ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

B Novikov, University of St Petersburg, Russia and J W Schmidt, University of Hamburg, Germany. (Eds)

Advances in Databases and Information Systems, Moscow 1996

Proceedings of the International Workshop on Advances in Databases and Information Systems (ADBIS '96). Moscow, 10-13 September 1996

Type Associations Identified to Support Information Resource Reuse in Megaprogramming

Leonid Kalinichenko

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Type Associations Identified to Support Information Resource Reuse in Megaprogramming

Leonid A. Kalinichenko

Institute for Problems of Informatics

Russian Academy of Sciences

Vavilova 30/6, Moscow, V-334, 117900

e-mail: leonidk@ipian23.ipian.msk.su

Abstract

Type associations supporting reuse decisions in course of the interoperable information systems design are identified. Type reducts (projections of object interfaces defined by types) are considered as patterns of reuse. The required federated composite objects are assumed to be created by integration of such fragments using views above their type definitions. Type associations introduced are based on an assumption of type specification completeness. The type refinement association is introduced as the fundamental concept for the others, such as subtyping, type reduct, type conformances and type inferencing associations. These associations are established to support the information system design with reuse ¹.

1 Introduction

Megaprogramming metaphor [13] reflecting the need for programming-in-the-large in an open distributed computing / telecommunication environment becomes now a strong motive force. The components used in megaprogramming, called megamodules, encapsulate the functionality of services, provided by various pre-existing information resources. Technically megaprogramming is based on interoperability providing for representation of megamodules as objects in the global object space interoperating through brokers [10].

In the SYNTHESIS project [3] considering the megaprogramming based on the pre-existing information resources we concentrate on semantic interoperability as a specific technique providing for meaningful and correct components composition to reach the required functionality. The semantic interoperation reasoning (SIR) method proposed [3] should lead to the concretization of specifications of requirements by views over the pre-existing information resources. These steps include: integration of an application domain and of an information resource ontological contexts, concretization view definitions supporting mediation of resource types to satisfy the requirements of specific application types, usage of declarative description style and predicative specifications for application and resource for generation and justification of concretization conditions capturing structural, extensional and behavioral properties.

The core interoperation technology [10] (addressing the system, technical level of interoperation) is based on the idea of total encapsulation of the underlying resources. No semantics of resources could be seen on the generalized level of description. In SYNTHESIS we are looking for methodological framework and architecture that should be SIR complete.

By completeness we mean that the specifications should be sufficient to reason that a resource is applicable to a given problem (perhaps, after some conceptual, functional, extensional, etc. reconciliation). Specifically it means that we are able to decompose the application specification and locate its fragments for which the resource could be considered a concretization. Or it means that the specifications should be sufficient to reason that a collection of resources is composable into a consistent, coherent entity reusable for a given problem (or its fragment) as a whole. Therefore, our solutions constitute complementary features to the core interoperation frameworks [10] and to existing

¹ This work was partially supported by the INTAS Program grant INTAS-94-1817

Object Analysis and Design methods. The process of concretization of an application specification by the pre-existing resources is critical issue to reach the semantic interoperation. The proposed procedure [4] of searching for resources with application relevant capabilities is based on the ontological context coherence and semantical similarity of resource / application specifications, on strict, justifiable conditions of structural, value and behavioral concretization of an application by the heterogeneous resources.

We assume that pre-existing information resource specifications (IRS) and information system requirements specifications (ISRS) are known. We assume that these specifications are complete and given in the uniform notation [5]. The basic constituents of the specifications are definitions of types representing ontological, state and behavioral semantics of pre-existing components and of the requirements to be developed. We consider specifications to be completely separated from the implementation definitions. We are focused here on reuse of specifications. The implementations that are behind them can be invoked through the specifications (one or multiple implementations can be associated with a type specification of a pre-existing resource).

