Meta Information Management

S. Crawley, S. Davis, J. Indulska, S. McBride, K. Raymond
CRC for Distributed Systems Technology (DSTC)
University of Queensland, Qld 4072, Australia
crawley@dstc.edu.au

Abstract

The increasing openness and heterogeneity of distributed computing systems requires the representation of meta-information in distributed environments. This paper presents a conceptual framework for discussing meta-information, and describes the DSTC's design for a universal Meta-Information Manager (MIM) with the key properties of heterogeneity, extensibility and openness. The DSTC MIM design supports a wide range of applications for meta-information, from trading and binding services to design repositories and data warehousing.

Keywords

Distributed systems (C.2.4), programming environments (D.2.6).

1 INTRODUCTION

The increasing openness and heterogeneity of distributed computing systems requires the representation of meta-information in distributed environments, as evidenced by current standards activity in the OMG [1] and ISO [2][3]. Meta-information is information that describes other information. Everyday examples of meta-information include programming language data types, CORBA or DCE interface definitions, formal specifications and design diagrams, and schemas for databases.

A *Meta-Information Manager* (MIM) is a service that provides a repository for meta-information in a distributed environment. A MIM needs to support a wide range of clients, including infrastructure services, domain specific services and end user applications; for example, trader service types, interface types, database schemas, user profiles in a command interpreter.

Although low volume, meta-information is typically complex and highly inter-related; it is typically viewed as a navigable tree or graph. In addition, certain "paradigms" such as naming and scoping are common to many kinds of meta-information.

The DSTC MIM is the third generation of universal meta-information or type managers designed and developed by the DSTC [4][5][6]. The DSTC MIM described here differs from earlier DSTC work in that it represents types in structured form. This entails adding a well defined meta-meta-information level to the type management model to allow the representational aspects of each target type system to be modelled. One consequence is that the MIM can produce "specific" meta-information interfaces that are tailored to each target type system.

Apart from the DSTC work, there is little published material on universal type management. A few type managers have been designed to support a single interface type system; for example the CORBA Interface Repository [7] supports CORBA interface definitions, and the Commandos Type Manager [8] supports a canonical type model for the Comandos supported languages. The only other example of a universal meta-information manager is the UNISYS Universal Repository [9], which is designed primarily as a repository for software design tools.

2 OVERVIEW OF THE META-INFORMATION MANAGER

The terminology of this document frequently prefixes words with "meta-". In general, the *meta-* prefix indicates that the term denotes "a description of a something". In some cases the prefix is repeated; for example, a meta-meta-object is "an object that describes a meta-object".

2.1 Abstract Overview

The DSTC MIM is based on the abstract model of *meta-information* as illustrated in Figure 1. The universe of entities ("things") in a given domain of interest can be classified into *types*; i.e. sets of entities that share some property or properties of interest. The types for a given domain of interest are defined in some formal or informal *type language* . The types will have associated domain specific semantics, some of which can be expressed as *type relations*, consisting of tuples of related types (*type relationships*). For example, the notion of subtyping in a programming language can be modelled as a binary relation between types; i.e. the set of valid "subtype" / "supertype" pairs. A *type system* consists of a type language for a domain of types, and a definition of the type relations that are meaningful over those types. Finally, a *type schema* is a collection of types and relationships between them that jointly describe some "system" of interest.

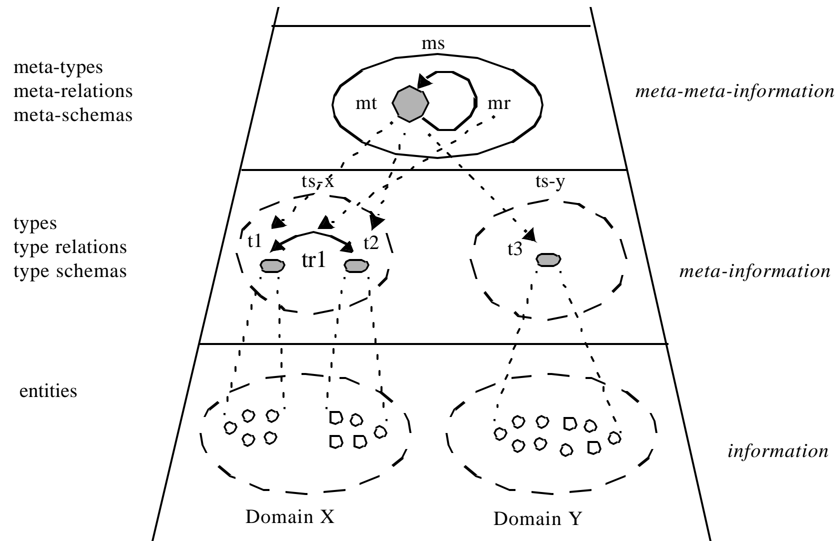

Figure 1 Meta-meta-information, meta-information and information.

