MEMOPS: Data modelling and automatic code generation.

Rasmus H. Fogh[1†], Wayne Boucher[1†], John M.C. Ionides[1†], Wim F. Vranken[2], Tim J. Stevens[1] and Ernest D. Laue[1*]

[1] Department of Biochemistry, University of Cambridge, 80 Tennis Court Road, Cambridge, CB2 1GA, UK

[2] PDBe group, EMBL-EBI, European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge, CB10 1SD, UK

[†]The first three authors contributed equally to the paper

Summary

In recent years the amount of biological data has exploded to the point where much useful information can only be extracted by complex computational analyses. Such analyses are greatly facilitated by metadata standards, both in terms of the ability to compare data originating from different sources, and in terms of exchanging data in standard forms, e.g. when running processes on a distributed computing infrastructure. However, standards thrive on stability whereas science tends to constantly move, with new methods being developed and old ones modified. Therefore maintaining both metadata standards, and all the code that is required to make them useful, is a non-trivial problem. Memops is a framework that uses an abstract definition of the metadata (described in UML) to generate internal data structures and subroutine libraries for data access (application programming interfaces - APIs - currently in Python, C and Java) and data storage (in XML files or databases). For the individual project these libraries obviate the need for writing code for input parsing, validity checking or output. Memops also ensures that the code is always internally consistent, massively reducing the need for code reorganisation. Across a scientific domain a Memops-supported data model makes it easier to support complex standards that can capture all the data produced in a scientific area, share them among all programs in a complex software pipeline, and carry them forward to deposition in an archive. The principles behind the Memops generation code will be presented, along with example applications in Nuclear Magnetic Resonance (NMR) spectroscopy and structural biology.

1 Introduction

In recent times, the combination of digitization, high-throughput approaches and modern computing techniques has revolutionized the relationship between scientists and data in terms of size and access. These advances present great opportunities but also create considerable problems. Most data now exists in electronic form at some point in its life, and it is therefore extremely important that data can be passed seamlessly between the many different programs that might be used to process and analyse it. If all scientific software was always written to some common data standard then this would not be difficult. In practice, however, this is a non-trivial problem. Science is primarily driven by the need to generate results rather than conform to standards, even if such standards existed and could be agreed upon in constantly evolving fields.

* To whom correspondence should be addressed. Email: e.d.laue@bioc.cam.ac.uk

The need for standards remains, however. As high throughput methodologies have proliferated, and networks have made it increasingly simple to move data to wherever it is needed, there has been increased interest in defining data standards across a large number of fields where there are immense amounts of data that need to be organised and exploited. Recent reviews by Brazma *et al*., [1] on data standards and by Swertz and Jansen [2] on software infrastructure give a good account of both current efforts and the underlying considerations. Pitfalls abound: Top-down imposed standards may be universally ignored, but just hoping that communities will self-organise can be a recipe for anarchy; new and developing fields are too fluid for standardisation, while mature fields are too settled to change, even for the better; minimal standards can leave too much out, while detailed standards may become too unwieldy to be adopted.

Data handling in practice starts with the individual program. Every program has a data model, if only implicitly, and every program needs to consider reading and validating input, organising internal data, and producing output. Much of the required work is trivial, yet input/output and data handling routines can make up a surprisingly large fraction of the code (and the bugs) for a program. Programmers thus have an incentive to choose the simplest possible data models in order to maintain an overview and minimize the workload. Standardising and automating the writing of data access code has the potential to save significant developer time, in addition to making the resulting code more clearly defined, and easier to maintain and to interface with. More generally, making it easier to maintain complex underlying models will free resources to support both interoperability and additional functionality.

There are other practical considerations when considering how models, and the scientific code that implements them, are developed and maintained in reality. Science constantly moves forwards, both in terms of knowledge and methodology, so it is unrealistic to expect data models that describe scientific areas to remain static. In addition data model development in an academic environment is normally carried out by very small teams - typically one or two people. It is therefore helpful to have a system that scales well in terms of model maintenance and that has good mechanisms for extension. These are areas where code generation can help enormously. Hand-written code tends to deteriorate over time as developers add quick patches following user requests etc. Eventually refactoring becomes essential, and this can be extremely time consuming and hard to fund for a large code base. Generated code is not as flexible but tends to remain uncluttered for longer. A practical solution to this trade-off is to provide hooks in the code generation machinery that allow for a limited amount of maintenance-intensive but flexible handwritten code.

Structural biology has a long track-record of data standardisation. The determination of atomic resolution structures is both complex and expensive, and it was appreciated very early on in the history of X-ray crystallography that global archives of structures would have considerable value. It was also appreciated that such archives would be much more useful if the data were represented systematically. Thus macromolecular X-ray crystallography was arguably the first biology-related community to embark on data standardisation. The Protein Data Bank formalised their internal procedures into the v2.3 PDB coordinate format [3] while a more extensive set of definitions were defined in mm-CIF [4]. At the same time as these standards for archiving were being developed, CCP4 [5] was coordinating the establishment of standards covering the intermediate stages of structure determination, defining software pipelines based on PDB and reflection files that contain data, and a standard command line

interface to run programs. CCPN, the project that has developed Memops, was created in part to emulate CCP4 in the NMR field.

