Using Template Metaprogramming to Enhance Reuse in Visitor-based Model Interpreters

James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

Aniruddha Gokhale
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
Emaila.gokhale@vanderbilt.edu

***Abstract*—This paper discusses an approach called *Metaprogrammable Interpreters for Model-driven Engineering (MIME)*, which integrates C++ meta-programming techniques into model interpreters for domain-specific modeling languages (DSMLs). The goal of MIME is to reduce reinvention of core model interpretation logic across model interpreters that use the Visitor software design pattern. Experience from applying MIME's approach to realistic DSMLs show it overcomes limitations of existing Visitor-based model interpreters by (1) decoupling interpreter-logic from generation-logic and (2) allowing developers to suppress either aspect of the model interpreter, while promoting maximal reuse of code.**

***Index Terms*—C++ template metaprogramming, model-driven engineering, model interpreters, Visitor software design pattern**

I. Introduction

Contemporary model-driving engineering (MDE) techniques, particularly those that leverage domain-specific modeling languages (DSMLs) [1], provide developers with modeling notations that are closer to their domain. This makes it easier and more intuitive to capture various artifacts of the target domain more accurately. Furthermore, the constraints incorporated within DSMLs enforce construction of valid models. If models constructed using the DSML are invalid, users are notified of such violations and they are either corrected automatically or manually (with guidance). Tools that facilitate contemporary MDE techniques include (but are not limited to): Generic Modeling Environment (GME) [2], Generic Eclipse Modeling System (GEMS) [3], Domain-Specific Language Tools included with the Visual Studio Team System [4].

In addition to facilitating intuitive modeling of domain-specific concepts, MDE frameworks support model interpreters that provide generative capabilities to synthesize a variety of artifacts. Different model interpreters with the same *interpretation intentions* (*i.e.*, interactions with the same model elements in a similar fashion) can transform a single model into different representations, such as application and platform configurations, performance evaluation, analysis of different system properties. For example, the platform independent modeling languages and tools, such as CBML [5], and V^{3}Studio [6], contain multiple model interpreters with the same interpretation intentions, but generate different artifacts to target different needs, such as opposing architectures, technologies, and tools.

Existing techniques for implementing model interpreters simplify model interpretation as much as possible using software design patterns [7]. A common implementation technique used by model interpreters is implementing the *interpreter logic* (*i.e.*, how the model is traversed) using the Visitor [7] design pattern. This enables the model interpreter to determine how and when to interact with elements of interest. Tools such as GEMS, GME, and Java Emitter Templates [8] apply this implementation technique.

Although the Visitor design pattern has its advantages, experience developing Visitor-based model interpreters indicates that it does not promote reusability for interpreters with the same interpretation intentions [5], [8], [9]. This is primarily because the model visitation and generation logic is tightly coupled. When other interpreters want to reuse the interpreter logic, it is *hard* to do so because the interpreter logic may not meet its needs (*i.e.*, the interpreter logic is immobile [10]). This forces each interpreter developer to reinvent the core interpreter logic.

A promising technique for promoting reuse of core model interpreter logic is to use generative programming [11] techniques. Generative programming is an attractive choice because it allows transparent alteration of core implementation (such as source code) on a per use case basis without affecting other entities that utilize the same core implementation. Moreover, it allows developers to compose model interpreters based on what interpreter logic features are needed.

This paper therefore investigates how generative programming techniques can be used to improve the limitations of Visitor-based model interpreters. The main contributions of this paper are as follows:

- It describes *Metaprogrammable Interpreters for Model-driven Engineering (MIME)*, which is a metaprogrammable model interpretation technique to address the challenge of reinvent core interpreter logic in Visitor-based model interpreters;
- It describes two contemporary approaches for developing Visitor-based model interpreters and outlines the reasons for reinvention manifested in these approaches; and
- It showcases how template metaprogramming can be extended to model interpreters by capturing the correct

visitation logic and generation logic of the target DSML, and enabling composition of model interpreters based on what features are needed.

Experience using the MIME to implement Visitor-based model interpreters show that it supports implementing model interpreters that not only promote reuse of core interpreter logic, but can be specialized on a per use case basis, *e.g.*, suppressing or altering element visitation and varying generated artifacts.

Paper organization. The remainder of this paper is organized as follows: Section II discusses interpreter writing techniques wherein we illustrate the reasons for reinvention of the interpretation logic that use the Visitor pattern; Section III discusses, in detail, MIME's approach to writing Visitor-based model interpreters; Section IV presents a case study that illustrates MIME in the context of a real-world example; Section V compares MIME with related research; and Section VI provides concluding remarks.

II. Reinvention Challenges in Visitor-based Model Interpreter Development

To help illustrate the limitations of Visitor-based model interpreters, Figure 1 illustrates an example DSML in GME for modeling messages, such as emails. This example, however, is not constrained to GME.

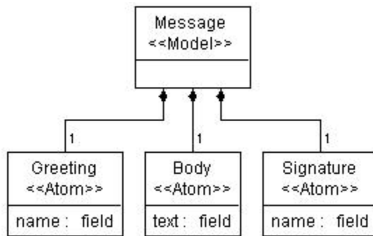

Fig. 1. Simple DSML in GME for modeling messages.

There are two primary techniques for implementing Visitor-based model interpreters: *single interpretation* and *strategized interpretation*. The following provides an overview of the two approaches and their limitations in the context of reinventing core interpreter logic.

Single Interpretation. Single interpretation is the simplest—and sometimes the quickest—approach for writing Visitor-based model interpreters. In single interpretation, core interpreter logic visits (or interprets) only elements of interest. The points in the interpreter code where visitation occurs are called *points-of-visitation*. When each element of interest is visited, the interpreter logic generates artifacts that correspond to that particular element. The points in the interpreter logic where an artifact is generated are called *points-of-generation*.