We assume that type specifications are based on an object model with well-defined semantics containing an object calculus as its basic part (such as, e.g., [11, 5]). We assume that in a type definition state and behavioral attributes are distinguished, invariant (constraints) definitions are provided and definitions of functions are given in a form of mixed pre- and postconditions expressed in an object calculus.

The intention of this paper is to identify basic associations between type specifications that are required to support reuse decisions in course of design. These associations can be provided directly by a specification repository or they can be inferred by analysis of type specifications and justified. The application domain semantic associations between type specifications (provided partially by ontological information) are beyond this paper: everywhere we assume that types considered for reuse are semantically coherent.

For reuse we apply a model of composite object integrating data and behavior from various sources. The data and/or behavior residing at each source is regarded as a fragment of a composite object. We consider reducts (projections of object interfaces defined by types) as patterns of reuse. So, we decompose type specifications into a collection of reducts that can serve as units of reuse and composition. Thus we specify types of the object fragments as reducts of the required type and provide their loose integration in a bottom up fashion. Federated composite objects are created by integration of such fragments using views above their type definitions. Basic techniques supporting such objects include delegation, substitutability, inheritance providing for composition and sharing of behavior and state of objects. Concentrating on reuse, object-centered programming should be combined with class-based programming to form a continuum between prototype- or delegation-based languages [8] and class-based ones. Applying such models, we can properly define autonomous information resources (software servers) whose functional capabilities can be shared by delegation.

We address type refinement [9, 1] as the fundamental property that is required to justify a correctness of substitution of instances of one type for another and of reuse of pre-existing objects state and behavior. We should discover suitable reducts of IRS types refining reducts of ISRS types so that we could apply the basic techniques above to form a refinement of the required type. Necessary functionality of objects of the required type can be provided by constructing of required behavior from scratch or reusing pre-existing behaviors relying on inheritance or delegation. The resulting type should be consistent (that is, the type should have a model and operations should preserve the type invariants). The resulting type should be a refinement of the required type. Thus we construct a type that is a subtype of the ISRS type. This construct provides for reuse by means of object substitutability.

In the paper we define basic type associations that should be established to make reuse decisions efficient. The paper is structured as follows. We start with the definition of type refinement as of the basic required association. Other associations introduced are based on the type refinement concept. Subtyping and reduct associations are defined further. Introducing type conformances, we deviate from conventional definitions of such associations basing their meaning on the type refinement associations. Finally we discuss type inference associations providing for necessary type calculations during the design. The type associations introduced are based on an assumption of completeness of the type specifications.

2 Type Refinement

The objective of the design is to have type specifications implemented eventually as software modules by means of some programming notation. So, we have to transform specifications so that they could eventually be implemented by means of a program. This will be done by a step by step restriction of the constructs that could be used further. This activity is called a *refinement*.

Algorithmic refinement is an approach of being more and more precise about the way our operations should be eventually made concrete.

Data refinement consists in removing completely all variables whose types are too complicated to be implemented as such and in replacing them by simpler variables whose types correspond to those found in programming notations.

The refinement of a type is defined as follows: a type t_j is said to refine a type t_i if a user can use values of t_j instead of values of t_i without noticing it.

Both types must have the same operation names (each with same input and output parameters respectively). Both types are said to have the same operational signature. But of course, they will not have the same variables. To define more precisely what a refinement is, we introduce the following definition of a type model.

For a type model M the set of all type specifications expressible in M is denoted by T .

Type state. The *type state* corresponding to a type specification $t_i \in T$ is a function $s_{t_i} : Id_{t_i} \rightarrow V_i$, defining for each state variable (attribute) of the type specification denoted by identifier $I \in Id_{t_i}$ its value v_i taken from the set of admissible values V_i of the variable type. It is essential that v_i in its turn can also be an analogous function. A set of admissible states corresponding to some type specification $t_i \in T$ is a set of functions $S_{t_i} : [Id_{t_i} \rightarrow V_i]$.

A *space of type states* expressible in M is a set of functions $S : [Id_i \rightarrow V_i]$, which may be considered as union of sets $s_{t_i} \in S_{t_i}$ for all $t_i \in T$. We consider only admissible states that satisfy the invariants related to types.