In macromolecular NMR spectroscopy, the interpretation and analysis phase of a project is traditionally done by a single person interactively over weeks and months, and the total amount of data generated in this phase runs to tens of Mb for a single project. The raw data are NMR spectra – generally in the order of tens, each up to a few hundred Mb in size. Each spectrum is a simple numerical matrix, calculated once and for all, and stored in binary form in one of a few proprietary formats. The need for standardisation is not so much here, but in the more varied and complex data used for spectrum headers, experiment description, interpreted data, generated structures, and validation output.

Automatic code generation is a well established software technique [6] increasingly used as a time-saving way of generating variants of verbose and repetitive code. It is related to the Model Driven Architecture (MDA) concept promoted by the Object Management Group (OMG). In MDA an abstract model of the system under study is used as a starting point first for platform-specific models and ultimately for the finished software product. MDA does not require code generation *per se*, and the specific models and final code are often generated completely or partially by hand.

The Memops framework [7] (from **ME**ta-**MO**delling **P**rogramming **S**ystem) is designed to enable a small development team to build and maintain a large complement of code libraries. This is achieved by generating the necessary subroutine code directly and automatically from an abstract data model. Memops is an easy way of getting fully functional libraries handling data access, I/O, consistency and validity checking for several different languages and storage implementations in parallel. It was initially proposed as part of a project to make a data standard for macromolecular NMR spectroscopy data. Memops reduces the overhead of developing standards, particularly in cases where the model is developing rapidly, in a number of ways:

- The use of an abstract model with a diagrammatic representation (in UML) makes it easier to oversee the structure of the model and to discuss changes. This is important, as rigorous initial definition of a model can drastically improve its long term usefulness.
- The automatic generation of data access, I/O, and validation code provides high quality libraries, allowing developers to concentrate on application development rather than housekeeping code.
- The model change process is extremely fast. This makes it easier to modify a model, which can be extremely useful both when fine-tuning the model, and when extending it to include new features. The data compatibility system ensures that old format data can be read by new versions, and for many changes (additions, deletions, renamings) compatibility code is generated automatically.
- In some cases it is possible to hide model changes, or complexities, from programs that are unaware of them. An attribute can be replaced transparently by a function call (a 'derived attribute'), which behaves as if the attribute was stored in the normal way. This feature can be used to present data in a simple manner even where the structure of the data model is actually more complicated.

The resulting data access subroutines provide efficient means for integrating software either sequentially into pipelines or through concurrent access, both due to the ability to maintain complex models and because of the nature of the data storage implementation. All related data are kept in a single network of interlinked objects with individual access to each object and precisely defined relations between data items. Programs can navigate the network starting from a root object, accessing only the data they need while maintaining consistent links to other data that are not relevant in this context. For instance a LIMS application, X-ray crystallography software, and NMR software could work off a single data set, keeping all the information consistent, without any need for e.g. the crystallography software to be aware of the NMR data structures. The facility to connect application-specific data to each data object further allows program-specific information to be stored and kept through a pipeline. Software integration and ultimately data quality are also improved by the precise and comprehensive nature of the data model. The combination of standard access code, precise definitions, model constraints, and built-in validation leaves much less scope for ambiguity. In writing the CcpNmr FormatConverter, for instance, it was found that the most difficult step was disambiguating and connecting up the information being read in from external formats. Once the information was inside the data standard and thus well defined, exporting to other programs was invariably straightforward. By contrast, even something as simple and well established as the PDB coordinate file format was in practice often used in ways that were incompatible between different programs, or that did not respect the specification.

This paper presents the version 2.0 of the Memops architecture, and evaluates its impact over a number of applications. A more detailed view of the implementation can be found in the Supplementary Material.

2 Development of standards in the NMR community

In order to describe how Memops has been used in practice, and to highlight the strengths and weaknesses of the approach, it is helpful to look at the development of the associated standard in the NMR community and how it tackles issues that had long proven difficult to solve.

Software for macromolecular NMR is dominated by isolated programs with little provision for integration or the forming of software pipelines. The effectiveness of NMR studies is often compromised by the difficulties of transferring data between programs. The answer to this problem lies in some form of data exchange standard. However, the precise nature of this standard needs to take into account the sheer complexity of the data under consideration, and the fact that NMR experimental methods are constantly evolving so that any standard has to be future-proofed. It must also take into account the relatively low level of resources available for not purely scientific objectives like ease of use or adherence to external standards. Memops was developed to address these problems by providing a framework for integrating existing NMR software through a highly complex, readily extensible standard that can reasonably be developed and maintained by a small team.