As illustrated in Figure 2, two different interpreters are defined for the example DSML, namely `Business_Msg_Visitor` and `Empty_Msg_Visitor`. Both implementations use the Visitor software design pattern for traversing models of the message DSML. `Business_Msg_Visitor`, however, visits Body elements

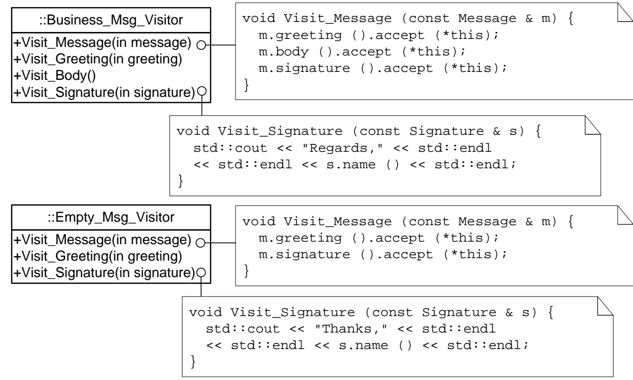

Fig. 2. Conceptual model of single interpreter's implementation.

of the DSML while `Empty_Msg_Visitor` ignores Body elements. `Business_Msg_Visitor`, therefore, has three points-of-visitation (*i.e.,* `Visit_Greeting`, `Visit_Body`, and `Visit_Signature`); whereas, `Empty_Msg_Visitor` has two points-of-visitation. In addition, both interpreters have the same number of points-of-generation as the number of points-of-visitation. All visit methods (*i.e.,* `Visit_*` methods) represent reinvented interpreter logic in both implementations.

The main advantage of single interpretation model interpreters is that each interpreter contains efficient model traversing logic (within the limits of the Visitor design pattern) because it visits only elements of interest. The drawback of this approach is that each interpreter has to reinvent the core traversing logic, e.g., `Visit_Message`, since each interpreter implements a different generation logic (*i.e.*, generates different artifacts) at each point-of-generation.

Strategized Interpretation. Strategized interpretation is an implementation technique that partially addresses the challenge of reinvention in core interpretation logic when targeting multiple metadata formats. In strategized interpreters, developers use the Strategy [7] design pattern to implement a base class that contains points-of-generation as methods. The core interpreter logic is then implemented in terms of the base strategy class. Each model interpreter implements concrete classes derived from the base strategy that override the points-of-generation to generate the appropriate metadata. Lastly, the concrete class is used to strategize the core interpreter logic implemented in terms of the base strategy class.

As illustrated in Figure 3, the base class `Message_Strategy` contains three points-of-generation: `print_greeting`, `print_body`, and `print_signature`. Each concrete class derived from the base strategy class can override each point-of-generation to generate the appropriate artifact(s). If a point-of-generation is not overridden, then the default method—usually an empty method—is used. Similar to the single interpretation implementation, the parsing logic for the DSML is implemented using the Visitor pattern. The main difference is that instead of coupling the interpreter logic with

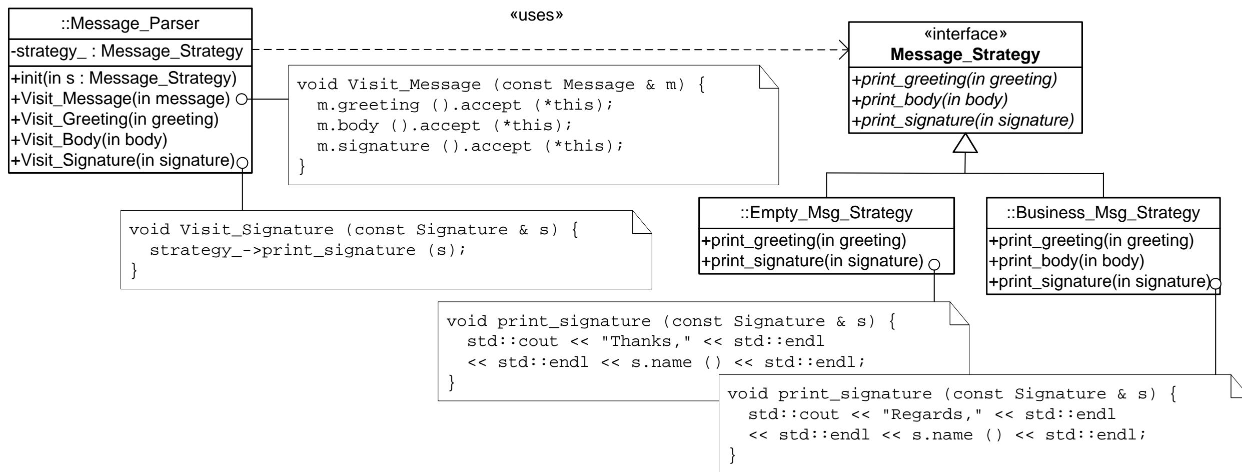

Fig. 3. Conceptual model of a strategized interpreter's implementation.

the generation logic, the parser (i.e., `Message_Parser`) contains a base pointer to the appropriate strategy for the generation logic (*i.e.*, `Message_Strategy`), and invokes the appropriate point-of-generation to generate the correct artifacts.

