

CC++: A database language supporting the development of collaborative, reusable applications

Waldemar Wiczerzycki

Department of Information Technology
The Poznan University of Economics
wicz@kti.ae.poznan.pl

Abstract. In the paper the CC++ database language is proposed that is addressed to databases supporting collaboration between users. The CC++ language offers high expression level and provides concepts required by teams of cooperating users. A very important advantage of the proposed approach is the increase of software reuse. The database application itself contains different sets of versions of data and functions which correspond to different user requirements, e.g. different hardware/software platforms, communication languages, user interfaces, provided functionalities, etc. Thus, in many cases the program behavior may be changed without the need for source program modification and re-compilation.

1 Introduction

The theory and technology of classical databases is very mature, commonly accepted and verified over many years, thus the following question naturally arises: can we apply this technology in collaborative systems, instead of re-implementing database functions from scratch and embedding them in collaborative systems? In other words: can we develop collaborative systems as database applications, thus probably saving time normally spent on re-implementation of selected database functions? We can obviously try, but there is one substantial drawback we have to take into account. The classical database paradigm assumes namely that database users are totally isolated.

In such situation, in order to develop collaborative database applications, we have to extend database technology. The required extensions should be applied simultaneously to both: data definition techniques, manipulation mechanisms and transaction management algorithms. Data definition techniques have to facilitate the representation of data structures that are specific to cooperation processes. Data manipulation mechanisms have to support the efficient development of collaborative database applications, with particular emphasis put on a high level of software re-usability. Finally, transaction management algorithms have to support human interaction and exchange of non-committed data.

There are many database models proposed in the literature that are addressed to advanced domains of database applications, in particular to computer aided design (CAD) and computer aided software engineering (CASE) [1, 4, 5]. These models (and languages) substantially support individual design and development activities of

database users. However, they do not sufficiently support group activities. It follows from the common assumption that database users communicate only via committed data. Since the users are totally isolated by the database system, each of them has an impression that the system is dedicated to him. When users collaborate to achieve a common goal, this approach is obviously too restrictive. Collaborators have to communicate directly before they agree on a data value.

Among basic drawbacks of the approaches mentioned above one can distinguish:

- the lack of data structures used to distinguish information particularly important to groups of collaborating users that could be simultaneously accessed by them without conflicts,
- difficulties with representing collaboration structure and collaboration organization in the data model; the lack of modeling concepts reflecting different levels of collaboration intensity,
- the lack of semantic relationships posed on data that could reflect relationships between the users operating on these data, following from different cooperation forms,
- the lack of operations reflecting collaboration techniques and collaboration management.

Notice that all the above drawbacks relate to both: data modeling techniques and manipulation mechanisms which are mutually strictly related. Data manipulation techniques emphasize structural aspects of the real-world being modeled in a database system, while data manipulation mechanisms emphasize its behavioral aspects.

To solve the problems mentioned above, in particular to efficiently support wide collaboration among database users a new data definition model, called *CDM model (Collaborative Data Model)*, has been elaborated [7]. The *CDM* model is oriented for the specificity of cooperation scenarios, cooperation techniques and cooperation management.

Parallel to the data definition model, a new transaction model, called *multiuser transaction model*, has been proposed [6], as well as new transaction management techniques aiming at the avoidance of data access conflicts and conflict resolution. These techniques are based mainly on users' negotiations. Multiuser transactions are flat transactions in which, in comparison to classical ACID transactions, the isolation property is relaxed.

Having (1) a data definition model relevant to the requirements of highly collaborative environments concerning information structuring and exchange, (2) a transaction model reflecting the specificity of cooperation processes, in particular simultaneous access to the same data and negotiations on the evolution of data values, there is still a need for (3) an appropriate data manipulation language, well suited to the level of expression required by collaborating database users, on one hand, and widely matching the data and transaction models, on the other hand.

The goal of this paper is to propose a database language fulfilling the aforementioned requirements, augmented by a natural need for a high level of software reusability, i.e. for a need to develop generic, scalable database applications that can be easily adapted to specific user preferences and expectations, without the necessity for source code modification and recompilation.

Taking into account commercial and prototype database systems, one can distinguish three typical approaches concerning database language development [2,

3]: (1) the application of existing programming language that is normally used in non-persistent environments, (2) the extension of existing programming language towards the specificity of database systems and database applications, and (3) the development of a new language from scratch. In first approach, a well known, commonly used programming language is typically chosen (e.g. C++, Smalltalk). This approach has one important advantage. Namely, the selected database language is usually the same as the language used to implement the DBMS. Thus, the entire system is very homogeneous and can be very efficient.

