Journal of Science and Technology
ISSN: 2456-5660 Volume 7, Issue 10 (December 2022)
www.jst.org.in DOI:https://doi.org/10.46243/jst.2022.v7.i10.pp35 - 45

The Dagstuhl Middle Meta model: A Schema For Reverse Engineering

Dr.A.Kalayan Chakravathi, S.Govinda Raju, B.Govinda rao
Professor[1], Assistant Professor[2,3],
Dept. of CSE,
mail-id:skraj999@gmail.com, mail-id:banothu.govind3@gmail.com
Anurag Engineering College,Anatagiri(V&M),Suryapet(Dt),Telangana-508206

To Cite this Article
Dr.A.Kalayan Chakravathi, S.Govinda Raju, B.Govinda rao **, The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering"** *Journal of Science and Technology, Vol. 0,7 Issue 10, December 2022, pp35-45*

Article Info
Received: 09-10-2022 Revised: 10-11-2022 Accepted: 10-12-2022 Published: 27-12-2022

Abstract

The Dagstuhl Middle Metamodel (DMM) is an extensible schema for static models of software. It is a middle-level metamodel since it captures program level entities and their relationships, rather than a full abstract syntax graph (lower level), or architectural abstractions (higher level). DMM can be used to represent models extracted from software written in most common object-oriented and procedural languages. This paper presents the main features of DMM.

Keywords*:* Metamodelling, Static Analysis, Exchange Formats, DMM, Reverse Engineering

Introduction

To enable software re-engineering tools to fully interoperate, an agreed-upon exchange format must be available. Many different parsers, shareable reposi- tories or databases, as well as analysis tools could then work together.

An exchange format needs both a schema (i.e. a metamodel) describing the objects and relationships, as well as a 'carrier' syntax describing how model elements will be transmitted or stored. This paper discusses the metamodel, leaving aside the question of syntactic form. For the latter we suggest TA [1] or GXL [2].

There have been many suggestions for metamodels to represent the static structure of source code. The metamodel presented here derives from pre- decessor work at several universities, e.g. [3] [4] [5] [6]. Ideas from these predecessors were incorporated into what was originally called the "Dagstuhl Middle Model" (DMM) at the Dagstuhl Seminar on Interoperability of Re- engineering Tools, Jan 22-26, 2001 [7]. Since then, there have been several revisions of DMM, and the final letter now stands for 'Metamodel'.

This paper discusses the main features of DMM version 0.007; additional information can be found at [8]. The version number will be changed to 1.0 if and when a commercial vendor supports DMM.

There have been several practical uses of DMM. For example, Moise and Wong [9] used it in an industrial reverse engineering case study. There is also a tool on the web [10]

that will take any C++ source code and convert it into DMM using GXL syntax. Several projects are also building schemas that extend or connect with DMM (e.g. [11]).
DMM is a middle metamodel since it represents neither complete syntax of code (lower metamodels) nor abstract architectural elements (higher meta- models). It represents the main program elements and their relationships.
DMM has been reasonably stable, so researchers may experiment with using it in interoperable tools without being concerned about large changes. We do, however, anticipate some further evolution. Our objective would be that it becomes a defacto standard in the community.
In the next section we present an overview of DMM. Section 3 then presents some of the main design decisions it embodies. Finally, Section 4 discusses some of the directions for future work.

Overview of DMM

DMM can represent information about the source code of most popular programming languages, ranging from C, C++ and Java to Fortran. It does not handle aspects of less widely used languages (such as functional or logic languages), although extensions could be created to handle these.

DMM does not represent programs completely; i.e. it does not store the abstract syntax tree. Nor does it represent very high-level architectural ele- ments like pipes, filters, clients, servers, etc. Other types of metamodels can be used for these low-level and high-level models, respectively. DMM is intended to be used for *middle*-level models. There is nothing in DMM, however, that precludes extensions which address high-level or lower-level concerns.

