

Modelling for Knowledge Discovery

A. Gerber[†], E. Glynn[‡], A. MacDonald[‡], M. Lawley[†], and K. Raymond[†]

[†]Distributed Systems Technology Centre,
email: adm@dstc.edu.au

[‡]School of Information Technology and Electrical Engineering
University of Queensland

1 Introduction

Modernization of legacy systems requires that a clear understanding be gained of those systems. In this position paper it is proposed that this understanding is best supported by modelling specific key aspects of the system. The validity or not of modelling is not argued, but rather the aspects that should be modelled are discussed. It is believed that the aspects will be of interest to all involved in modernization of legacy systems and not only those from the modelling community. The modelling approach taken in this work is derived from an initial submission [1] to the Object Management Group's (OMG) Request for Proposal on Knowledge Discovery for Architecture-Driven Modernization [3].

Knowledge Discovery for Architecture-Driven Modernization (ADM) should include models covering at least the following system attributes:

- Static behaviour of the system: association, containment, annotated call graphs (method calls and access/modification of data), inheritance, error handling, exceptions, real-time/concurrency.
- Dynamic/run-time behaviour of the system: deciphered polymorphic calls, profiling to show component usage, unravelled concurrent/distributed/real-time behaviour.
- Supporting materials: test materials, requirements and design documentation, etc.
- Environment: actual files, their locations, interaction with the user/file system/database/etc., build mechanisms (makefiles and other dependencies).

It is with the collection of Knowledge Discovery Meta-models (KDM) that a system can (in theory) be completely represented and it is instances of these models that will maintain the information required for modernization.

The remainder of this paper will focus on how the static behaviour of the system can be modelled in a manner that encourages reuse and extension of models, and transformation of models, and hence the systems being modelled. Section 2 describes the basic philosophy that underpins the

development of this modelling strategy, with Sections 3 and 4 presenting the General static and Object-oriented static models. The paper ends with a summary of the work presented.

2 Modelling philosophy

The modelling philosophy that underpins this approach is that all model items which derive from common high-level concepts should be modelled as extensions of high-level models. For example, a common understanding of a *program* is a set of instructions to *do* things to *data*.

This very general notion of programming can then be refined into various styles of programming: imperative, functional, and logic programming. These in turn can be further refined. For example, styles of languages that fit within the imperative programming paradigm include procedural, object-oriented, scripting languages. Concrete programming languages (such as Java, C++, or COBOL) are then further refinements of these abstract notions. Finally there may be different versions or variants of these programming languages as implemented by specific tools.

Thus programs written for the Java Development Kit (JDK) version 1.4 contain elements which are specialisations of general Java concepts which are specialisations of object-oriented concepts, which are specialisations of imperative programming constructs, and so on. Accordingly, the model for programs written for JDK 1.4 should be refinements of a general Java model, which is in turn a refinement of an object-oriented model which is in turn a refinement of an imperative model and so forth.

This approach to knowledge discovery modelling requires leaf-level models for every programming language variant that might be used as a source or target programming language in a modernization effort, in addition to the higher-level models from which they are derived. Clearly this is a massive (and perhaps unachievable) task hence the aim of the Knowledge Discovery specification is to establish the higher levels of this modelling framework, and demonstrate the approach of refinement by standardising a small selection of lower-level models. These lower-level models should be chosen to reflect the extent of their

popularity as the source or target of modernization efforts. For example, if there is an overwhelming desire to replace COBOL programs with Java programs, then COBOL and Java should be included in the specification. Other specific languages can be standardised subsequently, if and when there is sufficient interest to undertake the work.

To illustrate this philosophy, this paper focuses on two models: a General static model and an Object-oriented static model. The General static model presented in Section 3 is intended to capture the concepts common to a broad range of programming languages, both procedural and object-oriented, e.g. COBOL, C and Java. Of particular interest when modelling the system statically, is the determination of the control flow and, possibly even more importantly, the inter-relationships between *segments* of code in the system. These relationships include whether a piece of code in the system calls a method from another piece of code, or accesses and/or modifies data from another segment. As such the internals of a method have been modelled to the point where methods calls and data access/modification can be represented.

The power of the model refinement approach can be illustrated by presenting the Object-oriented static model as a refinement of the General static model, in Section 4. The Object-oriented static model specialisation requires only the addition of classes, inheritance and explicit visibility. These additions require only an extension of one small section of the General static model. The Object-oriented static model could then be further refined into a Java and a C++ model.

