33

ODP Types and Their Management: an Object-Z Specification

W. Brookes, J. Indulska

Department of Computer Science, CRC for Distributed Systems Technology,
The University of Queensland, Brisbane 4072, Australia

Defining a type model and maintaining a persistent type repository is an approach taken by a number of distributed platforms for managing the heterogeneity present in distributed systems. In particular, the Reference Model of Open Distributed Processing (RM-ODP) defines a basic type model and type repository function for supporting application interoperability in an open, heterogeneous and autonomous environment. This paper presents the specification of one approach to type description and type management that is compliant with the current version of the RM-ODP standard. The specification extends the scope of the RM-ODP type model by introducing relationship types. It shows their description and role in type matching. A summary of the important features of this specification is also presented.

Keyword Codes: C.2.4; D.1.3; D.2.1
Keywords: Distributed Systems, Concurrent Programming, Requirements/Specifications

1. INTRODUCTION

The Reference Model of Open Distributed Processing (RM-ODP) is an emerging ISO standard that recognises the need for applications and services to be able to interwork in an open, heterogeneous and autonomous environment [14]. In an open system little commonality can be assumed, as systems can be developed independently. Therefore, in order to use services globally, there must be a common understanding of the nature of those services, independent of their representation or provision by a particular network. Some distributed computing platforms have addressed this issue by building type management systems [2, 3, 18]. They differ in their type models and the functionality of those systems. RM-ODP introduces a Type Repository function which provides a framework for describing and relating the types of information, services and entities in an open distributed system [14].

With such a repository of type information, end-users and components of a distributed system infrastructure can use the type information to support *interoperability* of applications

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).

K. Raymond et al. (eds.), *Open Distributed Processing*

(since there is a common agreement on the types), *resource discovery* (by providing a repository of information about types of services that exist), and *system evolution* (by recording information about different types which provide compatible functionality). ODP infrastructure components requiring support from the type repository are the ODP Trader (which selects services on behalf of users) and functions responsible for binding (to bind interfaces in a type-safe manner).

We refer to the type repository function as a *Type Manager*, however its functionality need not be implemented by a single ODP object. This paper presents an overview of a formal specification of types which are basic for interoperability (and therefore have to be described in the Type Manager) and of type management functions. The latter includes two components: the type description repository and the relationship repository.

The issue of formal specification of the ODP type model and management of types has already been addressed in other research. Najm & Stefani [17] and RM-ODP Annex A [14] present basic specifications of the ODP type model assuming that it is a first order type system with the structural ODP subtyping relationship defined in the form of inference rules on judgements. It addresses operational interfaces only. Z and Object-Z specifications of an ODP trader [9, 16] include some specification of type management functions. They are simplified and focus on the trading function only; they do not specify types and specification of type management is limited to adding and deleting types as well as a high level specification of service type matching.

Our goal is to provide a formal specification for aspects of the whole type management system including both the type model and management of types, to precisely and unambiguously describe ODP type management concepts. We use an object-oriented approach to provide a definition of types and a set of management operations. The type model presented is compliant with the ODP type model, but provides a more general approach to relationship types. As ODP defines a built-in subtyping relationship with one particular semantics, we extend this by allowing a definition of various kinds of relationships (*e.g.* different kinds of subtyping relationships) to be introduced. This creates a basis for mapping between subtyping relationships from different domains (*e.g.* mapping between an ODP-like system and OMG CORBA [19] or DCE [20]).

Several candidate specification techniques exist for the type model including ACT.ONE [8], and the lambda calculus [6]. There are also various languages which could be used to specify the whole Type Manager including LOTOS [11] and Z [21]. However, Object-Z was chosen for the following reasons: (a) the language Z (as developed by Spivey, [21]) is able to readily accommodate object-oriented concepts including inheritance [10, 15], (b) some work involving Z specifications already exists in ODP [9, 16], and (c) Object-Z [22] is an object-oriented extension of the Z specification language and it "displays sufficient expressive power for use in modelling ODP systems" [23]. Other factors which influenced the decision in favour of Object-Z were the difficulties in interpreting inheritance and subtyping in LOTOS [7, 15] and the lack of facilities for semantic specification in the other techniques mentioned (specification of the semantics of ODP types, not the semantics of the language in which the types are described). Semantic specification is not exploited in this paper as the specification presented considers only the syntax of types, but is necessary for future research.