DMM is described using the four UML class diagrams shown in Figures 1 through 4. As is conventional in UML, abstract classes are shown in italics. The four diagrams are as follows:
Figure 1 is a top level view, showing how the other three figures fit together. It shows the three classes at the top of the hierarchies: **SourceObject** (representing high level syntactic entities specific to a particular piece of source code), **ModelObject** (representing conceptual entities that would exist even if the code were translated into a different language), and **Rela- tionship**.

Figure 1. Top-level classes in DMM

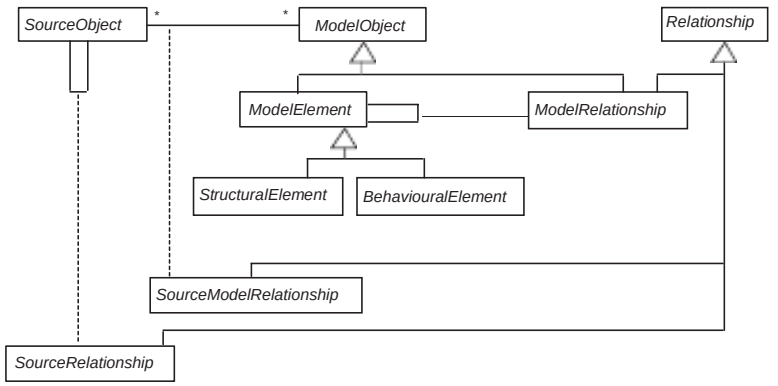


Figure 2 shows the subclasses of ModelObject, and their associations. The most important thing to notice about this hierarchy is the division between StructuralElement

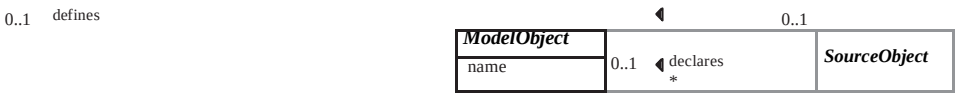
and BehavioralElement. StructuralElement has such subclasses as Variable and Type, of which Class is a further subclass. The main subclass of BehavioralElement is Routine, of which Method is a further subclass.

Figure 3 shows the subclasses of SourceObject, and their associations. The most important SourceObject subclasses are SourceFile and Source- Unit (the latter being used to represent blocks of code editable by the user

in repository based environments that don't store code as a collection of files). It is possible to entirely omit SourceObjects other than Source- File or SourceUnit. In the next section we will see that this is one of the ways in which DMM is designed to be flexible. However, most implemen- tations will want to add instances of other SourceObject subclasses such as MacroDefinitions or objects that specify where in a given SourceFile or SourceUnit any given ModelObject is defined or declared.

Figure 4 shows the Relationship classes. These are all UML association classes arranged in a generalization hierarchy. The domain and range of each relationship is shown in each class box 5 . The relationships are di- vided into SourceRelationship, ModelRelationship and SourceMod- elRelationship. The most important ModelRelationship subclasses are Invokes (to model caller-callee relationships), various IsPartOf relation- ships such as IsMethodOf and IsFieldOf, as well as the Accesses re- lationship (e.g. to model which Routines access which Variables; this will be discussed further later). The ModelRelationships also appear as association labels in Figure 1. The Includes relationship is a key subclass of SourceRelationship, i.e. it is a relationship between SourceObjects. Defines and Declares are the main subclasses of SourceModelRela- tionship. The 'inheritanceType' attribute indicates whether inheritance is private, public, protected, etc. as in C++.

5 This is a departure from UML syntax, but is very helpful in making the DMM diagrams more expressive. Normally, only attributes and operations appear in class boxes.