Data standard: Organisation. The goal of a data exchange standard is interoperability between programs from different origins, within a software pipeline that *generates* data. The set of exchanged data must be so comprehensive, detailed, and consistent that it can be used directly as input for calculation. An exchange standard must work with programs that use different approaches and architecture, and must maintain consistency for continuously changing data without relying on data curation. It has to be possible to validate data files electronically against the standard, to ensure compliance in a heterogeneous environment. Any standard that satisfies these requirements must of necessity be large and complex. Yet the

fragmented state of macromolecular NMR software, which is what makes a data exchange standard desirable in the first place, means that the standard would have to be adopted by a large number of independent, under-resourced groups using different programming languages and storage implementations. We felt that our goals could only be achieved if storing data in the standard became an integral part of each application. We therefore needed to make the standard as attractive and useful for application programmers as possible.

To satisfy these goals it was decided:

- to target the standard at application programmers, who are presumably more tolerant of complex models than research biologists
- to establish a standard object-oriented Application Programming Interface (API) rather than a standard format
- to describe the standard in an *Abstract Data Model* and ensure that the API(s) exactly reflected the model
- to provide working API implementations for several languages, and for data storage in at least XML formatted files and SQL relational databases
- to include complete validation of all model rules and constraints in the API implementations
- to make data access transparent and independent of the underlying storage mechanism
- to provide utility code and features, such as event notification, backwards compatibility support, and distribution of reference data

Data standard: Content. The actual content of the data standard for macromolecular NMR was determined by CCPN over a series of community workshops where a wide range of stakeholders were consulted [7,8,9]. The resulting standard was built on the work of the BioMagResBank on NMR data [10] and the PDB on macromolecular topology [3]. It covers NMR spectroscopy and structure generation; macromolecules (topology, structure, coordinates, and simulation); laboratory information management; and utility data (citations, people etc.). The structure of the data model is illustrated in Supplementary Figures 4 and 5. For further information on the *contents* of the model see the CCPN web site and the API documentation found there. Note that in the particular context of macromolecular NMR it was critical that the standard was extremely detailed; it had to be able to be mapped onto by a large number of subtly different implicit data models within the various NMR software packages, and accommodate extensions as the science developed. However, the Memops approach can be applied to any field, and to far simpler data models.

Code generation in the context of NMR is made more complex by the fact that external software developers require implementations for a number of different computing languages. This means that the generation machinery has to be capable of maintaining and synchronising several API implementations in tandem. The need for backwards compatibility at the application level further complicates matters. If each implementation was written and maintained by hand, coding and synchronisation testing would require resources far beyond what could realistically be provided. As the number of implementations increased, even providing separate code generators for each implementation would become a demanding task; ideally the system should also make it possible to add new implementations with a minimum of additional work.

Fortunately code for data access (getting and setting values, creating and persisting objects) is highly repetitive and can be easily deduced from the data model. This makes it possible to replace handwritten API implementations by a combination of generic code and model lookup. The advantages are clear: There is less code to write and maintain; the generic code does not depend on the model and so can be debugged once and for all; and synchronisation between API implementations and data model is guaranteed.

3 The Memops Machinery

The initial development of the Memops code generation framework has been described previously [7]. The generation process is summarized in Figure 1. Details can be found in the 'Memops Machinery' section of the Supplementary Material. Memops currently supports parallel generation of Python+XML, Java+XML, Java+database, and C+XML implementations, all from a single model. The source for all generated code is the data model that describes the structure of the data to be stored, including constraints on which data values are allowed. The machinery can work with any model that follows the Memops modelling rules, whatever the underlying subject matter.