The benefit of this family of model refinements is that systems can be modelled precisely by using a *leaf* model specific to the precise version (or variant) of a given programming language, yet retaining the ability to view aspects of the system using common higher-level concepts. This ability to view aspects of a system using higher-level concepts allows for the transformation of systems using a combination of general and language-specific transformation rules. High-level transformation rules can be expressed in terms of the high-level common constructs for the general case (and therefore re-usable in many modernization efforts) which can then be *extended* or *superseded* by rules based on specific subtypes for handling special cases or exceptions to the general rule.

As an example, classes are conceptually very similar across a range of object-oriented languages. While object-oriented languages such as Java, C++ and Eiffel all have class constructs, the syntax of these constructs differ. To capitalise on the existence of common constructs, such as classes, refactoring transformations should be able to be written in a language-independent manner. This allows the knowledge discovery modelling efforts to leverage techniques and tools developed within the wider Model Driven Architecture (MDA) community. The ability to write concise, extensible and coherent transformations is particularly important to allow the use of the MDA Queries/Views/Transformation (QVT) [2] Standard for modernization.

3 General static model

Figure 1 presents a general model that supports the modelling of a range of languages that are (or can be) block structured in the traditional sense of the word, but are not object-oriented. Examples of such languages are COBOL and C. The object-oriented extension to this general model (presented in Section 4), for Object-oriented languages such as Java and C++ require extensions to this general model. An Object-oriented static model is presented in Section 4.

At the most abstract level a System inherits a Namespace and a NameElement within that Namespace. A System can contain Systems and this mechanism is designed to model Systems that rely on existing (possibly third-party) software. An example of this is the use of existing libraries such as the standard C libraries. In this case the complete system can be seen as a composition of smaller systems: code specific to the problem could be a sub-system and code provided by others could be other sub-systems, or code specific to the problem could be in the current system and code provided by others in sub-systems. A System is also composed of Documents. A Document also defines a Namespace and the Namespace for a Document is a potentially equal subset of the Namespace of the System that contains it. Note this Namespace restriction also applies to contained Systems. A System is not required to contain a Document as a System could simply be a collection of Systems held together by build mechanisms.

A Document can contain Blocks and a Block can only belong to a single Document. Note this restriction does not apply to Documents or Systems which can belong to multiple Systems. If a programming language or programming environment supported a finer granularity of sharing this restriction could be pushed down the inheritance hierarchy as appropriate. A Document maps to a File in the Environment model. Blocks optionally have a NameElement associated with them, can contain other Blocks and have a visibility relationship to other Blocks. In this model, the meaning of visibility is programming language dependent. It is suffice to model that one Block can see another, i.e., code from one Block can call methods from and/or access/modify data from another Block. Visibility in this general model can only be calculated at the program level by doing a complete analysis of how each block interacts with each other block.

A PrimitiveBlock is essentially an optionally typed Block utilising the 'is of' association with the Type class. This separation may seem arbitrary at this time, but it supports easy extension of the model, especially to object-oriented systems and does not unnecessarily complicate the general model. PrimitiveBlock is sub-classed into one of NamedData, Data, or MethodSpecification. This subclassing supports Documents being as simple as a program (in COBOL or C) or a C header file or a Modula2 module with contained modules.