The remainder of this paper is organised as follows. Section 2 presents a specification and discussion of types and the type description repository component of a Type Manager. Section 3 takes a similar approach in presenting the relationship repository. Section 4 contains a short description of the specification of a complete ODP-based Type Manager. Finally, section 5 highlights the most important sections of the specification, and presents areas for future work.

2. THE TYPE DESCRIPTION REPOSITORY

The Type Manager keeps type descriptions of types which are basic for ODP interoperability: Datatypes, Operations, Flows, Signals, Interfaces, Objects and Relationships.

Datatypes are essential as they are the building blocks for other types (for example, the arguments and results of operations). Descriptions of operations, flows, signals, interfaces and objects are necessary in an ODP-based type model since they are fundamental concepts for ODP, and provide the basic units of interaction in an ODP system. Our work extends the RM-ODP type model to include relationship types. Since the Type Manager should support many different relationships, it is necessary to store the description of each relationship. As defined by RM-ODP [13], a type is described by a predicate. Relationship descriptions can be treated as types because they may easily be described as a predicate (defining the relationship).

The Type Manager must be able to learn about new kinds of types, *e.g.* service types required by a Trader, therefore the set of kinds of types is extensible to accommodate definitions of new kinds. Types should be uniquely identified. Having type identifiers allows applications to reference types by names, and allows a relationship repository to be built in a more flexible manner since the relationships may be stored separately from the descriptions [1]. For the purposes of this paper a type is identified by a *TypeId* (which is unique in its domain). The issues of context-based naming (which is necessary to reduce the scope of unique names) and name versioning are not addressed in this paper.

$[TypeId, Name]$

The type description repository maintains type descriptions (each with an associated name). From the type descriptions together with type identifiers a repository is constructed with operations to manipulate the contents (*e.g.* Add, Delete, Lookup). In the following specification, bold subscripted numbers associated with schema names represent the RM-ODP clause on which the schema is based. The specification of object types has been omitted.

2.1. Type descriptions

We start by defining the notion of a type in general. The only thing the different kinds of types have in common is that it is possible to identify to which kind a particular type belongs.

TypeDefn

$$kind : TypeKind$$

As discussed previously, the type kind is one of the following:

$$\begin{aligned} TypeKind ::= & \ Datatype \mid Operation \mid Flow \mid Signal \\ \mid & \ Interface \mid Object \mid Relationship \mid \ldots (other\ types) \end{aligned}$$

2.1.1. Operation type description

There are two basic kinds of operation in RM-ODP: interrogation and announcement. An interrogation is what is widely known as a remote procedure call. It consists of two interactions: an invocation followed by a termination. An announcement consists only of the invocation, and no termination action is expected (or allowed).

An invocation action consists of the operation name, and the number, names and types of the argument parameters. The following specification constrains the argument parameter names to be unique and that each parameter name has exactly one type.

OperationInvocationTemplate 7.1.26

$$opName : Name$$
$$nrArguments : \mathbb{N}$$
$$arguments : Name \nrightarrow TypeId$$
$$\#(\mathrm{dom}\ arguments) = nrArguments$$

A termination action is similar to an invocation. Each termination has a name, and the number, names and types of result parameters. This specification assumes that each termination is uniquely identified by a name.

TerminationTemplate 7.1.26

$$termName : Name$$
$$nrResults : \mathbb{N}$$
$$resultTypes : Name \nrightarrow TypeId$$
$$\#(\mathrm{dom}\ resultTypes) = nrResults$$

Having defined invocation and termination actions, we can now define interrogations and announcements. An interrogation consists of one invocation, plus possibly many terminations.

InterrogationSignature 7.1.26

TypeDefn

$$invocation : OperationInvocationTemplate$$
$$responses : \mathbb{F}_1\ TerminationTemplate$$
$$kind = Operation$$

An announcement contains only a single invocation action.