The advantage of strategized interpretation is that it avoids reinvention of core interpreter logic by decoupling interpretation and generation logic. The drawback of this approach is that all points-of-generation are invoked for each strategy—even if it is not explicitly overridden. As illustrated in Figure 3, `Empty_Msg_Strategy` does not override `print_body`; however, `Message_Parser` is not aware that `Empty_Msg_Strategy` ignores this method. Instead, `Message_Parser` uses the default implementation for `print_body` because it contains a base pointer to a `Message_Strategy` object. Moreover, it is *hard* to modify the points-of-visitation (such as suppressing) similar to the way the points-of-generation are modifiable.

A special case of strategized interpretation is to use a template approach, which is called *parameterized strategy interpretation*. As illustrated in Figure 4, the main interpreter logic is written as a template class that is parameterizable by concrete types (*e.g.,* `Business_Msg_Strategy` or `Empty_Msg_Strategy`). Similar to the strategized interpreter, the interpreter logic is implemented by invoking points-of-generation on the template class, *i.e.,* `STRATEGY_TYPE`, where the template class is implemented using the strategy design pattern previously explained. This is similar to the Template Method design pattern [7].

The advantage of parameterized strategy interpretation is that each interpreter is capable of reusing the core interpreter logic and does not pay the penalty of invoking each point-of-generation. For example, `Empty_Msg_Strategy` is not concerned with the `print_body` point-of-generation and does not override this method. When `Message_Parser` is parameterized with `Empty_Msg_Strategy`, it will use the default implementation for `print_body`—an empty method.

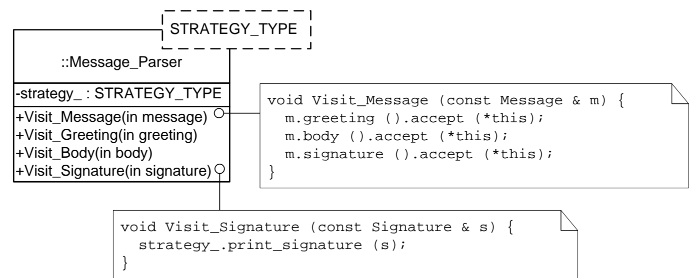

Fig. 4. Conceptual model of a template method interpreter's implementation.

Since `print_body` would be considered *dead code* [12], the C++ compiler will eliminate it from the executable as an optimization [13].

The disadvantage of this approach is that each interpreter is still not able to modify points-of-visitation because it reuses core interpreter logic that is often implemented for the general case. For example, `Empty_Msg_Strategy` is not able to suppress visiting the Body element because the `accept` method is performing a double-dispatch on `Message_Parser` through its virtual table [7]. Although `Empty_Msg_Strategy` will not generate any metadata in `Visit_Body` (not pictured), the method is unnecessarily invoked.[1]

Challenges in Enabling Interpreter Logic Reuse

Table I summarizes the advantages and disadvantages of the current techniques for implementing visitor-based model interpreters discussed above. There exist two main techniques for implementing the interpreter logic of a model interpreter using the Visitor pattern. The most simple and usually the most common method is single interpretation. Although this method

[1]It can be argued that runtime complexity of a model interpreter does not matter as long as it generates artifacts faster than a human who does it manually. Ideally, however, you do not want simple interpreters to incur the same runtime complexity as complex interpreters when reusing interpreter logic—especially when traversing large models.

TABLE I

TABLE SUMMARIZING THE ADVANTAGES AND DISADVANTAGES OF DIFFERENT TECHNIQUES FOR IMPLEMENTING VISITOR-BASED MODEL INTERPRETERS WHILE TRYING TO PROMOTE MAXIMAL REUSE OF CORE INTERPRETER LOGIC AND IMPLEMENT THE MOST SUCCINCT MODEL INTERPRETER FOR DIFFERENT NEEDS

Technique	Reuse of Core Interpretation Logic	Suppress Points-of-Generation	Suppress Points-of-Visitation
Single Interpretation			
Strategized Interpretation	X		
Parameterized Strategy Interpretation	X	X	

is simple, it forces developers to reinvent the interpreter logic on a per interpreter basis (as illustrated in Figure 2). To address the challenge of reuse in interpreter logic, it is possible to use parameterized strategy implementation. The main limitation of this approach is its difficultly to customize points-of-visitation.

In order to ensure reusability of interpreter logic across multiple DSML model interpreters (including composite model interpreters), developers must address the key challenge of customizing the interpreter logic without physically modifying it (*i.e.,* changing its source code). Different model interpreters that use the same interpreter logic may not be interested in the same points-of-generation and points-of-visitation as other model interpreters. As discussed above, the strategized interpretation partially addresses the challenge of promoting reuse in the interpreter logic. It promotes reusability because it separates the interpreter logic from the generation logic. It, however, disallows developers to strategically alter points-of-visitation.

When striving to achieve reuse of the interpreter logic, it is ideal to implement it for the general-case and let it interact with all the necessary model elements in a predetermined order. This will allow as many interpreter implementations as possible to reuse the interpreter logic. As shown in Section II, the `Empty_Msg_Strategy` interpreter does not handle the same number of elements as the `Business_Msg_Strategy` interpreter. In order to promote reuse of interpreter logic, Visitor-based model interpreters need a technique to transparently customize interpreter logic for their needs. In addition, the customization must be done without altering existing interpreters that use the same interpreter logic.

In the case of the generation logic, different interpreters generate different artifacts from a DSML. As illustrated in Section II, the `Empty_Msg_Strategy` interpreter generates different artifacts from the `Business_Msg_Strategy` interpreter. If the interpreter logic is implemented for the general case to promote reusability, it must also facilitate customization of its generation logic since the reusable interpreter logic will not know how to generate correct artifacts for each individual model interpreter. Successful reuse of the interpreter logic using the Visitor design pattern, therefore, should allow specialization of points-of-generation and points-of-visitation while keeping the two decoupled.

