Memops: Data modeling and automatic code generation in multiple languages

Rasmus H. Fogh[1], Wayne Boucher[1], Wim F. Vranken[2], Anne Pajon[2], Tim J. Stevens[1], T.N. Bhat[3], John Westbrook[4], John M.C. Ionides[2] and Ernest D. Laue[1]

[1]Department of Biochemistry, University of Cambridge,
80 Tennis Court Road, Cambridge, CB2 1GA, UK
{r.h.fogh, wb104, tjs23, e.d.laue}@bioc.cam.ac.uk
[2]MSD group, EMBL-EBI, European Bioinformatics Institute,
Wellcome Trust Genome Campus, Hinxton, Cambridge, CB10 1SD, UK
{wim, pajon, jmci}@ebi.ac.uk
[3]Biotechnology Division (831), NIST,
100 Bureau Drive, Stop 8310, Gaithersburg, MD 20899-8314, USA
bhat@nist.gov
[4]Department of Chemistry and Chemical Biology, Rutgers University, Rutgers,
State University of New Jersey, 610 Taylor Road, Piscataway, NJ 08854-8087, USA
jwest@rcsb.rutgers.edu

Abstract The Memops framework is a tool for data modelling and the fully automatic generation of subroutine libraries for data access in multiple computer languages. The data model is entered in a UML subset similar to XMI. Code is generated automatically for several languages, with Python and Java being supported so far, and C/C++ and Perl support planned. The product includes an object-oriented data interaction API and its implementation, complete with data validation and checking and a notifier facility. Data storage in either XML files or relational databases is integrated in the data access subroutines. XML and database schemas and documentation is also generated from the UML model.

To achieve long-term maintainability across different platforms, Memops uses a single platform-independent model directly as the basis for code generation. Platform-specific information, which cannot be completely dispensed with, is entered in the UML model as a series of tagged values. As an example, model-specific, language-specific code is kept in the model as code snippets. These amount to less that 1 per cent of the final generated code. The approach is successful because Memops is targeted to a limited field - data modelling and data access. Memops is currently used for a data model in the structural biology field with 300 classes. A Python API (250 000 lines), and a number of applications based on it have been released.

1 Introduction

1.1 Project Goals

Memops is a product of the CCPN project [1], which was funded by the BBSRC to create a data exchange standard for the field of macromolecular NMR spectroscopy. Such a standard should allow a conforming application to modify data in a plug-and-play manner, with all modifications being kept for eventual database deposition. As might be expected in a developing scientific field, the situation facing CCPN was characterised by a substantial agreement on the kinds of data that needed to be stored, a great variety of potential uses and algorithms for exploiting the data, and the expectation of significant future changes for both. Organisationally, existing software in the field was developed by a large number of poorly resourced academic groups, each making its own choices with respect to platforms, programming languages, and data representation and storage. The resulting programs tended to be closely attuned to the needs of local users, but to have severe problems with respect to interoperability and long-term maintenance because of the lack of coordination and resources. With the rise of structural biology and high-throughput methods, however, there was an increasing need for automation, for joining different analysis programs together into software pipelines, and for large-scale harvesting and deposition of data.

1.2 MDA and Autogeneration

Model-Driven Architecture and automatic code generation seemed the only way of achieving a data exchange standard capable of being adopted and used in the field. In the absence of a mechanism for enforcing compliance, a standard could only hope to be adopted if it allowed programmers to continue working with their favourite platform. To make the changeover attractive the model must come with enough functionality in its subroutine libraries to actually make it easier to develop applications with the Memops libraries than without them. With MDA the underlying model could be precisely specified to serve as a standard, and at the same time implementations could be provided for a variety of programming languages and storage platforms. As a corollary, something very close to fully automatic code generation is indispensable to allow supporting highly functional subroutine libraries across multiple platforms with a realistic expenditure of resources. Not finding a suitable application at the start of the project, we decided to develop Memops to meet the twin requirements of simultaneous multi-platform support and 100% automatic code generation.

2 The Data Model

A data model is a description of the data for a particular subject area, how they are defined and organized, and how they relate to one another. In Memops, the data model serves as the specification for all generated code, in keeping with the Memops strategy of providing a data access layer rather than a complete application

2.1 Model organization - packages

A Memops data model is represented as a platform-independent model in UML. Memops uses a UML subset very similar to the XMI subset used for metamodel definition, with some additional tagged values. The model is generated with a standard UML editing program.

The model is subdivided in packages, which ideally should represent separate domains of knowledge and be loosely coupled to other packages. Packages serve to organize both the model description, the generated subroutine libraries, and the storage of the actual data. The purpose of this organization is to allow an application (or a data modeler) to work on part of a multidisciplinary project without having to consider either code or data for packages that are not relevant in the context. This also facilitates the production of integrated data standards for large areas of knowledge, since widely separated domains can have full control over their own packages, while sharing packages for domains that are in common.

