componentized into the IMS.

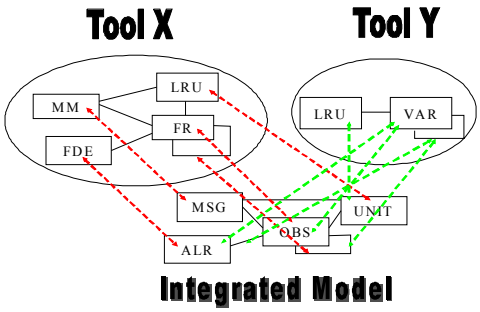


Figure 1. Semantic mapping of tools using an integrated domain model

The Common Model Interface (CMI) is a collection of CORBA interfaces that provides tools with the capability of exchanging models with the server via a network connection. The CMI is specified in the CORBA Interface Definition Language (IDL). It defines the data structures and rules for accessing the IMS. As can be seen in Figure 2, new "integrated" tools can be created that access the IMS directly through the CMI. An example of such a tool is the Java-based Integrated Model Browser, which provides a view of the contents of the IMS using a standard web browser.

Legacy tools that were developed without knowledge of the IMS must have their models transformed by a tool adapter into a form that can be sent via the CMI. Each tool adapter must convert the data in native storage format into a structure that is valid with respect to the CMI. This process is a simple syntactic transformation, thus tool adapters are focused on syntactic issues. Currently, we have created five different tool adapters that permit the integration of tools within the domain of avionics fault analysis. The native storage formats for these tools have been in the form of an Access database, an Excel spreadsheet, a comma-separated file, a proprietary textual specification language, and a Microsoft COM-based modeling tool. The IMS persistently stores the translated models into a database that is built on top of Microsoft

Repository. The underlying database can be either SQL Server or Access.

This subsection presented a very brief overview of the framework. There are many points that have not been explained. More details can be found in [12]. The remainder of this section describes the DSLs that are used to represent the concepts of each tool (see section 2.2) and the transformations that are performed in conversion between the tool and the IMS (see section 2.3).

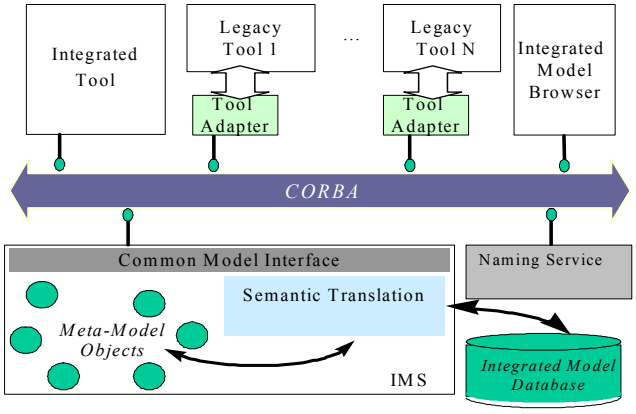


Figure 2. Tool Integration Framework (TIF) with Integrated Model Server (IMS)

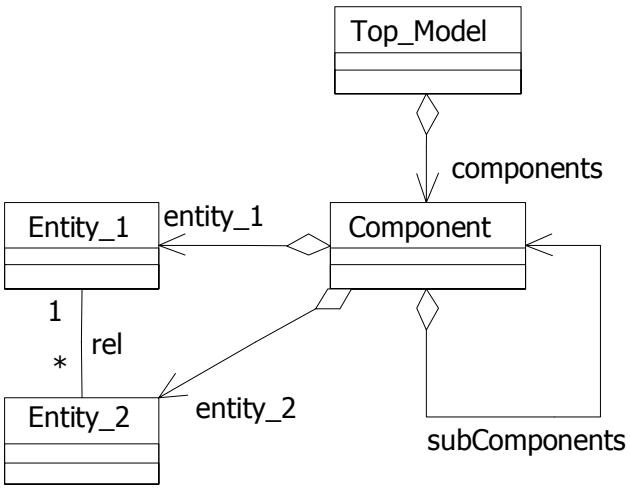
2.2 Model specification language

The first order term in the success equation of reuse is the amount of domain-specific content and the second order term is the specific technology chosen in which to represent that content. [5]

