Structured Specification of Model Interpreters

Gabor Karsai
Institute for Software-Integrated Systems
Vanderbilt University
PO-Box 1829
Nashville, TN 37235,USA
gabor@vuse.vanderbilt.edu

Abstract

Model interpreters play an essential role in model-integrated systems: they transform domain-specific models into executable models. The state-ot-the-art of model interpreter writing needs to be advanced to enhance the reusability and maintainability of this software. This paper presents an approach which makes this possible through the use of structured specifications. These specifications let the programmer express traversal strategies and visitation actions in very high-level terms. From these specifications efficient traversal code can be automatically generated.

1. Introduction

Model-integrated computing [SZ97] relies on the interpretation and use of domain-specific models in run-time environments. The domain models can be considered as objects that are mapped into run-time objects. The mapping can take many forms, ranging from configuring the attribute values of run-time objects to actual generation of code that defines classes and creates instances of run-time objects. This mapping process is performed by a component called the model interpreter that acts as a transformation engine. While the input of the interpreter is known (the model objects), the output is difficult to define in general: it can be, for instance, a text file, a list of objects created in a running system, or a sequence of messages in a distributed system. The exact nature of the output of the interpreter is specific to the domain and the run-time environment.

Writing a model interpreter is a non-trivial task. One has to understand the structure of the models, (i.e. the *data model* of the model database), the intricate details of the expected output, and the relationship between the two. Next, this understanding has to be translated into software that performs the desired mapping. Additionally, the software has to perform the transformation with reasonable performance.

The model interpretation process is somewhat similar to the back-end of compilers. The models capture information in a structured form, typically in the form of hierarchically organized objects. This graph of objects should be traversed, perhaps transformed, and output generated. While the process is very easy to describe in general, it is highly non-obvious how it can be implemented.

This paper shows a generic technique, which helps in the writing of model interpreters by offering a high-level, concise notation for capturing the relevant steps of an interpreter in a structured form. The technique does *not* generate the entire the model interpreter. This would be a rather impossible task because of the widely different outputs expected from an interpreter. Instead, it focuses on the “mechanistic aspects” of model interpretation and simplifies the task of the interpreter writer by generating a large and uninteresting portion of the interpreter code automatically.

2. Background

Model interpreters are transformation programs that walk a graph (the model objects), and perform actions during this process. This activity is, of course, performed routinely in various software systems. Indeed, probably it is fair to say that it is one of the most frequently occurring tasks in any system that transforms data.

Attribute Grammars

The first and foremost application of graph traversal and actions is the code generator part, the “back-end”, of compilers [AH86]. After building the syntax tree from the input text, compilers perform various analysis steps on the data structure (typically for the purpose of semantic checks), then traverse it and output the generated code. Compiler research literature provides a great source for efficient traversal and transformation algorithms. On the other hand, the area of automatically generated compilers provides some interesting technologies for the structured specification of graph traversals.

One approach, as a widely and successfully used one, is apparent: Attributed Grammars (AG) [AH86]. AGs, invented by Knuth, tie semantic specification to the syntactical rules of a programming language. Suppose the syntax of a language is specified in the form of context free grammar, with production rules, terminal and nonterminal symbols, and a start symbol. Now the parser stage of a compiler builds a syntax tree from the input. This represents, in a tree form, what production rules have been applied from the grammar, starting from the start symbol. The application of these rules leads to the sequence of terminal symbols, which is equal to the input string. The syntax tree captures the syntactical structure of the input to the compiler, if the input was syntactically correct. Obviously, one grammatical production rule may appear many times in the tree, showing how a nonterminal (on the left side of the rule and at the local root in the tree) was used to "generate" non-terminals and terminals (on the right side of the rule and at the local leaves). With each symbol in the grammar, we can associate a set of *attribute values*, and in the rules we can define how these values are to be calculated. Through these calculations attribute values can depend on other attribute values, including attribute values of other symbols. Because attributes are attached to symbols in the production rules, they can be considered as values associated with the nodes in the parse tree. Attributes can be *inherited* or *synthesized*. The value of a synthesized attribute is calculated from the attribute values of the children of a node in the parse tree, while the value of an inherited attribute is calculated from the attribute values of the parent and the siblings of a node. Note that the inherited/synthesized properties of attributes implicitly define a *data dependency* among the attribute values. This dependency implicitly describes a traversal sequence on the nodes of the parse tree. Thus, attribute specifications determine how the tree must walked, and imply a visitation sequence for calculating attributes. Circular dependencies lead to infinite loops in the traversal, but these are the result of incorrect specifications. One can also insert into the attribute specifications any code to be executed when the traversal is performed. From the attribute specifications a traversal code can be generated that walks the tree and evaluates the attributes in the necessary order.