Second approach consists in augmenting a well known programming language by new semantic and syntactic concepts, specific to a particular DBMS, especially to its data model (e.g. *O₂C* language in *O₂* DBMS, *OPAL* language in *Gemstone* DBMS). This approach in general preserves all advantages of the first approach. A new important advantage is the increase of modeling (programming) expression level and the increase of system homogeneity level, as a result of embedding specific database concepts into a database language, e.g. integrity constraints, triggers, database schema operations.

Third approach leads to the development of new database languages devoted to a specific database application domains, frequently without any reference to existing data models and programming languages. In this approach, the relationship between a data model and a database language is typically reversed. A database language is elaborated first, and then a relevant data model is proposed next. The main advantage of this approach is a high relevance of a database language to the database application domain. An obvious drawback of the approach is its reduced applicability, since a database system is addressed to very specific domain. That is probably why this approach is used only in case of prototype (non-commercial) DBMS.

In the paper the second approach is used. The paper proposes a database language, called *CC++* (Collaborative C++). The language is a natural extension of a commonly used classical programming language, namely C++, towards operating on persistent and multiversion data, in particular data structured according to the *CDM* data definition model and operated in the scope of multiuser transactions.

The paper is organized in the following way. Section 2 presents basic semantic concepts of the proposed database language. Section 3 discusses persistency aspects of storing data in a database. Section 4 details syntax extensions of the C++ language necessary to develop *CC++* language. Section 5 concludes the paper.

2 Basic Concepts of the Database Language

In the proposed approach a database application (a program) is composed of a *program body* and a set of logically independent *program contexts*. The program body contains global functions and global data structures. A particular global function, called the *main function*, is distinguished which starts and ends the program execution. Each program context contains exactly one variant of every class defined in the program and one variant of every non-global function, called a *context function*. It means that logically the number of both classes and context functions is the same in every program context.

During the program execution only one context is active. It may be, however, changed dynamically by a particular context switching command, which can be used only in the scope of global functions. Users of the same application can invoke this command independently, thus they can simultaneously operate on different contexts of the same database domain. In other words, during the entire database session every user perceives the application as a sum of the program body and exactly one program context.

Variants of the same class/function belonging to different contexts need not be different. In order to avoid redundancy, the same variants are physically shared by program contexts. In case of classes, there are two possible levels of sharing: total and partial. *Total class sharing* is used when the class definition is exactly the same in two program contexts. *Partial class sharing* is used when the class definition is partially different in two program contexts, i.e. some class variables and/or methods are modified, while other remain unchanged. In the former case a single class is a unit of sharing, while in the latter case two units of sharing are used: a single variable and a single method.

In order to correctly resolve dynamic context switching the late binding technique is used and some restrictions on versioning are imposed. The late binding technique is commonly used in object-oriented languages: some of them (like Smalltalk) use it widely, while other (like C++), for the sake of efficiency, constrain its scope only to virtual methods. Because this technique is well-known, it does not require any additional comments.

Restrictions imposed on versioning are due to required language simplicity and implementation. In general, it is assumed that all objects (being class instances) and functions, no matter which context is currently used, react possibly in different ways to the same set of calls from the program body. As a consequence, constraints on variants of context functions and classes are imposed. Different variants of the same context function must have the same signature, i.e. the same number and type of arguments, and a returned value, in any. Their implementation, however, may differ freely. In other words, variants of the same context function may embed different local variables, nested functions and instructions.

In case of classes, two main constraints are distinguished. First, the class inheritance hierarchy (*DAG*) is common for all contexts. It substantially simplifies the process of class definition: the inheritance hierarchy is orthogonal to contexts and their derivation rules. Second, different variants of classes must preserve the same interface, i.e. they must provide the same number and the same signature of public methods and variables. The implementation of these methods, however, may be arbitrarily different. Also, all other elements of the class definition which are not available in the class interface, like private methods, private variables, may differ freely.

Like classes, contexts are defined statically in the program source file. A particular context, called *root context*, is distinguished. Its definition is divided into two distinct parts. In the first part, interfaces of all classes used in the program are specified, and signatures of all context functions are given. As it was mentioned above, these properties are common to all contexts, which means that they are logically copied to all other contexts subsequently defined in the program. The second part of the root context definition contains, on one hand, the definition of private variables of all

classes and signatures of private methods, and, on the other hand, the implementation of both: context functions and methods (public and private).