Engineering design tools manipulate models. A model can be thought of as a graph structure. Each node in the graph represents some entity in the model, and each link represents some relation between entities. The links can represent explicitly defined relations, or they may denote a more implicit link that is a result of a hierarchical containment. The models follow a data model (or schema), which is expressed in the form of a Model Specification File (MSF). The MSF is written in a declarative DSL that captures the data model for the various entities and relationships within a tool. It is an example of a type of DSL that is used for data structure representation [23]. The specification in Figure 3a illustrates a simple example of an MSF, and the bottom of the figure is a corresponding representation in the UML. The first step in building a domain-specific integration solution is to create an MSF for the concepts within the domain of the set of tools to be integrated; this is the domain schema. The domain schema is then sent to a generator that produces C++ code of the equivalent CMI representation. This C++ code defines and implements classes that allow the construction and manipulation of CORBA data structures that are compliant with the CMI definitions. An MSF file is also specified for each tool that is participating in the integration. The MSF files for all of the tools are also passed into the code generator and translated into a corresponding C++ representation that is "wrapped around" CMI data structures.

```
paradigm Foo;

model Top_Model {
 part Component components;
}

model Component {
 part Entity_1 entity_1;
 part Entity_2 entity_2;
 part Component subComponents;
 rel Rel rel;
}

entity Entity_1 { ... }

entity Entity_2 { ... }

relation Rel {
 Entity_1 src 1
        <->
 Entity_2 dst *;
}
```

a) MSF representation of tool domain



b) UML representation of tool domain

Figure 3. Sample tool definition (a) with corresponding UML class diagram (b)

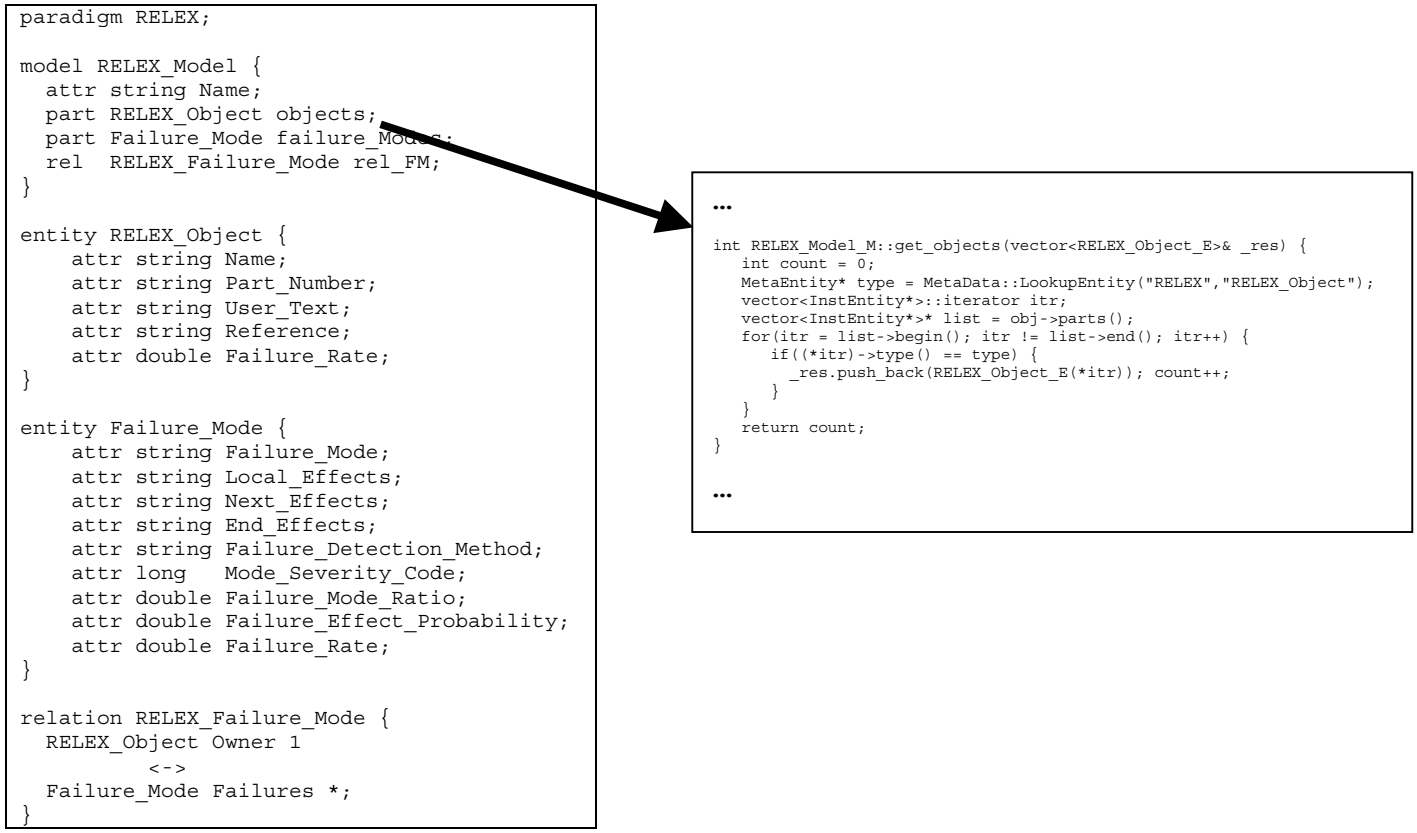


Figure 4. MSF for the Relex reliability analysis rool (with subset of generated C++)