III. DETAILED DESIGN AND FUNCTIONALITY OF MIME

This section discusses MIME's approach to resolving the challenge of reinventing interpreter logic in Visitor-based model interpreters. MIME builds on the parameterized strategy implementation technique discussed in Section II, but uses template metaprogramming techniques to address its limitations.

A. Specializing Visitation Logic via Template Metaprogramming

Specialization of the interpreter logic is the ability to transparently customize how the model is traversed without actually modifying the existing interpreter logic. In order to promote specialization of the interpreter logic one must first determine the different points-of-visitation. For example, in Figure 1 there are three points-of-visitation: `Visit_Body`, `Visit_Greeting`, and `Visit_Signature`.

Some model interpreters may want to visit all the elements in the model; whereas, other model interpreters may only care to visit certain elements, such as not visiting the Body element as exemplified by the `Empty_Msg_Strategy` interpreter. In order to facilitate transparent customization of interpreter logic on a per interpreter basis, MIME applies template metaprogramming techniques to the points-of-visitation, as shown in Figure 5.

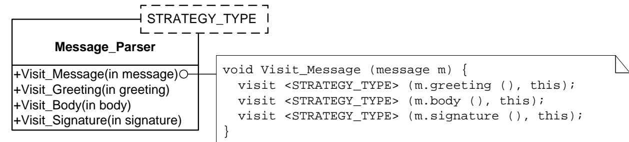

Fig. 5. Conceptual model of using template metaprogramming to specialize points-of-visitation.

As alluded to in Figure 5, each point-of-visitation is wrapped by a parameterizable `visit` function. The `visit` function determines if the element of interest is visitable by the interpreter of `STRATEGY_TYPE` type. If the `STRATEGY_TYPE` interpreter is interested in the specified element type, then the interpreter logic will visit it. Otherwise, the interpreter logic will ignore it (*i.e.*, remove its implementation at compile time).

```
1  // legend:
2  // S = interpreter; T = element(s); V = visitor
3  template <typename S, typename T>
4  struct visit_type {
5    static const bool result_type = true;
6  };
7
8  template <typename S, typename T, typename V>
9  inline bool visit (T t, V v) {
10   if (!visit_type <S, T>::result_type)
11     return false;
12
```

```
13     t.accept (v); // visit the element
14     return true;
15   }
```

Listing 1. Example point-of-visitation template.

Listing 1 illustrates an example `visit` function in MIME. As shown in this listing, each model interpreter can control whether or not to visit a particular element by specializing `visit_type`. By default, each element is visited. MIME also provides visit functions that iterate over a collection of elements. Finally, each visit function can provide model domain-specific functionality through customization using template specialization techniques.

Applying visitation logic templates to message example. Listing 2 shows a code snippet of the message example that uses the template approach highlighted in Figure 5. As illustrated in this listing, `Message_Parser` is implemented as a template class that is parameterizable by the concrete interpreter's type (*e.g.,* `Business_Msg_Strategy` or `Empty_Msg_Strategy`). The `visit` function is then used to insert customizable points-of-visitation into the interpreter logic (*i.e.,* lines 5–7). By default, all elements will be visited by the interpreter logic. Since the `Empty_Msg_Strategy` interpreter does not visit Body elements, it specializes the `visit_type` trait. This causes `Message_Parser`'s interpreter logic to suppress that point-of-visitation when constructing the implementation of the `Visit_Message` method for `Empty_Msg_Strategy`.

```
1   // Message_Parser_T.cpp
2   template <typename S>
3   void Message_Parser <S>::
4   Visit_Message (const Message & m) {
5     visit <S> (m.Greeting (), *this);
6     visit <S> (m.Body (), *this);
7     visit <S> (m.Signature (), *this);
8   }
9
10  // Empty_Message.h
11  template < >
12  struct visit_type <Empty_Msg_Strategy, Body> {
13    static const bool result_type = false;
14  };
15
16  typedef Message_Parser <Empty_Msg_Strategy>
17    Empty_Message_Parser;
```

Listing 2. Applying a points-of-visitation template to message example.

Because of MIME's points-of-visitation template metaprogramming technique, both example model interpreters can use the same interpreter logic. Moreover, the each model interpreter can transparently specialize the interpreter logic without impacting the other model interpreter.

B. Specializing Generation Logic via Template Metaprogramming

The previous section discussed how template metaprogramming is used to promote reuse of interpreter logic for different model interpreters. This generalized interpreter logic, however, did not contain any generation-logic because different interpreters generate different artifacts. If the generalized interpreter logic contained generation-logic, then it would mean all interpreters that used the same interpreter logic would generate similar artifacts, which is not always the case.

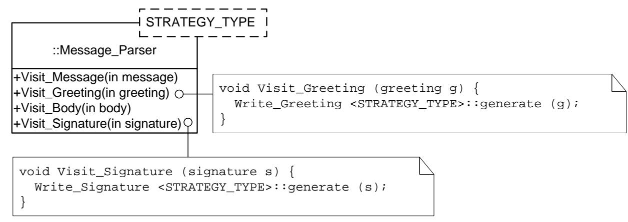

Fig. 6. Conceptual model of using template metaprogramming to specialize points-of-generation.

Similar to how template metaprogramming is used to specialize points-of-visitation, it can also be used to specialize points-of-generation. Figure 6 illustrates MIME's approach for specializing points-of-generation. As highlighted in this figure, each point-of-generation is an object that contains a static `generate` function. The point-of-generation object is parameterized by the `STRATEGY_TYPE`. If the point-of-generation is not specialized, then the default implementation of the point-of-generation—usually an empty method whose implementation is suppressed at compile time—is used.