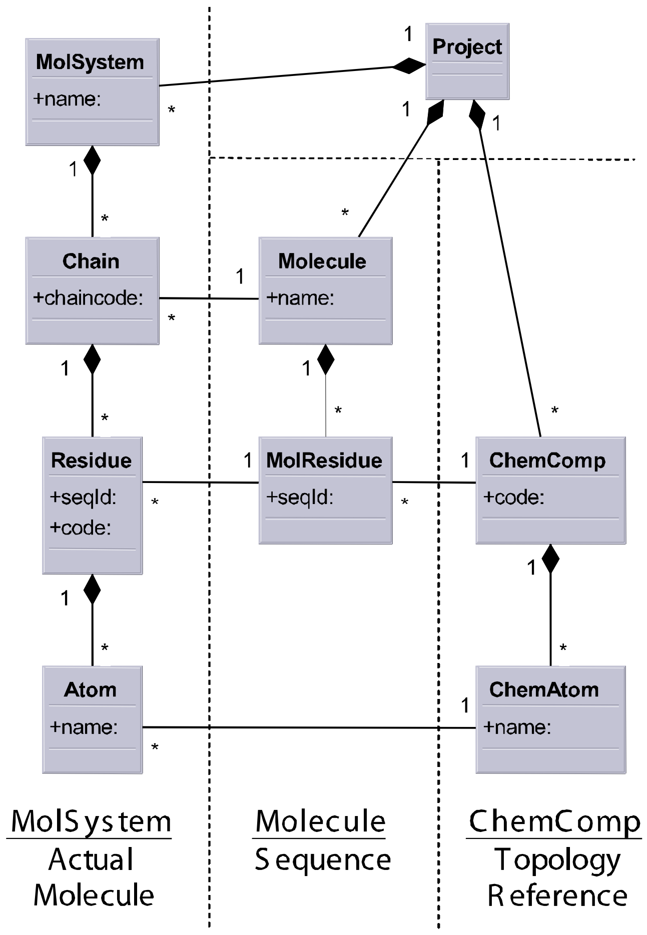

Fig. 1 A simplified part of the CCP macromolecular Data Model. Only composition ('parent') links, attributes making up the class key, and some of the more important links are shown. Dotted lines separate different model packages.

2.2 Model Organization - Relationships between Classes

There are some constraints on the allowed models to permit simple and efficient API implementations (see Fig. 1 for an illustration). All classes must have a composition association to another class, known as the 'parent' class (not to be confused with inheritance). The 'parent' links connect all data objects into a tree with a single root object. This has the dual purpose of providing a clear navigation path between any pair of objects, and of specifying a containment hierarchy for

XML storage. There is a further requirement that any class must have a set of attributes (or links) that uniquely identifies each object relative to sister objects with the same parent. If no natural key is present, an integer ‘serial’ must be provided. Combined with the tree of ‘parent’ links this provides a unique, persistent, composite identifier for each object without relying on absolute URLs or locally generated random integers, either of which may change with time. These identifiers are used to specify inter-file links between objects for XML storage.

2.3 Methods and Constraints

Class methods are mostly implicit in the model, as the methods needed for data access (see section 4.5) can be generated fully automatically once the data type and cardinalities of an attribute are known. Methods are specified explicitly if their behavior differs from the standard, or if it is desired to provide additional functionality. A case in point is derived attributes and links. These are specified to behave like normal attributes as far as the interface is concerned, but are calculated on-the-fly rather than stored; here the necessary derivation functions must be specified. When specifying a method (or a constraint) code snippets are added for the supported languages (currently Python and Java). For the future it is considered to enter code snippets in OCL, and to provide automatic translation to the supported languages [2].

Constraints may be entered on attributes, links, classes and data types, in the same way as for methods. These constraints are then evaluated either before modifying . ed/data or in a validity checking step, and serve to prevent illegal data from being entered.

3 Automatic Code Generation

As illustrated in Figure 2, subroutines for data interaction (APIs), data storage, and documentation are all generated automatically from the abstract data model. Autogeneration guarantees that all of the generated documents are synchronized, greatly simplifying the maintenance of the project. For API implementations, I/O routines, and even documentation, over 99% of the final code (or documentation) can be generated fully automatically from the data model itself. The remaining 1% is added to the model in the form of tagged values with code snippets or documentation strings, or written to a separate file as backward-compatibility I/O code. As a result there is no post-generation editing, and the generated code is ready for use immediately after generation.

