

## Types and their management in open distributed systems

This article has been downloaded from IOPscience. Please scroll down to see the full text article.

1997 Distrib. Syst. Engng. 4 177

(<http://iopscience.iop.org/0967-1846/4/4/001>)

View [the table of contents for this issue](#), or go to the [journal homepage](#) for more

Download details:

IP Address: 38.107.179.213

The article was downloaded on 20/02/2012 at 07:47

Please note that [terms and conditions apply](#).

# Types and their management in open distributed systems

Wayne Brookes<sup>†</sup>, Stephen Crawley<sup>‡</sup>, Jadwiga Indulska<sup>†</sup>,  
Douglas Kosovic<sup>§</sup> and Andreas Vogel<sup>§</sup>

<sup>†</sup> CRC for Distributed Systems Technology, School of Information Technology, The University of Queensland, Brisbane, Queensland, 4072, Australia

<sup>‡</sup> Defence Science and Technology Organisation, PO Box 1500, Salisbury, South Australia, 5108, Australia

<sup>§</sup> CRC for Distributed Systems Technology, Level 7, Gehrman Laboratories, The University of Queensland, Brisbane, Queensland, 4072, Australia

Received 28 May 1996

**Abstract.** Open distributed processing aims to support cooperation within and between large-scale heterogeneous and autonomous computing environments. An inherent issue in such environments is enabling the interoperation of objects whose interfaces have been defined in different type models. In this paper, we present a type management system which provides a means for representing, storing, retrieving and translating types, and for expressing and evaluating relationships between types in a heterogeneous distributed computing environment. This system allows multiple type languages and models, and can relate types expressed in different ones. The type management system is designed to support the instantiation and dynamic binding of objects, run-time type checking of object interactions, and the discovery of new resources (e.g. services) within the system.

Current approaches to interface definition in distributed systems are mainly based on the use of a single interface definition language (IDL). While this provides a level of common agreement about the types of system interfaces, the type models of existing IDLs are not rich enough to model either the overall architecture of a system or the behaviour of objects. We illustrate this by briefly describing some aspects of an enhanced type model with the emphasis on the model's impact on the design of the type management system.

## 1. Introduction

Current trends in distributed computing are towards systems which extend cooperation of applications beyond the boundary of one system, i.e. beyond one homogeneous platform for distributed computing or one distributed operating system. This open cooperation has to deal with heterogeneity and autonomy of systems present in the environment. The RM-ODP work on modelling open distributed systems [19], takes an object-based view of the components of distributed systems. While we will use an object-based approach throughout this paper, we note that some popular distributed-system modelling paradigms are not object-based. Furthermore, our approach to type management is quite capable of supporting these non-object-based paradigms.

The open, heterogeneous nature of a distributed environment introduces many problems which are not encountered (or are less complex) in closed or homogeneous systems. The most important of these problems are as follows.

- Meaningful and type safe interactions are often difficult due to mismatching component object interfaces.
- The evolution of systems entails the introduction of

new objects with new types, often as replacements for older versions. A variety of interface compatibility problems can arise when these new objects need to interwork with existing objects. Such problems need to be solved with a minimum of effort and disruption to normal working.

- Discovery and re-use of objects (e.g. services) is much more difficult and requires supporting services for management of object and object types.

The above problems can be variously ascribed to differences in the interface type models or languages used to describe object interfaces, to lack of expressiveness in the type models, and to inconsistencies in the system's overall object model or schema (if it exists). When component object interfaces are defined using different type languages, the major areas of discrepancy include [9, 24]:

- differences in the notion of an object, e.g. visibility or non-visibility of the object state, attributes, separate model constructs representing object relationships, generic functions, number of interfaces allowed (one or many),
- differences in type systems including differences in the basic types and type constructors, and in the meanings of inheritance and subtyping,
- differences in modes of interaction between objects

(e.g. synchronous or asynchronous communication),

- differences in the granularity of objects supported by the type model.

There are two ways to facilitate the interoperation of distributed systems that use different type models. One approach is to define a canonical type model, and to define mappings between this model and the other type models. The second approach is to define one-to-one mappings between various type models. In either approach, when the differences between type models are too great it is impossible to define perfect mappings. Fortunately, it is often sufficient to use imperfect mappings; i.e. ones in which some constructs are not supported, or some interactions involve precision, type safety or type constraint compromises.

To allow large-scale interoperability of heterogeneous distributed systems it is necessary to support

- *dynamic type checking*, which compares two types to determine whether they satisfy some relationship; e.g. one that ensures type safe interaction between dynamically bound objects,

- *dynamic type matching*, which searches for a set of types that satisfy a relationship with a given type; e.g. to facilitate discovery and trading of resources.

It is desirable that dynamic type checking and type matching should support both generic types and subtyping relationships, where these are applicable.

A considerable amount of work on interoperability in heterogeneous systems already exists. The Concert/C [3] approach was to add extra types to the C programming language to support distributed objects. A more popular alternative is to develop an interface definition language (IDL) for interface specification that is separate from the system implementation language(s). Well known examples of this approach include DCE [31,32], CORBA (and OMA) [26,29], ANSAware [2] and ILU [39]. These typically allow interoperation between different programming languages, and across different operating systems and hardware platforms. Manola and Heiler [24] explore the canonical type model approach for mapping between type models in distributed object systems, while the CORBA/COM interoperability proposal [30] and ILU's CORBA interoperability [39] are examples of one-to-one mappings between type models.

There has also been significant work on extending interface type definitions beyond syntactic signatures. RM-ODP [19], [21] and [23] cover aspects of the specification of behaviour, and [41] discusses interface matching based on behavioural signatures.

Our research on the interoperability of large-scale, heterogeneous systems has two aspects. First, we are working on type management systems to provide a number of important interoperability-related functions as part of a distributed systems infrastructure. We have developed a prototype type management system (hereafter called the type manager) that provides a means for the description, storage, retrieval, and translation of types and their relationships. The type manager manages a persistent repository of type information which has been designed to facilitate:

- dynamic type matching and type checking by various clients; e.g. the runtime binding functions of distributed computing environments, support services such as traders and service bridges, and end users using resource browsers or 'universal client' tools,

- use of types expressed in many type models and languages and relationships within and between models,

- integration with tools that support the software engineering process.

The long-term aim is to support interoperability of objects across different type models, evolution of system components, and service resource discovery. This work is the main focus of this paper.

The second aspect of our research is on type models with more expressive power than current generation IDLs. We have developed an enhanced type model and type description language that supports behavioural descriptions and complex bindings. This work is only discussed briefly in this paper; for more detail refer to [8,9]. The role of this description is to show the impact of the model on the design of our type management system.

The remainder of this paper is organized as follows. Section 2 briefly presents our type model and section 3 focuses on type relationships that are important to the issue of interoperability. Section 4 describes the role of type management infrastructure in future distributed computing environments, and section 5 presents our approach to the design and implementation of our type manager. Section 6 discusses the potential role of the type manager in the software engineering process and other issues.

## 2. Type model

Our main goal in developing a new type model is to enhance 'signature-based' object types with additional behavioural information. In particular, we want to be able to describe the way an object communicates with other objects and to express quality of service (QoS) constraints for this communication. This behavioural information can be used for more accurate interface type checking and to improve the quality of type matching results. If we can express the interaction patterns of objects as types, comparing these types increases the likelihood that object interactions will be meaningful.

Our type model defines object types as message signatures (syntax) expressed using a standard set of basic data types and constructors, behaviour expressed in terms of an object's patterns of interaction with other objects together with other functional or non-functional constraints. The model defines the types of complex runtime bindings involving two or more objects. Finally, it defines the types of relationships between groups of types, and metatypes for classification.