To summarize, AGs provide very high-level specification formalism for the traversal of a tree through the use of dependency among the attributes. Unfortunately, while intellectually appealing, AGs have very serious practical limitations. For a specific traversal sequence, it is highly non-trivial how the attributes should be set up and how should they depend on each other. Sometimes one has to introduce extra attributes just for forcing a particular kind of traversal. Referencing to attributes that are calculated at remote nodes in the parse tree is rather problematic. Thus, while Attributed Grammars offer a very high-level formalism for the structured specification of traversals of graphs, their usability is limited.

Adaptive Programming

When object-oriented languages started to gain acceptance, it has been observed that OO programs are structured very differently than "traditional" procedural programs [LIE96] Specifically, it has been noted that OO programs follow a pattern of *collaborations* where multiple objects of different classes cooperate to achieve a certain goal. Unlike in traditional approaches, complex behaviors are implemented by a set of simpler behaviors distributed over a set of classes of objects. This appears to be an essential property of all object-oriented approaches. This structure is both a benefit and a liability. It is beneficial because very complex behaviors can be built from trivial ones, but it is a liability because OO languages typically lack the syntactical constructs to express them. Adaptive Programming (AP) [LIE96] offers a solution, which also provides relevant techniques for the model interpretation problem as well. In AP, collaborations are expressed using two specifications: *class graphs* and *traversal strategy graphs*. Class Graphs describe what classes are available in the system, and how they are related to each other through inheritance and associations. Traversal Strategy Graphs are subgraphs of Class Graphs that also specify what the precise strategy is to traverse that subgraph. The strategy is a very compact and high–level specification of the traversal: it simply refers to the classes involved, omitting all implementation details. For example, if class A is associated with class B which is associated with class C, a strategy can simply specify "from A visit C", without mentioning intermediate classes. In addition to this specification one can also include code in the strategy which gets executed when the traversal happens. From the class graph and the strategy graph specifications, a tool can synthesize all the traversal code, which is distributed across classes as methods. From each strategy graph a set of methods is created (assigned to the classes), which implement the strategy. The automatic generation of this code removes the mundane tasks from the programmer: iterating over lists, invoking methods on objects in the list, and hand-coding the traversal of a quite complex graphs with the help of small, distributed methods. The code also incorporates the user-specified code fragments that are executed during traversal. The technology has been developed by Lieberherr and others, and has been termed as "Adaptive Programming" (AP).

With respect to the specification of model interpreters, we can recognize the relevance of AP as follows. AP solves the task of traversal specification in a compact and efficient way that has many applications in object-oriented programs. The actual traversal code is synthesized, and the user is not burdened with low-level details.

Intentional Programming

Intentional Programming, developed at Microsoft Research [SIM96], offers another paradigm for program development. The central thesis here is that software is written as a collection of *intentions*, which are then refined into actual implementations. The intentions capture what a programmer wants to "say" in a particular context, in a language independent manner. Once the intentions are expressed, the programmer (or the development environment) should refine those intentions into actual implementation. Technically, intentions are intermediate nodes in a parse tree. The refinement is expressed by specifying how the intentional data structure should be transformed into a structure that can be directly used in a code generator. This refinement is currently expressed in the form of actual code that performs the transformation. IP shows similarities to model interpretation in many respects. If the "models" stand for "intentions", the transformation of those into implementations is the task of the model interpreter. Unfortunately, the current implementation of IP offers a very low-level interface for implementing the transformation engines, i.e. their model interpreters.