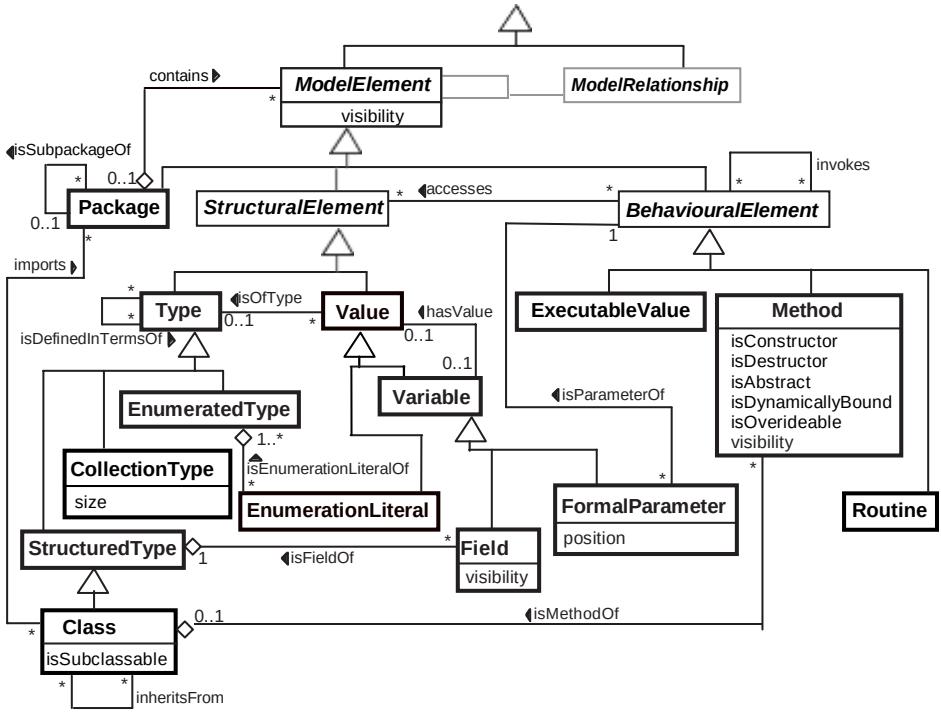


Figure 2: The DMM ModelObject hierarchy. ModelObjects represent program-level entities, independent of any particular source code.

Experience has shown that the intent behind most DMM classes is reasonably clear to developers building DMM-based reverse engineering systems. Such developers would generally create a parser or scanner for the source code of the programming languages they are interested in. They would then build data structures representing instance of the various DMM classes.

The next section explains some aspects of DMM in more detail.

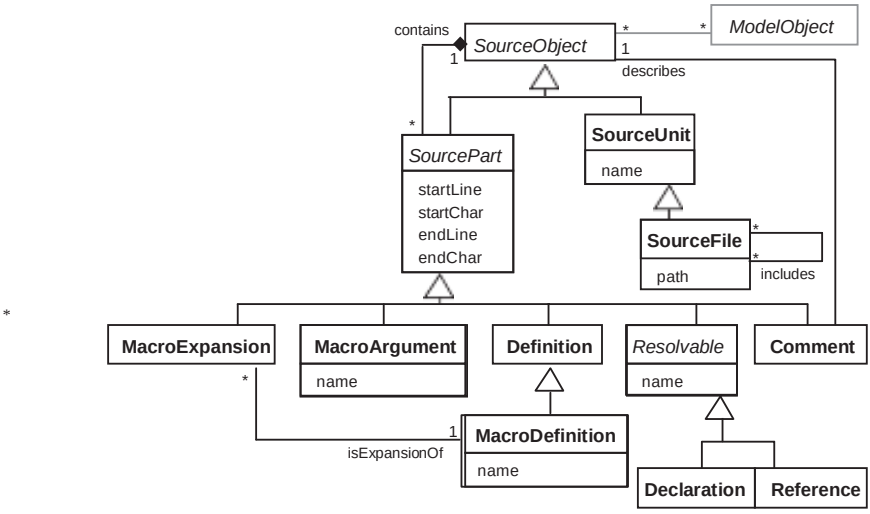


Figure 3: The DMM SourceObject hierarchy. These represent chunks of sourcecode.