The remainder of this section briefly describes the types which constitute our type model.

### 2.1. Basic data types and type constructors

The basic data types and constructors of our model were chosen by surveying a number of existing type models and

languages [10, 9] including those of CORBA [29], ASN.1 and GDMO [17, 18], DCE [31, 32] and Concert/C [3], and designing a set of types that gave good coverage. The set chosen is not maximal (i.e. is not a superset of the data types of all the aforementioned systems), but rather a set which is useful for describing interactions in open systems. Our analysis indicates that it is possible to define mappings between our basic data types, and those of other type languages. However, we do expect that there will be some inevitable loss of information (e.g. range and precision), or loss of type constraints in some cases.

## 2.2. Complex types

Our type model defines a number of complex types (e.g. types of objects, interactions, interfaces and bindings), which are necessary for describing object interactions in an open system. In addition, the model defines relationship types that describe relationships between types that are managed by the type manager. The model may be extended with new kinds of complex types in the future.

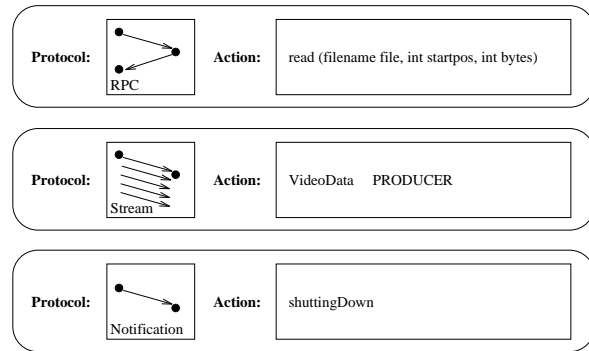
**Object types.** Objects are the basic providers of behaviour in an open distributed system. Object types describe the visible portion of an object's behaviour, first in terms of the manner in which other objects may interact with it, and secondly by the description of other functional and non-functional constraints on the object's visible behaviour which are not described by the behaviour of the individual interactions.

In our model, an object type defines the set of interactions (described by interaction types below) that the object is able to participate in. These interactions are partially ordered sets of messages exchanged with other objects. An object type's interaction types can be divided into one or more interfaces by which the object interacts with its environment. An object can provide multiple interfaces, and interact simultaneously using more than one of them.

**Interaction types.** An interaction type describes a pattern of communication that constitutes some meaningful behaviour that is visible to an external observer. For example, a file-server client performing an RPC operation to read part of a file would typically be described as one interaction. Interaction subsumes concepts such as 'operations', 'remote procedure calls', 'streams', 'notifications', as well as more complex patterns such as remote procedure call with a callback, or a multiparty commercial EDI transaction.

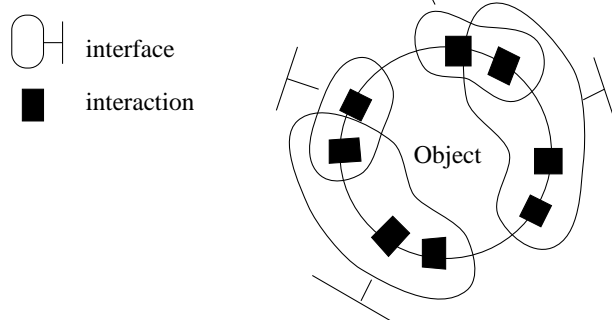
There are two distinct parts to an interaction type: the protocol (the communication pattern) and the action. Action types describe *what* the interaction does, while protocol types describe *how* the action is to be carried out (how the object communicates with other objects). Action types consist of an action signature as well as an informal description of the semantics of the action.

In practice, most interactions in distributed systems can be expressed using a small repertoire of common abstract patterns or protocols; e.g. 'RPC', 'streams' and 'broadcast'.



**Figure 1.** Example interaction types, (a) RPC, (b) stream, (c) notification.

### Legend:



**Figure 2.** Objects, interactions and interfaces.

For this reason, the type model allows the definition of *protocol types* which consist of partially ordered sets of typed messages and associated characteristics. Complex protocol types can be built up from simple typed messages, and from other protocols. For example, one way to construct a callback RPC protocol is by combining two instances of an RPC protocol. This approach allows us to model the entire range of message exchanges that occur in distributed systems, without having to rely on predefined protocol primitives.

Interactions in distributed systems are causally ordered and can have associated timing and QoS constraints. For example the definition of a stream interaction type which defines a behaviour of an object sending a video may include QoS parameters such as latency, jitter and error rate. As this interaction type is part of the video interface type definition, the QoS parameters may be taken into account when matching interface types for a binding.

Some example interaction types are shown graphically in figure 1.

**Interface types.** With object types and interaction types as defined above, we can define interface types as groupings of an object's view of its interactions. An interface is a view of an object's behaviour that is useful for a particular purpose. The corresponding interface type consists of a set of interactions and any other object constraints that pertain to the interface. An object type may offer multiple interface types, and the sets of interaction types

```

interface FTPClient {

    type errorcode : ENUM { LostConnection, TooManyErrors, ClosedByRemote };
    type FileData : SEQ OF CHAR;

    RPC.client get (filename:STRING) : OK() | noSuchFile() |
        permissionDenied()
    [ ... ];

    Stream.sink getFile (indata:FileData) : OK() | NotOK(err:errorcode)
    [
        maximise frame_rate > 100 < 1000,
        minimise loss = 0
    ];

    Notification.receive shuttingDown ()
    [ ... ];

    behaviour [

        ( get ; if (get.OK) {getFile} )
        [> ( shuttingDown )

    ]
}

```

**Figure 3.** An example interface type definition.

in these interfaces may overlap. The relationship between interfaces, interactions and objects is shown in figure 2.

An example of an interface type is shown in figure 3. This type description defines a client side of a hypothetical file transfer protocol. There are three different protocols involved: RPC, stream (for file transfer), and notification about errors. It is assumed that all these types have been already defined (including their behaviour). The interface type definition specifies that the objects will communicate using these protocol types and also specifies (using a subset of Basic LOTOS) how they are related together in this interface type.

**Binding types.** In general, a binding is something that connects a number of objects together so they can interact. More precisely, our chosen distributed architecture model defines a binding to be a context in which a particular interaction occurs [5]. Most existing distributed application platforms have a limited view of binding (e.g. ANSAware [2], ‘RISC’ Object Model [3], COMANDOS [12], CORBA [29], DCE [31,32]). These systems typically define a binding as a connection between two objects, possibly also allowing one-to-many connections to model broadcast or multicast.

The distributed architecture model [5] designed in the Distributed Systems Technology Centre (DSTC) supports more complex bindings involving multiple-participant objects, each with a distinct role in the binding. A binding type defines the roles in terms of the required types of the participant objects (including their respective interaction types), and includes a specification of the binding’s information flows between the roles. Binding types may also contain a contract type to further constrain the interactions which may occur in the context of the binding [25].

**Relationship types.** A key requirement of the type manager is that it can represent useful relationships between types. There will be many kinds of types in an open, heterogeneous system, and many kinds of relationships between those types. Examples of the relationships that are useful in the distributed systems context include type equivalence, compatibility, substitutability (i.e. subtyping) and managed\_by. These relationships may have substantially different meanings in different type models; i.e. for types from models other than the model described in this section. Other useful relationships will involve three or more types.

Our type model recognizes the need for flexibility and openness in expressing type relationships by defining a special kind of type to describe these relationships. Our definition of relationship types is influenced by the ISO general relationship model [15], one of the OSI management series of standards. The definition given here is an adaptation and extension of the ISO model.