An example of a tool that was included in our integration effort is shown in Figure 4. The left side of this figure specifies the entities and relationships for a subset of Relex – a commercially available reliability analysis tool that uses Access for persistent storage [21]. This figure also shows a portion of the generated code from a single line of the Relex MSF. The MSF code generator will build wrapper classes (e.g., `RELEX_Model_M` and `RELEX_Object_E` in the right side of the figure) that are extensions of the CMI representation in CORBA. These wrapper classes provide the definition of attributes and relationships, as well as the access methods needed to retrieve the attribute values using the CMI. Our contention is that there are many advantages to writing the pertinent characteristics of a tool using the MSF, and then having the code generator produce the details for building the scaffolding to interact with the underlying CMI data structures within CORBA (i.e., we would rather write the specification on the left side of Figure 4, rather than the code on the right side).

2.2.1 Comparing MSF and generated code. To our knowledge, there have been very few studies that have quantified the actual productivity improvements offered by DSLs. One of the earliest studies demonstrated an order of magnitude difference [11]. The most detailed study of this topic can be found in [3], where it was discovered that a DSL for specifying data structures led to a reduction of programming time by a factor of 3. It was also determined in that study that the number of lines of code needed to represent a specific intention was reduced by a factor of 4.

Table 1 lists several measurements taken between the MSF and the generated C++ along the criteria of lines of code, and size of code (number of bytes). A ratio of differences between the sizes of these two representations is also provided within each cell of the table. The representative samples come from the five tools that were integrated in our initial effort. An examination of these comparisons shows a significant improvement in the conciseness of representation when using a DSL like the MSF.

Table 1. Comparison of MSF to generated code

	Lines of Code	Bytes of Code
Advise	MSF: 33 C++: 506 Ratio: 1::15.33	MSF: 761b C++: 14.79k Ratio: 1::19.44
Relex	MSF: 34 C++: 538 Ratio: 1::15.82	MSF: 819b C++: 17.54k Ratio: 1::21.42
FMECA	MSF: 44 C++: 802 Ratio: 1::18.22	MSF: 1.26k C++: 27.32k Ratio: 1::21.68
AEFR	MSF: 49 C++: 639 Ratio: 1::13.04	MSF: 870b C++: 21.42k Ratio: 1::24.62
GME	MSF: 58 C++: 922 Ratio: 1::15.89	MSF: 1.19k C++: 28.71k Ratio: 1::24.13

2.3 Semantic translation specification language

The final stage of the process for integrating a new tool into the IMS is focused on the creation of a mapping strategy between the various tools and the specific IMS domain schema. The developers who perform this task must have an understanding of the tool semantics and the IMS schema semantics. The translation process must link the entities and relations in the tools with the corresponding modeling elements in the IMS (or vice versa). The process for creating semantic translators is at the core of our tool integration technology. The most difficult task in creating a semantic translator is the specification of a strategy that will traverse/visit one graph and transform it into a different graph. To assist in this process, we have constructed a generator for another DSL, which is based on Adaptive Programming (AP) [16].

In AP, a key focus is the separation of behavior from structure. To aid in the modularization of this concern, visitor and traversal strategies are used. This modularization prevents the knowledge of the program's class structure from being tangled throughout the code, a desirable property that is called "structure shyness." Traversal strategies can be viewed as a specification of the class graph that does not require the hardwiring of the class structure throughout the code [17]. An example of a traversal/visitor language for supporting such modularization is described in [19]. Our application of the idea of AP is being applied toward the tool integration problem and the transformation of models. Our approach differs from traditional AP, however, in that our focus is on model representations of tools, not programs written in traditional languages.