Type behavior. The *type behavior* corresponding to a type specification $t_i \in T$ is a function $b_{t_i} : O_{t_i} \rightarrow [S_{t_i} \times S_{t_k} \times \dots \times S_{t_n} \rightarrow S_{t_i}]$ defining for each operation in the type specification the related state transformation for this type. $t_k, \dots, t_n$ are type specifications known in a context of the type specification t_i . A set of admissible behaviors corresponding to some type specification $t_i \in T$ is a set of functions $B_{t_i} : [O_{t_i} \rightarrow [S_{t_i} \times S_{t_k} \times \dots \times S_{t_n} \rightarrow S_{t_i}]]$.

A *space of type behaviors* expressible in M is a set of functions $B : [O \rightarrow [S \times \dots \times S \rightarrow S]]$, which may be considered as union of sets $b_{t_i} \in B_{t_i}$ for all $t_i \in T$.

Type model. *Type model* M is a triple $\langle T, Ms, Mb \rangle$ where $Ms : T \rightarrow S$ is a semantic state function of M , $Mb : T \rightarrow B$ is a semantic behavior function of M .

Now, considering type t_j being a refinement of type t_i in M , we assume that the following conditions hold:

— type state diagram is commutative:

— type behavior diagram is commutative:

- Θ mapping is a bijective abstraction function of a data refinement;
- β mapping is an algorithmic refinement.

3 Reduct Associations

To characterize fragments of objects to be reused we introduce a notion of a *type reduct* or simply a *reduct*. Roughly a reduct is a subset of an interface of a type instance. A reduct can be formed taking a projection of a type specification on certain state and functional attributes.

On another hand, a reduct should be a proper type specification that is always a supertype (immediate or transitive) with respect to an original type. A reduct should be properly placed in a type lattice.

Concerning reusability, we consider two reduct extremes: a functional reduct containing only behavioral attributes of the original type and a state reduct containing only state attribute of the original type. These different kinds of reducts can be formed to be focused on pragmatically different primary intention of reuse: to reuse a data source or a functional source.

Actually in course of a design the reducts can dynamically be included into the type lattice to support searching for reusable components. After the search completion the reducts are removed from the type lattice.

Generally, we consider reducts as the basic reuse patterns. Different constructs for reduct aggregation and composition providing for reuse (such as paths, compositions of functions) are applicable for the views definition.

4 Subtyping Associations

The basic meaning of a subtyping is based on the following assumptions [2]:

- subtyping is a partial order on types;
- if $\tau \leq \sigma$ then all values (objects) of a type τ can be used in any context in which values (objects) of a type σ can be used;
- if $\tau \leq \sigma$ then instances of type τ have at least all properties that instances of type σ have. In another words, a theory of the supertype is contained in a theory of the subtype, i.e. $\tau \leq \sigma \Leftrightarrow Th(\sigma) \subseteq Th(\tau)$. Therefore, subtyping implies a subset relationship between instances.

A subtype τ can be obtained from the supertype σ by:

- adding of new functions to an interface of σ ;
- refining the states;
- refining the assertions;
- refining the functions.

We denote a type specification $t = \langle Id_t, S_t, O_{f_t}, O_{p_t} \rangle$ where Id_t denotes for a type t a set of state variables, S_t - a state space, O_{f_t} - a set of operations, O_{p_t} - a set of invariants.