Formally, a type relationship is a tuple of types of predefined kinds (metatypes) from our type model. A *relationship type* consists of a set of roles, each one constraining the types that may fill that role in a relationship of that type. The roles in a relationship type consist of a role name, a metatype for the role and the role’s cardinalities (required and permitted). The relationship type may also specify additional semantics for the relationships. For example, homogeneous binary relationships (i.e. 2-tuples of the same kind of type) have reflexivity, symmetry and transitivity properties. If it is possible to automatically derive a particular relationship from the respective definitions of type (e.g. ODP subtyping), then the relationship-type definition can also specify a method or tool for deriving the relationship.

The relationship types in our model define relationships between types belonging to the model. The concept,

however, is easily extended to relating types in different type models.

**Metatypes.** In order to support classification of types, we introduce the notion of metatypes. A metatype is the type of a type, or more precisely, a predicate which defines a class of types, e.g. ‘interface types’, or ‘relationship types’. These metatypes may then be used when referring to the class of types as a whole (e.g. when defining relationships between classes of types). This classification concept used in our type model has been extended for the type management system to include metatypes belonging to various type models. Examples of such classifications (metatypes), are presented in the sections describing the type management system.

### 3. The role of type relationships

Relationships between types play a central role in type management for open systems<sup>†</sup>. In homogeneous systems (i.e. systems with a single type model), there are typically only one or two relationships that are important, e.g. interface equivalence and interface compatibility. These relationships are typically implicit in the type model (e.g. allomorphy in GDMO [18]) or the implementation of the model (e.g. interface compatibility in DCE).

In an open, heterogeneous environment, type relationships are more complex. Common relationships such as equivalence and subtyping have different semantics in different type models. In addition, interoperation of objects from different platforms implies that there are relationships between object types in different type models.

Possibly the most important relationship in distributed systems is the substitutability relationship. This is the relationship between object types that determines whether an object with one type can be safely substituted for an object of a second type. This relationship is also referred to as compatibility or subtyping.

Different type models have widely varying notions of substitutability. In the most commonly used type models [2, 19, 26, 29, 31, 32], substitutability is based on the syntax of an object’s operations, i.e. on the names and (concrete) types of the operation arguments and results. In a few cases, substitutability also includes some semantic aspects [1, 21, 22].

By describing each form of substitutability as a distinct relationship type, we allow users to distinguish between the forms of subtyping, and to select the form that is best for a given context. In some cases, the relationship will be a computable function. In other cases, especially those involving semantic aspects, the relationship instances may need to be asserted by a system administrator or domain expert. The difference between the relationship types from our type model and the type manager’s view of relationship

<sup>†</sup> Relationships between objects, or between types and objects are separate issues. Relationships between objects can be represented variously using standard interfaces to relational and object-oriented database products, or other services such as the CORBA relationships service [28]. The relationship between objects and their types is typically supported directly by the IDL to programming language mapping.

types is that the type manager supports types from different type models.

When we consider interoperability across type models (e.g. between CORBA and DCE), substitution of types will involve a mapping between the type domains [11]. This will typically need to be expressed as a ternary relationship, e.g. ‘CORBA-I is substitutable for DCE-I under mapping M’. The mapping type ‘M’ could be expressed in a mapping language that can be interpreted by a (hypothetical) universal protocol bridge. In some situations, this type could possibly be generated automatically or semi-automatically. Alternatively, the mapping type could simply encode the name of a programmer-defined bridging service. In each case, extra functionality is needed in a distributed system’s binding function to establish network connections through the appropriate protocol bridges. In practice, this functionality is likely to be a subset of that needed to handle complex bindings as described previously.

### 4. The role of the type manager

In the previous section, we explained the importance of type relationships in open, heterogeneous distributed systems. In this section we discuss the functionality needed to manage types and their relationships in a practical system.

In typical first- and second-generation distributed systems (e.g. those epitomized by socket-level programming and RPC respectively) interface-type checking mechanisms were relatively crude. Typically, an interface type was represented by an interface identifier with an associated version number or timestamp. Interface checking consisted of comparing interface identifiers tendered by the client and server. Type safety often depended on the programmer obeying certain conventions such as updating version numbers when interfaces were changed. Little if any interface type information was available at runtime, meaning that interface mismatch problems were generally fatal errors, and many ‘dynamic’ services were impossible to implement.

One of the major contributions of the RM-ODP model [19] is that it highlighted the need for a persistent repository of types and type relationships in open distributed systems. The type repository is seen as underpinning the functionality of both the distributed computing infrastructure, and a variety of user applications. The main users of the type management services provided by such a repository are as follows.

- The distributed system infrastructure, which requires type information to:
  - perform the runtime type checks needed to ensure that dynamic binding and invocation of services occur in a type safe manner,
  - instantiate services that have transient existence,
  - establish complex bindings involving multiple participants,
  - support interoperability (mapping) of incompatible interfaces, both within and between type domains.
- Service objects that support the distributed environment (e.g. traders), which need type information to perform their functions.

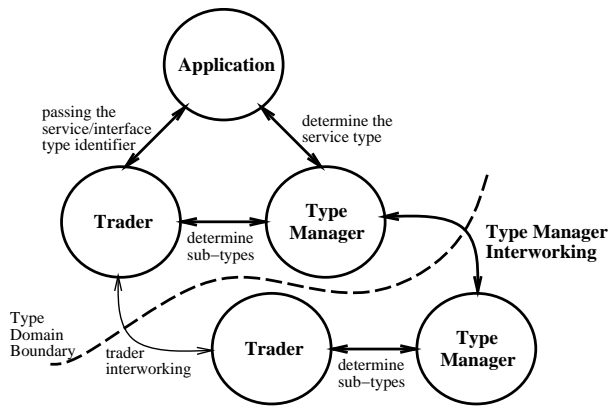


Figure 4. A scenario of interworking traders.

- End users (e.g. using resource browsers), who need type information for resource discovery of both services and information.

- Type manager administrators, who need to access and define type information, both to configure a type manager, and to federate it with type management services for other domains.

In addition, the type manager can support a number of aspects of the software engineering lifecycle, including system modelling, interface design, program design and implementation, debugging, system installation configuration and maintenance. Indeed, the level of integration of the software engineer's tools with type management services is likely to considerably influence their ability to manage the system's evolution.

We can illustrate the utility of type management services with an example involving trading for application services. A trader [4, 14, 16] is an infrastructure service that acts as a broker for services available in a distributed system. Service providers export to the trader a description of the services they wish to make available to clients. The trader records these service offers, and uses them to satisfy import requests (i.e. service location requests) from potential clients. To do this, it needs to know whether a service requested by a client is compatible with that offered by a provider. This can be determined by using the type manager to query the appropriate compatibility relationship for the pairs of types, or by obtaining the set of all types that match as compatible with the client's request type.

A large-scale distributed environment will typically be divided into domains, with a domain containing one or more traders to manage the service offers in the domain, and one or more type managers to handle type information.

A client in a given domain uses its local trader to request services independent of their location. To support this, the local trader must interwork with remote traders to obtain service offers from other domains. In addition, since the service types for offers from remote domains may not be recorded locally, the local type manager must interwork with remote type managers to resolve queries. An example of interworking traders is shown in figure 4.