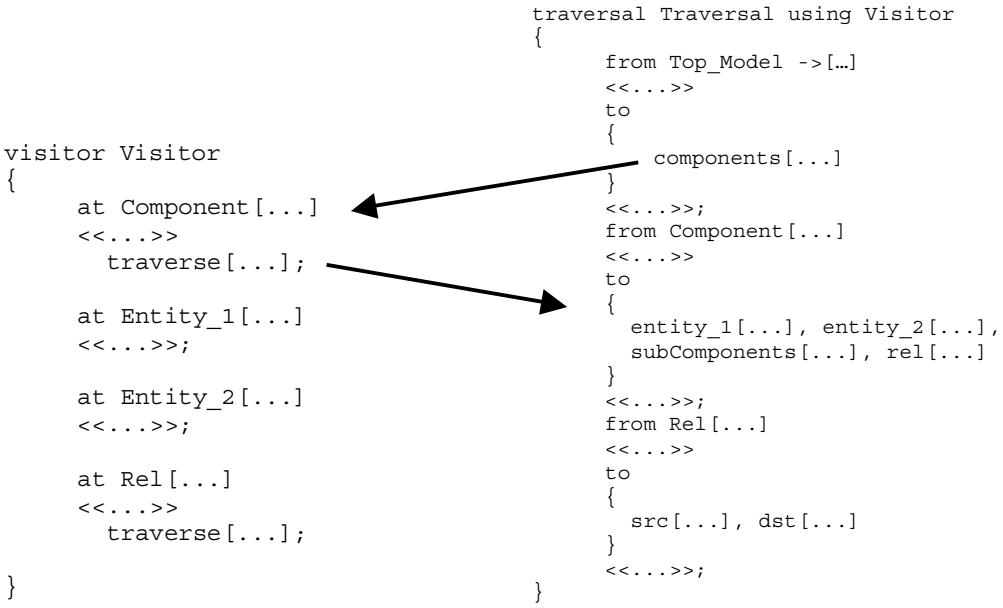
In a semantic translator, the specification of the traversal, and the actions to be performed at each traversed node, are separated. Separation of concerns is evident in our tool integration process in the following ways:

- Separation of the *structure* of the models - what are the *possible* paths for traversals? (see the left side of Figure 3)
- Separation of the *traversal* sequences - what are the *desired* paths for traversals? (see the right side of Figure 5)
- Separation of the *visitors* - what are the transformation actions at each node? (see the left side of Figure 5)

An instance of another DSL is shown in Figure 5, which demonstrates the traversal/visitor specifications that appear within a translator. This DSL is called the Traversal/Visitor Language (TVL). The translation process begins with the `Top_Model` and follows along the traversal specifications. At visitor nodes, a specific action is performed that executes the required transformation (these are elided inside of the inline code, denoted as <<…>>). For nodes that contain other entities (like Component), it is necessary for the respective visitor to further traverse the contained entities (see the lower arrow in Figure 5). In Figure 5, the first two steps in the model translation are shown by two arrows. The remaining traversal/visitor sequence would follow similarly. Although it is not shown in Figure 5, there are also constructs in the TVL that permit multiple passes through the model structure.

The code shown in Figure 6 represents an actual piece of a semantic translator that converts a model from a tool into the IMS schema for representing fault-analysis tools. This code is just one of several traversal specifications included in this tool's semantic translator – in this case, the partial specification describes the manner in which a Component is to be traversed.

The generated C++ code in Figure 7 corresponds to the traversal fragment of Figure 6. There are a few things to notice about this generated code. Perhaps the most obvious observation is that the majority of the code is concerned with iterating over collections. In fact, a large percentage of the code is a replicated template for iteration over vectors. Manual construction of repetitive code, like that in Figure 7, is often a ripe area for introducing errors. Automatically generating such code can offer more assurance that the translator is correctly constructed. A second observation that can be made from the generated code is that the code generator for TVL also knows about the tool definitions contained in the MSF file. Notice names like `GME_4_0::Component_M` and `GME_4_0::FailureMode_E` in Figure 7. These names represent classes that were generated from the MSF file for a specific tool (the GME).



```
visitor Visitor
{
      at Component[...]
      <<...>>
        traverse[...];

      at Entity_1[...]
      <<...>>;

      at Entity_2[...]
      <<...>>;

      at Rel[...]
      <<...>>
        traverse[...];

}
```

```
traversal Traversal using Visitor
{
      from Top_Model ->[…]
      <<...>>
      to
      {
        components[...]
      }
      <<...>>;
      from Component[...]
      <<...>>
      to
      {
        entity_1[...], entity_2[...],
        subComponents[...], rel[...]
      }
      <<...>>;
      from Rel[...]
      <<...>>
      to
      {
        src[...], dst[...]
      }
      <<...>>;
}
```

Figure 5. Traversal/Visitor specifications (based on the model from Figure 3)

```
…

  from Component[IMS::Component_M& parent]
         to {
               failureModes[parent,fMap],
               discrepancies[parent,dMap],
               monitors[parent,mMap],
               faultReports[parent],
               subComponents[parent,pcMap],
               fmMonitor[parent,fMap,mMap],
               fmDiscrepancy[parent,fMap,dMap]
          };


…
```