It is possible to use the parameterizable strategy interpreter technique (see Section II) to implement specialization of points-of-generation. By specifying point-of-generations in parameterizable objects, however, it is possible to define traits that explicitly suppress points-of-generation, similar to the points-of-visitation. This approach, therefore, provides greater flexibility and control for customizing the generalized interpreter logic on a per interpreter basis.

Applying generation logic templates to message example. Listing 3 contains a code snippet of the message example that uses MIME's approach illustrated in Figure 6 to customize points-of-generation on a per model interpreter basis. As shown in this listing, `Write_Body` and `Write_Signature` are two points-of-generation that can be specialized by concrete interpreters to generate the appropriate content.

```
1   template <typename S>
2   struct Write_Body {
3     static bool generate (const Body &) {
4       return false;
5     }
6   };
7
8   template <typename S>
9   struct Write_Signature {
10    static bool generate (const Signature &) {
11      return false;
12    }
13  };
```

Listing 3. Parameterizable points-of-generation in the Message example.

Listing 4 shows how the points-of-generation in Listing 3 are integrated into the interpreter logic. As shown in this listing, `Visit_Body` (line 3) and `Visit_Signature` (line 9) invoke their respective generation methods. Both `Business_Msg_Strategy` and `Empty_Msg_Strategy` implement the

`Write_Signature` point-of-generation (line 28 & 43, respectively). The `Empty_Msg_Strategy` interpreter does not visit `Body` elements (see Listing 2, line 12). `Empty_Msg_Strategy` therefore does not implement the `Write_Body` point-of-generation. On the other hand, `Business_Msg_Strategy` specializes the `Write_Body` point-of-generation (line 19), which enables its interpreter to generate the necessary content for the `Body` element.

```
1  // Message_Parser_T.cpp
2  template <typename S>
3  void Message_Parser::
4  Visit_Body (const Body & b) {
5    Write_Body <S>::generate (b);
6  }
7
8  template <typename S>
9  void Message_Parser::
10 Visit_Signature (const Signature & s) {
11   Write_Signature <S>::generate (s);
12 }
13
14 // Business_Message.h
15 struct Business_Msg_Strategy {
16   static std::ofstream file;
17 };
18
19 template < >
20 struct Write_Body <Business_Msg_Strategy> {
21   static bool generate (const Body & b) {
22     typedef Business_Msg_Strategy BMS;
23     BMS::file << "\n" << b.text () << "\n";
24     return true;
25   }
26 };
27
28 template < >
29 struct Write_Signature <Business_Msg_Strategy> {
30   static bool generate (const Signature & s) {
31     typedef Business_Msg_Strategy BMS;
32     BMS::file << "Regards,\n\n" << s.name () << "\n";
33     return true;
34   }
35 };
36
37 // Empty_Message.h
38 struct Empty_Msg_Strategy {
39   static std::ofstream file;
40 };
41
42 template < >
43 struct Write_Signature <Empty_Msg_Strategy> {
44   static bool generate (const Signature & s) {
45     Empty_Msg_Strategy::file
46       << "Thanks,\n\n" << s.name () << "\n";
47     return true;
48   }
49 };
```

Listing 4. Applying points-of-generation templates to message example.

Because of MIME's point-of-generation template technique, it is possible to reuse the generation-logic with both example interpreters. Moreover, it is possible to transparently specialize the generation-logic of the either model interpreter without affecting each other.

C. Reusing the Metaprogrammable Interpreter Logic within Composite DSML Model Interpreters

Composite DSMLs are those created from one or more existing DSMLs. This allows the parent DSML (*i.e.,* the one created from multiple DSMLs) to use elements from the child DSML when constructing valid models. One of the main benefits of this modeling technique is reuse of existing modeling languages that capture a domain.

Although DSMLs can be composed from other DSMLs, the parent DSML interpreter is responsible for interpreting the child DSML. It is ideal for the parent DSML to reuse the interpreter logic of the child DSML. This would eliminate the need for parent DSML from having to reinvent the interpreter logic of the child DSML, which can be a complex task if the child DSML is complex.

In Section II we showed how current techniques for implementing DSML model interpreters using Visitor pattern do not support reuse of interpreter logic for single DSMLs. Existing techniques for implementing DSML model interpreters, therefore, do not fully meet the needs of composite DSML model interpreters. In Section III-A, however, we showed how MIME can support interpreter logic reuse.

Because MIME's implementation techniques allowed us to reuse the interpreter logic for single DSMLs, the interpreter logic of the child DSML can also be reused by the composite DSML. To illustrate this concept, we have created a simple composite DSML that uses the Message DSML introduced in Figure 1 of Section II. Figure 7 illustrates our composite DSML example that we created in GME.

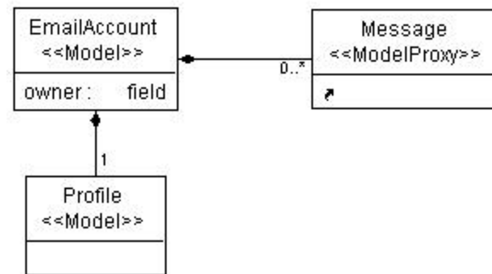

Fig. 7. Example composite DSML created in GME.

As illustrated in Figure 7, the `EmailAccount` model element has one attribute named `owner` and contains two elements: `Profile` and `Message`. The `Profile` element is a GME Model element that contains other elements (not shown in Figure 7) to provide details about the email account. The `Message` element is a GME Model Proxy (*i.e.,* a placeholder for external models) that references the `Message` GME Model element introduced in the DSML illustrated in Figure 1 of Section II.