The problem of federating type managers is similar to that of federating traders, and the two problems cannot

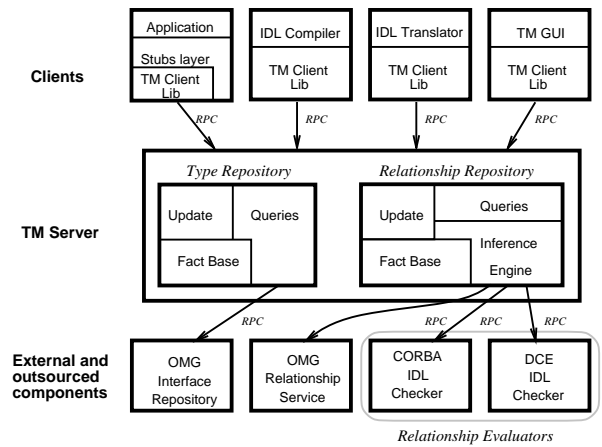


Figure 5. Architecture of the type manager.

be solved independently of each other. A more detailed discussion on type problems with interworking traders is given in [36]. The ODP trader is currently under standardization by both ISO and ITU. This will result in a standard high-level interface specification [16] which can be expressed in various IDLs. The issues of federation of type managers are discussed in section 6.1.

## 5. Architecture and functionality of the type manager

To date, we have designed and implemented two generations of the type manager. The first generation met the basic requirements of storing and retrieving types and relationships between types [6]. It used a simple representation for type descriptions (byte strings) and allowed relationships between types to be asserted by users and inferred from the relationship characteristics such as transitivity. The system was an advance on the primitive type managers used in distributed platforms such as ANSAware, but it could not cope with

- multiple languages for expressing type descriptions,
- individual types with descriptions in multiple languages,
- type relationship queries that required understanding of type definitions (i.e. beyond string comparison),
- federation of type managers, or
- cooperation with other sources of type information, e.g. the interface type repository provided by the CORBA/OMA architecture [26].

The following sections describe the design and implementation of our second-generation type manager. The design aims to satisfy the general requirements of supporting large-scale, open, heterogeneous distributed computing, as well as addressing the specific shortcomings of the first-generation type manager. The design was influenced by our enhanced type model, without being dependent on it.

## 5.1. Type manager architecture

The type manager consists of a number of functional components integrated under a common interface as illustrated in figure 5. The main components of our type manager are as follows.

- The control and management component: this integrates the functionality of the other components into a single API.
  - The type repository: this provides persistent storage and retrieval of type definitions, including those for the types of relationships in the relationship repository.
  - The relationship repository: this provides persistent storage and retrieval of relationships between types. The repository includes an inference engine that can infer type relationships based on a combination of known relationship type characteristics, asserted relationship ‘facts’ and relationships derived by the relationship evaluator.
  - The relationship evaluator: this supports the dynamic derivation of type relationships based on the definitions of types.
- The overall type management architecture also includes a number of functions provided by separate programs.
- The user interface tool (TM GUI): this allows the user to interactively add, update, query and browse type descriptions and relationships.
  - The DCE IDL compiler: this compiles DCE IDL into an intermediate form and stores this and the original IDL source code in the type repository.
  - The IDL translator: this performs syntactic translations between DCE and CORBA IDLs using the type manager to hold the intermediate form.

While the type manager was designed to allow interoperation with out-sourced components like the OMG interface repository [27] and OMG relationship service [28], this interworking has not yet been implemented. Interoperation with (for example) the OMG interface repository would allow re-use of existing type information, and tools such as proprietary IDL compilers that interact with it. However, it would also introduce the possibility of inconsistency between the different sources of type information.

The type manager is currently implemented for the DCE platform. The server software and associated programs are largely coded in C. The exception is the inference engine module which is a hybrid of C and Prolog. In the following subsections we will discuss four basic components of our type management architecture, namely the type repository, the relationship repository, the relationship evaluator and the type translation components.

## 5.2. Type repository

The type repository stores *type definitions*, whose information content is illustrated by figure 6.

A type definition is an object with a globally unique identifier<sup>†</sup>. The object consists of a set of zero or more

<sup>†</sup> In the current implementation, type identifiers are DCE universal unique identifiers (UUIDs) in printable string form, although any suitable globally unique identifier would suffice.

*type descriptions*, each describing some aspect of the type; e.g. its signature, behaviour, etc. The descriptions are represented as byte strings with an associated textual *language name*. A type definition object may also include one or more textual annotations of a non-definitive nature.

The type descriptions that make up a type definition object may be expressed formally or informally (e.g. in English). The descriptions may express complementary type information or present different views of the same type, using different languages. For example, an interface type definition might contain an IDL component (expressing the interface’s syntax), a LOTOS [7,20] component (expressing aspects of the interface’s behaviour) and an English-language component. In addition, as we shall see later, a type definition can include both ‘source’ and ‘compiled’ forms of a given description.

When a type object is created, the client supplies the initial set of descriptions and annotations, along with the type identifier for a metatype, and a list of textual names for the type. The metatype tells the type manager what ‘kind of type’ the newly created type object denotes. The names allow a client to refer to type identifiers by name. (The current type manager has its own persistent type name space. Future versions should use a standard directory service.)

The core type manager has no knowledge of the individual or collective meanings of the type descriptions or of the languages in which they are expressed. Indeed, the current implementation does not even know what sets of descriptions constitute a valid definition for a given metatype. This knowledge is assumed to be vested in non-core components (e.g. IDL compilers and relationship evaluators), and in the clients of the type manager.

For the purposes of this paper, we will express type definitions in a Prolog-like syntax [13,33]. In the following example of a type with two descriptions,  $\langle dce\_interface\_type \rangle$  denotes the type identifier for all DCE interface types,  $\langle foo\_interface \rangle$  denotes a type identifier, and the matched pair of square brackets (Prolog list syntax) encloses the types’ descriptions.

```
type(<dce_interface_type>, <foo_interface>,
    [desc("Informal",
          "The Foo service processes equibalanced
          elephants"),
     desc("DCE IDL",
          "[version(1.1), uuid(...)] interface
          foo{...}")
    ]).
```

Similarly, we will use Prolog-like notation to express *predicates* of interest. For example we can define a predicate for complete (fully specified) DCE interface types as follows, where *descs\_include\_lang* is a predicate that matches lists containing a description in the specified language.

```
complete_dce_type(T) :- type(<dce_interface_type>,
                             T, DL),
                        descs_include_lang(DL,
                                             "DCE IDL").
```

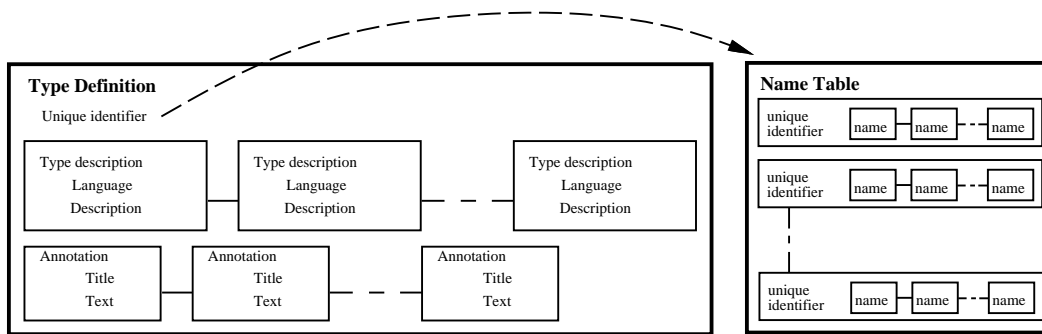


Figure 6. Data structure for type definitions.

The type repository provides a set of operations which allow a client to add and delete types, and also to browse and query the repository. The query operations are limited to string and pattern matching on the language and description strings. The human user can browse, query and update the type repository using the graphical user interface tool described in section 5.6 below.

### 5.3. Relationship repository

While the type repository holds information about individual types, the relationship repository contains the information about how they relate to one another. This information is represented by the type manager as tuples of type identifiers known as *relationships*.