Type is subclassed to PrimitiveType and this is also to

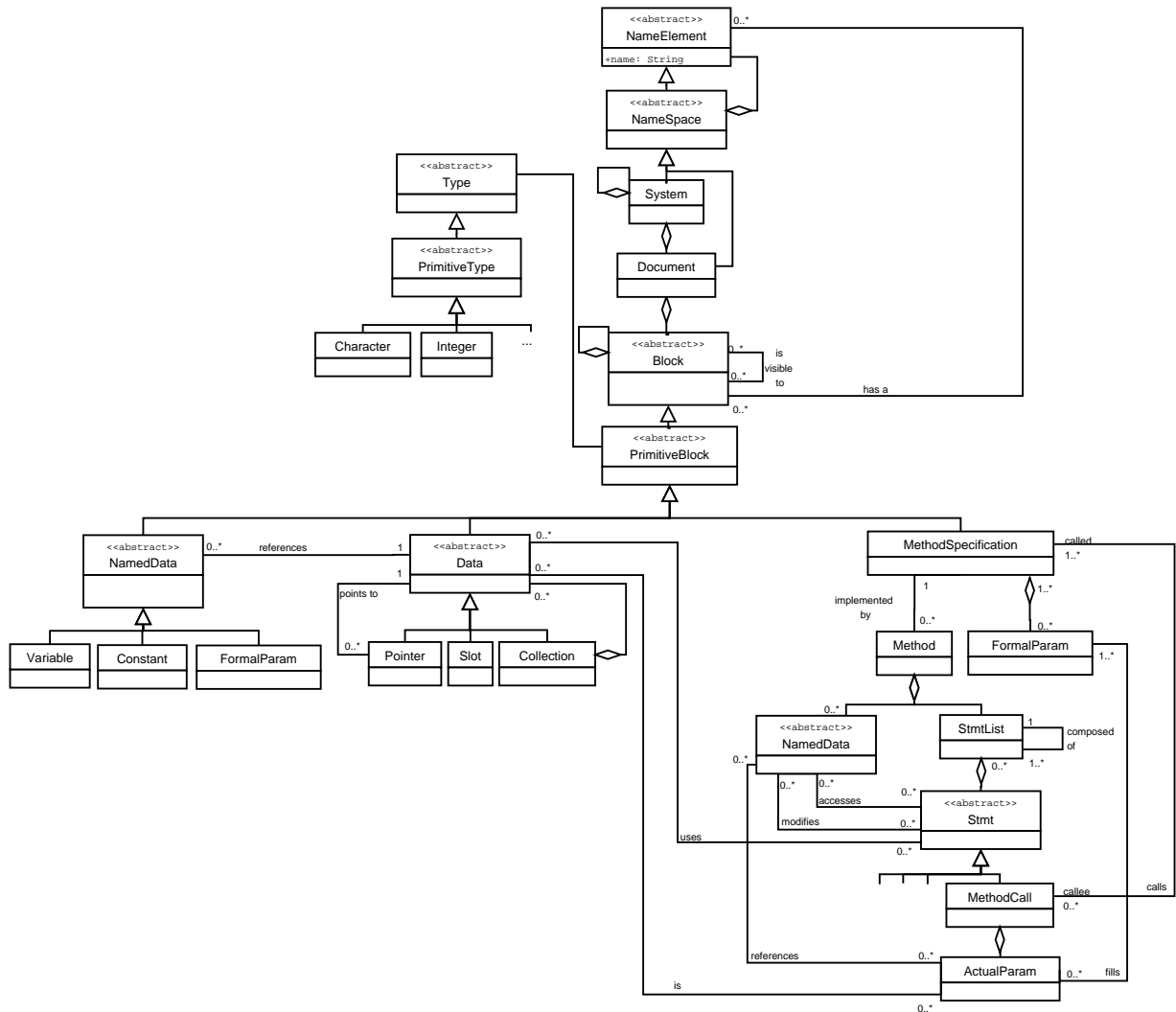


Figure 1. General static model

support ease of extension. In the General model there are only the primitive types, but extensions (such as the object-oriented extension of Chapter 4) may wish to add other non-primitive types. PrimitiveTypes represent the base types provided by programming languages. Character and Integer provide an example PrimitiveType. The full set of PrimitiveTypes would be defined when the model is specialised for a specific programming language or class of programming languages.

Data is partitioned into its two aspects: the actual data and the way to access that data. The actual data is represented by the class Data and is restricted to being anonymous (cannot have a name). Data exists as at least three subclasses: Pointer, Slot and Collection. A Pointer is a piece of Data that points to another piece of Data and can only point to one piece of Data, but one piece of Data can be pointed to by many Pointers. A Slot (as defined in UML 2.0) is the place/location of a piece of Data. A Pointer will often point to a Slot, though it is not restricted to this. A Collection (also from UML 2.0) is a mechanism for grouping a number of related pieces of Data. Data is referenced by NamedData with similar constraints to

those on a pointer, i.e., NamedData references one piece of Data, but one instance of Data can be referenced by many instances of NamedData. NamedData is not allowed to be anonymous, hence named data. Two types of NamedData are Variables and Constants.

Methods have been separated into the MethodSpecification and the Method. This separation allows the modelling of C header files with method prototypes or the pre-definition of methods in a C implementation file. Methods can only implement one MethodSpecification, but a MethodSpecification can be implemented by multiple Methods. A MethodSpecification also has FormalParameters which are another form of NamedData.