Analogously, a type system can be said to define a domain of *meta-information* whose "values" are the types and relationships expressible in the type system. By applying the approach above, it can be reasoned the values in the meta-information domain can themselves have types. These types over the meta-information domain (i.e. the types of types, relationships) are known as *meta-types* and *relation* or *relationship types*, and are examples of *meta-meta-information*. The meta-meta-information domain is the one in which MIM type system definitions would be expressed. Further layers may be added to the abstract model, though three layers are generally sufficient for practical purposes.

The meaning of the abstract concepts in Figure 1 may be understood using some real world illustrations. If the meta-meta-information layer defined a type system for CORBA IDL, then the meta-information layer would contain schemas of CORBA IDL types and relationships between them, and the entities in the information layer would be CORBA objects. Alternatively, if the meta-meta-information layer defined a type system for design notations, then the meta-information layer would define specific notations, and the entities in the information layer would be design diagrams.

2.2 Concrete Overview

The DSTC MIM needs to represent types, relations and schemas in a form that is convenient for client programs. Since the MIM is intended to be "universal", it also needs to represent type system definitions and provide a mapping from these to meta-information interfaces. It needs to allow derived relationships between types;

e.g. based on analysis of the types or on inference from type system properties. Finally, the MIM needs to allow type system definitions to be augmented with semantics.

The DSTC MIM interfaces are all expressed in terms of the CORBA object model, and defined in CORBA IDL. Meta-information is represented as CORBA objects that are called *meta-objects*. Types and components of types are represented by meta-objects called *type objects* that have a fixed set of strongly typed properties. Relations are represented by meta-objects called *relation objects*. Collections of related types and relations are represented by *schema objects*.

Meta-meta-information is represented as CORBA objects that are called *meta-meta-objects*. Each kind of MIM type, relation and schema object in a type system is defined by a meta-meta-object; i.e. a *meta-type object*, *meta-relation object* or *meta-schema object* respectively. The closure of a meta-schema (i.e. the set of meta-meta-objects it depends on) constitute a *MIM type system definition*.

2.3 Type System Instantiation

The main reason that the DSTC MIM represents meta-meta-information is to allow programmers to define their own type systems. The programmer first defines a type system using a language and tools provided with the MIM. The type system definition is then *instantiated* to produce a group of CORBA services that manage the meta-objects which represent types, relationships and schemas. The programmer would then develop the tools to populate the meta-object services, and use the meta-information as required. Type system instantiation is illustrated in Figure 2.

The type system instantiation process could be automated using a *MIM toolkit*, or it could be entirely manual. This allows the DSTC MIM specification to be used for both "universal" meta-information repositories, and type system specific repositories. Assuming that it exists, a MIM toolkit could provide type system specification languages, compilers and IDL generators, along with tools for meta-object server implementation, and generic tools for browsing and managing meta- and meta-meta-objects. In either case, the steps in the type system instantiation process are as follows:

1) The programmer designs the conceptual structure of the types and relations in the new type system and causes the corresponding meta-meta-objects to be created. For example, the type system could be specified in a textual language (e.g. DSTC's "MODL" language [10]) and then compiled to produce meta-meta-objects.

2) The programmer translates (using a tool or by hand) the meta-meta-objects to the IDL for the new type system's meta-objects, following the mapping rules in the MIM specification. This IDL can then be compiled using a standard IDL compiler.