AnnouncementSignature 7.1.27

TypeDefn

$$invocation : OperationInvocationTemplate$$
$$kind = Operation$$

Now we can define a more general notion of an operation. An operation type signature is either an interrogation signature or an announcement signature.

$$OperationSignature_{7.1.28} ::= Interrogation\langle\langle InterrogationSignature\rangle\rangle \mid Announcement\langle\langle AnnouncementSignature\rangle\rangle$$

2.1.2. Flow type description

RM-ODP defines stream interfaces to support modelling of applications which manage continuous flows of data (*e.g.* multimedia applications). Each stream interface consists of smaller building blocks called "flows". Each flow contains the type of information which will pass along the flow, as well as an indication of the direction of data flow.

FlowSignature 7.1.16
TypeDefn

$$
\begin{array}{|l}
\hline
type : TypeId \\
causality : Direction \\
\hline
kind = Flow \\
\hline
\end{array}
$$

The causality of the flow is indicated by its direction. An object may be either producing the flow of data (initiating) or consuming it (responding).

$$Direction_{7.1.9} ::= Initiating \mid Responding$$

2.1.3. Signal type description

A basic unit of interaction defined in RM-ODP is a signal. It consists of a single, atomic action (message) between a basic computational object and a binding object. Signals are named, contain parameters (number, names and types) and have causality (direction).

SignalSignature 7.1.9
TypeDefn

$$
\begin{array}{|l}
\hline
sigName : Name \\
nrArguments : \mathbb{N} \\
arguments : Name \pfun TypeId \\
causality : Direction \\
\hline
kind = Signal \\
\#(\mathrm{dom}\ arguments) = nrArguments \\
\hline
\end{array}
$$

2.1.4. Interface type description

There are three kinds of interface types. ***Operational interfaces*** consist of a set of operations (interrogations or announcements). Thus, operational interface types are defined as a set of operation signatures. The second kind of interface is a ***stream interface***, which consists of a set of information flows. The final kind of interface is a ***signal interface*** which contains a set of signal signatures.

ComputationalInterfaceSignature 7.1.10, 7.1.17, 7.1.31
TypeDefn

$$
\begin{array}{|l}
\hline
defn : OperationalInterface(\!(\mathbb{F}\ OperationSignature)\!) \\
\quad \mid StreamInterface(\!(\mathbb{F}\ FlowSignature)\!) \\
\quad \mid SignalInterface(\!(\mathbb{F}\ SignalSignature)\!) \\
\hline
kind = Interface \\
\hline
\end{array}
$$

Including a description of some interface semantics as part of an interface description is for further study, and not presented in this specification.

2.1.5. Relationship type description

ODP-based systems should support system evolution to reflect the changing nature of an open system. Thus an ODP-based Type Manager should allow new relationships to be introduced at run-time by a user and for these relationships to be later used for type matching and type checking. One approach to facilitate this is to store a meta-level description of each relationship and to allow new relationship descriptions (type descriptions) to be added at run-time.

Relationship types consist of two parts: syntactic and semantic. The syntactic elements of a relationship are captured by a set of role types. The following definition of relationship types is an extension of the ISO General Relationship Model [12].

RoleSignature

$$
\begin{array}{l}
rolename : Name \\
roletype : TypeId \\
required\_cardinality : \mathbb{N} \\
permitted\_cardinality : \mathbb{N}
\end{array}
$$

Each role models one participant in the relationship, and has a name, type, and cardinalities associated with it. Required cardinality corresponds to the minimum number of participants in that role, and permitted cardinality defines the maximum number. Each role may have additional described behaviour associated with it, however that is not modelled in this specification.

The semantics of a relationship are captured in two parts: a declaration of the characteristics of the relationship, and a definition rule which may be used to determine membership in a given relationship. The characteristics applicable for all relationships include the method of handling deletion of one of the roles. When a specified role is deleted, the possible options for dealing with the relationship are: (a) delete other roles as well; (b) release other roles (*e.g.* set them to be NULL); (c) prevent the deletion from occurring unless other roles are empty (NULL).