The *Visitor* Design Pattern

The *Visitor* design pattern [GOF95] codifies a prototypical solution to a frequently occurring design problem: A graph consists of nodes of heterogeneous types. We need to traverse this graph, possibly multiple times, and perform operations on the node. For example, the graph can be the syntax tree generated in a compiler, and the actions can be "optimize" or "generate code". The solution is to encapsulate the operations in a set of *Visitor* classes that can visit nodes of specific types. Once a visitor is created, it can be "handed to" a graph node, which will invoke the proper visitation operation on the visitor. The graph node should also incorporate the actual traversal operation: it should "know" how its neighboring nodes should be visited. As a design pattern, *Visitor* can be implemented in many ways (none of which is supported directly by a tool like an AP). The most obvious implementations have serious shortcomings in terms of scalability, but the pattern is a conceptually powerful technique.

The *Visitor* design pattern shows what is important: separating structure (the graph) from the traversal of the structure (the Visitor object), and the encapsulation of the operations in the latter. However, it is merely a design pattern, and thus in itself does not offer a way for the structured, high-level specification of model interpreters.

From the above four background technologies one can draw the following requirements for model interpreter specification:

- There is a need for the formal, high-level specification of the traversal. This specification should be the input to a code generator that synthesizes the actual traversal code.
- The traversal specification must be explicit (for maintainability), and concise. All intermediate code (for iterations, etc.) should be automatically synthesized.
- There is a need for writing multiple interpreters for the same structure. Just like one can have multiple visitors for a syntax tree, one should be able to define multiple interpretations for the same models.
- The traversal and the operations to be taken during that traversal should encapsulated in classes. This encapsulation offers a context, which can be built dynamically as the traversal proceeds, and provides a way for capturing the "state" of the traversal.

3. The approach

Based on the observations made above, the following approach is proposed. When specifying a model interpreter, the following components should be defined:

- *Model structure.* The model structure defines what classes of model objects we have, and how they are related to each other. One can use, for example, UML class diagrams to express model structure. In this paper a simple textual language is used.
- *Traversals.* Traversals capture how the models should be traversed. The specification should address the following question: If we are at node of type X, which node do we go to next? These traversal specifications can be made very concise (as shown in AP), and the actual traversal code can be generated from them. Traversals are objects that encapsulate the traversal code fragments, and can also encapsulate state information.
- *Visitors.* Visitors capture the actions to be taken when visiting a node of a particular type. Visitors are also objects that encapsulate the operations to be performed, and they can also provide a context for the traversal.

These three components can be encapsulated as classes, as shown on Figure 3below.

The *Traversal* and *Visitor* objects are also directly linked to each other, and are operated in a co-routine like manner. Suppose the Traverser starts at a specific type of model node. Based on its specification, it determines how to follow pointers emanating from that type of node and call the visitor on the objects that the pointers are pointing to. The visitor might take an action, and/or activate the traverser to proceed from the accessed node. So the control flow oscillates between the traverser and the visitor: the traverser determines where to go next, the visitor "visits" (i.e. takes actions) and can call back the traverser to proceed further.