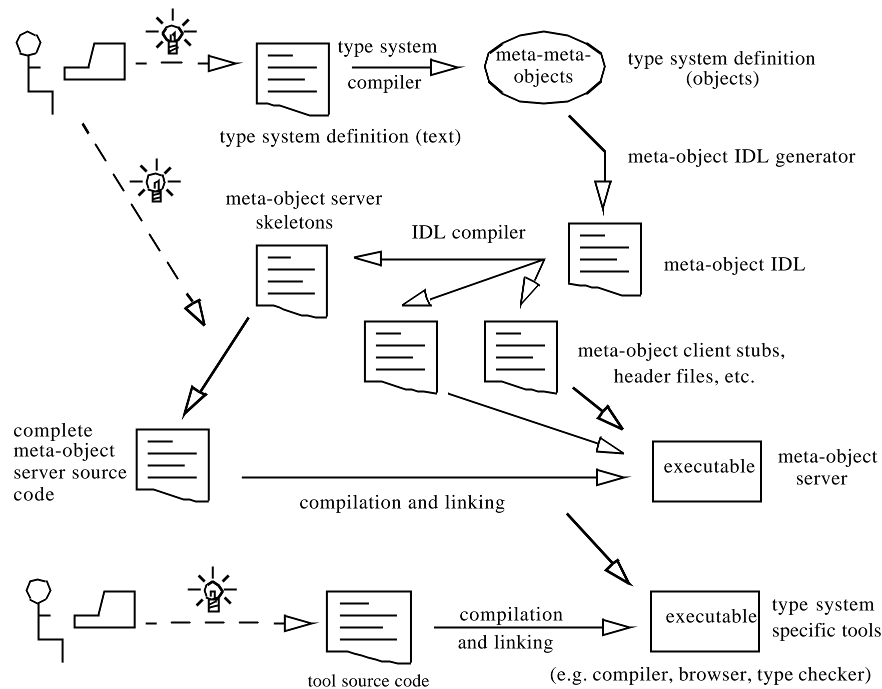

Figure 2 MIM type system instantiation and tool building.

3) The programmer implements the meta-objects services for the type system; e.g. by starting from server skeletons generated by the IDL compiler, or by using a meta-object implementation language and tools from the MIM toolkit.

When the type system has been instantiated, the programmer needs to implement any tools (e.g. browsers, compilers, type checkers, design tools, etc.) needed to create and use meta-information for the new type system. Once again, this could be automated using tools in the MIM toolkit.

3 META-OBJECTS

In this paper, the term *class* is used to refer to the interface type or signature of an object. In the CORBA / OMA world [7], object interfaces are typically expressed in the CORBA Interface Definition Language (IDL). Interfaces have operations with typed parameters, results and exceptions, and typed attributes. CORBA provides multiple interface inheritance. This paper uses the terms *super-class* and *sub-class* to denote the respective roles.

While a typical CORBA class can be instantiated as CORBA objects, some classes (known as *abstract classes*) are defined solely to be inherited by other interfaces. An *abstract base class* is a class defined to create a common super-class for group of classes. A *component class* is defined to provide a set of related features that will be reused in other classes.

The DSTC MIM represents meta-information using CORBA objects called *meta-objects*. There are many meta-object classes, all descended from the abstract base class "MO" (Meta-Object), as illustrated in Figure 3. The MO interface provides two attributes; one that links a meta-object to the meta-meta-object that describes it, and another that links it to the type schema that "owns" it.

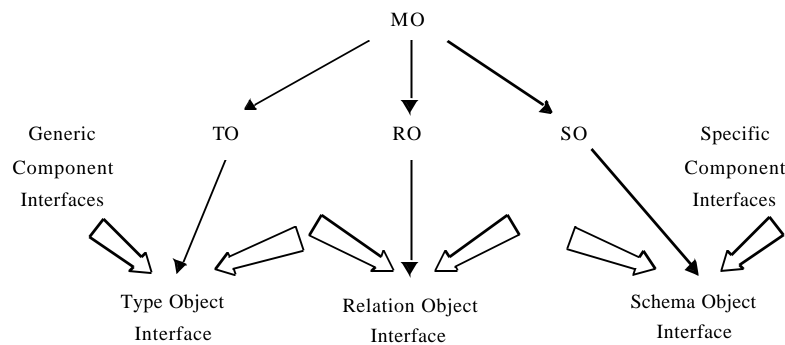

Figure 3 The MIM meta-object class hierarchy.

The full (i.e. most derived) interface for any MIM meta-object will inherit from two distinct kinds of interfaces. The *generic meta-object interfaces* provide a type system independent view of the meta-information, using textual names, unbounded sequences and the CORBA "any" type to achieve genericity. These interfaces are provided to support generic tools such as meta-information browsers that work with all kinds of meta-information.