RelationshipSignature

TypeDefn

$$
\begin{array}{l}
roles : \mathbb{F}\, RoleSignature \\
delete\_all\_in\_roles : RoleSignature \nrightarrow \mathbb{F}\, RoleSignature \\
release\_all\_in\_roles : RoleSignature \nrightarrow \mathbb{F}\, RoleSignature \\
only\_if\_none\_in\_roles : RoleSignature \nrightarrow \mathbb{F}\, RoleSignature \\
\hline
kind = Relationship \\
\#roles > 0 \\
\mathrm{dom}\ delete\_all\_in\_roles \in roles \land \mathrm{ran}\ delete\_all\_in\_roles \subseteq roles \\
\mathrm{dom}\ release\_all\_in\_roles \in roles \land \mathrm{ran}\ release\_all\_in\_roles \subseteq roles \\
\mathrm{dom}\ only\_if\_none\_in\_roles \in roles \land \mathrm{ran}\ only\_if\_none\_in\_roles \subseteq roles
\end{array}
$$

Homogeneous binary relationships (having exactly two roles, each of the same type) are a special case. For homogeneous binary relationships, additional information can be maintained about the characteristics of the relationship, such as whether it is reflexive, irreflexive, symmetric, *etc.* The definition may also record whether the relationship is intended for supporting type matching. The schema on the left shows the definition of a homogeneous binary relationship, while the definition on the right shows the possible characteristics.

BinaryRelationshipSignature
RelationshipSignature

R : *Reflexivity* ::= *Reflexive* | *Irreflexive* | *Antireflexive*
S : *Symmetry* ::= *Symmetric* | *Asymmetric* | *Antisymmetric*
T : *Transitivity* ::= *Transitive* | *Intransitive* | *Antitransitive*
type_matching : $\mathbb{B}$

$\#roles = 2$
$\exists r_1, r_2 \in roles \bullet$
$r_1 \neq r_2 \wedge r_1.roletype = r_2.roletype$

The semantics of constraints imposed by the binary relationship characteristics is captured by the relationship repository operations. When an instance of a relationship is added, it is checked to ensure that it does not violate the characteristics of the relationship description. For example, the RM-ODP subtyping relationship is reflexive, antisymmetric and transitive. Therefore when adding a new instance of this relationship, appropriate checks should be made to ensure that these constraints are not violated. As mentioned earlier, each relationship should have a definition rule associated with it. This rule can be used to enforce checking whether types can legally participate in the relationship (as explained below). For binary relationships it is modelled as a Z relation between two type identifiers. If two types may legally participate in the type relationship, they are members of the Z relation (the definition rule).

The set of characteristics includes an indication of whether automatic type matching is possible (*e.g.* if matching is based on syntax only). This knowledge, together with the definition rule provides the mechanism for allowing the Type Manager to automatically determine membership in a relationship, as can be done for structural subtyping (as just one example). If full automation of matching is not possible, the definition rule can be used by a tool supporting semantic type matching (*e.g.* a browsing tool for a user). Note that defining the semantics of each relationship allows users to define their own compatibility relationships, and allows multiple definitions of subtyping relationships to co-exist in the repository and be used for different matching purposes when necessary. This approach facilitates federation of different distributed environments which can use different compatibility relationships. The proposed Type Manager provides a tool for mapping between these relationships. It does not, however, prescribe a set of compatibility relationships, as this would contradict the goal of openness.

As an example, the RM-ODP structural subtyping relationship (as applied to operational interfaces) is shown by the following definition. Note that the definition is recursive: subtyping on operational interfaces will depend on subtyping of operations, which in turn will depend on subtyping of arguments and results. The definition of subtyping of operations, arguments and results is omitted from the following definition rule. These subtyping rules are not always straightforward to define (*e.g.* when recursive types are introduced). A description of RM-ODP subtyping rules for operational interfaces, including recursive types, can be found in Annex A of RM-ODP Part 3 [14].

The following rule first retrieves the two type descriptions to be compared ($t1defn$ and $t2defn$). If both type descriptions are for interfaces, then $t1$ is a subtype of $t2$ if for every operation in the supertype $t2defn$ (called x) there exists a corresponding operation (with the same name) in the subtype $t1defn$ (called y) such that operation y is a subtype of operation x.