3.1 The Generation Process

The automatic code generation is a two-stage process. In the first stage the information describing the model is extracted from the UML modeling tool (ObjectDomain [3]), transformed into a set of Python objects in memory, and then written to a set of files. In the second stage these files are read to recreate a set of in-memory Python objects, which then form the basis for the various generation scripts. This approach decouples the generation process from the UML modeling tool, and allows the substitution of other tools at the price of changing only a single module of the generation software.

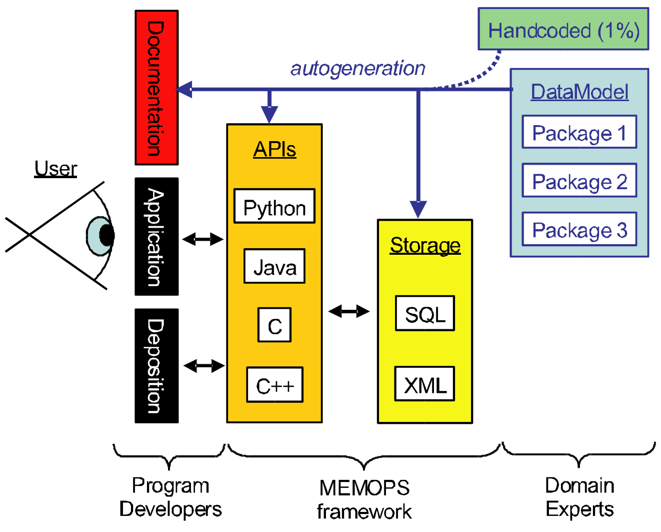

Fig. 2 Implementation of Memops code generation. Users interact with applications or deposition tools as before, while software developers use the APIs to interact with the underlying data. The actual data model is written by domain experts in a separate process with limited programming input. APIs and their implementations, storage format descriptions, I/O routines and documentation are all generated automatically from the UML data model, to the extent of over 99%. The APIs will remain stable over time even when the underlying data formats or data model change, thus insulating application programs from future changes.

3.2 Generated Libraries

Generated libraries include Python and Java API implementations, XML and SQL schemas, subroutines and mappings for I/O, and documentation. Most of these are essentially one-to-one mappings of the model. A class in the model will correspond to a Java or Python class, an XML element, or an SQL table. The same name, or an automatic derivation of it, is used throughout, to avoid the need for special mapping files. Given the nature of the platforms a one-to-one mapping is not, however, enough. XML requires extra elements for some attributes and links, relational databases require extra tables for many-to-many associations *etc.*, but in each case the extra code follows directly from the nature of the model without requiring (or allowing) extra input. There is of course an infinite number of ways of making *e.g.* Python API implementations or XML schemas that correspond to a given data model. The goal of MEMOPS is in each case to derive one useful implementation in a simple and fully automatic way, rather than to make the process customizable by the application programmer or data model developer.

4 The API implementation

The use of APIs (rather than data formats or models) as the invariant target for application programmers' efforts has a number of advantages for software integration and interoperability. APIs can be designed to be less tied to the precise detail of the underlying model than e.g. a parser would be, as they represent a higher level of abstraction. This allows the API to protect applications that use it from having to modify their code even as the data model changes. Additions to the model are especially easy to handle, since the addition of new functions to an API does not interfere with the existing ones. Changes in names, or in which data are stored and which are calculated on the fly are also relatively unproblematic, and it will frequently be possible for the API to accommodate even more fundamental changes in the structure of a data model.

4.1 General Architecture

For an application programmer the impact of using the Data Model is determined mainly by the APIs. The quality and ease of use of the API implementations is therefore extremely important. Memops API implementations are optimized for querying, for maintaining consistency in the presence of continuously changing data, and for supporting multiple projects with multiple users using different approaches and techniques. Automatic code generation in itself reduces the potential for bugs and guarantees a consistent style across the entire body of code. In addition, the APIs have been designed to include a wide range of functionality. Comprehensive validity checking is incorporated in all operations that modify data, to ensure that the data remain in a consistent and legal state. Data loading is done automatically, and the API keeps track of which data packages are modifiable, or have been modified and thus require saving.

The Memops APIs were designed as interfaces not to a specific XML file, but to a single, consistent representation of the data in a project. The prototype use case in structural biology research, where applications should be able to work directly off the generated API, accessing all relevant data, leaving the project accessible to any other conformant program, with information carried along towards an eventual deposition of the data. The emphasis on consistency checking, on persistent identifiers, and the decision not to use URL-based link mechanisms, arise from these considerations.