A subtyping association for a supertype $t_{sup} = \langle Id_{sup}, S_{sup}, O_{f_{sup}}, O_{p_{sup}} \rangle$ and a subtype $t_{sub} = \langle Id_{sub}, S_{sub}, O_{f_{sub}}, O_{p_{sub}} \rangle$ holds iff there exists an abstraction function Abs mapping the subtype state space (S_{sub}) into the supertype state space (S_{sup}) such that the following conditions are satisfied:

1. For a subtype invariants to be a refinement of a supertype invariants the following condition should be proved ($P_{sup}(P_{sub})$ is a conjunction of all predicates in $O_{p_{sup}}(O_{p_{sub}})$):

$$\forall S_{sub}, S_{sup} \bullet P_{sub} \wedge Abs \Rightarrow P_{sup}$$

2. To preserve supertype operations behavior, subtype operations should be a refinement of the corresponding supertype operations [7]. To show that an operation of a subtype $O_{sub} \in O_{f_{sub}}$ is a refinement of the corresponding operation $O_{sup} \in O_{f_{sup}}$ of the supertype when a subtype and supertype state spaces S_{sub} and S_{sup} are related by an *Abs* mapping, proof obligations should be generated according to the following templates [12]:
The first template should state that a subtype operation should terminate whenever a supertype operation is guaranteed to terminate (X and Y denote input and output spaces of the operation):

$$\forall S_{sub}, S_{sup} \bullet preO_{sup} \wedge Abs \Rightarrow preO_{sub}$$

This is known as the law of weakening of precondition. The second template ensures that the state after the subtype operation (marked by ') represents one of those abstract states in which an operation of a supertype could terminate (here $spc(O)$ denotes a predicative specification of O expressed in a mixed pre- and postconditions):

$$\forall S_{sub}, S_{sup}, S'_{sub} \bullet preO_{sup} \wedge Abs \wedge spc(O_{sub}) \Rightarrow \exists S'_{sup} \bullet Abs \wedge spc(O_{sup})$$

The second condition is the law of strengthening of postcondition. If these conditions are satisfied then a subtype operation is suitable for all purposes for which a supertype operation is suitable. If we model a type and subtype by an abstract machine and its refinement then the subtyping conditions above will be correctly established.

Actually what we said above means that type t_j is a subtype of a type t_i iff a reduct of t_j to state attributes and functions corresponding to t_i is a refinement of t_i .

5 Conformance Associations

Here we introduce a notion of type conformance associations that deviates from given sometimes in the literature [11]. The reason for the new notion is that we look here not only for the object substitutability, but mostly for reusability of object fragments. For a given type we would like to associate a collection of other type definitions partially conforming to our type and forming a reusability perspective for the type. Therefore, we redefine conformance as follows.

A *conformance association* between types t_1, t_2 holds (we say " t_2 conforms to t_1 ") iff there exists a common reduct of t_1 and t_2 .

A *common reduct* $t = CR(t_1, t_2)$ is such type (other than the null type) that t is a reduct $r_t(t_1)$ of t_1 and there exists a reduct $r_t(t_2)$ that refines $r_t(t_1)$ ².

A *conformance relation* on a set of types is reflexive and transitive. The conformance relation is not antisymmetric. It means that from the facts that t_1 conforms to t_2 and t_2 conforms to t_1 it does not follow that $t_1 = t_2$. On the contrary, for subtyping relation that is reflexive, transitive and antisymmetric (a partial order relation), if t_1 is a subtype of t_2 and t_2 is a subtype of t_1 then $t_1 = t_2$.

A particular conformance t_2 for a particular type t_1 gives a reuse possibilities for t_1 . The reducts defined for t_1 through conformance association define states and behaviors of t_2 that can be reused to implement the type t_1 . A type t_1 can have many conformances with a type t_2 . However, we are interested in a conformances in which any extension of a common reduct will not lead to new conformances. Such conformances are called *the most common conformances*.

²Here $r_x(y)$ denotes a projection of the type y to the attributes of the type x .

A *most common reduct* $t = MCR(t_1, t_2)$ is a reduct $r_i(t_1)$ of t_1 such that there exists a reduct $r_i(t_2)$ that refines $r_i(t_1)$ and there can be no other reduct t_i of t_1 ($r_{t_i}(t_1)$) such that t is a reduct of t_i , t_i is not equal to t and there exists a reduct $r_{t_i}(t_2)$ that refines $r_{t_i}(t_1)$.