$$
\begin{array}{|l}
\_\ IsRMODPOperationalInterfaceSubtypeOf\ \_ : TypeId \leftrightarrow TypeId \\
t1defn, t2defn : \downarrow TypeDefn \\
\hline
\exists\, t1, t2 : TypeId \bullet \\
\quad t1defn = typedb(t1) \\
\quad t2defn = typedb(t2) \\
\quad t1\ \underline{IsRMODPOperationalInterfaceSubtypeOf}\ t2 \Leftrightarrow \\
\qquad ((t1defn.kind = Interface \land t2defn.kind = Interface) \Rightarrow \\
\qquad\quad (\forall x \in \operatorname{ran} t2defn.defn \bullet \\
\qquad\qquad \exists y \in \operatorname{ran} t1defn.defn \bullet \\
\qquad\qquad\quad y.invocation.opName = x.invocation.opName \\
\qquad\qquad\quad y\ \underline{IsRMODPOperationSubtypeOf}\ x \\
\qquad\quad)) \\
\end{array}
$$

2.2. The type database

Types known to the Type Manager are stored in some form of specialised database. This is modelled as a partial function which maps type identifiers (*TypeId*) into type descriptions (*TypeDefn*). Initially the database is empty.

Only the operation to add a type to the database is shown here. Specification of other operations (*e.g.* deleting a type, performing a lookup of a type description) may be found in [5].

TDB

$$typedb : TypeId \pfun \downarrow TypeDefn$$

INIT

$$typedb = \varnothing$$

Add

$$
\begin{array}{l}
\Delta(typedb) \\
tid? : TypeId \\
defn? : \downarrow TypeDefn \\
\hline
tid? \notin \operatorname{dom} typedb \\
typedb' = typedb \cup \{tid? \mapsto defn?\}
\end{array}
$$

3. THE RELATIONSHIP REPOSITORY

There are two aspects to consider about relationships between types: the definition of the relationship and the instances of the relationship. The definition of a relationship includes the roles of the relationship, the predicate defining a membership rule as well as a description of the properties of the relationship (*e.g.* reflexivity, symmetry). This information is maintained in

the form of a 'relationship type', and is stored in the type description repository as described in the previous section. The Type Manager also stores the set of instances for relationships between types. RM-ODP defines a Relationship Repository function, however it differs in scope from the following description in that RM-ODP relationships are defined between objects and interfaces, whereas our approach is for relationships between types, as required for type matching (*e.g.* subtyping relationships).

The following section discusses relationship instances, focusing on homogeneous binary relationships only. Binary relationships such as subtyping, inheritance and compatibility are the most interesting from the type matching point of view. These concepts are then used to specify the database of relationships maintained by the Type Manager.

3.1. Homogeneous binary relationship instances

Homogeneous binary relationship instances are building blocks of type hierarchies and can be stored in a directed graph structure. The following discussion is based around the assumption of storing binary relationship instances as a general directed graph (not necessarily fully connected), that is specialised for recording type relationships.

Using the previous definition of relationship types, we assume that the two roles of a binary relationship are called '*from*' and '*to*'. We use the Z notation $\succ$ as the name of the reflexive transitive closure of the relationship, and use an infix notation (*i.e.* $from \succ to$).

Integrity checks are performed when adding a new instance of the relationship. Namely, the types being added to the graph are verified to ensure they are defined in the type database, and the pair of types being added is verified to ensure that it does not violate the characteristics of the relationship. For instance, the graph must be acyclic for a reflexive, antisymmetric and transtive relationship (*e.g.* subtyping). The reflexive transitive closure ($\succ$) is checked to ensure that this condition is not violated.

The relationship characteristics can be used for two purposes. The first is to ensure that adding a given pair of types to the relationship will not violate the relationship characteristics (*e.g.* adding $\langle a, a \rangle$ to an antireflexive relationship).

$$
\begin{array}{l}
defn?.R = Antireflexive \land defn?.T = Antitransitive \Rightarrow from? \neq to? \\
defn?.R = Antireflexive \land defn?.T \neq Antitransitive \Rightarrow from? \not\succ to? \\
defn?.S = Antisymmetric \Rightarrow to? \not\succ from? \\
defn?.T = Antitransitive \Rightarrow from? \not\succ to?
\end{array}
$$

The second use of the characteristics is to ensure duplicate information is not stored in the graph. A new instance is added if it cannot be deduced by reflexivity, symmetry or transitivity.

$$
\begin{array}{l}
((defn?.R \neq Reflexive \lor from? \neq to?) \land \\
\quad (defn?.S \neq Symmetric \lor to? \not\succ from?) \land \\
\quad (defn?.T \neq Transitive \lor from? \not\succ to?)) \\
\qquad \Rightarrow \succ' = \succ \cup \{(from?, to?)\}
\end{array}
$$

3.2. The relationship database

Relationship instances are stored in a database, ***RelnDB***, defined in this section. Initially the relationship instance database is empty.