A MethodCall is associated with the MethodSpecification of the Method it is calling and may contain ActualParameters. ActualParams 'use' Data and NamedData (e.g. addPair(x,1)) and as such is linked to Data by the 'use' association and NamedData by the 'references' association. ActualParams are also related to FormalParams via the 'fills' relationship through which the ActualParams of a MethodCall must satisfy the FormalParams of the MethodSpecification.

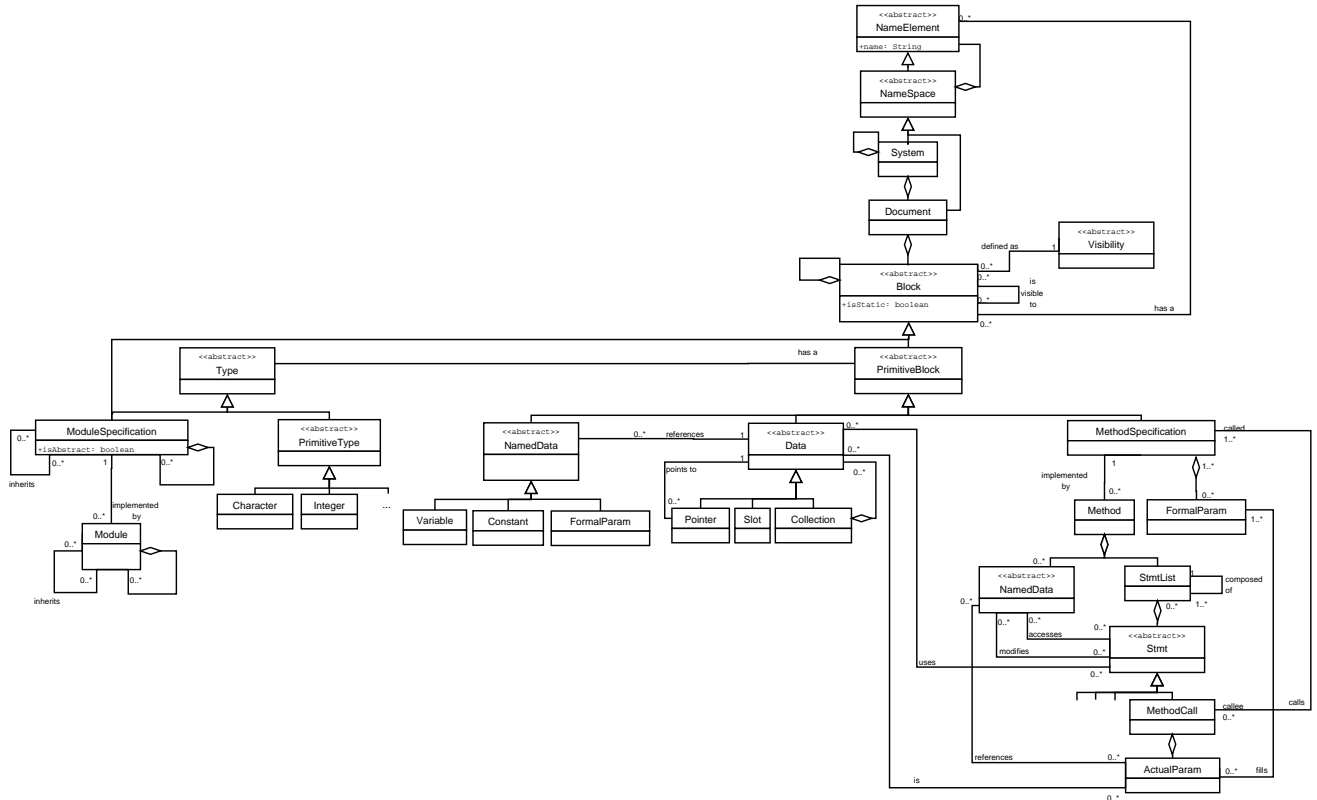


Figure 2. Object-oriented static model

A Method contains both NamedData and a list of statements (StmtList). In this context, NamedData refers to any Variables, Constants etc. which exist within the local scope of the Method. The NamedData is optional. The StmtList is compulsory, but it may be composed of zero statements (Stmts) and hence be empty. This feature allows Methods with empty bodies to be modelled.

