The Dagstuhl Middle Model (DMM)

Version 0.006 – July 2002

Active contributors:

Timothy C. Lethbridge <tcl@site.uottawa.ca>
Erhard Plödereder <ploedere@informatik.uni-stuttgart.de>
Sander Tichelaar <tichel@iam.unibe.ch>
Claudio Riva <claudio.riva@nokia.com>
Panos Linos <linos@butler.edu>
Sergei Marchenko <smarchen@site.uottawa.ca>

There are other interested who subscribe to the DMM web site
See the DMM web site (http://scgwiki.iam.unibe.ch:8080/Exchange/2.)

Overview

The DMM is a middle-level model for representing software in reverse engineering applications. It is intended to allow for exchange of information among tools. It can be contrasted with low-level models consisting of abstract syntax graphs (ASGs), and high level models representing architecture.

Data conforming to the DMM schema can be encoded using various different formats. It was explicitly designed to be a schema for GXL (http://www.gupro.de/GXL/), but could also be converted to an XML schema. DMM is described here as a seto of UML class diagrams that describes the possible nodes and edges that will be represented in GXL.

'Dagstuhl Middle Model' was originally a working name but has become permanent. The name was chosen because the model was first developed at the Dagstuhl Seminar on Interoperability of Re-engineering Tools, Jan 22-26, 2001. DMM represents a merger of ideas from several pre-existing models developed by the active contributors.

Changes

The change process is as follows: Proposed changes and comments are sent to the mailing list of active participants (dmm@iam.unibe.ch) and interested observers. All changes are posted to the Wiki site http://scgwiki.iam.unibe.ch:8080/Exchange/2. Changes from the previous version of this document should alwways be highlighted.

New changes reflected in the current version:
• Documents some proposed changes that need to be discussed.

Work plan

A draft paper about DMM has been written and should be submitted for publication soon.

Encodings

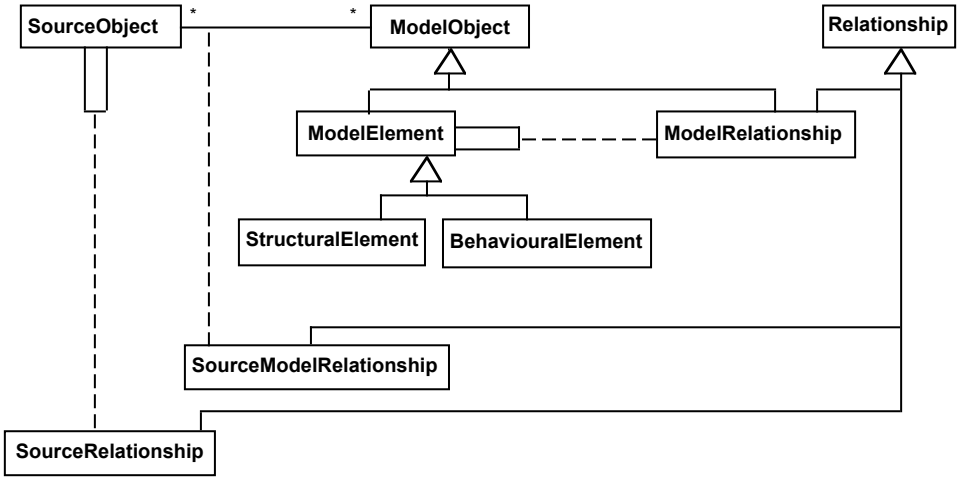
In this document, DMM is encoded in UML. At the web site you will find encodings in GXL (using the GXL metamodel) and in TA.