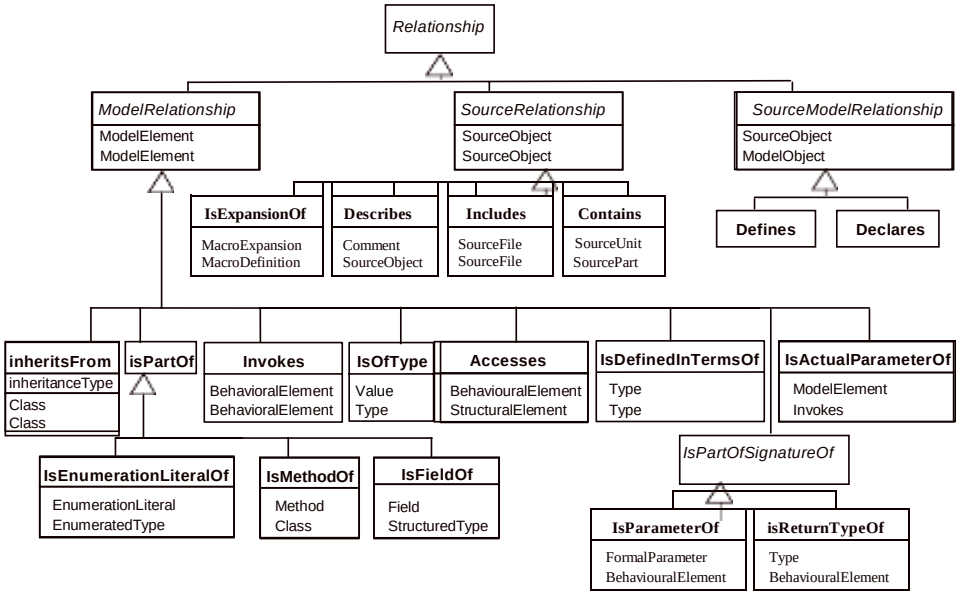


Figure 4: The DMM Relationship hierarchy. All classes are association classes, and show the domain and range.

Key DMM Design Decisions

The structure of the Dagstuhl Middle Metamodel was derived after several key decisions were made. These are detailed below.

Separation of source and other elements

DMM separates objects representing source code (class SourceObject and its subclasses) from those representing abstract elements of a program or system (class ModelObject and its subclasses). This is illustrated at the top of Figure 1. The two separate hierarchies are shown in Figures 2 and 3.

Such a separation is very useful in reengineering tools, since it facilitates:

- Modelling of the various syntactic representations or references to the same software element, e.g. a **Definition**, several **Declaration**s, and numerous **Reference**s, (places where **Access**es or invocations occur in the code).
- Mapping of the same software into different source representations (e.g. before and after restructuring or editing, or even after translation from one language to another).
- Ignoring of the source code when necessary. For example building abstract models of class hierarchies without reference to implementation in any language.
- Dealing with source-level (pre-compilation) information such as **Comment**s and **MacroDefinition**s that have no existence in the compiled version of code. This has been found to be particularly important to make reverse- engineering and re-engineering tools useful and adoptable. Our studies have shown [12] that maintainers want to see models of the actual code, not code after it has already been pre-processed. Models based on DMM have been found particularly useful for searching through large volumes of code; tools that facilitate this must provide search results as pointers to locations in the actual non-preprocessed source.

Some users of DMM may elect to simplify their models by omitting all but the most essential **SourceObject**s. Rather than storing one or more **SourceObject**s for every single **ModelObject** (e.g. specifying where in a file each variable, method, invocation etc. is located), one could just store the **SourceObject**s corresponding to top-level **ModelObject**s (i.e. just the **Class**es). If this option is chosen, then tools using such models will not be able to pinpoint the exact location in files of lower-level **ModelObject**s without further searching; such searching can, however, be done easily in near real time.

Implementations of DMM that take the above simplifying option can represent the needed **SourceObject**s (instances of **SourceFile**) simply as string attributes of the respective **ModelObject**s.

An interesting issue that arose when storing source code information was how, in the **SourcePart** class, to store pointers to the start and end of blocks of source code text. One strategy is to use character offsets from the beginning of a file. This makes seeking to a particular character easy in some programming languages. The choice that was instead made is to use line number plus character offset in the line. This has the advantage that no confusion arises when the size of lines in a file change due to the different line-ending conventions (i.e. CR vs. CR/LF).

Inclusiveness of multiple languages, including OO and non-OO languages

DMM can represent the key features of systems written in object-oriented languages such as C++, Smalltalk and Java. But it can just as easily represent

non-object-oriented systems written in imperative languages such as C and Fortran.