A conformance between pairs of types t_1, t_2 is *most common* iff these types are related through the most common reduct. It is clear that for a pair of types t_1, t_2 there exists at most one MCR . If most common reduct of t_1 and t_2 is equal to t_1 then we say that t_2 is a subtype of t_1 .

Now we can extend the definition of a conformance association of a type t_1 to a collection of types T .

We define a *conformance of a type* t as a collection of types $S = \{t_1, t_2, \dots, t_n\}$ such that any type t_i in S conforms to t .

It is possible that some of the states and behaviors may be shared among the types in the conformance because of subtyping and conformance associations that may exist among them. A type can have many conformances. However, for any type there exists a conformance such that adding a type to the conformance does not add any additional conformance information to the original type and removing a type from the conformance would lose such information. This conformance is called the *most common conformance for the type*.

We define a conformance $S = \{t_1, t_2, \dots, t_n\}$ of a type t to be the *most common conformance of* t iff we take the most common conformance of each type in S to t and there does not exist a type t_v not belonging to S such that $MCR(t, t_i)$ is a reduct of $MCR(t, t_v)$ where t_i is a type belonging to S . A most common conformance of a type t is denoted by $MC(t)$.

The $MC(t)$ characterizes a reuse perspective for a type t .

6 Type Inferencing Associations

These associations usually focus on issues of a type inferencing during object calculus formulae evaluation under assumption of the type specification completeness. Two kinds of formulae are distinguished: object-preserving and object-creating ones. The object-preserving formulae are limited to returning existing objects from an object base while object-creating formulae may create new objects during their evaluation [11]. The object-creating formulae can form new types that were not integrated with the types from which they were formed. Based on the notion of a well-defined type lattice we should have a systematic possibility to integrate these new types into the existing lattice.

The type operations and expressions usually are intended as a technique for target-creating formulae evaluation, view and schema evolution support. Here we mostly emphasize these associations for the process of design with reuse - to form reducts introducing them properly into the type lattice, to produce composition of reducts and to express formulae leading to views serving as refinements of the required types.

Here following [11], we introduce type operations that are suitable for incomplete type specifications containing only signatures of functions.

Let $T_i (1 \leq i \leq n)$ denotes types.

- An operation $T_1 \& T_2$ produces a type T as a *meet* of the operand types. T includes functions that are common for T_1 and T_2 . If $T_2 (T_1)$ is a subtype of $T_1 (T_2)$ then $T_1 (T_2)$ is a result of a type meet. Type T is placed in the type lattice as a supertype of types being operands of the operation.
- An operation $T_1 | T_2$ produces as a result a type T that is a *join* of the operand types. T includes a union of functions of T_1 and T_2 . If $T_2 (T_1)$ is a subtype of $T_1 (T_2)$ then $T_2 (T_1)$ is a result of a join operation. A type T is placed in the type lattice as a subtype of the operand types.
- An operation $T_1 * T_2$ produces as a result a type T that is a *product* of the operand types. A type T is placed in the type lattice as a subtype of other product types (but not of the operand types) in accordance with the interfaces defined for the product types.

A product type $T_1 * T_2 * \dots * T_n$ is placed in the type lattice as a subtype of a product type $T'_1 * T'_2 * \dots * T'_m$ if $m \leq n$ and T_i is a subtype of T'_i for $1 \leq i \leq m$. It is placed in the type lattice as a supertype of $T''_1 * T''_2 * \dots * T''_k$ type if $n \leq k$ and T_i is a supertype of T''_i for $1 \leq i \leq n$. If a product type cannot

be placed into a type lattice as a subtype of another product type it is placed there as a subtype of the root type of the lattice.

Complete type specifications include specifications of assertions (predicates constraining admissible type values) and specifications of functions serving as type operations. In this case type inferencing operations should produce *correct* types placed into type lattice in accordance with the definition of subtyping association given above that is clarified as follows.