The model

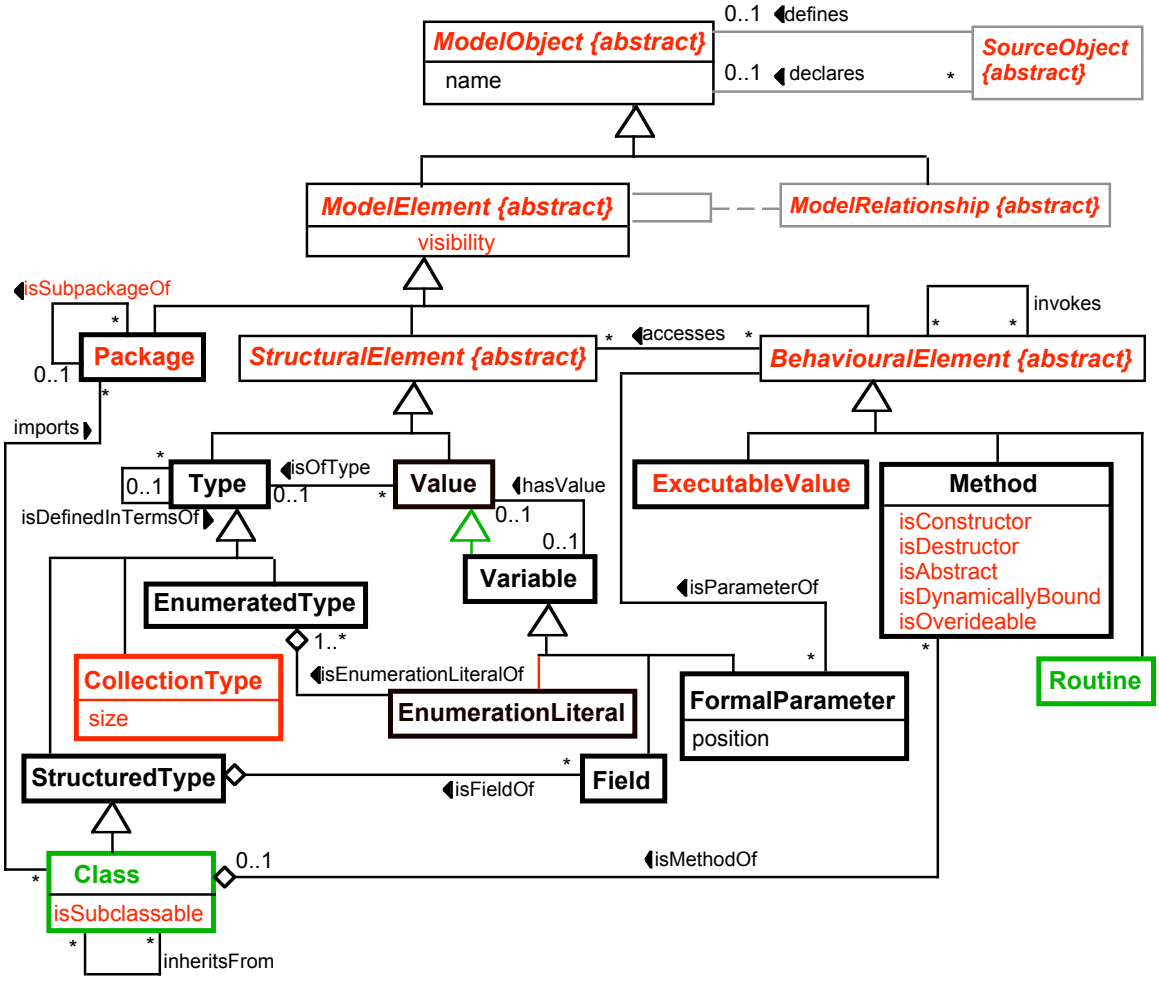
The model has been divided into four sub-hierarchies:

- **The top level model**: This shows the top-level classes and their relationships
- **ModelObject hierarchy**: Represent the abstract view of a program, independent of the specific layout of the source code.
- **SourceObject hierarchy**: Pieces of source code that represent such things as definitions, declarations and references to the model objects
- **Relationship hierarchy**: Relationships among the objects.