To achieve this multi-language transparency, DMM generalizes several concepts. For example, the notions of *routine*, *function* and *subroutine* are all treated the same. Also, a **Method** is very much the same as a **Routine** except that it has a relationship to a class. Similarly, a **Class** is a **Struc- turedType** that has a few other features, such as methods. Although various programming languages have minor semantic differences regarding how they implement these ideas, DMM abstracts these differences away.

Some people have proposed even abstracting away the differences between **Method** and **Routine**, as well as between **StructuredType** and **Class**. If this were done, true structured types would be represented as classes that happen not to have any methods. This simplification has not been made in the current DMM version. A reason for keeping all four classes is that we believe it helps people to understand DMM better.

Multi-language inclusiveness has many benefits for the user, including the ability to work with multiple-language systems, and the ability to design tools that work in the same way no matter what programming language is employed.

A separate hierarchy of relationships

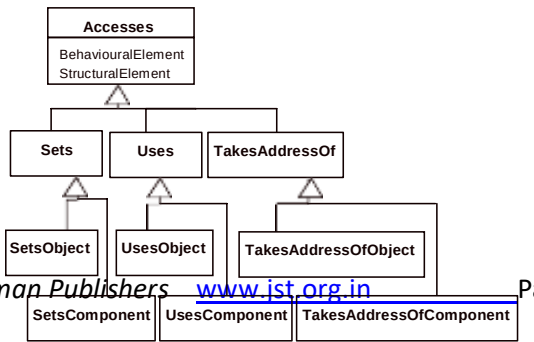
As discussed in Section 2, Figure 4 shows the hierarchy of **Relationship**s. Each of these is an association class which can therefore have its own at-tributes. The presence of such a large hierarchy reflects the fact that in re- engineering, relationships are as important at the things related.

As one moves down the hierarchy in Figure 4, the relationships become more specialized. The domain of a sub-relationship is the same as or a sub-classes of the domain of a higher-level relationship. The same is true of ranges. Models using DMM do not have to represent information about each relationship shown. Also, if a model wants to model accesses, it could use either the higher level Accesses relationship, or its more specific subclasses.

Flexibility to allow for variants and extensions

DMM has several dimensions of flexibility:

- Any DMM class can be subclassed if needed. A tool importing DMM data



with subclasses it does not 'understand' would still be able to interpret the data as the appropriate superclass or superclasses. Figure 5 gives an example of several subclasses of Accesses; not all tools will need or want to support these, but they are available for tools that want to do more sophisticated analysis. Figure 6 shows an extension to represent instances of **Property** which can be treated as both variables (they can be accessed to get or set their value) and methods (they can invoke other methods); properties exist in various programming languages, such as Delphi.

Figure 5: Standard DMM extension to represent different types of access that BehaviouralElements can make to StructuralElements.

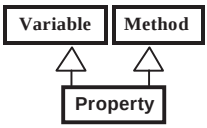


Figure 6: Standard DMM extension to represent Properties, which are programconstructs found in several different programming languages.

- Instances of many DMM classes can be omitted in a valid model in order to simplify the model. We already discussed in Section 2 how most SourceObject classes can be omitted. It would also be quite reasonable to generate data without, for example, information about method parameters.
- New classes can be added to represent different types of information by creating associations to DMM classes. This is exactly what we did to represent dynamic information in [11].

A schema that uses higher level DMM classes but omits some lower-level classes and adds its own classes would be called a variant. A variant could be *consistent* with DMM (the new classes represent different things than those already in DMM), or *inconsistent*. An inconsistent variant might be needed if DMM classes are not able to capture certain concepts and need to be substituted by classes with more representational power. However it would be betterto have an inconsistent variant that nevertheless reused some DMM classes, than a model that was completely different (with unnecessary inconsistencies.) It would be useful if all DMM-compliant tools were developed with an ability to work as best as they can with data containing extensions and vari- ants, and also with data that omits certain DMM classes. Data that contains anything other than basic DMM will have to be transmitted along with its schema. For example, a tool that reads data containing instances of **Property** (Figure 6), but which does not know how to manipulate these instances internally, would nevertheless be able to read the extended schema and process them as separate instances of both **Method** and **Variable**.