Using the operations above, type expressions are introduced taking into account that meet and join operations are commutative and associative and a product operation is neither commutative nor associative. A product operation has a higher priority comparing to the operations of type meet and join. The type inferencing operations defined above should be extended with the following transformations (justifications) [6]:

- Type meet operation:
Assertions created in the resulting type (obtained from operand type assertions) should be formed by a maximal subset of assertions common for the operand types and relevant to a reduced set of functions of a resulting type.
- Type join operation:
A set of assertions inherited from operand types should be consistent (the resulting type should have a model).
- Any type operation:
A specification of any interfacing function of a resulting type can be chosen among corresponding interfacing functions of operand types if the functions are in a refinement order (for a supertype (subtype) more abstract (more refined) function should be chosen). If the functions are not in a refinement order they are considered to be different interface functions.

In case of a specification completeness type inference operations should produce correct supertype (subtype) of the operand types. To do that the operations should perform nontrivial transformations of specifications and justification of their correctness.

7 Conclusion

The paper identifies basic associations between types that are required to support reasoning of semantic interoperability and reuse in course of a megaprogram design. The pre-existing information resource specifications (IRS) and information system requirements specifications (ISRS) are assumed to be complete. By completeness we mean that the specifications should be sufficient to reason that a resource is applicable to a given problem. For reuse we apply a model of federated composite object integrating data and behavior from various sources. To identify reusable fragments we consider reducts (projections of object interfaces defined by types) as patterns of reuse. Basic techniques supporting creation of such objects include delegation, substitutability, inheritance providing for composition and sharing of behavior and state of objects.

For the basic associations between types required to reason about the object fragments reusability we identified reducts, refinement, subtyping, various conformances and type inferencing associations.

We treat type refinement to be the fundamental type association that is required to justify correctness of reuse of pre-existing object fragments (object state and behavior) as well as to establish other type associations (such as subtyping, conformances and type inferencing).

The type associations defined support the semantically interoperable design in the SYNTHESIS project.

References

- [1] Abrial J.-R. B-Technology. Technical overview. BP International Ltd., 1992, 73 p.

- [2] Enrich H.-D., Engels G., Paredaens J., and Wegner P. Fundamentals of Object-Oriented Languages, Systems and Methods. Dagstuhl Seminar Report, 95, August 1994.
- [3] Kalinichenko L.A. Emerging semantic-based interoperable information system technology. Computers as our better partners. Proceedings of the International IISF/ACM Symposium, Tokyo, World Scientific, 1994
- [4] Kalinichenko L.A. Constructing of Concretization of an Application Domain Description by Information Resources., Advances in Database Systems. *USIM*, 6, 1994, Kiev.
- [5] Kalinichenko L.A. SYNTHESIS: the language for description, design and programming of the heterogeneous interoperable information resource environment.- Institute for Problems of Informatics, Russian Academy of Sciences, Moscow, 1995, 103 p.
- [6] Kalinichenko L.A., Zadorozhny V.I. Type Inferencing Based on Complete Type Specifications. Proc. of the Second International Workshop "Advances in Databases and Information Systems", Moscow, 1995, Springer
- [7] B. Liskov, J. M. Wing. Specifications and their use in defining subtypes. *OOPSLA'93*, 1993.
- [8] Malenfant J. On semantic diversity of delegation -based programming languages. *OOPSLA'95*, Austin, TX, USA
- [9] Morgan C. Programming from Specifications. Prentice Hall, 1994
- [10] Object Management Group, "The Common Object Request Broker: Architecture and Specification", OMG Document Number 91.12.1, December 1991.
- [11] R. J. Peters, A. Lipka, M. T. Ozsu, and D. Szafron. The query model and query language of TIGUKAT. *Technical Report TR 93 -01*, Computer Science Department, University of Alberta, Canada, June 1993.
- [12] J. M. Spivey. *The Z Notation. A reference manual*. Prentice-Hall, 1989.
- [13] Wiederhold G., Wegner P., Ceri S. Toward megaprogramming. *CACM*, v. 35, N 11, November 1992