Although the composite DSML presented in Figure 7 is a simple DSML (*i.e.,* contains few elements), the composite DSML model interpreter is responsible for implementing the interpreter logic for both the parent and child DSML. If we apply MIME's implementation technique, we can then reuse the interpreter logic of the child DSML (*i.e.*, Listing 2) to simplify implementing the composite's interpreter logic. Listing 5 shows a code snippet of the composite DSML model interpreter for Figure 7 implemented using MIME.

As illustrated in Listing 5, we implemented a single interpretation style interpreter for the `EmailAccount` DSML.

When the interpreter visits `EmailAccount` elements (line 3), it first visits the contained Profile element (line 4). After visiting the Profile element, it interprets the contained Message elements (or DSML). Because the child DSML (*i.e.,* the Message DSML) provides a reusable interpreter logic, the `EmailAccount_Parser` reuses `Message_Parser` by parameterizing `Message_Parser` with itself. This allows `EmailAccount_Parser` to customize the points-of-visitation and points-of-generation in `Message_Parser` as needed using the techniques discussed in previous sections.

```
1  // EmailAccount_Parser.cpp
2  void EmailAccount_Parser::
3  Visit_EmailAccount (const EmailAccount & e) {
4    e.Profile ().accept (*this);
5
6    // reuse of Message_Parser
7    Message_Parser <EmailAccount_Parser> generator;
8    e.Message ().accept (generator);
9  }
```

Listing 5. Example of a composite model interpreter.

Although we use the single interpretation technique in our composite DSML model interpreter, it is also possible to use MIME's template metaprogramming technique to promote reuse. Similar to the `Message_Parser` implementation, `EmailAccount_Parser` would be declared as a template class. Likewise, instead of directly invoking the accept method on the Profile element (line 4), the interpreter logic would apply the templates presented in Section III-A. Finally, instead of parameterizing `Message_Parser` with itself (line 7), it would be parameterized with `EmailAccount_Parser`'s template parameter. This would allow reuse of the `EmailAccount_Parser`'s interpreter logic by other `EmailAccount` model interpreters and DSMLs that use `EmailAccount` as a child DSML. Listing 6 shows the composite model interpreter in Listing 5 updated to use MIME's template metaprogramming technique.

```
1  // EmailAccount_Parser_T.cpp
2
3  template <typename S>
4  void EmailAccount_Parser <S>::
5  Visit_EmailAccount (const EmailAccount & e) {
6    visit <S> (e.Profile (), *this);
7
8    // reuse of Message_Parser
9    Message_Parser <S> generator;
10   visit <S> (e.Message (), generator);
11 }
```

Listing 6. Example of a parameterized composite model interpreter.

D. Generalization of the MIME's Implementation Technique

Situations will arise that offer an opportunity to add customization to interpreter logic that the current templates discussed above can not handle. For example, one interpreter may want to visit the elements of a container in reverse order; whereas, another interpreter may want to set an upper bound on the number of elements visited in a container, or change the container type. To address this concern, the following are guidelines for handling such situations:

1) Create a trait class or function with a name that reflects its purpose. This makes it easier to understand how it should be used.
2) Regardless of using a trait class or a template function, the first template parameter is the concrete interpreter's type. This allows the concrete interpreter to be the primary specialization artifact.
3) If there are any remaining template parameters, they are classified as properties that can influence the template's final value (or behavior) via template specialization.

E. Simplifying MIME's Customization Techniques

One major challenge of template metaprogramming is comprehending its implementation. Model interpreters that leverage MIME's implementation technique will have vast amounts of template logic embedded throughout its implementation. Although a design goal is to make it easy to identify and understand the templates used by MIME, it may not appear as simple to novice developers.

To address this concern, it is feasible to define a high-level language with constructs that allow developers to define the general parsing logic for the DSML. This high-level parsing logic would then be transformed into a interpreter that uses MIME's implementation technique, which is similar to template-based generators operator. This would also alleviate the need for understanding how to write templates for MIME, and make MIME's implementation technique more acceptable to developers who have little experience with template metaprogramming.

IV. Use Case: Generating Different Metadata Formats from the Same Behavior and Workload DSML

This section showcases how MIME's approach was applied to a more complex and realistic DSML.

A. Overview of the Component Behavior Modeling Language and the Workload Modeling Language

The Component Behavior Modeling Language (CBML) and the Workload Modeling Language (WML) [5] are two DSMLs we developed in GME. CBML is a DSML for capturing the behavior (*i.e.,* the actions and states) of a component at a high-level of abstraction. WML is a DSML for capturing the workload of operations (or actions). WML complements CBML because the workload modeled in WML can parameterize the operations (or actions) in CBML to create behavior that has realistic workloads.

CBML and WML are both complex DSMLs in terms of the number of modeling elements. For example, CBML contains 69 different GME modeling elements and attributes combined that model interpreters can interpret. WML is less complex in that it contains only 19 different modeling elements and attributes combined. Lastly, both languages tend to evolve as more is learned about their domain.

CBML and WML together are primarily used to generate source code for component-based distributed systems [5] and

TABLE II
TABLE COMPARING MIME'S APPROACH AGAINST CURRENT TECHNIQUES FOR IMPLEMENTING VISITOR-BASED MODEL INTERPRETERS

Technique	Reuse of Core Interpretation Logic	Suppress Points-of-Generation	Suppress Points-of-Visitation
Single Interpretation			
Strategized Interpretation	X		
Parameterized Strategy Interpretation	X	X	
MIME	X	X	X

early integration testing [14]. Although the primary use of CBML and WML is generating source code, the following are other interpretation needs for both DSMLs:

- **Documentation.** Users want the option to generate documentation of the behavior and workload in a natural language (such as English). This will help simplify documenting the behavior and workload of individual components and the entire system for technical reports.
- **Real-time scheduling analysis.** Several users want to generate configuration files for proprietary real-time scheduling analysis tools. This will allow them to verify if a system with the modeled behavior and workload will honor deadlines under different real-time scheduling policies.
- **Alternative architectures.** Stakeholders want to use CBML and WML to capture the behavior of Data Distribution Service [15] and Infospherics Common API (www.infospherics.org/api) client applications. This will enable them to rapidly create and generate client implementation code for different architectures instead of writing it manually by hand.