The *specific meta-object interfaces* provide a view that is tailored to each specific type system. These interfaces provide specifically named operations and attributes with ordinary (static) CORBA types, and are designed to be more "user friendly" for applications working in a particular type system. The specific interfaces are derived from the type system's definition in the type system instantiation process (see Section 2), and should conform to the detailed rules for mapping meta-meta-objects to meta-object IDL as defined in [10].

3.1 Type Objects

A *type object* is a meta-object that represents a type or a component of a type. It has a fixed number of *properties*, where the property name and property type are specified in the type system definition. The type of a property may be any CORBA data type including a CORBA object type.

An instantiated MIM type system will define one or more type object classes. As Figure 3 shows, each type object class is descended from the "MO" class by way of the "TO" class. It also inherits a number of "generic" and "specific" component classes, depending on its properties, and on the relations that it may belong to. In the case of properties, the component classes provide operations to "get" and "set" the property values of the type object.

3.2 Relation Objects

A *relation object* is a MIM meta-object that is used to represent type relations. A type system definition will typically include a number of relation object classes. Some will serve to aggregate and connect component type objects to form complete types. Others may represent relationships between whole types; e.g. type equivalence or subtyping. Others still may be used to annotate types with other meta-information; e.g. descriptions or version stamps.

A MIM relation object represents a relation, and has roles with fixed role names and role types as explained in Section 3. The role types may be the classes of MIM type objects or general CORBA data types. A relation may be viewed as a "table" as shown in Figure 4. This example shows relationships between some type objects that are intended to represent record types and their component fields. Each row in the table represents a single relationship tuple, and each column represents a role in the relation.

	Roles		
	record type	field type	field name
Relationships	X	Y	"f1"
	X	Z	"f2"
	P	Q	"f1"
	...	...	...

Figure 4 The tabular view of a relation.

A MIM relation may be defined to have one or more *unique key constraints*. A unique key constraint ensures that no two relationships can have the set of values in the unique key roles. (In the example in Figure 4, a unique key consisting of the "record type" and "field name" roles would ensure that field names are unique

within a record type.) Every relation object implicitly has a *global uniqueness constraint* that ensures that no relationship tuple can appear more than once in the relation.

Relation object classes are descended from the "MO" class via the "RO" class as shown in Figure 3. They inherit component interfaces that define generic and specific operations for querying the relation (i.e. for finding all relationships with given values in particular roles), and for adding or removing relationships from the relation. When a relationship is added, the meta-object server for the relation must check that the new tuple does not violate the relation's uniqueness constraints.

3.3 Specialised Relations

Two kinds of type relationship occur very frequently in type systems. Types can often be modelled as *containers* for other types. For example, a CORBA interface type "contains" the types of the interface's operations. Types also commonly provide *name spaces* for other types. For example, a record type is a notional "name space" for the record fields.

The DSTC MIM defines three specialised kinds of relation object to support these uses. A *containment object* is a relation object in which two roles are designated the *container* and *contained* roles, with the contained role being a unique key, Thus a "contained" object can only be in one "container" at a time. A *naming object* has three roles designated the *namer, named* and *name* role respectively. The namer and name roles form a unique key for the relation. The naming and containment functions can be combined to give a *naming containment object* which has at least three roles, and two unique key constraints.

The interfaces for specialised relations (as defined in the IDL mapping rules) provide extra generic and specific operations to support the naming or containment paradigms. For example, a naming object has name lookup and reverse name lookup operations. The DSTC MIM implementation can be changed to support other specialised relations by extending the mapping rules.

3.4 Schema Objects

A *schema object* is a MIM meta-object that represents a type schema. A schema object is a container for a collection of types and relationships belonging to an instantiated type system. It provides "object factory" operations for creating type and relation objects in the schema, and provides the mechanisms for finding all type and relation objects in the schema. Schema objects inherit from "MO" by way of the "SO" class as shown in Figure 3.

A schema object contains a relation object for each of the type system's relation object classes, and may contain many type objects for each type object class. When these meta-objects are created using the schema's factory operations, they are

automatically added to per-class attributes of the schema object. In addition, each meta-object's "my_SO" attribute is set to refer to schema object that created and owns it. Finally, a schema object may create only one relation object of each class.

3.5 Schema Factories

Each instantiated MIM type system defines a *schema factory* class*;* i.e. an object interface for creating new schemas for the type system. This interface consists of a single operation to create a schema object. The DSTC MIM specification does not say whether the factory object for a type system is unique, or how a client would go about locating it. Indeed, there are many potential uses of the MIM where no schema factory object would be required.