Each relationship known to the type manager belongs to a *relation*, with a similar meaning of relation as in the relational database (RDB) community. The relation has a fixed number of named *roles* (cf relational attributes) that may be filled by types that belong to a specified metatype (cf relational attribute types). Each individual relationship is analogous to a tuple in an RDB relation.

Relations in the type manager are described by *relationship types*. These give the number and type of the relation's roles, and each role's *cardinality*. A relationship type may also specify one or more *properties* for a type, and if the relationship can be derived by computation, the name of a *relationship evaluator*.

The cardinality of a role consists of a pair of natural numbers that constrain the number of times a given type object may appear in a role. For example, if role 1 of relation 'R' has cardinality [0..3], then for any given type 'T' with the appropriate metatype, there can be between 0 and 3 relationships of the form  $R(T, \dots)$  in the relationship repository. In practice, the type manager treats cardinality constraints as advisory only<sup>†</sup>.

Relationship properties are used to represent information about relationships that can be used to express consistency constraints and inference rules. For example, all

<sup>†</sup> Strict enforcement of non-zero minimum cardinalities presents some tricky problems. For example, if a given role has minimum cardinality of 1, when a type object is created with the role's metatype a corresponding relationship must be created simultaneously. The only way to support this is to make updates to the type and relationship repositories transactional, and check the cardinality constraints at commit time.

homogeneous binary relationships have properties of transitivity, symmetry and reflexivity.

While a relationship's properties can be expressed in an open-ended way (e.g. as  $\langle name, value \rangle$  pairs), the meaning of the properties must be known to the relationship repository if they are to have any effect. Currently, support for properties such as transitivity, symmetry and reflexivity is 'hard wired' into the relationship repository in the form of Prolog consistency and relationship inference rules. While we would like users to be able to define new properties, this is not supported in the current implementation.

As we have previously stated, the core type manager has no knowledge of the representation of type descriptions. However, it does have the capability of invoking an external relationship evaluator to see if a particular relationship holds for a given type tuple. To allow this, the relationship type specification may include the name of an evaluator service for the relation. Relationship evaluators are discussed in more depth in section 5.4.

In fact there is one exception to the above paragraph. The types of the relationships managed by the relationship repository are actually stored as types in the type repository. A simple relationship description language (RDL) has been defined and the core type manager includes a parser and unparsers for this language.

As a first example of how relationships are defined and used, let us consider how the type manager handles metatypes. Recall that when a type object is created, the client supplies the type identifier for the type's metatype. In the current implementation, the association between a type and its metatype is represented by the *is\_of\_type* relation. The relationship type for this relation has the form:

```
type(<relationship_type>, <is_of_type>,
  [desc("Informal",
    "The built-in is_of_type relation
    associates types with
    their immediate metatypes"),
  desc("RDL",
    "relation is_of_type {
      roles
        type:    <type> [1 .. 1],
        metatype: <type> [0 .. infinity];
    };")
]).
```

This says that (*is\_of\_type*) denotes a relationship type that describes a binary non-homogeneous relationship between

types and their corresponding metatypes. The cardinality constraints on the roles mean that every type must have precisely one metatype. (The astute reader will notice that the `is_of_type` relationship is underspecified. To fully specify the relationship we need to express different ‘kinds’ of type; e.g. a type lattice with multiple dimensions.)

Our second relationship example describes substitutability of DCE interface types, i.e. the relationship that must exist between DCE client and server interfaces for binding and type-safe RPC. DCE interface substitutability is a binary homogeneous relationship, i.e. a relationship between two DCE interface types. DCE interface substitutability is transitive, antisymmetric and reflexive, and is a computable function (for this meaning of substitutability). We can define DCE interface types and the substitutability relationship type as follows:

```
type(<metatype>, <dce_interface_type>,
  [desc("Informal", "The type of all DCE
    interface types")]).

type(<relationship_type>, <dce_substitutable>,
  [desc("Informal",
    "This relationship determines whether
    a DCE client can use
    a DCE server in a type-safe fashion"),
  desc("RDL",
    "relation dce_substitutability {
      roles
        server_type: <dce_interface_type>
          [0 .. infinity],
        client_type: <dce_interface_type>
          [0 .. infinity];
      properties
        "transitivity": "transitive",
        "symmetry": "antisymmetric",
        "reflexivity": "reflexive";
      evaluator
        "DCE-interface-subst-eval-1.1";
    };")
  ]).
```

Assuming that each distinct DCE interface definition (expressed in DCE IDL [40]) has a DCE interface type, the first clause says that the `<dce_interface_type>` identifier denotes a metatype that represents the type of all DCE interface types. The second clause then defines the substitutability relation.

Given the above relation definition, we can add facts to the relationship repository to assert that particular types are substitutable. For example, given the following DCE interface types:

```
type(<dce_interface_type>, <foo_v1.1>,
  [desc("DCE IDL", "[version(1.1), ...]
    interface foo{...}")]).
type(<dce_interface_type>, <foo_v1.2>,
  [desc("DCE IDL", "[version(1.2), ...]
    interface foo{...}")]).
```

we can assert that `<foo_v1.2>` is substitutable for `<foo_v1.1>` as follows:

```
dce_substitutable(<foo_v1.2>, <foo_v1.1>).
```

The type manager provides operations for adding and deleting relationship types and specific relationship information. Other operations allow a client to browse the relationship repository, and to issue a wide range of relationship queries. For example, the client can ask ‘is `<foo_v1.2>` substitutable for `<foo_v1.1>`’, ‘what types are substitutable for `<foo_v1.1>`’, and even ‘what relationships exist between `<foo_v1.2>` and `<foo_v1.1>`’.

Depending on the relationships involved, the type manager uses a combination of methods to satisfy queries. It can consult the relationship repository to see if any of the asserted relationships are relevant to the query. If the relationship type has an evaluator, it can use that to compute further answers to the query. If the relationship type has properties, the type manager can use logical inference to deduce the relationship from others. (For example, if `<foo_v1.2>` is substitutable for `<foo_v1.1>` and `<foo_v1.3>` for `<foo_v1.2>`, then the type manager can deduce that `<foo_v1.3>` can be substituted for `<foo_v1.1>`.)

The current type manager uses multiple copies of a Prolog-based inference engine to satisfy relationship queries. Relationship types and asserted relationships from the persistent repositories are mirrored as ‘facts’ in the Prolog databases. Knowledge of relationship properties (e.g. transitivity, etc) is embedded in the inference engine as Prolog ‘rules’.

The relationship inference system also serves to maintain the consistency of the relationship repository. When a relationship is asserted, the inference engines check that the type objects conform to the relation’s role type and cardinality constraints. In addition, if the relation has defined properties, the inference system checks that these properties are not contradicted by the new ‘fact’. The inference system also avoids recording facts that can be inferred from other facts.

#### 5.4. Relationship evaluator

The relationship repository manages asserted facts about relationships between types, and can infer further facts from these and the relationship properties. However, since the relationship repository does not understand type descriptions stored in the type repository, it cannot directly deduce relationships from them.

In our type manager, algorithmic analysis of type definitions to determine relationships is performed by the relationship evaluator component. This component consists of a small amount of control code in the core type manager, and a number of non-core evaluator servers that implement specific relationship evaluations.

On receiving an RPC request from the core type manager, a relationship evaluator retrieves the required type definitions from the type repository. Next, it analyses the type definitions using its built-in knowledge of the type description language(s) and the relationship semantics. The result of the analysis is returned to the core type manager allowing it to continue with the rest of the client query.