Robustness

A metamodel needs to be *robust* as opposed to *fragile*. Robustness means the widest possible variety of tools can use it without the need for inconsistent variants. We hope that robustness was increased by the fact that the devel- opers of several metamodels got together and worked out a metamodel that could be

used by all the groups. The fact that several projects have used DMM with little change, is initial evidence for its robustness.

Conclusions and Future Work

DMM is a metamodel for software reverse engineering that has been provedin practice to be useful. However there are still various areas for research that could lead to changes or extensions. These are discussed below.

It will be important to continue to examine other metamodels to improve DMM to the point where it achieves widespread acceptance. Examples of metamodels that have been widely studied include Columbus [13] and the UML metamodel [14]. Unlike DMM, Columbus is explicitly for C++. The UML metamodel overlaps DMM in places; however, it is designed for forward

engineering, and omits DMM's **SourceObject**s, and various other classes. It is also rather more complex than what appears to be needed for simple reverse engineering tools.

It might be useful to model certain features of programming languages that DMM does not currently support. Examples include generic types (e.g. C++ templates), as well as concerns and aspects (from Aspect Oriented Programming languages).

Finally, documents could be written giving a more precise semantics for each class, and a mapping from various programming languages to DMM. It has been proposed that in order for all the relationships to be modelled consistently by all parsers and other tools, a reasonably formal specification of each should be produced.

References

R. C. Holt, "An Introduction to TA: The Tuple Attribute Language", Technical Report, Department of Computer Science, University of Waterloo and University of Toronto, 1998.

R. C. Holt, A. Winter, A. Schu¨rr, " GXL: Toward a Standard Exchange Format", Proc. 7 th Working Conference on Reverse Engineering (WCRE), 2000, 162-171

S. Tichelaar, S. Ducasse, S. Demeyer, "FAMIX and XMI", Working Conference on Reverse Engineering, 2000. Nov. 2000, 296–298

S. Tichelaar, "Modeling Object-Oriented Software for Reverse Engineering and Refactoring", PhD. Thesis, University of Berne, 2001, http://www.iam.unibe.ch/~scg/Archive/PhD/ tichelaar-phd.pdf.

T. Lethbridge, “Requirements and Proposal for a Software Information Exchange Format”,
http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal.html.

J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke, E. Pl¨odereder, D. Simon, J.-F. Girard,
M. Wu¨rthner, “Data Exchange in Bauhaus”, Working Conference on Reverse Engineering, WCRE 2000, November 23-25, Brisbane, Australia, IEEE Computer Society Press, 2000.

Dagstuhl Seminar 01041, Interoperability of Reengineering Tools, http://www.dagstuhl.de/ 01041/

DMM: The Dagstuhl Middle Metamodel. http://www.site.uottawa.ca/~tcl/dmm/

Moise, D, and Wong, K, “An Industrial Experience in Reverse Engineering”, proc. Working Conference on Reverse Engineering, WCRE2003, November 2003, Victoria, Canada, IEEE Computer Society Press.

C/C++ to GXL Converter Online, http://www.site.uottawa.ca:4333/parser_online/

Hamou-Lhadj, A. and Lethbridge, T.C., “A Metamodel for Dynamic Information of Object- Oriented Systems”, Electronic Notes in Theoretical Computer Science (this volume), expanded from a paper presented at 1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM), Victoria, Canada, 2003.

Singer, J., and Lethbridge T.C. (1998), “Studying Work Practices to Assist Tool Design in Software Engineering”, 6th IEEE International Workshop on Program Comprehension, Italy, 173-179. Ferenc, R., Besz´edes, A´., Tarkiainen, M., and Gyimothy, T. “Columbus - Reverse Engineering Tool and Schema for C++”, Internatioonal Conference on Software Maintenance: ICSM 2002, 172-181.

OMG, “OMG Unified Modeling Language Specification” version 1.5, March 2003, http:// www.omg.org.