3.6 Known Relations

The DSTC MIM supports two distinct models of relationships between type objects. The *relational model* is characterised by general query operations over a table of relationships for many types. The *navigational model* is characterised by operations for navigating a graph of related type objects. While the relational model is a more powerful model, it is rather heavyweight if the client has no need to perform searches. For example, using queries to find the fields of a record type object in Figure 4 would be cumbersome and inefficient.

A type object interface can "know about" the relationships that the object participates in. For example, the record type objects in Figure 4 can "know about" their relationships with field names and type objects. The type object is said to have a *known relation* or simply a *known*. The effect of providing a type object with a known is that it acquires a virtual "link" to the objects it is related to as shown in Figure 5.

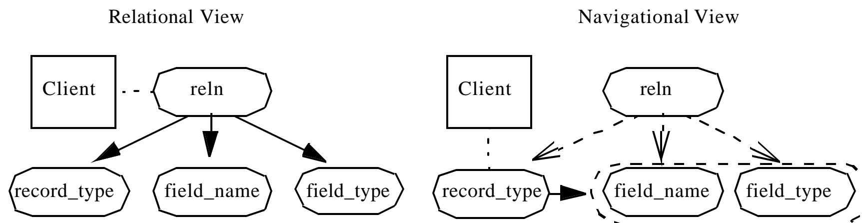

Figure 5 The relational and navigational views.

The definition of known relation of a type object specifies the role that the type object fills in the relation (the *known role*), and a *known name* by which the type object interface will refer to it. The known causes extra operations to be inherited by the type object class. These allow a client to find objects related to "this" object to create relationships between "this" object and others.

A complete and detailed description of these objects is available in [10].

4 CONCLUSIONS

In summary, this paper has introduced a conceptual framework for describing meta-information and meta-meta-information, and has described the design for a meta-information manager, which was successfully prototyped in CORBA. The key features that differentiate the DSTC MIM design from previous type management and repository research are as follows:

- Heterogeneity: the design supports multiple, arbitrary type systems.
- Extensibility: the design allows new type systems to be added at any time, either starting from scratch or by composing or extending existing type systems.
- Openness: the design allows multiple MIM servers to be federated into a seamless distributed meta-information service.

ACKNOWLEDGEMENTS

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

REFERENCES

[1] “Common Facilities RFP-5: Meta-Object Facility”, OMG TC document cf/96-05-02, Object Management Group, June 1996.

[2] “Open Distributed Processing: Reference Model”, ISO/IEC 10746, 1996.

[3] “ODP Type Repository Function” ISO/IEC CD 14769, 1997.

[4] “A Type Management System for an ODP Trader”, by Jadwiga Indulska, Mirion Bearman and Kerry Raymond, in Proc. of the IFIP TC6/WG6.1 Int. Conf. on Open Distributed Processing (ICODP-93), Berlin, September 1993, North-Holland, pp. 169-180.

[5] “Types and Their Management in Open Distributed Systems” by Wayne Brookes, Stephen Crawley, Jadwiga Indulska, Douglas Kosovic and Andreas Vogel, Journal of Distributed Systems Engineering (to appear in 1997).

[6] “Type Management using Type Graphs” by Stephen Crawley, in Proc. DSTC Symposium ‘96, Brisbane. 11-12 July 1996.

[7] “The Common Object Request Broker: Architecture and Specification”, Revision 2.0, Object Management Group, April 1995.

[8] “The COMANDOS Distributed Application Platform”, Vinny Cahill, Roland Balter, Neville Harris and Xavier Rousset de Pina (eds.), ESPRIT Research Reports (Project 2071), Volume 1, Springer-Verlag, 1993.

[9] "Universal Repository (UREP)" (web pages), UNISYS. http://www.unisys.com/marketplace/products/urep/

[10]"Meta-Object Facility", OMG TC document cf/97-01-01, Linnæus Project, DSTC, January 1997.

[11]"Object Oriented Modelling and Design", J.Rumbaugh et al, Prentice Hall, 1991.

BIOGRAPHY

Stephen Crawley is a Senior Research Scientist at the Defence Science Technology Organisation. Scott Davis is a Professional Officer at the Defence Science Technology Organisation. Jaga Indulska is a Senior Lecturer in the School of Information Technology, University of Queensland. Simon McBride is a Research Scientist in the CRC for Distributed Systems Technology. Kerry Raymond is a Senior Consultant at CiTR Pty Ltd.