Top-Level Hierarchy of DMM classes



Hierarchy of ModelObjects



Descriptions of classes

Top level classes

- **ModelObject** {abstract}: Any logical object in program. DMM can represent its location in source code one or more instances of **SourceObject**. A **ModelObject** can have many **SourceObjects** that **declares** it, but only one that **defines** it.
- **ModelElement** {abstract}: A **ModelObject** that is not a relationship. These are the most fundamental elements in the source code, classes, methods, types etc. The name of a model element is a character string; there many be many **ModelElements** with the same name in a system.
- **ModelRelationship** {abstract}**:** A **ModelObject** that relates two other **ModelObjects**. Note that since it inherits from **ModelObject**, this has a place in the code where it is defined (e.g. a the place where an instance of Invokes is defined is the place where one Method calls Another)

Subclasses of **ModelElement**:

- **Package**: A logical grouping of **ModelElement**s. Corresponds to a Java package or C++ namespace. The contains relationship shows that it can contain any **ModelElement**; this allows for flexibility; in existing languages only **Class**es would be contained. The **isSubpackageOf** relationship allows for nested subpackages.
- **StructuralElement** {abstract}**:** A **ModelObject** that represents some form of data or data structure.
- **BehaviouralElement** {abstract}: A ModelObject that executes. While executing it accesses a set of structural elements. More details of the accesses relationship will be described later.

Subclasses of **StructuralElement**

- **Type**: A **StructuralElement** that constrains what can be assigned to a **Variable**. it can be that one **Type isDefinedInTermsOf** another: This would be the case when the C/C++ typedef construct is used, or when a **CollectionType** type is defined to have members that are of some other type. A union type is modelled by having a type defined in terms of several others. A type can be, in some cases, unnamed; the name attribute would then be the empty string. For so-called untyped languages, like Smalltalk, types are simply omitted from the DMM data.
- **Value**: A piece of data that **isOfType** a certain **Type**. Given that DMM represents the static view of a system, run-time values are not represented. However, if a value is known, e.g. because it is a constant, then it can be represented in DMM. To represent a manifest constant, the text used in the source code, whether it be a string or a number, becomes the **name**.

Subclasses of **Type**

- **EnumeratedType**: A **Type** that has a fixed set of **EnumerationLiteral**s as its possible values.
- **CollectionType**: A **Type** that has members (whose type it **isDefinedInTermsOf**) and a size. A pointer type is shown as a collection type of size 1.
- **StructuredType**: A **Type** (e.g. a C struct, Pascal or Cobol record,) that is broken down into a list of named fields. Note that in C++ a struct can have methods (little known to most people) so it should be a class. It is proposed to merge this with Class.

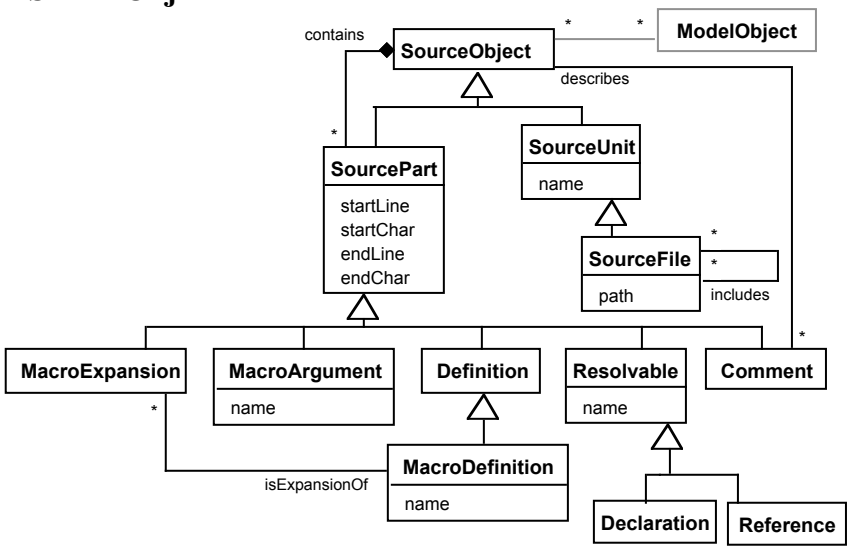
Subclasses of **Value**

- **Variable**: A variable is modelled in DMM as a **Value** that not only has a name, but also, at least at run time, **hasValue** some other value. A constant is modelled as a variable whose value is fixed.
- **EnumerationLiteral**: A constant **Variable** that **isEnumertionLiteralOf** an **EnumeratedType**. In some programming languages, such as Pascal, these only have a name ; in languages like C, they can be given a value as well.
- **Field**: A **Variable** that **isFieldOf** a **StructuredType**
- **FormalParameter**: A **Variable** that **isParameterOf** a **BehaviouralElement**