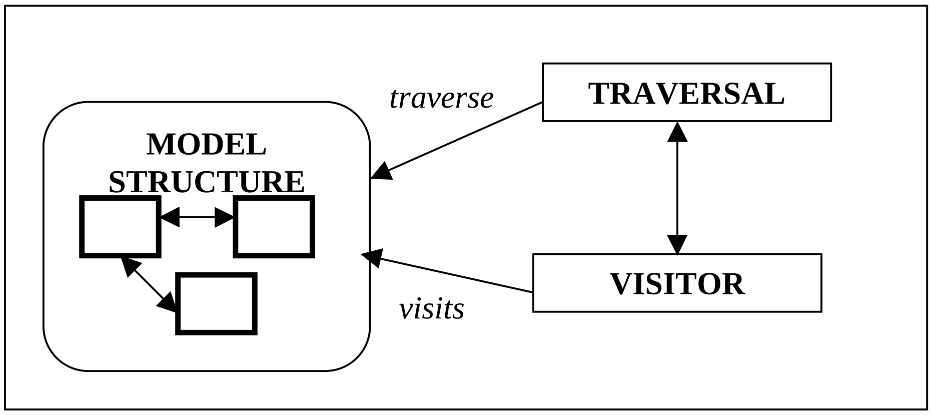

Figure 1:Components of a Model Interpreter

The *model structure specification* should capture what kind of model classes are available, how they are composed of simpler entities or other models, and how they are associated with each other (beyond composition). Models, entities, and relations might have attributes: key-value pairs that capture non-structured information. Thus, model structure can be easily specified using a standard specification technique; for instance UML class diagrams [FOW97]. For the sake of simplicity, we will use a simple; entity-relationship based textual language for specification. In the language, one can define entity types (which are named collections of attributes), relation types (that relate entities and models to each other), and model types (that contain entities, relations and possibly models). All types can have attributes and entity types, and model types can be organized into inheritance hierarchies. Figure 2 below shows a trivial specification. The specification introduces an entity, a model, and a relation, respectively. The relation is specified in terms of objects it relates (Signal to Signal) and the name of roles they play in the relations (src and dst).

Traversal specifications should answer the following question: “if we are at node of type X, where do we go next?” The “next” should be an object that is reachable from objects of type X. Thus it should be associated with X either directly or indirectly, possibly through inheritance. From “Compound”, for instance, one can access each “Dataflow” object via the “flows” association.

Thus, one traversal specification might be like “from Compound to flows”. This specification results in a code fragment in the interpreter, which is invoked when one wants to traverse a graph starting from a Compound node. When specifying a traversal, one specifies what visiting action to take *indirectly*: it is not explicit what to do, but the traverser “expects” that a corresponding visiting action is available. In a traversal specification one would also want to visit multiple nodes. For example “from Compound to {locals, blocks, flows}” might be a suitable specification. As was mentioned above, model and entity types can be organized in an inheritance hierarchy. In the example, Compound is a Block, inheriting attributes, parts, relations, and associations from the base type. When traversal of a derived type is specified, it is useful for specifying that the derived type objects should first be traversed as a base type object, for instance: “`from Compound do Block to {locals, blocks, flows}`”. This capability simplifies the specifications, because base class related traversals could be specified only once, and then invoked from derived class traversals.

```
entity Signal {
  attr string name;
}
model Compound : Block {
  part LocalSignal locals;
  part Block blocks;
  rel Dataflow flows;
}

relation Dataflow {
  Signal src * <-> Signal dst *;
  -- constraints for connections
}
```

Figure 2: Example Model Structure specification

Visitor specifications should capture what should be done when visiting a particular kind of object. There are basically two options: either take a “user action” (i.e. execute a piece of user-supplied code), or it can proceed with the traversal (i.e. call the traverser with the object being visited). These can be intermixed and/or omitted completely. The visitor specification should thus enumerate actions using the form:

```
at Dataflow <<USERCODE-1>> traverse
        <<USERCODE-2>>
```

Each clause after the type name is optional. The << and >> are special brackets which surround user-defined code. User-defined code can contain any C++ code to be executed at the start of the visit (USERCODE-1), or at the end of the visit (USERCODE-2).

A model interpreter generator can translate each traversal specification into a method of a Traverser class. The methods can take one parameter: a reference to the type of model object where the traversal starts. The visitor specifications can also be translated into methods of a Visitor class, which gets one parameter: a pointer to the object visited. One can even use the same name for the methods (e.g. visit() for Visitor methods, and traverse() for Traverser methods), because the C++ or Java overloading mechanism can correctly resolve the call based on the type of the parameter.

The overloading gives rise to an interesting capability: One can supply extra formal parameters in traversal and visitor specifications with the "origin point" of the action, and can supply actual parameters when "calling" a visitor or a traverser. An example is shown on Figure 3.

```
Traverser:
from Compound[int x] to blocks[2*x];
from Block[int j] to locals[j+1];

Visitor:
at Block[int x] traverse[x+3]
```

Figure 3: Specification with parameters

The parameters are simply added as extras for the generated method's parameter list, and, again, the overloading mechanism will be used to select the correct alternative.

User-defined actions can be added to the visitor specifications, as indicated above, but occasionally it is also useful to add them in traversal specifications. The syntax for traversals allows this in the following way:

```
from Compound do Block << USERCODE-1 >>
     to << USERCODE-2 >> locals
        << USERCODE-3 >> ;
```

USERCODE-1, USERCODE-2 are executed in sequence, and USERCODE-3 is executed after visiting all the LocalSignal nodes.