Our approach of implementing relationship evaluation in non-core services has some important advantages. First, it allows us to keep the size of core type manager

manageable. Second it allows us to implement new type languages and new relationships without changing the core type manager software. Third, it allows us to separate the (relatively large) computing overheads of relationship evaluation from other parts of query processing and spread them over a number of machines. One disadvantage of this approach is the overheads of the multiple RPCs involved in relationship evaluation.

Relationship evaluators can be built for a wide range of computable relationships between types. For example, we have shown that it is relatively easy to implement syntactic interface equivalence and substitutability within a single IDL (e.g. DCE or CORBA) by adapting known type checking algorithms. Similarly, it is postulated in [38] that it should be possible to determine if interface types expressed in different IDLs are interoperable under some mapping of operations and types. Given well defined semantics for construct mapping and a user-specified correspondence of types and operations, this kind of computation is clearly feasible.

To date we have implemented two generations of relationship evaluators for DCE IDL. The first generation of evaluators translated the DCE and CORBA IDL source code into abstract syntax trees (ASTs) using the *kimwitu* compiler generator [35] and then used these trees as the basis for comparison. One evaluator implements interface equivalence by checking for syntactic isomorphism of the respective interface ASTs. A second evaluator tests for DCE interface substitutability by pairwise comparison of the operation ASTs and comparison of the interface's UUID and version numbers.

The problem with the AST-based approach is that it does not cope with cosmetic differences in the IDL specifications. In the case of DCE IDL, it was also clear that a purely syntactic analysis could not cope with equivalent but non-isomorphic attribute specifications.

Our second generation of relationship evaluators uses an intermediate representation for DCE interface types designed to make deep type analysis easy. The representation is graph based rather than tree based, with name references resolved to pointers to the objects (typically type nodes) that they refer to, and all constant expressions evaluated. We also convert DCE attribute information to a canonical form and propagate it to the appropriate nodes. This type graph information is used by a family of DCE interface relationship evaluators that can (for example) determine whether two interfaces use compatible network representations. This can be used to provide a sanity check on the programmer-assigned interface version numbers.

Both generations of relationship evaluator actually use a 'compiled' form of the types stored in the type repository. The programmer uses a free standing IDL compiler to translate an IDL source code file into a flattened form; i.e. a flattened AST or type graph. This is then stored in the type repository as one description of the interface type. This reduces the cost of comparing two interface types and the complexity of the evaluation algorithm. It also means that syntax and semantic errors in the type specifications will typically be detected earlier.

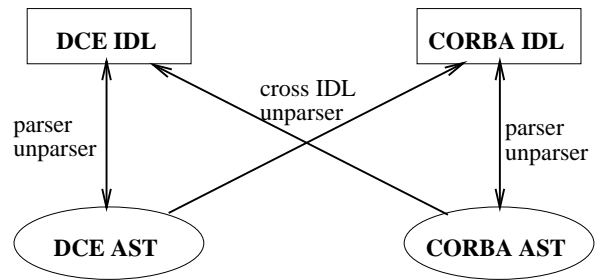


Figure 7. Translation tools for DCE IDL and OMG IDL.

Evaluation of other kinds of relationships may or may not be feasible. Comparison based on the abstract syntax of the type language is easy, but this approach is generally inadequate, as illustrated by our first-generation evaluators. (Most type languages will allow a given interface type to be expressed in many syntactically distinct forms.) The algorithms for type checking interfaces defined in terms of type signatures are well understood. However, type descriptions which incorporate behaviours (e.g. expressed as event patterns) are likely to require more advanced type checking algorithms. (This is an area for future research.) In the extreme, evaluation of relationships between interface types specified in general specification languages such as Z [34] would appear to be an intractable problem.

### 5.5. Type language translation

In a heterogeneous distributed computing environment we need to support interoperability of interfaces across different IDLs, e.g. a CORBA client calling a DCE server, or vice versa. Ideally we would like to have a tool that automatically finds a mapping between the two interfaces and generates the software for bridging any differences in network representations used by the client and server. In practice, this is only likely to be achievable when the IDLs in question have similar syntactic constructs with similar semantics.

As an extension to our first-generation relationship evaluators, we tried to implement automatic mapping between DCE and CORBA using the IDL transformation approach. Tools were built to unparse ASTs produced by the CORBA-to-AST and DCE-to-AST compilers described previously. Then we built a pair of cross-IDL unparsers, i.e. tools to unparse DCE IDL ASTs as CORBA IDL source code, and vice versa. figure 7 shows the translation steps involved.

The details of the DCE/CORBA mappings used in the IDL transformation tools are described in [38]. It should be noted that the focus was on the syntactic aspects of IDL transformation. Issues of mapping semantics (e.g. conversion between the network encodings used by DCE and CORBA) were beyond the scope of initial research. To date, our experience with these IDL translation tools is limited to producing IDL descriptions of application-level bridges that were then implemented by hand.

In hindsight, it is clear that there are fundamental problems with the semantics of DCE/CORBA transformation. While we were able to find clean mappings for many DCE and CORBA types, others are difficult to deal with. For example:

- DCE's *full pointers* are hard to map onto CORBA data types. Any DCE data structure that uses full pointers may contain aliases and cycles. These are not transmissible by CORBA using standard marshalling mechanisms.

- There are no types in DCE IDL corresponding to the CORBA *any* or *TypeCode* types. Implementing them fully in DCE would at best involve non-standard use of internal marshalling mechanisms in the DCE RPC runtime system. (Vogel *et al* [38] proposed that *any* types be implemented as a union of the DCE types that correspond to CORBA's basic and built-in types.)

- The proposed mapping for CORBA *object references* in [38] is an open data structure. To use the object reference, a conventional DCE client would require an inordinate amount of code to locate the object and create a binding handle for it. In practice, this would be best handled by modifying the DCE IDL stub generator and runtime libraries.

- While there is often a notional correspondence between CORBA *object references* and DCE *context handles*, it is hard to know when to map between them. In addition, semantic differences make the mapping difficult. For example, object references can be freely passed between machines and can persist, but this is not true for context handles.

In addition to the DCE-specific problems above (and others), there are two general problems with the automatic IDL transformation approach:

- When the IDLs assume different interface design paradigms IDL transformation will give interfaces that are non-intuitive and hard to use. For example, in the DCE/CORBA case (where the paradigms are RPC and distributed objects respectively), the mapping of 'objects' causes problems for both worlds as illustrated above.

- Automatic IDL transformation is not useful for finding mappings between pre-existing interfaces. For example, there are many ways that a given DCE IDL interface could be expressed in CORBA IDL interface, and vice versa. Therefore, if a given DCE IDL interface is automatically transformed into a CORBA IDL, the chance that the result will be compatible with a pre-existing interface is vanishingly small.

However, the above observations do not mean that interoperation is impossible across platform boundaries. For any pair of incompatible (but sufficiently similar) interfaces expressed in the same or different IDLs, it should be possible to construct a bridge between a client of one interface and a service of the other. Type managers have the following important roles in the description and use of these bridges.

- For bridges that are implemented by hand, we can record a relationship between a pair of interface types and a bridge type in the type manager in a ternary relation. For example:

```
bridge(<interface_type_1>, <interface_type_2>,
      <bridge_type_1>).
```

This relation could be queried at bind time to find a bridge between incompatible interfaces. Furthermore, the type manager could infer that it is possible to connect a client and server via a 'pipeline' of bridge services.