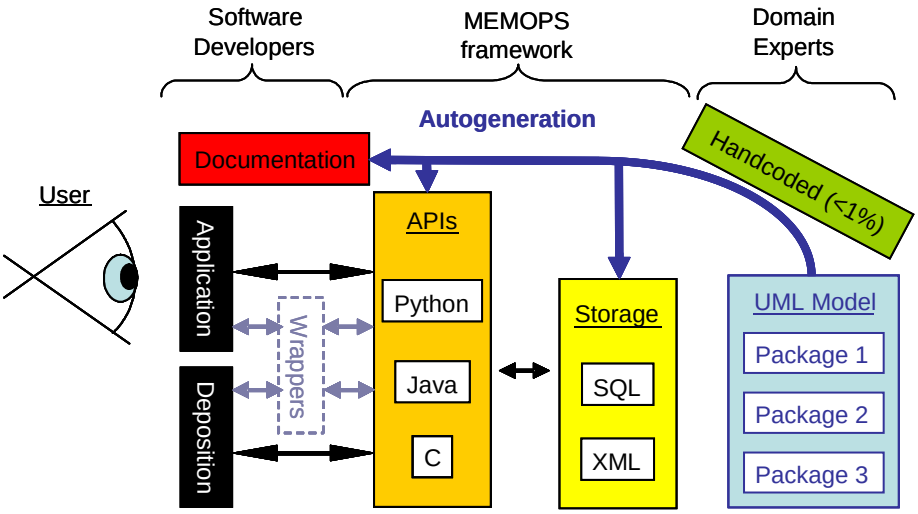


Figure 1: Organisation of the Memops framework. The data model, edited in UML [11] is split into packages, so that different domain experts can curate different areas, and so that applications can choose to work with only a subset of the model. Packages have one-way 'import' links to other packages that they depend on. XML and database schemas, persistence code, and the actual API implementations are generated automatically from the model. Over 99% of the final code is generated automatically. The remainder, including code for complex model constraints and non-standard functions, is added as language specific code snippets to the UML, so that the generated code is fully functional without manual modifications. Applications, written by third parties, ideally do all their data access by direct API calls.

Memops produces object-oriented code. Information is organised as a set of linked data structures (objects) with attributes, using methods to get, set, and modify values and links. For the C implementation object orientation is not possible and so is emulated using structures and complex function definitions. The model can be enriched by adding 'derived attributes' that behave like stored attributes but are calculated from the stored information at runtime – together with the function code needed to derive them. Data are fetched from disk automatically when needed, one data package at a time. The implementations include full validation of new data against all model constraints and protection of internal data structures from casual modification ('encapsulation'). A model event notifier (callback) facility has been integrated with the API. Any application can register a function to be called when a given API

function is executed or a given type of object is modified. This allows e.g. graphical user interfaces to keep themselves up-to-date as the model data are modified.

Our first prototype was written as generic code that interrogated the model description at run-time (an 'interpreter'). However, this approach proved to be slow, complex to program and debug, and hard to extend from the original Python to a statically typed language like Java. For subsequent implementations we opted for a generator that interrogates the model at code generation time and writes the run-time code for each API function out in full (a 'compiler'). This approach provides faster execution times and easier debugging, at the cost of having a large and repetitive body of code in the API. For simplicity we concentrated on generating a single, efficient subroutine library for each language+storage combination, albeit with some possibility for fine-tuning the database structure in the database implementation.

The degree of synchronization we need requires essentially 100% automation, as any manual post-generation modifications to the code would have to be repeated endlessly every time the generation script was re-run. When the project started no existing framework could provide for this need. Frameworks like Hibernate can now produce a functional data access API implementation for Java+database automatically from a single input file. Memops is unique in providing several data access implementations in parallel from a single model specification. In addition Memops makes it relatively simple to add generators for new implementations.

Additional features. Since the first publication [7], the data model and generation machinery have been extensively tested and optimised through practical use. Key changes to the internals of Memops have been made at two levels. At the level of the code generation machinery itself, the Python code has been refactored to make it simpler to add support for additional languages (see Supplementary Figures 2 and 3). At the level of the generated APIs, backwards compatibility code for data files from older model versions is now integrated with the Python+XML API. Most backwards compatibility can be handled automatically as part of XML file loading. Large model changes may be beyond the capabilities of this mechanism, but these can be handled by a more complex installation. For example, changing data from version 1 to 2 of the CCPN framework is possible through a web service.

A Java API over a database persistence layer has also been developed. In outline, Hibernate is used to map between the database and the Java layer. Java code that is compatible with the standard XML I/O routines is generated along with a Hibernate mapping file that is also used to generate the database schema using standard Hibernate tools. For additional customisation, hooks within the Memops generation machinery can be used to define additional Hibernate mappings and database triggers.

Further changes include the new C+XML API, complex object types that compare by value, globally unique identifiers (GUID), simplifying the core model package, and various optimisations.

Scalability. Memops is most effective where the handled data are complex, but not particularly large. This is precisely the situation in the NMR field, where a typical project may contain 2 million objects covering 300 classes and taking up 70 Mb when stored as XML files. Loading such a project with the Python+XML implementation takes roughly 30 seconds on a modern Linux PC with 2.16 GHz dual-core processor, and requires 600 Mb of memory when fully loaded. However, these load times are somewhat misleading as individual data files are only loaded when required, massively reducing the dwell times in actual application. The load times for the Java+database implementation depend largely on the behaviour of Hibernate. Currently searching starts to slow down noticeably for tables above 10^6 rows,

where performance becomes limited by of the overhead of creating proxy objects within the Hibernate layer. However, by judicious use of customisation, and using the Hibernate layer to query SQL directly when most appropriate it is possible to work with larger data sets.

Availability. The Memops framework is available under the GPL license, the data exchange standard with generated subroutine libraries under the LGPL license. Both may be found wither on the CCPN web site, or at SourceForge.