Subckasses of **BehaviouralElement**

- **UnknownBehaviouralElement**: Allow for the representation of items such as calls to routines that are stored in variables, or calls to polymorphic operations (where one of several methods may be ultimately called).
- **Routine**: Anything executable, that is not part of a class, including a procedure, a function, a Cobol paragraph etc.

Hierarchy of SourceObjects

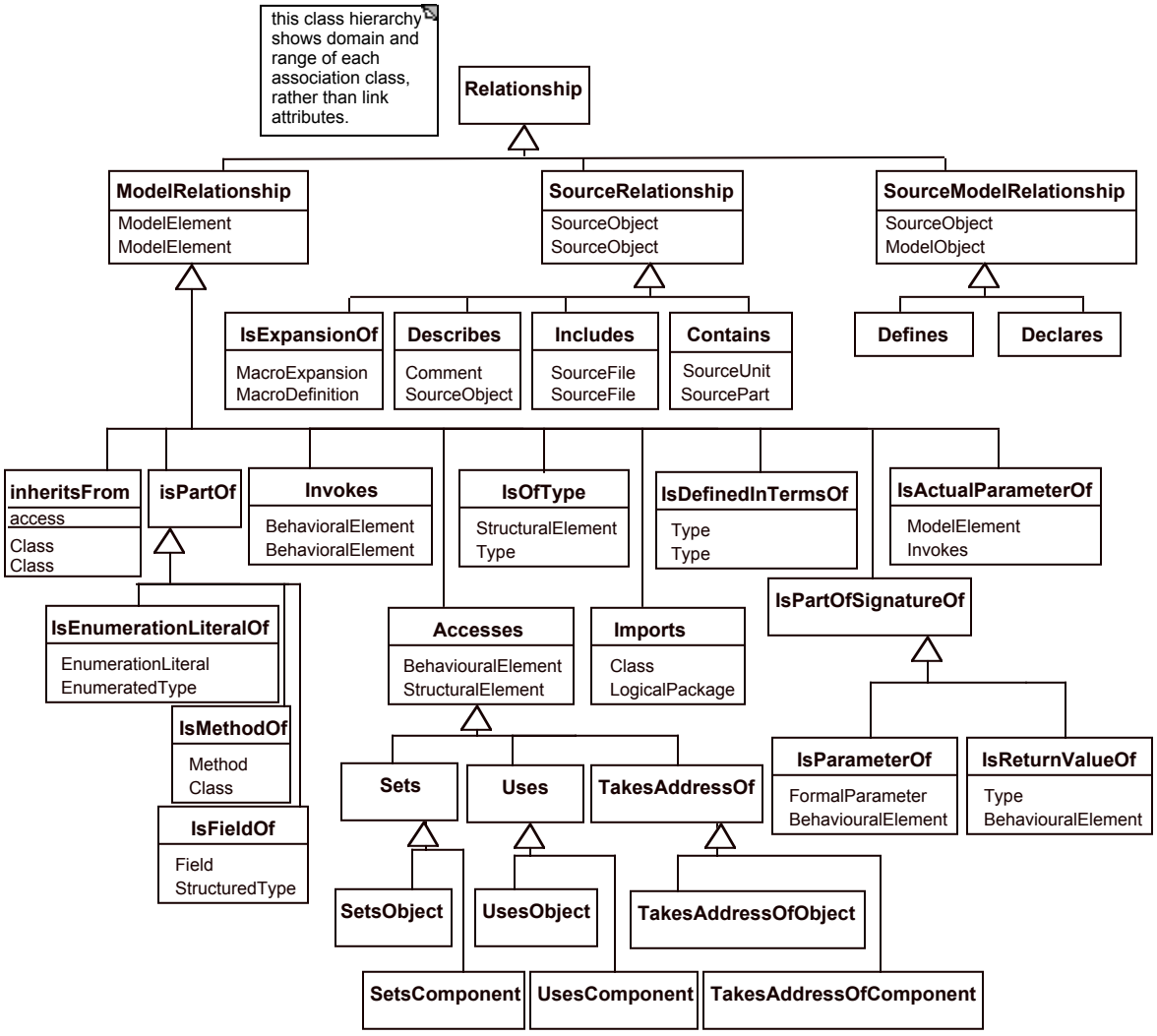


Descriptions of classes.

- **SourceUnit**: Represent editable chunks of code that can be displayed by the user. These are normally files, but may not be in languages where source is retrieved from a repository.
- **SourceFile**: Represent files that can have inclusion relationships. If the path attribute is omitted, it means that it is unknown (e.g. when including a file, and the include path will be defined later)
- **SourcePart**: Represent specific parts of source units that represent model objects.
- **Definition**: Any part of the source code that explicitly defines something of interest.
- **MacroDefinition**: A definition of a macro at the purely source-code level. Will disappear following pre-processing (optional).
- **MacroArgument**: An argument to a MacroDefinition (optional).
- **Resolvable**: A named element of source code, that refers to some model element (e.g. a variable access or a routine call), but which requires a linker to make a definitive resolution. A resolvable could resolve to different model elements depending on the particular SourceUnits included in a build.
- **Declaration**: A declaration that some model element exists.
- **Reference**: A reference to a model element (e.g. routine call, variable access).
- **Comment**: Source code attached to some other source code, that acts as a comment (optional).

Hierarchy of Relationships

In the following diagram, the domain and range are shown. Showing this inside the class box is not conventional UML, however it seemed the easiest way to represent the information.



Descriptions of classes

- **ModelRelationship (ModelElement -> ModelElement)**: Superclass of all relationships between model elements.