- Given the existence of a language for describing mappings of operation parameters and values in (say) CORBA IDL, and a 'universal mapping service' that interprets the language, many bridges could be implemented simply by populating a relation of the form:

```
universal_bridge(<interface_type_1>,
                <interface_type_2>,
                <mapping_type_1>).
```

- If we further assume that the type repository stores a concept hierarchy for the relevant domain, along with the relationships between these concepts and operation parameters in IDL interface, the type manager could potentially derive candidate mappings between interfaces, i.e. by 'computing' the mapping types above.

## 5.6. Graphical user interface

We have implemented a graphical user interface for the type manager that provides a user-friendly tool for browsing and modifying the contents of the type manager. The tool is implemented in C for the X/Windows Motif environment. Figure 8 is a snapshot of the tool, showing windows for querying the type and relationship repositories.

There is also a hyper text markup language (HTML) interface with limited functionality which is customized for service location and access [37].

## 6. Future work

### 6.1. Scalability and interworking

Type management is particularly important in heterogeneous distributed systems with large (possibly global) extent. This makes scalability an important issue for any type management system.

Our proposed approach to scalable type management is to implement a federation of interworking type managers. Type managers need to be able to access each others' type and relationship repositories, and to perform type and relationship queries that make use of the combined information base. Ideally, a type management client would not need to be aware of federation. The client would simply send a request to a local type manager instance, which would contact others as required.

The following issues have to be addressed as a prerequisite for successful federation of type managers:

- The type managers need agreed interfaces and protocols for repository access and for negotiating federation issues. These protocols need to support queries over multiple type manager repositories.

- The type managers need a common language for defining relationship types.

- The type manager administrators need to agree on common type taxonomies, e.g. agreed names and identifiers for the common metatypes, agreed definitions of the

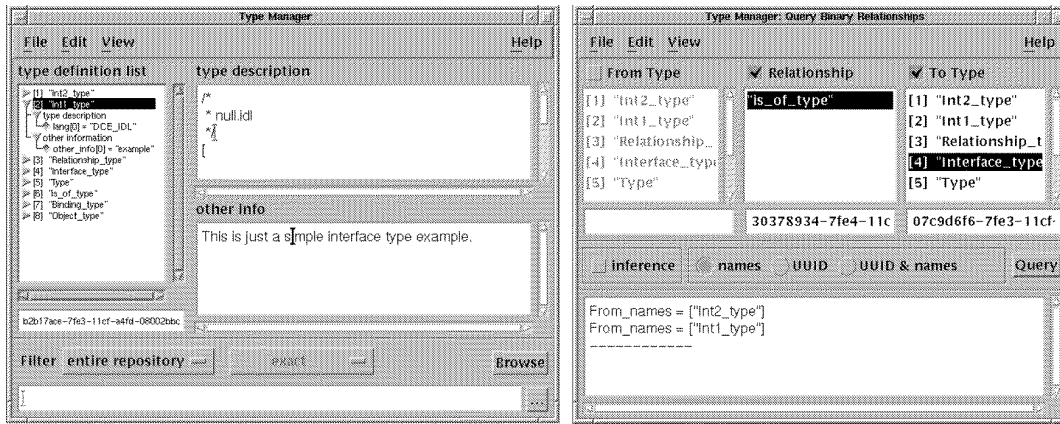


Figure 8. Graphical user interface to the type manager.

common relationship types, and agreed names of type languages.

While these issues have not yet been researched, we believe that our current type management model will be amenable to federation. The current type manager was designed with this kind of interoperation in mind, and should be amenable to the required changes in the future.

## 6.2. Software engineering issues

While our current type manager only supports runtime binding and resource discovery, it can potentially support a far wider range of software engineering activities in the distributed systems context.

Type information is needed at all stages of the software engineering lifecycle. In current-generation software-development environments, type information (i.e. meta-information) for a system is typically dispersed throughout the host-file system, e.g. in design documents, header files, 'makefiles', IDL files and repositories and binary files (debug symbol tables). This dispersal leads to a variety of problems, including difficulties with finding information and with configuration control of the information.

In a system based around a type manager, this type information could all be centrally managed, removing considerable burden from the developer of software engineering tools, and making it a lot easier to perform resource discovery and to re-use type information. The COMANDOS project [12] is an example of applying this in a homogeneous environment. COMANDOS has a unified type model and provides a type manager that is used to maintain a persistent type information base in support of its basic software engineering activities.

Our type management model responds to a broader set of requirements than COMANDOS, i.e. supporting large-scale, open, heterogeneous distributed systems. Since this kind of environment has broader software engineering problems (especially in the interoperability area), our scope for supporting the software engineering process is correspondingly greater. We would envisage that the type manager could be used to hold, for example:

- system requirement specifications,
- analysis and design models for system components,

- system architecture models (e.g. [23, 8]),
- formal and informal software specifications,
- software source code, configuration and build information,
- symbol tables,
- component interface types, binding types,
- types that express interface mappings, etc,
- types that describe dynamically instantiable objects and services,
- application specific types, e.g. document types and user interface types.

This information could be used at all phases of the software engineering lifecycle, from requirements capture through to maintenance. It would be accessed and updated by requirements capture tools, design tools, programming tools (editors, compilers, debuggers) and system-management tools. The type manager would support tools such as browsers and interactive resource-discovery tools that rely on having type descriptions at runtime. Finally, it would support the increasing number of components that need dynamic type matching and type checking, e.g. interpreters for scripting languages, service traders and the distributed system's runtime environment.

## 7. Conclusions

In this paper we have presented a type manager designed to support interoperability in a large-scale, open, heterogeneous distributed computing environment, and we have described some aspects of an advanced type model for distributed systems.

Our type model is an enhancement of the type models of existing IDL languages that adds an open set of complex types to support interoperability, e.g. types of interactions and bindings, and the types of relationships between types. The model allows the capture of both structural and behavioural elements of interface types, making them available for type matching and type checking. However, from the type management viewpoint, the idea of relationship types is the most important contribution of the type model.

Relationship types allow the expression of arbitrary relationships between types; e.g. as represented in a type management system. A relationship type defines the number, type and cardinality of the roles in a relationship. It can also declare properties for the relation, and can name a function that can compute the relation. The information in a relationship type can be used by a type manager to maintain consistency of a database of 'facts' concerning relationships between types, and to infer and compute new relationships.

The type manager provides clients in a distributed environment with services for storing and retrieving information about types and their relationships. Types may be described in multiple languages, and relationships between types expressed in any language are supported. The system administrator may add new type languages and associated tools to the type manager, and may define new type relations (using relationship types).

The type manager has the ability to resolve a range of type and type-relationship queries. In the case of relationships, it uses an inbuilt inference engine that can deduce relationships using the type managers 'fact' base, known properties of relationships, and information provided by external relationship evaluators. The latter give the type manager the ability to compute type relationships from the descriptions of the types.

We described our early attempts to use the type manager for providing interoperability between CORBA and DCE IDLs by translation of abstract syntax trees. While this approach had only limited success, we were able to identify alternative (less ambitious) approaches that are more likely to succeed. Future work in this area is planned within the context of a different project.

In the 'future work' section of the paper, we discussed our proposed approach for scalable type management by federating a number of type manager instances. We also sketched our ideas on how type management services could underpin the software engineering process in large-scale distributed systems.

Finally, we should mention that our work on type management systems was used as contributions to the ISO standardization of the ODP type repository function. We are currently working on a submission in response to the Object Management Group's 'Meta-object Facility' RFP. Our submission is strongly influenced by the work described in this paper.

## Acknowledgments

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. It was also partially supported by an Australian Government Postgraduate Research Scholarship (APRA).