4 Applications

Memops provides application developers with a set of APIs consistent with a formal data model and with a high level of built-in housekeeping functionality. In the context of macromolecular NMR, this makes it relatively easy for programs to communicate with each other, either directly through the standard API I/O functionality generated by Memops, or through wrappers. This approach has proven extremely successful in developing new applications and addressing the data exchange problems that plagued the NMR community. Some examples are described below:

The CcpNmr suite was developed as part of the CCPN project. It was based entirely on the CCPN API from the start, using the Python+XML implementation (Supplementary Figure 6, top). The CcpNmr programs served as pilot applications for the NMR and molecular parts of the data model.

CcpNmr Analysis is a program for visualisation and assignment/analysis of macromolecular NMR data, and is often used to set up structure generation. The program includes a rich set of features supporting various NMR tasks, and has several hundred users worldwide. Version 2.1.3, based on the most recent Memops APIs has recently been released. Macromolecular NMR spectrum analysis is arguably a matter of viewing spectrum contour plots and filling information into a highly complex data model. The demands on the user interface are high, as a single project may require many weeks of continuous interaction with the program. Data access in CcpNmr Analysis is carried out through the Memops-generated API (except for the large numeric matrices of the NMR spectra), with some limited transfer to internal data structures for speed. The graphical user interface (GUI) is built on the Memops notifier facility, and relies on the internal checking in the API implementation for enforcing data consistency. CcpNmr Analysis needs to store both user profiles and session information like window positions, current colour settings etc. - neither of which has any place in a general data exchange standard. To this end CCPN has added two Analysis-specific packages to the model. Using the Memops machinery also for this purpose was faster than writing file I/O by hand and had the advantage that all data were available through a single interface.

Overall, using Memops aided development in three main ways. Firstly, the complexity allowed by Memops made it possible to develop the data model to represent the underlying science correctly from the start. Simplifying assumptions were confined to higher level code, and so could be changed with relative ease as users needed ever more complicated cases to be taken into account. Secondly, the time required to extend the model is extremely fast. Once the desired semantics have been confirmed, the code update can take as little as half a day, even for a complex extension. Where problems were found in the model – Analysis had its own bespoke model packages and was co-developed with central parts of the data model – this again sped up the development of new features considerably. Thirdly, the generated code

is completely systematic, which means that developers can easily and accurately understand what each function does without constantly having to check documentation. The users of Analysis are mainly academics, many of whom require new features to test out ideas, and using Memops means that feature requests can be handled quickly and accurately.

CcpNmr FormatConverter is a universal format converter for NMR and structural biology that can read data from and write data to over 30 different current and legacy data formats. It works by first converting all input data to CCPN format, querying the user where necessary for disambiguating the data, and then re-exporting to the target format. It is a single application with its own format parsers and writers. The architecture exploits the fact that the CCPN format is precisely defined, highly detailed, and able to store all relevant data. Ambiguities can be resolved at the import stage, simplifying the subsequent export. The use of a central data definition format means that you can convert between n formats with 2n (rather than n**2) converters.

A key point here is that for FormatConverter to work, it is essential that the underlying data model is complex. Indeed, each piece of existing software has its own, often implicit, underlying data model, and the role of FormatConverter is to resolve these assumptions in the context of an all-encompassing data model that can handle all the individual cases. The ability to convert between formats efficiently is a key step for both meta-analysis, and for data exchange / pipelining. FormatConverter is widely used in the field of macromolecular NMR and in the eNMR project, in particular the CASD-NMR structure calculation competition [12] (see Supplementary Figure 8). The ability to consistently store large amounts of information from external data files is especially powerful; the FormatConverter and CCPN framework are used for data curation at the NMR deposition database BioMagResBank [13,14], and have provided data for the RECOORD structure recalculation project [15]. They are essential tools in new data organisation and analysis efforts [16,17].

Extend-NMR is a software development and integration project for macromolecular NMR spectroscopy funded under the EU FP6 program. The integration aspect involves combining scientific software from eight different developers into a single integrated pipeline, encompassing NMR data acquisition, processing, analysis, structure generation, docking, validation, and deposition (see Supplementary Figure 7). The pipeline includes a shared GUI that can launch all the different applications. It is an example of using the API implementations generated by Memops for data exchange, and involves pre-existing applications with their own separate code base. The end-of-project integrated pipeline has just been released. The integrated programs include:

TOPSPIN (Bruker BioSpin GmbH). Bruker is a major equipment manufacturer for NMR spectrometers, and TOPSPIN is the Bruker software for data acquisition, processing, and analysis of NMR data. The current TOPSPIN release (v2.1) can export NMR spectrum and peak data to a CCPN project, and the upcoming release (v3.0) will have increased export capabilities. TOPSPIN is written in Java, and the data are exported to a CCPN data structure in memory by direct calls to the CCPN Java+XML API implementation.