Figure 6. Single traversal specification in GME2IMS

```
…
void Traversal_T::traverse(GME_4_0::Component_M& self,
                           IMS::Component_M& parent) {
    vector<GME_4_0::FailureMode_E> _lst;
    self.get_failureModes(_lst);
    vector<GME_4_0::FailureMode_E>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
     GME_4_0::FailureMode_E arg=GME_4_0::FailureMode_E(*_itr);
     vis->visit(arg,parent,fMap);    }
    vector<GME_4_0::Discrepancy_E> _lst;
    self.get_discrepancies(_lst);
    vector<GME_4_0::Discrepancy_E>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
     GME_4_0::Discrepancy_E arg=GME_4_0::Discrepancy_E(*_itr);
     vis->visit(arg,parent,dMap);    }
    vector<GME_4_0::Monitor_E> _lst;
    self.get_monitors(_lst);
    vector<GME_4_0::Monitor_E>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
     GME_4_0::Monitor_E arg = GME_4_0::Monitor_E(*_itr);
     vis->visit(arg,parent,mMap);    }
    vector<GME_4_0::Fault_Report_E> _lst;
    self.get_faultReports(_lst);
    vector<GME_4_0::Fault_Report_E>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
    GME_4_0::Fault_Report_E arg=GME_4_0::Fault_Report_E(*_itr);
     vis->visit(arg,parent);    }
    vector<GME_4_0::Component_M> _lst;
    self.get_subComponents(_lst);
    vector<GME_4_0::Component_M>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
     GME_4_0::Component_M arg = GME_4_0::Component_M(*_itr);
     vis->visit(arg,parent,pcMap);    }
    vector<GME_4_0::FMMonitor_R> _lst;
    self.get_fmMonitor(_lst);
    vector<GME_4_0::FMMonitor_R>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
     GME_4_0::FMMonitor_R arg = GME_4_0::FMMonitor_R(*_itr);
     vis->visit(arg,parent,fMap,mMap);    }
    vector<GME_4_0::FMDiscrepancy_R> _lst;
    self.get_fmDiscrepancy(_lst);
    vector<GME_4_0::FMDiscrepancy_R>::iterator _itr;
    for(_itr = _lst.begin(); _itr != _lst.end(); _itr++) {
  GME_4_0::FMDiscrepancy_R arg=GME_4_0::FMDiscrepancy_R(*_itr);
     vis->visit(arg,parent,fMap,dMap);    }
}
…
```

Figure 7. Generated C++ code from GME2IMS traversal specification

2.3.1 Comparing TVL and generated code. A comparison between the TVL specification, and the generated code, is presented in Table 2. The five rows represent the five semantic translators that are used to import the tool model into the IMS. There also exist semantic translators for the reverse direction (i.e., from the IMS back to the tools), but are not shown in this table (although they have similar ratios).

Table 2. Comparison of TVL to generated code

	Lines of Code	Bytes of Code
Advise2IMS	TVL: 155 C++: 355 Ratio: 1::2.29	TVL: 4.03k C++: 8.78k Ratio: 1::2.18
Relex2IMS	TVL: 351 C++: 523 Ratio: 1::1.49	TVL: 10.15k C++: 17.54k Ratio: 1::1.73
FMECA2IMS	TVL: 248 C++: 435 Ratio: 1::1.75	TVL: 7.85k C++: 12.10k Ratio: 1::1.54
AEFR2IMS	TVL: 192 C++: 497 Ratio: 1::2.59	TVL: 6.49k C++: 13.39k Ratio: 1::2.06
GME2IMS	TVL: 251 C++: 523 Ratio: 1::2.08	TVL: 7.22k C++: 14.27k Ratio: 1::1.98

3. Aspect-oriented domain modeling

Separate from the tool integration research described previously, this section introduces our work on using a DSL to improve separation of concerns in visual modeling tools. The following subsections provide only a brief overview of our work in Aspect-Oriented Domain Modeling (AODM). We invite the reader to consult [10] for more comprehensive details.

3.1 Problem: Crosscutting modeling constraints

The core research area at the Institute for Software Integrated Systems (ISIS) is Model-Integrated Computing (MIC) [15]. For over a decade, a major focus of MIC has been on domain-specific modeling environments that are created from metalevel specifications of a particular domain. The Generic Modeling Environment (GME) is a metaprogrammable CASE tool that supports the generation of new modeling environments. Using the GME, code generators (interpreters) for domain-specific *visual* languages are used to synthesize applications from models.

One key application area of MIC is that of real-time embedded systems. Here, MIC is applied in the modeling, analysis, and synthesis of the system. Several of the domain models that we have created using the GME are embedded real-time systems that are highly adaptive. In many real-time embedded systems, it is advantageous to model the design space of an application. In fact, this is mandatory for self-adaptive systems that must choose at run-time among numerous alternatives [18]. Our approach to modeling self-adaptive embedded systems uses a form of OCL [26] constraints to help prune the size of the design space during exploration. These constraints stipulate design criteria and limit design alternatives.