A StmtList is a composition of statements (Stmts) which is can be viewed as a line by line composition of statements to make a list (or sequence) of Stmts (0..*). Additionally Stmts may contain other (or sub) statements such as can be demonstrated by the code example setX(getX()+1) which represents a method call that uses another method call's result as an ActualParameter. Stmts are associated with Data and NamedData as they 'use' Data and 'access'/'modify' NamedData. The access/modified relationships are key relationships to model, if a clear understanding of behaviour is to be extracted from the system. Many types of Stmts exist, representing the different standard programming language statements. The key subclass of Stmt for this model is MethodCall. This class enables a second key relationship for understanding the system (calls) to be derived. The derivation of the 'calls' association further allows the development of Control flow graphs, and Call graphs in addition to supporting the computation of the actual build dependencies that exist within the system (as compared with the explicitly defined build dependencies of the Environment model).

4 Object-oriented static model

This section outlines the extensions required to the General static model to support the representation of object-oriented programs. The General static model was designed to support ease of extension and the only conceptual changes required to support object-orientation are at the Block level. The Object-oriented static model, shown in Figure 2 adds three concepts not supported in the General static model:

- Module and ModuleSpecification to represent the structuring concept that is both a Block and a Type (i.e. the class concept)
- inheritance of Modules and ModuleSpecifications
- visibility determined at the Block level

Modules allow the introduction of the class concept and provide for Blocks which define Types. As such a ModuleSpecification inherits from both Type and Block. This extension allows classes to be treated as Types and can be used as such within the system, e.g., a Module can provide the type of a FormalParam, referenced by NamedData, etc. A ModuleSpecification is also a Block and due to the composition of Blocks may contain further MethodSpecifications, Methods, NamedData, Data, etc. ModuleSpecifications can also contain ModuleSpecifications supporting the Java inner class concept.

A ModuleSpecification may contain a Module that implements the specification. In fact a Module can only implement one ModuleSpecification, but a ModuleSpecification can be implemented by more than one Module supporting the Java `interface` concept. A ModuleSpecification can be abstract, while (arguably) this is derivable, it is considered important design information that should not be discarded.

Inheritance is supported from ModuleSpecification to ModuleSpecification and Module to Module. The two types of inheritance allow for the modelling of systems developed in languages that have type and/or implementation based inheritance. At this point the model can fully model inheritance and due to the general nature of the initial model no further changes are needed to support/model polymorphism. It should be noted that the identification of the particular instance of a Method called in a polymorphic hierarchy cannot be determined statically, but the set of possible called Methods can be determined and modelled.

The final concept added to the object-oriented model is the concept of explicit Block level visibility. Object-oriented programming languages encompass the concept of methods and data being explicitly marked to define their Visibility to other Blocks. A key benefit of this explicit Visibility is that the visibility can be determined at the Block level independent of the system rather than as a analysis of the complete system as in the General static model. Visibility has been left abstract as the range of behaviours across programming languages, affecting both composition and inheritance based visibility, is quite large (public, private, protected, named public, friends, non-inheritable, etc.).

The second item introduced in this segment is the ability of a block to be static (have only one instance). This concept supports both the Singleton Architectural pattern and the main method concept in Java/C++. The static concept exists on the boundary between the Static and Dynamic models of the system, but can be derived through static analysis, useful in understanding the system and hence has been included in the model.

5 Summary

In this paper, we have outlined a highly extensible approach to knowledge discovery modelling. The General and Object-oriented static models that were presented are designed to be extended to cater for a wide variety of programming languages and tools. The highly extensible modelling framework also enables exploitation of the power of rule extension and rule superseding in OMG's Query/View/Transformation (QVT) specification (currently in development), which leads in turn to greater opportunities for re-use and extension within modernization efforts. By using QVT to abstract KDM models to PSMs and PIMs, it will be possible to evolve legacy systems using established MDA techniques and tool support.

Acknowledgement

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

References

- [1] A. Gerber, E. Glynn, M. Lawley, A. MacDonald, and K. Raymond. Knowledge Discovery MetaModel - Initial Submission. OMG Submission admtf/04-04-01, Object Management Group, 2004.
- [2] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In A. Corradini, H.-J. Kreowski H. Ehrig, and G. Rozemberg, editors, *Proceedings of ICGT'02*, volume 2505 of *Lecture Notes in Computer*, pages 90–105. Springer Verlag, 2002.
- [3] Object Management Group (OMG) ADM Taskforce. Request for Proposal - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM). OMG RFP It/03-11-04, Object Management Group, 2003.