- **SourceRelationship (SourceObject -> SourceObject)**: Superclass of all relationships between chunks of source code.
- **SourceModelRelationship (SourceObject -> ModelElement)**: Superclass of all relationships that relate a source object to a model element.
- **Sets (BehavioralElement -> Variable)**: BehavioralElement contains code that sets Variable (e.g., with an assignment). This means that the value of Variable (potentially) changes.

- **SetsObject (BehavioralElement -> Variable)**: BehavioralElement contains code that sets Variable as a whole. (Could be assignments of whole arrays or StructuredTypes if the language provides for it.)
- **SetsComponent (BehavioralElement -> Field)**: BehavioralElement contains code that sets the Field of a Variable that has a StructuredType.
- **Uses (BehavioralElement -> StructuralElement)**: BehavioralElement contains code that uses (i.e., does a read access to) StructuralElement (e.g., it accesses a Variable or EnumerationLiteral).
- **UsesObject (BehavioralElement -> StructuralElement)**: BehavioralElement contains code that does a read access to StructuralElement as a whole.
- **UsesComponent (BehavioralElement -> Field)**: BehavioralElement contains code that uses the Field of a Variable that has a StructuredType.
- **TakesAddressOf (BehavioralElement -> StructuralElement)**: BehavioralElement contains code that takes the address of StructuralElement (e.g., the address of a Variable or a Routine). (Obviously, not all programming language provide constructs that allow this kind of functionality.)
- **TakesAddressOfObject (BehavioralElement -> StructuralElement)**: BehavioralElement contains code that takes the address of StructuralElement. This address denotes StructuralElement as a whole.
- **TakesAddressOfComponent (BehavioralElement -> Field)**: BehavioralElement contains code that takes the address of Field of a Variable that has a StructuredType

Issues to resolve:

- What elements **must** be transmitted by compliant tools? We need a clearly defined set of minimal information; for example a tool could just generate Accesses relationships instead of its subclasses.