Unfortunately, we have found that such constraints are tangled throughout the model hierarchy [9]. These constraints cut across the modular boundaries of a model. The crosscutting nature of these constraints makes it difficult to maintain and reason about their effects and purpose.

It is often the case that a global property, such as processor assignment, is scattered across all nodes in a model. This creates a difficulty because any change to the model, or to the details of the global requirement, will necessitate the modification of multiple nodes in the model. This would require the modeler to visit, by hand, each modeling element in the GME. This is a time consuming task that, in some cases, makes it impossible to view the effect of different constraints.

3.2 Solution: Aspect-oriented techniques

Several new modularity technologies have been proposed that improve separation of concerns in programming languages. In particular, research in Aspect-Oriented Programming (AOP) has been promoted as a means toward the separation of concerns that crosscut the modularity of an implementation [13]. In AOP, a translator called a weaver is responsible for taking code specified in a traditional programming language and additional code specified in an aspect language, and weaving the concerns together. We are uniting our core research area with the powerful new techniques offered in AOP by extending the purview of applicability by developing weavers for constraints in domain-specific models.

Domain-specific weavers are created as a particular instantiation of a metaweaver framework. A core component of this framework is a code generator that translates high-level descriptions of strategies, specified as a DSL, into C++ source code. We call this DSL the Embedded Constraint Language (ECL). It is based on the OCL [26].

Our solution to the problem of tangled constraints involves the separation of constraints from modeling elements. The solution allows modular specifications of constraints to be propagated throughout a model via a domain-specific weaver, whose purpose is to integrate constraints back into a model. Domain-specific weavers

rely on specification aspects and strategies to carry out their duty. *Specification aspects*, similar to pointcuts in AspectJ [14], are used to specify where the constraints will be applied in the model. *Strategies* describe how a constraint is applied in the context of a particular node in the model. The description of specification aspects and strategies allows a modeler to quantify properties of the model in a module that is separate from the model structure.

```
…
components.models("")->select(c |
         c.id() == refID)->DetermineLaziness();
…
```

Figure 8. Fragment of the EagerLazy strategy

Figure 8 contains a single statement from a strategy defined in [10]. This statement finds all of the models that match a specific id and then calls the DetermineLaziness strategy on those selected models. The amount of C++ code that is generated by our code generator, however, is far from being concise or simple (see Figure 9). Much of the code for implementing this strategy statement is focused on iterating over a collection and selecting elements of the collection that satisfy the predicate. The C++ code calls an XML Parser wrapper class that retrieves a set of all models.

```
CComPtr<IXMLDOMNodeList> mods=XMLParser::models(components,"");
nodeTypeVector selectVec1 = XMLParser::ConvertDomList(mods);
nodeTypeVector selectVecTrue1 = new std::vector<nodeType>;
vector<nodeType>::iterator itrSelect1;
for(itrSelect1 = selectVec1->begin();
    itrSelect1 != selectVec1->end(); itrSelect1++) {
  nodeType selectNode1 = (*itrSelect1);
  nodeType c;
  c = selectNode1;
  CComBSTR id0 = XMLParser::id(c);

  ClData varforward1(id0);
  ClData varforward2(refID);
  bool varforward3 = varforward1 == varforward2;
  if(varforward3)
     selectVecTrue1->push_back(*itrSelect1);
}

vector<nodeType>::iterator itrCollCall1;
for(itrCollCall1 = selectVecTrue1->begin();
    itrCollCall1 != selectVecTrue1->end(); itrCollCall1++)
  DetermineLaziness::apply(…);
```

Figure 9. Sample of generated C++ code (generated from ECL in Figure 8)

3.3 Comparing ECL and generated code

Similar to the previous two tables in Section 2, the data presented in Table 3 is a comparison of the conciseness offered by DSLs like ECL. The subjects of this study were a subset of several of the strategies that were created to support our research on aspect-oriented domain modeling. The details of each strategy can be found in [10].

Table 3. Comparison of ECL to generated code

	Lines of Code	Bytes of Code
Power Distribution	ECL: 43 C++: 140 Ratio: 1::3.25	ECL: 859b C++: 3.08k Ratio: 1::3.50
Processor Assignment	ECL: 39 C++: 137 Ratio: 1::3.50	ECL: 954b C++: 3.28k Ratio: 1::3.44
Eager/Lazy	ECL: 85 C++: 230 Ratio: 1::2.71	ECL: 2.03k C++: 6.24k Ratio: 1::3.07
Exhaustive State Transition	ECL: 70 C++: 184 Ratio: 1::2.62	ECL: 1.92k C++: 5.14k Ratio: 1::2.68
State Generation	ECL: 128 C++: 242 Ratio: 1::1.89	ECL: 3.42k C++: 6.76k Ratio: 1::1.98

4. Observations

In many pieces of code the problem of disorientation is acute. People have no idea what each component of the code is for and they experience considerable mental stress as a result. [8]

It is reasonable to assume that any language which raises the level of abstraction will be more concise than the underlying representation unto which it is mapped at generation time. A simple analogy of this would be a comparison of any high-level programming language to the equivalent assembly or object code that resides closer to the execution space. Typically, the representation of a single executable statement in a programming language translates to several assembly instructions, or more than a few bytes of object code. The same is true regarding the constructs offered by a DSL and their equivalent mapping to a programming language.