Because we have identified different interpretation needs for CBML and WML we realized it would be easier to reuse existing interpreter logic, as opposed to reinventing it. Table II shows how MIME's technique compares against the current approaches for implementing Visitor-based model interpreters. Because of the needs discussed above and the information presented in Table II, MIME's technique is an appropriate solution for reusing core interpreter logic while customizing it for the different interpretation needs.

B. Interpreter Reuse in CBML and WML using MIME

It is clear that each interpreter requires its own interpreter logic that generates artifacts based on the needs presented in Section IV-A. Each interpreter is also not interested in the same elements. For example, the C++ code generator interpreter preprocesses the model to gather the required information when generating C++ source code; whereas the documentation interpreter does not need to preprocess the model to generate a textual description of the behavior and workload. The real-time scheduling interpreter is only concerned with the actions and workload in the behavior; whereas the XML-based interpreter is concerned with every element in the behavior and workload model.

Because all interpreters are similar with different requirements, MIME's technique was used to implement the interpreter logic once and reuse it for each concrete interpreter. The following steps, which are also illustrated in Listing 7, were used to implement the interpreter logic for CBML and WML:

1) (line 1) The interpreter logic was initially implemented in terms of the C++ source code generator since it was the initial use case. At this point, the generation logic was coupled with the interpreter logic to acting as a placeholder for respective points-of-generation.
2) (line 13) The single interpreter was converted into a template method interpreter that is parameterized by the concrete interpreter's type (line 14). Instead of making direct calls on the template parameter's methods, point-of-generation were inserted to allow customization (line 20). The concrete template can opt to define this point-of-generation.
3) (line 26) Finally, all the direct visitor method calls on elements were transformed into points-of-visitation. This allows each model interpreter to selectively enable and disable them as needed (line 36).

```
1  // step 1 (single C++ version)
2  void CUTS_BE_Execution_Visitor::
3  Visit_Effect (const PICML::Effect & effect) {
4    std::string postcondition = effect.Postcondition ();
5
6    if (!postcondition.empty ())
7      outfile << postcondition << ";" << std::endl;
8
9    // visit next state in the behavior
10   effect.dstEffect_end ().Accept (*this);
11 }
12
13 // step 2 (parameterize point-of-generation)
14 template <typename S>
15 void CUTS_BE_Execution_Visitor <S>::
16 Visit_Effect (const PICML::Effect & effect) {
17   std::string postcondition = effect.Postcondition ();
18
19   if (!postcondition.empty ())
20     Postcondition <S>::generate (postcondition);
21
22   // visit next state in the behavior
23   effect.dstEffect_end ().Accept (*this);
24 }
25
26 // step 3 (parameterize point-of-visitation)
27 template <typename S>
28 void CUTS_BE_Execution_Visitor <S>::
29 Visit_Effect (const PICML::Effect & effect) {
30   std::string postcondition = effect.Postcondition ();
31
32   if (!postcondition.empty ())
33     Postcondition <S>::generate (postcondition);
34
35   // visit next state in the behavior
36   visit <S> (effect.dstEffect_end (), *this);
37 }
```

Listing 7. Code snippets of steps used to implement CBML and WML interpreter logic using template metaprogramming.

The final interpreter logic for CBML and WML spanned 32 files and contained approximately 1,000 source lines of

code (SLOC).[2] Due to the generality of the interpreter logic, it contains 39 points-of-visitation and 50 points-of-generation.[3] This creates a total of 1950 different customizations for the implementation (although it is clear many will not be used) because each point-of-visitation and generation is customizable on a per interpreter basis.

The C++ interpreter uses all the defined points-of-visitation and generation. The documentation interpreter uses all the points-of-visitation, but uses only 19 points-of-generation. The XML-based and real-time scheduling model interpreters are currently under development. It is safe to assume they will use fewer points-of-generation than the C++ interpreter and possibly use fewer points-of-visitation. Although each interpreter uses a different number of points-of-generation and interpretation, the key fact is that each interpreter reuses the same interpreter logic. Each interpreter is neither rewriting the interpreter logic each time nor suffering from the limitations of existing Visitor-based model interpreters.

V. Related Work

Visitor implementation. Nguyen et al. [16] present a technique for addressing the limitations of Visitor-based implementations in the context of grammar parsers to create flexible and extensible parsers. In their technique, when a term in the grammar is visited, the parsing token uses the Abstract Factory design pattern [7] to generate the correct visitor for that token. This provides specialization on a per term basis. MIME is similar to theirs because it can strategize the visitation of each element type in a model (similar to a term in a grammar) and the artifacts generated from the model element. MIME differs from their approach since it completely decouples generation logic from the visitation logic. When Nguyen's technique is applied to large models, it suffers from the excessive memory allocation anti-pattern [17]. MIME does not incur this overhead.

Neff [18] and Schordan [19] present a technique for implementing grammar parsers using the *bivisit* Visitor design pattern. It allows them to perform preprocessing/postprocessing operations before/after visiting terms in the grammar. The visitation logic however is still coupled with the generation logic. The bivisit variant of the Visitor can be used in conjunction with aspects in C++ [11] to achieve similar goals as MIME. MIME is also able to separate the generation logic from the visitation logic. Moreover, MIME can transparently customize the generation and visitation logic on a per use case basis.