ARIA [18] is one of the most popular programs for generating macromolecular structures from NMR data. It reads a molecular sequence, NMR shifts, peaks and structural constraints, and runs an iterative calculation to generate an ensemble of structures and a filtered version of the

input data. ARIA versions from v2.2 (2007) support the complete input and export of data and analyses through either ARIA files or a CCPN data structure. ARIA is a Python program with associated CNS scripts, calling the CNS structure calculation engine [19, 20]. It uses its own data structures internally, and transfers data to and from CCPN by calling the Python+XML API implementation.

HADDOCK [21,22] is an information-driven macromolecular docking program. It was developed from an earlier version of ARIA (see above) and uses the same architecture. Haddock and CCPN have jointly developed a Haddock-specific model package to hold both input data and Haddock switches and parameters, so that all information needed for launching Haddock can be held in a CCPN project. This model is used by the CCPN Analysis software to create all necessary input files to launch HADDOCK directly or, alternatively, to generate a single HADDOCK server parameter file that can be uploaded to the HADDOCK web server for a fully automatic docking run.

CING is a validation suite for NMR-derived structures, encompassing and expanding on a number of pre-existing validation tools. The top layers of CING are written in Python, and data transfer to and from the CCPN data standard is through direct API calls. A data model package for structure validation data – not specific to CING – has been developed in collaboration between the CCPN and CING teams.

The successful completion of the Extend-NMR project relied on the all-encompassing, detailed nature of the data model and the facility for tailoring additions to specific programs. The availability of implementations in different programming languages made it possible to distribute the work required to integrate the various components of the pipeline to the individual software development teams, each with their specific language expertise.

EUROCarbDB aims to create a distributed deposition database for glycobiology and glycomics data (currently NMR, MS and HPLC), with associated bioinformatics tools. The project includes an atomic-level molecular description framework for carbohydrate fragments, developed in collaboration with CCPN. The NMR component of EUROCarbDB consists of a Memops Java+database API for the NMR and molecular description packages of the standard CCPN data model (Supplementary Figure 6, bottom). The database with the NMR data is merged into the core EUROCarb database by merging the CCPN and EUROCarbDB Hibernate mapping files. Additional links connect the Memops root object to the central EUROCarb tables. Information is mainly uploaded one CCPN project at a time, with some facility for editing the deposited data. Data extraction is mostly through Java API calls.

The Java+database API was tested in EUROCarbDB with over 1300 data sets loaded. Some data-intensive or time-critical operations required special-purpose queries in Hibernate Query Language (HQL). These can be customized at individual sites. Where required, hooks within the Memops generation machinery were used to generate and maintain denormalised tables in the database. These tables allowed rapid searching across key areas of the database without needing to load large amounts of data through the Hibernate layer, and served to identify objects of interest for later retrieval.

5 Conclusion

Code generation techniques are a valuable tool for speeding up development and maintaining the integrity of code. As with any structured system, there is a trade off between flexibility and efficiency. In Memops this manifests itself in a number of ways. Firstly, it is relatively difficult to "hack" short term changes into the generated APIs. In terms of the long-term development of the code this should actually be seen as an advantage, as multiple layers of such changes inevitably lead to bugs, and eventually the need for refactoring. In other words, it enforces a systematic approach that may seem inconvenient in the short term, but is beneficial in the long term. Secondly, Memops makes it easier to maintain data models that accurately describe complex relationships and include difficult and unusual cases. The complexity is no longer limited by your ability to maintain oversight of the model. The main problem becomes the need for handling common situations simply while still including all relevant subtleties, and even this can be alleviated by the use of derived attributes. The case studies above demonstrate that being able to support a complex data model can be a critical feature in data exchange and meta-analyses where the underlying data are fundamentally complex, as is often the case in the life sciences. As web services and the GRID evolve, and networks permit the movement of large datasets, questions of interoperability are expected to become increasingly important. In the case of NMR there would be considerable advantages to developing the Extend-NMR pipeline further so that arbitrary workflows could be defined and the individual steps farmed out to remote services.

Memops implements a general approach for the generation of housekeeping code from an abstract data model, and as such is potentially applicable to a wide range of domains. However, it is anticipated that it would be most powerful in cases where the underlying data are described by a complex data model which needs to be maintained by a small, highly trained development team – a very common scenario in the life sciences.

Acknowledgements

We thank the BBSRC for funding the CCPN project since its inception, and the EU for additional support through the 'Quality of Life and Management of Living Resources' program (contracts QLRI-CT-2001-00015 'TEMBLOR' and QLK2-CT-2000-01313 'NMRQUAL') and the FP6 program (STREP contract 18988 'EXTEND-NMR'). We also thank Astra-Zeneca, Genentech, Dupont Pharma (now Bristol-Myers-Squibb) and GlaxoSmithKline for supporting the CCPN project.