As Dick Gabriel observed in the above quote, stress can result from the disorientation caused by the mismatch of expression between the intention of an objective and the underlying implementation needed to realize that objective. This is particularly evident with respect to the maintenance and evolution of a piece of software. For example, the right side of Figure 4, the method of Figure 7, and the method in Figure 9 are representations of implementation details that are at a level of abstraction much lower than their counterparts expressed in a DSL. The maintenance of such code would intuitively seem to be more problematic.

The three languages introduced in this paper each highlight a specific type of benefit that can result from

using a DSL. Each of these advantages is discussed in the following three sections.

4.1 Generation of data structures from higher-level specifications

The examples of the MSF, as shown in Figures 3 and 4, draw attention to the succinct expression of the pertinent characteristics of a modeling tool. The corresponding translation into a programming language contains many details that complicate expressibility. By hiding these details, the user of the MSF can focus their attention more on the essential elements that need to be specified. The nastier minutiae of moving into the execution/implementation space are concealed and abstracted away by the MSF.

Our observations from working with the MSF lead us to the conclusion that there are many advantages of generating data structures from specifications written in a DSL. This finding is also confirmed in [23]. The code generator for a DSL can contain the detailed knowledge needed to create the intricate wrappers for a complex set of inter-related data structures (like the CMI and its underlying CORBA interfaces).

4.2 Synthesis of iterative representations

There are often programming tasks that are repetitive in nature. That is, a pattern emerges as a technique for implementing a commonly occurring situation. An example of this can be seen in the code of Figure 7, where a common form of iteration is performed over different collections. The tedious nature of such repetitive duplication of code can be a source for introducing programming errors. With respect to iterating over collections, we have found much benefit in the ability to concisely specify our intention and have a generator create the solution.

The visitor actions that are specified in the TVL often consist of inlined C++ code. The inlined code is directly copied by the translator into the generated file. Conciseness should be improved by the ability to specify, at a higher level of abstraction, the functional changes needed in the transformation. With such an addition, it is possible that the TVL could offer even more benefit than is made evident in Table 2. This is an area of future investigation.

Considering Table 3, an observation can be made regarding the State Generation strategy. Its translation yielded the lowest ratio in comparison. This strategy also contains the least amount of ECL collection statements, suggesting the somewhat obvious fact that all of the code needed to iterate over a collection increases the amount of generated C++ code.

4.3 Wrapping of API calls

Any programmer who has written an application that makes frequent use of the XML DOM will testify that it is not a pleasant experience. This is often true of any library that offers a rich, yet complicated, set of APIs. It takes concentrated discipline to follow the strict sequence of API calls that are needed to accomplish a specific task. This can sometimes force a programmer to spend their time tangled in a morass of implementation details. The XMLParser adapter methods (two of these can be found in Figure 9) shield the ECL programmer from the concerns of calling the DOM to retrieve values. The ECL generator is able to make use of these wrapper methods in order to permit statements, like the one in Figure 8, to be more abstract. This is also true of the example in Figure 4. Many of the CORBA method calls are collected in adapters and facades.

5. Conclusion

We must recognize the strong and undeniable influence that our language exerts on our ways of thinking and, in fact, delimits the abstract space in which we can formulate – give form to – our thoughts. [27]

Domain-Specific Languages gain their power by raising the intentionality of programmer expression. With a DSL, it is argued, a programmer can express their objective in a concise manner using a language that is much higher in expressiveness than that typically offered in a traditional programming language. Because of this, it is often asserted that programs written in DSLs are much easier to maintain and modify.

As described in this paper, observations from our work on three different DSLs suggest that an upward shift in abstraction does indeed permit a more concise specification of an intention. The paper also contains observations that suggest situations that would best benefit from a generative approach using DSLs (e.g., isolating the programmer from the details of complicated data structures and API calls). This is our first attempt at using our experience to categorize the essential characteristics that make DSL use beneficial.