The root context is the only one which is defined from scratch. All other contexts are derived either from the root context or from another non-root context. Whenever a new context (a child) is derived, logical copies of all class and context function variants of its parent are associated with it. Next, particular subset of classes and context functions may be re-implemented in the child context, thus introducing new class and function variants. In case of static environments, program contexts must be defined before the compilation. As a consequence, the time relationship corresponds to the order of context definition in the program source file. In case of dynamic environments, the user works interactively with the programming system. Whenever a new program context is created, a time stamp is associated with it. As a consequence, the time relationship corresponds to the order of time stamps.

Finally, it is worth to emphasize that versioning is transparent to the user. At a program run-time only one context is active. It contains exactly one version of every class and function, thus it may be considered as a monoversion environment. The user addressing a particular class or function need not be aware that it has different implementations (versions) in other contexts.

3 Non-Persistent and Persistent Objects

In this section we present how non-persistent and persistent objects are distinguished in a source code of a database application. A non-persistent object life time is limited by the application execution time. A persistent object is stored in the DBMS directly after its creation by the application until it is explicitly deleted. A class of non-persistent objects is represented only at the application level, while a class that has at least one persistent instance is represented at the database schema level. An object can become persistent only if it is created at the outermost level of the application structure, i.e. an object that is globally accessible anywhere in the application, or if it is directly or indirectly (i.e. via other objects) assigned to a variable of a global scope. As a consequence, objects created and used in local functions and methods are always non-persistent. We distinguish two ways of specifying if an object is persistent or not: explicit and implicit.

Explicit specification of persistent objects can be achieved by the use of the ***persistent*** keyword in a source code. This keyword may be used:

- in a definition of a single object,
- in a class definition,
- in a context definition.

In first case the ***persistent*** keyword associates an identifier of a newly defined object. If the object is a global class instance, then it belongs logically to a program body, which means it is a persistent, non-versionable object. Similarly, if an object is a context class instance, then it belongs logically to contexts, thus it is persistent multiversion object.

In second case the ***persistent*** keyword associates an identifier of a newly defined class. It indicates so called ***persistent class***, i.e. a class which all instances are

persistent. In this situation, there is no need to apply the *persistent* keyword in a class instantiation. By analogy to a single object definition, a place where a persistent class is defined is meaningful. It decides whether we deal with a class of non-versionable or multiversion objects.

In case of defining both: a persistent object and a persistent class, a class definition is automatically introduced into the database schema during first execution of the database application. During next application execution the DBMS checks only if the class definition has been modified. A class definition modification implies in general a deletion of all class instances from the database and database schema upgrade.

In a final third case the *persistent* keyword associates a name of a newly defined context that becomes so called *persistent context*. For a persistent context the DBMS stores corresponding versions of all context class instances, that have been defined in one of the two aforementioned ways. For a particular version of a context class, in order to be persistent, two conditions have to be fulfilled. First, a corresponding multiversion object has to be defined as persistent (or a class it belongs to). Second, a corresponding context containing a considered object version has to be persistent.

Persistent object definition is illustrated in Figure 1. A database application is composed of two contexts: Cx_1 and Cx_2 containing two context classes: C_1 and C_2 . In the application body C_3 class is defined. Persistent application elements are denoted by the letter **P**: object O_{11} being an instance of class C_1 , class C_2 and object O_{31} being an instance of class C_3 . Moreover, context Cx_1 is persistent. Objects stored in the DBMS are shaded in the figure. We mean here: classes C_1 and C_2 in versions corresponding to the context Cx_1 , class C_3 , instances O_{11} , O_{21} , O_{22} in versions corresponding to context Cx_1 , and non-versionable instance O_{32} .

Besides explicit specification of persistent objects, implicit specification is also available. First, all component objects of a persistent object become automatically persistent. For example, if an object O being an instance of the *Book* class is explicitly persistent, then all respective instances of classes: *Section*, *Subsection*, *Figure*, *Table*, *Paragraph* embedded in object O become implicitly persistent. Second, if a particular class is persistent, then all its direct and indirect subclasses become automatically persistent, due to the semantics of the *IS-A* relationship binding classes and subclasses. For example, if the *Figure* class is persistent then all classes derived from *Figure* become automatically persistent: *Graph*, *Diagram*, *Picture* etc. In both cases implicit specification of persistent object is recursively applied for object composition hierarchies and class derivation hierarchy, respectively.