Model interpretation. AndroMDA [20] and oAW [21] provide tools that allow developers to implement model interpreters using template parsing and workflow engines. Developers use high-level constructs that dictate how to transform existing models into artifacts. MIME is similar to both AndroMDA and oAW because all three rely on generative programming techniques to implement model interpreters. We have learned, however, that reusability is less of an issue in these tools. Finally, MIME is tailored to Visitor-based model interpreters.

Agrawal et al. [22] and Muller et al. [23] each present a technique that use DSMLs and model transformations to implement model interpreters. Developers use a DSML that models how to transform existing models into artifacts. Model interpreters are then generated that perform the specified transformations. Although their technique is a level of abstraction higher, neither has explicitly addressed reusing transformations and generated interpreters within other use cases as MIME did with the Visitor design pattern.

[2] SLOC were obtained using SourceMonitor (http://www.campwoodsw.com/sourcemonitor.html).

[3] We manually counted the number of points-of-generation and visitation.

VI. Concluding Remarks

This paper presented a generative programming technique, called MIME, that used C++ template metaprogramming to promote reuse in Visitor-based model interpeters. The main goal of MIME is to enable reuse of core interpreter logic so that different interpreters for the same DSML with the same interpretation goal do not have to reinvent it. Each individual model interpreter can then customize the interpreter logic to meet its needs while reusing the core interpreter logic. Based on experience using MIME to implement several Visitor-based model interpreters, the following lessons were learned:

- If a DSML has only one use case, then single interpretation is the preferred implementation technique for its model interpreters. If multiple use cases have been identified, then the template metaprogramming technique is the preferred implementation technique for Visitor-based model interpreters.
- It is not *hard* to convert a single interpretation style interpreter to a template metaprogramming style interpreter because the main interpreter logic is already defined. It is "harder" to create a template metaprogramming interpreter from scratch because the different use cases for interpreters may not fully be understood a priori.
- C++ template metaprogramming incurs a steep learning curve for developers. When combined with a complex visitor hierarchy generated by modeling tools for a given modeling language, significant efforts must be expended to develop the interpreters. It is desirable for modeling frameworks to hide the complex visitor hierarchy but expose only the points-of-visitation and points-of-generation to the developers, which can enable rapid development of interpreters. This forms part of our future work.

MIME's templates has been integrated into the CUTS system execution modeling tool and is available in open-source format for download from the following location: http://cuts.cs.iupui.edu.

References

[1] Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-Integrated Development of Embedded Software. Proceedings of the IEEE **91**(1) (January 2003) 145–164

[2] Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer **34**(11) (2001) 44–51

[3] White, J., Schmidt, D., Gokhale, A.: Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study. In: MODELS 2006: 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, IEEE/ACM, ACM Press (October 2005)

[4] Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, New York (2004)

[5] Hill, J.H., Tambe, S., Gokhale, A.: Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems. In: Proceedings of 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 07), Tucson, AZ (Mar 2007) 307–316

[6] Alonso, D., Vicente-Chicote, C., Barais, O.: V^3Studio: A Component-Based Architecture Modeling Language. In: Proceedings of the 15th Internatoinal Conference and Workshop on the Engineering of Computer-based Systems (ECBS '05). (2008) 346–355

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)

[8] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley, Reading, MA (2003)

[9] google-ctemplate: google-ctemplate. code.google.com/p/google-ctemplate (2007)

[10] Martin, R.: Design principles and design patterns. Object Mentor (2000)

[11] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, Massachusetts (2000)

[12] Chen, Y.F.R., Gansner, E.R., Koutsofios, E.: A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In: Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, Springer-Verlag New York, Inc. (1997) 414–431

[13] Veldhuizen, T.L.: Five Compilation Models for C++ Templates. In: First Workshop on C++ Template Programming, Erfurt, Germany (October 2000)

[14] Hill, J.H., Slaby, J., Baker, S., Schmidt, D.C.: Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In: Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications, Sydney, Australia (August 2006)

[15] Object Management Group: Data Distribution Service for Real-time Systems Specification. 1.2 edn. (January 2007)

[16] Nguyen, D.Z., Ricken, M., Wong, S.: Design Patterns for Parsing. In: Proceedings of the 36th Technical Symposium on Computer Science Education (SIGCSE 05), New York, NY, USA, ACM Press (2005) 477–481

[17] Smith, C., Williams, L.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley Professional, Boston, MA, USA (September 2001)

[18] Neff, N.: Attribute Based Compiler Implemented Using Visitor Pattern. In: Proceedings of the 35th Technical Symposium on Computer Science Education (SIGCSE 04), New York, NY, USA, ACM Press (2004) 130–134

[19] Schordan, M.: The Language of the Visitor Design Pattern. Journal of Universal Computer Science **12**(7) (2006) 849–867

[20] Kozikowski, J.: A Bird's Eye View of AndroMDA. galaxy.andromda.org/docs-3.1/contrib/birds-eye-view.html

[21] openArchitectureWare: openArchitectureWare. www.openarchitectureware.org (2007)

[22] Agrawal, A., Levendovszky, T., Sprinkle, J., Shi, F., Karsai, G.: Generative Programming via Graph Transformations in the Model-Driven Architecture. In: Workshop on Generative Techniques in the Context of Model Driven Architecture (OOPSLA 02). (2002)

[23] Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: International Conference on Modeling Driven Engineering and Languages Symposium (MoDELS 2006). (2006) 98–110