RDB

$$relndbinst : TypeId \pfun RelnInst$$
$$typedb : TypeDB$$

$$\forall r : \mathrm{dom}\ relndbinst \bullet (r \in \mathrm{dom}\ typedb.typedb \land typedb.typedb(r).kind = Relationship)$$

INIT

$$relndbinst = \varnothing$$
$$typedb.INIT$$

One constraint that must always be true on the database is that for every relationship instance known in the instance database, the relationship definition for that relationship must also be known. Operations are provided to add an instance to the graph, to delete an instance, to find all types that are related to a given type and to find all types to which a given type is related. Specification of these operations, and their integrity checks may be found in [5].

4. THE TYPE MANAGER — A GLOBAL VIEW

The type description repository consists of the underlying type data base represented by the *typedb* function, and all the operations to manipulate/query the database. Similarly the relationship repository consists of the relationship database (*relndb*) and the operations upon it. A complete specification of these operations is given in [5].

TypeDB

$$TDB$$
$$TDB\_Add$$
$$TDB\_Delete$$
$$TDB\_Lookup$$
$$TDB\_Exists$$

RelnDB

$$RDB$$
$$RDB\_AddInst$$
$$RDB\_DelInst$$
$$RDB\_AreRelated$$
$$RDB\_RelatedTo$$
$$RDB\_RelatedFrom$$

The Type Manager consists of the type description repository (an instance of the class *TypeDB*) and the relationship repository (an instance of the class *RelnDB*). These two databases are separate, but related (*e.g.* relationship instances in RelnDB must be for a relationship whose type is defined in TypeDB). A summary of the functionality provided by the ODP-based type manager specified in this paper is given by the protocol diagram in Figure 1, showing the operations defined and their input and output parameters. Using these simple operations, type descriptions can be added, deleted and queried, and relationships can be stored and retrieved. More complex query operations can be built using this basic set of operations.

5. CONCLUSION

This paper presents an overview of the Object-Z specification of both basic types supporting interoperability of ODP-based systems and a Type Manager providing a persistent repository of

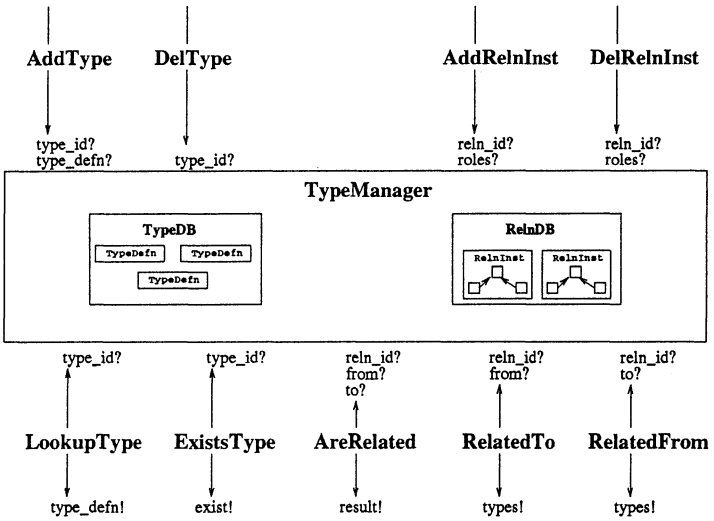

Figure 1: Type Manager Protocol

types. Users and ODP infrastructure components are able to access type information to provide type-safe dynamic selection of services, dynamic binding and dynamic operation invocation, while supporting system evolution.

The specification satisfies and extends the functionality required by the RM-ODP model. A specification of the two major components of an ODP-based Type Manager is given. The type description repository showed a concrete specification of some of the more important types referred to in the RM-ODP documents. In addition, it introduced the concept of relationship descriptions as types themselves since they can be expressed by a predicate. The relationship repository described the specification of a general directed graph of type identifiers. It introduced a specialised relationship database which associates a relationship name with a graph of instances and presented a specification of necessary integrity constraints.