A persistent database object can be easily addressed at the application level, provided a class it belongs to is known. For this purpose, in the proposed approach, *getInstances* message has to be sent to the respective class. A corresponding class method determines as a result of its execution a set of references (pointers) to all class instances stored in the database.

Finally, we discuss two particular sets of database classes that make it possible to store global and context function in a database. If the body of a program identified by $\langle program_name \rangle$ contains at least one function whose identifier is preceded by the *persistent* keyword, then a special non-versionable class is automatically created having the name *body* $\langle program_name \rangle$. This class comprises methods representing all global functions preceded by the *persistent* keyword. Similarly, if a program contains at least one context function whose identifier is preceded by the *persistent*

keyword, then a special multiversion class is automatically created with the name *context<program_name>*. This class comprises methods representing all context functions preceded by the *persistent* keyword. The number of versions of the class *context<program_name>* matches the number of persistent contexts at the application level.

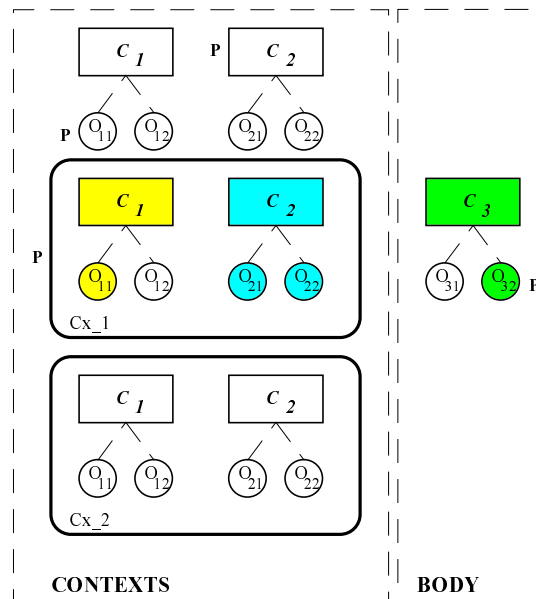


Fig. 1. Persistent objects

Classes: *body<program_name>* and *context<program_name>* are not instantiated. Since they are typical objects, messages can be addressed to them, in particular a message that retrieves a set of references to all class methods. Next, it is possible to execute a function selected from the database at the application level.

To summarize, in the proposed approach both: classes, class instances and functions may be persistent. As a consequence, it is possible to store in a database not only data (objects) operated by the application, but parts of application code as well.

4 CC++ Language

In this section we show C++ language extensions necessary to support the concepts proposed in sections 2 and 3.

The concept of the program body relates to a single section of a program which contains typical C++ statements enclosed in two braces and prefixed by a new keyword - *body*. Global functions correspond to typical C++ functions, while global data structures correspond to outer level C++ variables. In some sense, a program

body may be viewed as a classical C++ program which does not contain class definitions, however, may contain definitions of class instances.

The concept of the program context relates to a section of a program which is uniquely identified, enclosed in braces, and prefixed by a new keyword - *context*. Context functions correspond to typical C++ functions with limited scope, while class variant definitions correspond to typical C++ class definitions restricted in such a way, that they are visible or not at a program run-time, depending on which context is active. In some sense, a program context may be viewed as a set of class and function definitions which have limited scope at a program run-time.

Syntactically, there is a difference between a root context and a non-root context definition: the latter one does not contain the first part of the root context definition, i.e. the specification of class interfaces and context function signatures. Moreover, due to partial class sharing, in case of introducing new class variants, only differences are specified.

The context switching is performed by the use of a special command - *select* with a parameter pointing to the context that will be activated. This command, of course, may be used only in the scope of a program body.

Now the general program syntax is described in a more detailed way.

A program is composed of three main parts: a header, contexts and a body. The header contains compiler directives, constant definitions, type definitions and enumerations which are global to the whole program. Next, all program contexts are defined starting with the root context. The definition of each context contains a unique identifier of the context, the way of its derivation, definitions of classes and context functions. Finally, the program body contains definitions of global data structures and global functions, including *main* function. The general program syntax is given below.

```
// root context definition
context root
    { // classes
      // context functions };
// derived contexts
context <context_name> parent <parent_context_name>
    { // classes
      // context functions }
...
context <context_name> parent <parent_context_name>
    { // default context - last one on the list
    }
body
    { // global data structures
      // global functions
      ...
      void main ()
      {
        // main function
      } }
}
```

Three new keywords are introduced: *context*, *parent*, *body* and the pre-defined context name *root*.