We thank the following collaborators for permission to present results on their software: Bruker Biospin GmbH (TOPSPIN), Michael Nilges (ARIA), Alexandre Bonvin (HADDOCK), Geerten Vuister (CING), and Bas Leeflang (EUROCarbDB; FP6 contract RIDS-CT-2004-011952).

References

[1] A. Brazma, M. Krestyaninova, and U. Sarkans. Standards for systems biology. *Nature Reviews Genetics*, 7:593–605, 2006.

[2] M. A. Swertz, and R. C. Jansen. Opinion: Beyond standardization: dynamic software infrastructures for systems biology *Nature Reviews Genetics,* 8:235-243, 2007. (doi:10.1038/nrg2048)

[3] J. Callaway, M. Cummings, B. Deroski, P. Esposito, A. Forman, P. Langdon, M. Libeson, J. McCarthy, J. Sikora, D. Xue, E. Abola, F. Bernstein, N. Manning, R. Shea, D. Stampf, and J. Sussman. *Protein Data Bank Contents Guide: Atomic coordinate entry format description.* Brookhaven National Laboratory, 1996.

[4] P. E. Bourne, H. M. Berman, K. Watenpaugh, J. D. Westbrook, and P. M. D. Fitzgerald. Macromolecular crystallographic information file. *Methods Enzymol.*, 277:571-590, 1997.

[5] COLLABORATIVE COMPUTATIONAL PROJECT, NUMBER 4. The CCP4 Suite: Programs for Protein Crystallography. *Acta Cryst.*, D50:760-763, 1994.

[6] J. Herrington. *Code Generation in Action.* Greenwich, Connecticut: Manning Publications, 2003.

[7] R. H. Fogh, W. Boucher, W. F. Vranken, A. Pajon, T. J. Stevens, T. N. Bhat, J. Westbrook, J. M. C. Ionides, and E. D. Laue. A framework for scientific data modeling and automated software development *Bioinformatics,* 21(8):1678-1684, 2005. (doi:10.1093/bioinformatics/bti234).

[8] R. H. Fogh, J. M. C. Ionides, E. Ulrich, W. Boucher, W. F. Vranken, J. P. Linge, M. Habeck, W. Rieping, T. N. Bhat, J. Westbrook, K. Henrick, G. Gilliland, H. Berman, J. Thornton. M. Nilges, J. Markley, and E. D. Laue. The CCPN project: an interim report on a data model for the NMR community. *Nat. Struct. Biol.*, **9**:416–418, 2002.

[9] W. F. Vranken W. Boucher T. J. Stevens, R. H. Fogh, A. Pajon, M. Llinas, E. L. Ulrich, J. L. Markley, J. Ionides, and E. D. Laue. The CCPN data model for NMR spectroscopy: development of a software pipeline. *Proteins,* 59(4):687-96, 2005.

[10] E. L. Ulrich, H. Akutsu, J. F. Doreleijers, Y. Harano, Y. E. Ioannidis, J. Lin, M. Livny, S. Mading, D. Maziuk, Z. Miller, E. Nakatani, C. F. Schulte, D. E. Tolmie, R. K. Wenger, H. Yao, and J. L. Markley. BioMagResBank. *Nucleic Acids Research,* 36:D402-D408, 2007. (doi: 10.1093/nar/gkm957).

[11] J. Rumbaugh, G. Booch, I. Jacobson. *Unified Modeling Language Reference Manual*. Amsterdam: Addison Wesley Longman 1999.

[12] A. Rosato, A. Bagaria, D. Baker, B. Bardiaux, A. Cavalli, J. F. Doreleijers, A. Giachetti, P. Guerry, P. Güntert, T. Herrmann, Y. J. Huang, H. R. A. Jonker, B. Mao, T. E. Malliavin, G. T. Montelione, M. Nilges, S. Raman, G. van der Schot, W. F. Vranken, G. W. Vuister and A. M. J. J. Bonvin. CASD-NMR: Critical Assessment of Automated Structure Determination by NMR. *Nature Methods,* 6:625-626, 2009.

[13] J. F. Doreleijers, A. J. Nederveen, W. Vranken, J. Lin, A. M. Bonvin, R. Kaptein J. L. Markley and E. L. Ulrich. BioMagResBank databases DOCR and FRED containing converted and filtered sets of experimental NMR restraints and coordinates from over 500 protein PDB structures. *J Biomol NMR,* 32(1)**:**1-12, 2005.

[14] J. F. Doreleijers, W. F. Vranken, C. Schulte, J. Lin, J. Wedell, C. J. Penkett, G. W. Vuister, G. Vriend, J. L. Markley and E. L. Ulrich. The NMR Restraints Grid (NRG) at BMRB for 5,266 protein and nucleic acid PDB entries. *J. Biomol. NMR,* 45:389-396, 2009.