A translator program that generates C++ code processes the model structure, traversal and visitor specifications. The model structure can be translated into C++ class definitions, with attributes translated into class members, and relations into class objects that contain pointers to the related objects. This, of course, is just one possible translation: for example, an OODB schema can also be easily generated. The traverser specifications are translated into a Traverser class definition, with associated methods: one for each traversal specification. The methods contain code that iterates over the associated objects, and calls the corresponding visitor method. The Visitor methods contain the user-supplied code, and the (optional) call back to the Traverser for continuing with the traversal. Interestingly, the translation algorithm, which generates this code, can also be written quite easily using the Visitor/Traverser style.

4. Example

In this section a simple example is presented, which shows how to write a model interpreter for a block diagram language. The full specification can be found in the Appendix.

The models are for representing hierarchically organized processing networks (hardware or software). The modeling paradigm includes entity types called Ports, which are sub-classed into InputPorts and OutputPorts. The model type Block represents a "generic" processing module, which has inputs and outputs. This model type is sub-classed into Primitives and Compounds. Primitives define elementary processing operations, identified by a string attribute called type. Compounds are also blocks that contain other blocks (i.e. Primitives and Compounds), and relations of type Connection. Connections relate Ports to each other thus they can represent flow of data among processing blocks. There are no Block instances only Primitive or Compound instances. Furthermore, a Compound instance cannot contain itself.

The task of the model interpreter is to traverse the network of model objects, starting from a root Primitive or Compound. During traversal it has to print out each primitive instance (called a "node") encountered with a unique id and the type string, and wiring instructions that connect wires to nodes on the numbered ports of that node. Note that Compounds can contain other Compounds and Primitives, but in the output only the primitives are needed, with the "flat" wiring connecting them.

To show how the specified interpreter works, suppose we start at a Compound that has some input ports, some output ports. It contains one Primitive whose input ports are "wired" to the input ports of the parent Compound, and its output ports to the output ports of its parent. The traversal starts at the Compound, and the first specification here forces a traversal as if the object were a Block (using the do Block clause). Note that the Compound traversal specification expects one extra parameter, of type PortMap, which maps object ids (IDs) into connection ids (Wires). This parameter is passed along to the Block traversal specification. That specification visits the input and output ports of the object. The corresponding Visitor action checks if the selected port has an entry in the map, if not it creates a new Wire and assigns it to the object. Thus, the input and output ports will all have a Wire assigned to them. Next, the traversal continues by visiting all the Connections, and then all the Blocks contained in the Compound. When a Connection is visited, the Visitor code checks the Port objects (on both "ends" of the Connection) what parent model they belong to. If the Port belongs to the same parent as the Connection itself, it associates the Wire generated in the parent with the other Port object (via the PortMap). If neither of the Ports belongs to the parent of the Connection, this must be an "internal" wire that connects two ports belonging to blocks under the Compound. Thus, it creates a new Wire and assigns that to both ports. In terms of the example, these Visitor actions will result in new entries in the PortMap that associate the ports of the embedded Primitive with the already existing Wires. After visiting the Connections, the Traverser visits all Blocks. In the example there is only one block, of type Primitive, whose visit action calls back to the Traverser. The Traverser first traverses the Primitive as if it were a Block, which will not result in the creation of new Wires, because all its Ports must already have a Wire assigned to them. Next, a new Node is created (and a message is printed), and then the input and output Ports are visited again. But now a different visiting action is taken: one, which has extra parameters. This visiting action print out the "wiring commands", that indicate which Node should be connected to which Wire, and at which Port index.

5. Conclusions and Future Work

In this paper, we have shown a new approach to model interpreter specification. Traversals and Visits should be specified (in addition to the model structure). High-level notations can be used intermixed with user-supplied code. A tool has been developed that understands these specifications, and generates all the low-level traversal and visitation code.

The approach described in this paper is a highly practical one: its purpose is to serve the software engineer. This does not mean that the specifications cannot be thoroughly analyzed and important properties of traversals and visits determined. In fact, the tool mentioned above already performs these checks, and code generated by it is always correct. (Naturally, it cannot check the correctness of user-supplied, embedded code.)