Our comparisons focused solely on the relative sizes of lines of code. It is not clear that a programmer would write code that is similar to that produced from a DSL generator. Although the results suggest a benefit for using DSLs, a future area for further research would investigate the usefulness of our DSLs with respect to improving programmer productivity. That is, an important question is: How much time, if any, can be saved in development when using our DSLs? We believe that our initial studies suggest that decreased development time will result, due to the reduced complexity of expressing an objective in

the problem space and then automatically translating that into the solution space.

A topic that we are currently studying concerns the use of visual DSLs, so-called domain-specific *visual* languages (DSVLs). In particular, we have several ideas that we are investigating with respect to notations for visually describing the semantic equivalent of the ECL using the GME. We do not have plans, however, to further explore DSVL equivalents for the tool integration DSLs. The language processors for these DSLs, and their respective outputs, are tied to Visual Studio projects in such a way that a textual specification works well. The tool integration DSLs also contain many fragments of inlined code, suggesting a textual solution.

Acknowledgements

We thank Sandeep Neema, Ted Bapty, and Beatrice Richardson for their assistance on portions of this work. We also thank the reviewers for their numerous helpful suggestions. The research presented in this paper was conducted at ISIS and partially supported by Boeing. This work has also benefited from support by the DARPA Information Exploitation Office (DARPA/IXO), under the *Program Composition for Embedded Systems* (PCES) program. Early versions of this work were supported by the DARPA EDCS MIC project.

References

[1] John Aycock, "Compiling Little Languages in Python," *Proceedings of the 7th International Python Conference*, Houston, Texas, November 1998, pp. 69-77.

[2] David Barstow, "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, November 1985, pp. 1321-1336.

[3] Don Batory, Jeff Thomas, and Marty Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, New Orleans, Louisiana, December 1994, pp. 111-120.

[4] Jon Bentley, "Programming Pearls: Little Languages," *Communications of the ACM*, August 1986, pp. 711-721.

[5] Ted Biggerstaff, "A Perspective on Generative Reuse," *Annals of Software Engineering*, Vol. 5, 1998, pp. 169-226.

[6] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[7] Susan Davidson, G. Christian Overton, and Peter Buneman, "Challenges in Integrating Biological Data Sources," *Journal of Computational Biology*, vol. 2., no 4., 1995, pp. 557-572.

[8] Richard P. Gabriel, "The Column Without a Name: Software Development as Science, Art and Engineering," *C++ Report*, July/August 1995.

[9] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, "Handling Crosscutting Constraints in Domain-Specific Modeling," *Communications of the ACM*, October 2001, pp. 87-93.

[10] Jeffrey G. Gray, "Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Metaweaver Framework," Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Vanderbilt University, March 2002.

[11] Robert M. Herndon and Valdis Berzins, "The Realizable Benefits of a Language Prototyping Language," *IEEE Transactions on Software Engineering*, June 1988, pp. 803-809.

[12] Gábor Karsai and Jeff Gray, "Component Generation Technology for Semantic Tool Integration," *IEEE Aerospace Conference*, Big Sky, Montana, March 2000.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer-Verlag, Jyväskylä, Finland, June 1997, pp. 220-242.

[14] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.

[15] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, November 2001, pp. 44-51.

[16] Karl Lieberherr, *Adaptive Object-Oriented Software*, International Thomson Publishing, 1996.

[17] Karl Lieberherr, Doug Orleans, and Johan Ovlinger, "Aspect-Oriented Programming with Adaptive Methods," *Communications of the ACM*, October 2001, pp. 39-41.

[18] Sandeep Neema and Ákos Lédeczi, "Constraint Guided Self-Adaptation," *International Workshop on Self-Adaptive Software*, Balatonfured, Hungary, May 2001.

[19] Johan Ovlinger and Mitchell Wand, "A Language for Specifying Recursive Traversals of Object Structures," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, November 1999, pp. 70-81.

[20] George Polya, *How to Solve It*, Princeton University Press, 1957.

[21] http://www.relexsoftware.com/

[22] Adam Siepel, Andrew Tolopko, Andrew Farmer, Peter Steadman, Faye Schilkey, Dawn Perry, William Beavis, "An Integration Platform for Heterogeneous Bioinformatics Software Components," *IBM Systems Journal*, vol. 40, no. 2, 2001, pp. 570-591.

[23] Diomidis Spinellis, "Notable Design Patterns for Domain-Specific Languages," *Journal of Systems and Software*, February 2001, pp. 91-99.

[24] Arie van Deursen and Paul Klint, "Little Languages: Little Maintenance?" *First ACM SIGPLAN Workshop on Domain-Specific Languages,* Paris, France, January 1997, pp. 109-127.

[25] Arie van Deursen, Paul Klint, and Joost Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices*, June 2000, pp. 26-36.

[26] Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[27] Niklaus Wirth, "On the Design of Programming Languages," *Proceedings of the IFIP Congress*, 1974, pp. 386-93.