## References

- [1] America P 1991 Designing an object-oriented programming language with behavioural subtyping *Foundations of*

- Object-Oriented Languages (Lecture Notes in Computer Science 489) (REX School/Workshop, Noordwijkerhout, The Netherlands)* (Berlin: Springer) pp 60–90
- [2] Architecture Project Management Limited 1989 *The ANSA Reference Manual* (Cambridge: Cambridge University Press)
- [3] Auerbach J S, Gopal A S, Kennedy M T and Russell J R 1994 Concert/C: supporting distributed programming with language extensions and a portable multiprotocol runtime *Proc. 14th Int. Conf. on Distributed Computing Systems (ICDCS-14) (Poznan)* (New York: IEEE) pp 152–9
- [4] Bearman M 1994 ODP-Trader *Proc. 1st Int. Conf. on Open Distributed Processing (ICODP'93) (Open Distributed Processing II) (Berlin, 1993) (IFIP Transactions)* ed J de Meer, B Mahr and S Storp (Amsterdam: Elsevier, North-Holland) pp 19–33
- [5] Berry A and Raymond K 1995 The A1! architecture model *Proc. 2nd Int. Conf. on Open Distributed Processing (ICODP'95) (Open Distributed Processing III) pp 55–66 (Brisbane, 1995)* (London: Chapman & Hall)
- [6] Biggs C J, Brookes W and Indulska J 1994 Enhancing interoperability of DCE applications: a type management approach *Proc. 1st Int. Workshop on Services in Distributed and Network Environments (SDNE'94)* (Los Alamitos, CA: IEEE Computer Society) pp 42–9
- [7] Bolognesi T and Brinksma E 1987 Introduction to the ISO specification language LOTOS *Comput. Networks ISDN Syst.* **14** 25–59
- [8] Brookes W 1996 A type description language supporting interoperability in open distributed systems *Proc. 1st Int. IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)* (London: Chapman & Hall) pp 20–35
- [9] Brookes W, Berry A, Bond A, Indulska J and Raymond K 1995 A type model supporting interoperability in open distributed systems *Proc. 1st Int. Conf. on Telecommunications Information Networking Architecture (TINA '95)* pp 275–89
- [10] Brookes W and Indulska J 1994 A type management system for open distributed processing *Technical Report 285* The University of Queensland, Department of Computer Science
- [11] Brookes W, Indulska J, Bond A and Yang Z 1995 Interoperability of distributed platforms: a compatibility perspective *Proc. 2nd Int. Conf. on Open Distributed Processing (ICODP'95) (Open Distributed Processing III) (Brisbane, 1995)* (London: Chapman & Hall) pp 67–78
- [12] Cahill V, Balter R, Harris N and Rousset de Pina X 1993 *The COMANDOS Distributed Application Platform. Volume 1. ESPRIT Research Reports (Project 2071)* (Berlin: Springer)
- [13] Clocksin W F and Mellish C S 1987 *Programming in Prolog* (Berlin: Springer) 3rd revision and extended edition
- [14] Indulska J, Bearman M and Raymond K 1994 A type management system for an ODP trader *Proc. IFIP TC6/WG6.1 Int. Conf. on Open Distributed Processing (ICODP'93) (Open Distributed Processing II) (Berlin, 1993) (IFIP Transactions)* (Amsterdam: Elsevier, North-Holland) pp 169–80
- [15] ISO/IEC CD 10165-7 1993 *Information Technology—Open Systems Interconnection—Structure of Management Information. Part 7: General Relationship Model*
- [16] ISO/IEC CD 13235 1995 *Committee Draft 13235: Open Distributed Processing—Reference Model—Trading Function*
- [17] ISO/IEC DIS 8824-1 1992 *Information Technology—Open Systems Interconnection—Abstract Syntax Notation One (ASN.1). Part 1: Specification of Basic Notation*
- [18] ISO/IEC IS 10165-4 1992 *Information Technology—Open*

- System Interconnection—Structure of Management Information—Part 4: Guidelines for the Definition of Managed Objects*
- [19] ISO/IEC IS 10746-3 1995 *International Standard 10746-3, ITU-T Recommendation X.903: Open Distributed Processing—Reference Model—Part 3: Architecture*
- [20] ISO/IEC IS 8807 1988 *LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*
- [21] Liskov B and Wing J M 1993 A new definition of the subtype relation *Proc. Eur. Conf. on Object Oriented Programming (ECOOP '93) (Lecture Notes in Computer Science 707)* ed O Nierstrasz (Berlin: Springer) pp 118–41
- [22] Liskov B H and Wing J M 1994 A behavioural notion of subtyping *ACM Trans. Programm. Lang. Syst.* **16** 1811–41
- [23] Luckham D and Vera J 1995 An event-based architecture definition language *IEEE Trans. Software Eng.* **21** 717–34
- [24] Manola F and Heiler S 1993 A 'RISC' object model for object system interoperation: concepts and applications *Technical Report TR-0231-08-93-165* GTE Laboratories Incorporated
- [25] Milosevic Z, Berry A, Bond A and Raymond K 1995 Supporting business contracts in open distributed systems *Proc. 2nd Int. Workshop on Services in Distributed and Networked Environments (SDNE'95)* (Los Alamitos, CA: IEEE Computer Society) pp 60–7
- [26] Object Management Group 1992 *Object Management Architecture Guide* 2nd edn, Revision 2.0, OMG TC Document 92.11.1
- [27] Object Management Group 1994 *Interface Repository Specification* OMG RFP Submission—OMG TC Document 94.11.7
- [28] Object Management Group 1994 *Relationship Service Specification* OMG RFP Submission—OMG TC Document 94.5.5
- [29] Object Management Group and X/Open 1992 *The Common Object Request Broker: Architecture and Specification*
- [30] Object Management Group 1995 *Component Object Model/CORBA Interworking RFP* OMG TC document 95-4-2
- [31] Rosenberg W and Kenney D 1992 *Understanding DCE* (Open System Foundation)
- [32] Shirley J, Hu W and Magid D 1994 *Guide to Writing DCE Applications* 2nd edn (Sebastopol, CA: O'Reilly & Associates)
- [33] Spivey J 1996 *An Introduction to Logic Programming through Prolog* (Englewood Cliffs, NJ: Prentice-Hall)
- [34] Spivey J M 1989 *The Z Notation: A Reference Manual (Int. Series in Computer Science)* (Englewood Cliffs, NJ: Prentice-Hall)
- [35] van Eijk P and Belinfante A 1990 The term processor kimwitu, manual and cookbook *Technical Report INF-90-45* University of Twente
- [36] Vogel A, Bearman M and Beitz A 1995 Enabling interworking of traders *Proc. 2nd Int. Conf. on Open Distributed Processing (ICODP'95) (Open Distributed Processing III) (Brisbane, 1995)* ed K Raymond and J de Meer (London: Chapman & Hall) pp 185–6
- [37] Vogel A, Beitz A and Ianella R 1995 Discovery and access of services in globally distributed systems *DSTC Symp. (DSTC, Pty Ltd, Brisbane)*
- [38] Vogel A, Gray B and Duddy K 1996 Understanding any IDL—lesson one: DCE and CORBA *Proc. 2nd Int. Workshop on Services in Distributed and Networked Environments (SDNE'96)* ed P Honeyman (Los Alamitos, CA: IEEE Computer Society) pp 114–21
- [39] Xerox Corporation 1996 *The Inter-Language Unification System* Internet web page  
URL <ftp://ftp.parc.xerox.com/pub/ilu/ilu.html>
- [40] X/Open Ltd 1993 *X/Open DCE: Remote Procedure Call X/Open Preliminary Specification*
- [41] Zaremski A M and Wing J M 1993 Signature matching: a key to reuse *Proc. SIGSOFT '93 (December)* (also CMU-CS-93-151, May)