The approach can be extended in many different directions. One is to incorporate *constraints* in the specification that can be used to determine semantic correctness of models. Much of the work in a real model interpreter deals with validating model correctness, and the formal constraint specifications could help in this. Another issue is the sequencing and precise control of traversals. Currently the tool supports *phases*, which are distinct passes through the model structure. It is the main program's responsibility to switch between the phases. Instead of using phases, one might use *conditional traversals/visits*, which are executed only when some conditions are true.

Specifying model interpreters in a structured way is key component for interpreter writing. While hand-coded actions may be needed for a long time, to impose a framework on the construction of interpreters offers long-term gains, especially in maintainability and code reuse.

Acknowledgment

The DARPA/ITO EDCS program (F30602-96-2-0227), The Boeing Company, Saturn Corporation, and the Arnold Engineering Development Center of USAF has supported the activities described in this paper.

References

[AH86] Aho,A., Sethi,R., Ullmann: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[FOW97]Fowler,M: *UML Distilled*, Addison-Wesley, 1997.

[GOF95]Gamma,E., Helm,R.,Johnson,R.,Vlissides,J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[LIE96] Lieberherr,K.: Adaptive Object-Oriented Software:The Demeter Method with Propagation Patterns,PWS Publishing Company, Boston,1996.

[SIM96] Simonyi,C.: Intentional Programming - Innovation in the Legacy Age, Presented at IFIP WG 2.1 meeting, June 4, 1996, http://research.microsoft.com/ip/ifipwg/ifipwg.htm

[SZ97] Sztipanovits, J., Karsai, G.: "Model-Integrated Computing", *IEEE Computer*, pp. 110-112, April, 1997.

Appendix

Model structure specification

```
paradigm Xdl;

entity Port { }
entity InputPort  : Port { }
entity OutputPort : Port { }

model Block {
  part InputPort  inputs;
  part OutputPort outputs;
}

model Primitive : Block {
  attr string type;
}

model Compound : Block {
  part Block blocks;
  rel  Connection conns;
}

relation Connection {
  Port src * <-> Port dst *;
}
```

Model Interpreter Specification

```
interpreter XdlInterpreter;
<< typedef long Wire; typedef long Node;
   typedef map<ID,Wire> PortMap;
   int newId()  { static int count = 0;
                  return count++; }
   int mkWire() { return newId(); }
   int mkNode() { return newId(); } >>

visitor Visitor {
  at Port [PortMap& sMap]
  << if(sMap.find(self.Id())==sMap.end())
        sMap[self.Id()]=mkWire(); >>;
  at Connection [const Block_M* parent, PortMap &sMap]
   << Port_E *src = self.src(), *dst = self.dst();
      Block_M* srcBlock = (Block_M*)src->Parent();
      Block_M* dstBlock = (Block_M*)dst->Parent();
      if((srcBlock != parent) && (dstBlock != parent)) {
         int tmp = mkWire();
         sMap[src->Id()] = tmp; sMap[dst->Id()] = tmp;
      } else if(srcBlock == parent) {
         sMap[dst->Id()] = sMap[src->Id()];
      } else if(dstBlock == parent) {
         sMap[src->Id()] = sMap[dst->Id()];
      } >>;
  at Primitive [PortMap& sMap] traverse[sMap];
  at Compound [PortMap& sMap]  traverse[sMap];
  at Port [int& count, Wire wire, Node node]
     << printf("connect wire:%d to node:%d[%d]\n",
              wire,node,count);
        count++; >>;
  at Port [Node node, Wire wire, int& count]
       << printf("connect node:%d[%d] to wire:%d\n",
               node,count,wire);
          count++; >>;
}

traversal Traverser using Visitor {
   from Block[PortMap& sMap]
        to { inputs[sMap], outputs[sMap] };
   from Primitive[PortMap& sMap] do Block[sMap]
    << Node node = mkNode(); int count;
         printf("node %d %s\n ",node,self.type()); >>
    to { << count = 0;>> I
           inputs[count,sMap[arg.Id()],node],
                                                                                                   << count =
           outputs[node,sMap[arg.Id()],count] };
   from Compound[PortMap& sMap] do Block[sMap]
        to { conns[&self,sMap], blocks[sMap] };
}
```