The complete specification was used as the basis of our prototype Type Manager which enhances service selection and interoperability in the DCE platform for distributed computing [4]. Further work is being carried out on including a description of some semantics in type descriptions. The major goal of this work is to add elements of semantics specification which can be automatically or semi-automatically matched, improving type-safety of interoperability. Including elements of semantics in type descriptions can also support a user in deciding which types are semantically related.

REFERENCES

[1] A. Albano, G. Ghelli, and R. Orsini. "A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language". Technical Report FIDE/91/17, FIDE Project, 1991.

[2] APM Ltd, Cambridge UK. *ANSAware 4.1 Application Programmer's Manual*, Mar. 1992. Document RM.102.00.

[3] R. Balter. "Construction and Management of Distributed Office Systems Achievements and Future Trends". In *ESPRIT '89*, Proceedings of the 6th Annual ESPRIT Conference, pages 47–58. Brussels, November 27–December 1, 1989.

[4] C. J. Biggs, W. Brookes, and J. Indulska. "Enhancing Interoperability of DCE Applications: a Type Management Approach". In *Proceedings of the First International Workshop on Services in Distributed and Network Environments, SDNE'94*, 1994.

[5] W. Brookes and J. Indulska. "A Formal Specification of an ODP-based Type Manager". Technical Report 286, The University of Queensland, Department of Computer Science, Jan. 1994.

[6] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism". *Computing Surveys*, 17(4):471–522, Dec. 1985.

[7] E. Cusack, S. Rudkin, and C. Smith. "An Object Oriented Interpretation of LOTOS". In *Proceedings of the Second International Conference on Formal Description Techniques (FORTE'89)*, pages 265–284. Vancouver, B.C., 5–8 December 1989.

[8] J. de Meer and R. Roth. "Introduction to algebraic specifications based on the language ACT ONE". *Computer Networks and ISDN Systems*, 23, 1992.

[9] J. S. Dong and R. Duke. "An Object-Oriented Approach to the Formal Specification of ODP Trader". *Proceedings of the International Conference on Open Distributed Processing, ICODP'93*. Berlin, Germany, 14–17 September, 1993.

[10] R. Duke, G. Rose, and A. Lee. "Object-oriented protocol specification". In *Proceedings of the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification*, pages 323–339. Ottawa, Ont., 12–15 June 1990.

[11] ISO IS 8807. LOTOS, A Formal Description Technique based on the Temporal Ordering of Observation Behaviour, 1988.

[12] ISO/IEC CD 10165-7. Information Technology—Open Systems Interconnection—Structure of Management Information. Part 7: General Relationship Model, 1993.

[13] ISO/IEC DIS 10746-2. Draft Recommendation X.902: Basic Reference Model of Open Distributed Processing — Part 2: Descriptive Model, Apr. 1994. Output of Geneva editing meeting 14–25 February 1994.

[14] ISO/IEC DIS 10746-3. Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing — Part 3: Prescriptive Model, Apr. 1994. Output of Geneva editing meeting 14–25 February 1994.

[15] ISO/IEC JTC1/SC21. Working Document on Architectural Semantics, Specification Techniques and Formalisms, N4887, 1990.

[16] ISO/IEC JTC1/SC21/WG7 N743. "ANNEX G: Z Specification of the Trader", Nov. 1992.

[17] E. Najm and J.-B. Stefani. "A formal semantics for the ODP computational model". *Computer Networks and ISDN Systems*, to appear.

[18] Object Management Group. *Object Management Architecture Guide*, second edition, Sept. 1992. Revision 2.0, OMG TC Document 92.11.1.

[19] OMG and X/Open. *The Common Object Request Broker: Architecture and Specification*, 1992.

[20] W. Rosenberg and D. Kenney. *Understanding DCE*. Open System Foundation, 1992.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Intl. Series in Computer Science. Prentice-Hall, 1989.

[22] S. Stepney, R. Barden, and D. Cooper, editors. *Object orientation in Z*. Workshops in Computing. Springer-Verlag, Published in collaboration with the British Computer Society, 1992.

[23] P. Stocks, K. Raymond, D. Carrington, and A. Lister. "Modelling open distributed systems in Z". *Computer Communications*, 15(2):103–113, Mar. 1992.