Now we present the syntax of the root context. In general, the root context definition is composed of two sections: an interface and an implementation. In the

interface section, denoted by the keyword *interface*, class interfaces and context function signatures are given. Every class interface is composed of public data members and public member functions, that is the elements which are shared by all the program contexts. In case of public data members, their identifiers and types are specified according to the normal C++ syntax. In case of member functions, their signatures are only given, that means their names, type of returned values and type of input arguments. It is not possible to follow the member function signature by its implementation, as in case of in-line C++ functions. In-line functions are still available, however they must be defined outside of the interface section.

In the implementation section every class having private or protected members, that is members which are not included in the class interface, occurs once again for the purpose of specifying those members. In fact, this is what differs variants of the same class in different program contexts. Next, the implementation of all member and context functions introduced in the interface section is given. Finally, the implementation of non-public member functions specified at the beginning of implementation section is shown. Member function identifiers are prefixed by corresponding class identifiers, according to the normal C++ syntax.

The syntax of non-root program contexts is simplified in comparison to the one presented above. First of all, the interface section is omitted. Thus, non-root contexts contain only definitions (and implementations) of new variants of context functions and classes. In case of context functions, their new implementation is given exactly in the same way as in the root context. In case of classes, the encapsulated part of their definition, that is all non-public members, may be potentially modified. New members may be also added, and some members of the parent context may be hidden. As a consequence, the class definition contains three sections: shared, private and protected. The shared section, prefixed by the keyword *shared* and colon, enlists the identifiers of members which are shared with the parent context. Two other sections, prefixed by the keywords *private* and *protected*, include re-defined or new data members and member functions, which are not shared with the parent context. Similarly to the root context syntax, every non-root context must contain the implementation of all member functions whose signatures are specified in the corresponding class variant definition.

5 Conclusions

In the paper the *CC++* database language has been proposed that is addressed to databases supporting collaboration between users, in particular databases structured according to the *CDM* data model and managed according to the *multiuser transaction* model. The *CC++* language widely matches the concepts of these models offering high expression level and providing concepts required by teams of cooperating users.

A very important advantage of the proposed approach is the increase of software reuse. The database application (program) itself contains different sets of versions of data and functions which correspond to different user requirements, e.g. different hardware/software platforms, communication languages, user interfaces, provided

functionalities, etc. Thus, in many cases the program behavior may be changed without the need for source program modification and re-compilation.

Other advantages of the approach are the following. First, the data structuring techniques are enriched in comparison to classical object-oriented languages. As a consequence, it is easier to model real-world objects (entities) precisely in the program, especially if they are available in different variants, revisions, representations, etc. Second, program functions may be written in a generic way. Their behavior depends on the context actually chosen. Finally, versioning is transparent to the application developer: there is no need to explicitly distinguish different variants of the same classes/functions. The programming system automatically identifies proper variants at program run-time, depending on the context addressed.

The proposed approach is general enough to be easily adopted in any imperative language, e.g. *Ada*, *Pascal*, *Smalltalk*. To achieve this, it is sufficient to extend the syntax of the considered language by clauses for context and body definition, on one hand, and by clauses for class variant definition, on the other hand, as illustrated for *C++* language. Thus, the entire semantics and functionality of the base language is preserved.

References

1. Agraval R. et al., *Object Versioning in Ode*, IEEE, 1991.
2. P. Atzeni, V. Tannen (Eds.): *Database Programming Languages (DBPL-5)*, Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995. Electronic Workshops in Computing, Springer 1996.
3. S. Cluet, R. Hull (Eds.): *Database Programming Languages*, 6th International Workshop, DBPL-6, Estes Park, Colorado, USA, August 18-20, 1997, Proceedings. Lecture Notes in Computer Science, Vol. 1369, Springer, 1998.
4. Chou H. T., Kim W., *Versions and Change Notification in Object-Oriented Database System*, Proc. of the Design Automation Conf., 1994.
5. Lamb C., Landis G., Orenstein J., Weinreb D., *The ObjectStore Database System*, Communications of the ACM, Vol. 34, No. 10, 1991.
6. W. Wiczerzycki, *Transaction Management in Databases Supporting Collaborative Applications*, Second East-European Symposium on Advances in Databases and Information Systems - ADBIS'98, Poznań, pp. 107-118, 1998.
7. W. Wiczerzycki, *Database Model for Web-Based Cooperative Applications*, Eighth International Conference on Information and Knowledge Management, CIKM'99, November 2-6, 1999, Kansas City, MO, USA.