4.2 Notifiers

A notification facility is built into the API, to facilitate the building of graphical user interfaces (GUIs). The notifier registers a function to be called, with the relevant object as a parameter, when a given method is executed or when a given type of object is created, modified, or deleted. This can be a great simplification for GUI coding. By registering a notifier for *e.g.* creation and deletion of e.g. Molecule objects, a GUI could keep a list of all current molecules without having to change the code actually handling the molecule objects.

4.3 Storage management

The current API interacts with data stored in a mixture of XML files and local or remote databases. The price for this flexibility is that data must be loaded essentially one file at a time, which would be appropriate for situations where each project is accessed mainly by one person at a time. Data storage is by package, and each package may be stored in an XML file or database, locally or remotely. The Implementation package, which is loaded first, contains the storage locations for all other data. These are then loaded automatically by the API when the data they contain are needed. The API keeps track of which packages have been loaded and which have been modified (and should therefore be saved). Packages can also be marked as read-only, which will prevent attempts to modify the data they contain.

An alternative API implementation (currently in alpha test) provides concurrency, security and fine-grained control for simultaneous, multi-user access, transaction control, and roll-back, but this implementation depends on all the data being kept in a single database.

4.4 Derived Attributes

'Derived' attributes and roles follow the same syntax as real attributes and roles, but are in practice a convenient way of executing function calls. In a data model for person data, for instance, one could store each person separately, with links from children to their parents. A derived attribute 'mothersMaidenName' could then return the appropriate value without making it necessary to store the mother's maiden name in the model. If the model is changed so that an attribute is no longer stored explicitly, a derived attribute that mimics it can be added to avoid breaking existing code. Derived attributes and roles are especially useful since it is recommended that models be fully normalized, so that each piece of information is stored in only one place. If a piece of data is of interest in several places, derived attributes can make it available in all of them without duplication of the stored information.

4.5 Example - the Python API

The Python API consists of a Python class for each class in the model. Each class comes with a creation method (an __init__ in Python parlance), a delete method, and a checkValid method. Attributes and roles can be accessed and set using the normal Python 'object.attribute=value' syntax, but the code is organized using the Python 'properties' mechanism, so that these accesses are intercepted and passed to the relevant 'set' and 'get' methods. Access methods are generated from the model depending on the cardinality of the attribute/role. A single attribute, *e.g.* 'name', will give rise to methods 'getName' and 'setName', as will a single role. Multiple attributes will have two additional methods, so that you have *e.g.* getKeywords, setKeywords, addKeyword, and removeKeyword methods. Multiple roles will have a further three, *e.g.* findFirstAtom, findAllAtoms, and pickAtom; these methods select one or more atoms, either by filtering on their attribute and role values (the two 'find' methods) or by index (the 'pick' method).

Data are organized for fast retrieval rather than fast modification. Associations are stored at both ends, so that an employer knows his employees and an employee his employer, as it were. The API makes sure that the two ends of associations are kept consistent even if only one of them is explicitly modified, so that *employer.addEmployee(newEmployee)* and *newEmployee.setEmployer(employer)* will have the same effect. Validity checking code is built into all commands that modify attributes and roles, so that modifications that make the data illegal are prevented. Newly created objects are checked for validity after creation. The delete method works in a different way: If deleting object A makes object B invalid (*e.g.* because there was a mandatory link from B to A), object B will be deleted as well in a cascading delete.

5 Conclusions

5.1 Project Status

The Memops project has already matured sufficiently to prove that the approach works. The autogenerated Python API has been released, in the version based on XML data storage. It serves as the foundation for a couple of major scientific applications developed by CCPN, and is being interfaced with a number of other applications in the core area of CCPN, macromolecular NMR spectroscopy. The data model is being expanded into the area of (bio)chemical laboratory information management, and a Java API based on database storage is released in an alpha version. To illustrate the size of the project, the current model contains 318 classes, with 290 000 lines of code in the Python API implementation and 819 000 lines of HTML documentation.

5.2 Discussion

The decision to use a single platform-independent model as the basis for automatic code generation for several platforms has proved to work in practice, and has contributed greatly to the maintainability of projects using Memops. Of course it could be argued that the use of implementation-specific tagged values has confused the issue. The crucial factor, in our opinion, is that Memops is limited to generating data access layers, in a broad sense. This makes the problem sufficiently small and well-defined to allow the generation of efficient code from the platform-independent model with an efficiency of over 99%. It does not follow that a similar approach would be appropriate (or successful) in projects with a wider scope.

References

1. http://www.ccpn.ac.uk and references mentioned therein.
2. Akehurst D.H., and Patrascoiu, O.: Tooling Metamodels with Patterns and OCL. Proceedings of 'Metamodelling for MDA', York, UK, November 2003.
3. http://www.objectdomain.com