[15] A. J. Nederveen, J. F. Doreleijers, W. Vranken, Z. Miller, C. A. E. M. Spronk, S. B. Nabuurs, P. Guntert, M. Livny, J. L. Markley, M. Nilges, E. L. Ulrich, R. Kaptein, and A. M. J. J. Bonvin. RECOORD: A Recalculated Coordinate Database of 500 Proteins from the PDB Using Restraints from the BioMagResBank. *Proteins,* 59:662–672, 2005.

[16] W. F. Vranken. A global analysis of NMR distance constraints from the PDB. *J. Biomol. NMR*, 39:303-314, 2007.

[17] W. F. Vranken, W. Rieping. Relationship between chemical shift value and accessible surface area for all amino acid atoms. *BMC Struc. Biol.,* 9:20, 2009.

[18] W. Rieping, M. Habeck, B. Bardiaux, A. Bernard, T. E. Malliavin, and M. Nilges. ARIA2: automated NOE assignment and data integration in NMR structure calculation. *Bioinformatics,* 23:381-382, 2007.

[19] A. T. Brunger, P. D. Adams, G. M. Clore, P. Gros, R. W. Grosse-Kunstleve, J. -S. Jiang, J. Kuszewski, M. Nilges, N. S. Pannu, R. J. Read, L. M. Rice, T. Simonson, and G. L. Warren. Crystallography & NMR System (CNS), A new software suite for macromolecular structure determination. *Acta Cryst.,* D54:905-921, 1998.

[20] A. T. Brunger. Version 1.2 of the Crystallography and NMR System. *Nature Protocols,* **2**:2728-2733, 2007.

[21] C. Dominguez, R. Boelens, and A. M. J. J. Bonvin. HADDOCK: a protein-protein docking approach based on biochemical and/or biophysical information. *J. Am. Chem. Soc.,* 125:1731-1737, 2003.

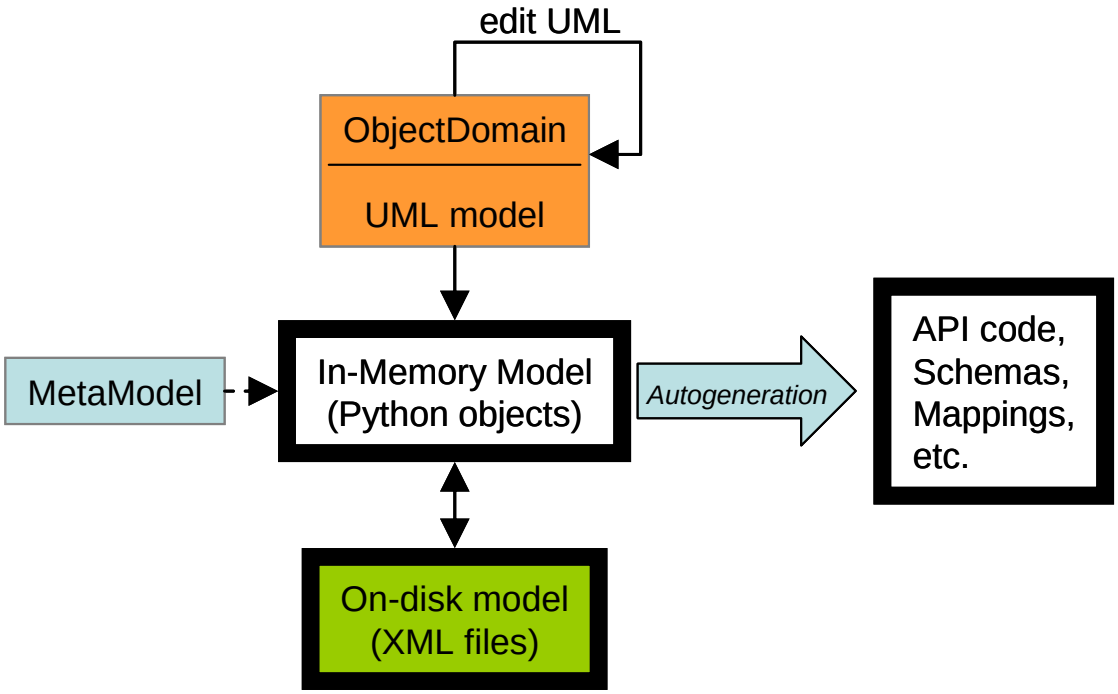
[22] S. J. de Vries, A. D. J. van Dijk, M. Krzeminski, M. van Dijk, A. Thureau, V. Hsu, T. Wassenaar, and A. M. J. J. Bonvin. HADDOCK versus HADDOCK: New features and performance of HADDOCK2.0 on the CAPRI targets. *Proteins,* **69**:726-733, 2007.

Supplementary Material: MEMOPS: Data modelling and automatic code generation

Rasmus H. Fogh, Wayne Boucher, John M.C. Ionides, Wim F. Vranken, Tim J. Stevens, and Ernest D. Laue

1 Memops machinery

The code generation process is shown in Figure 1.

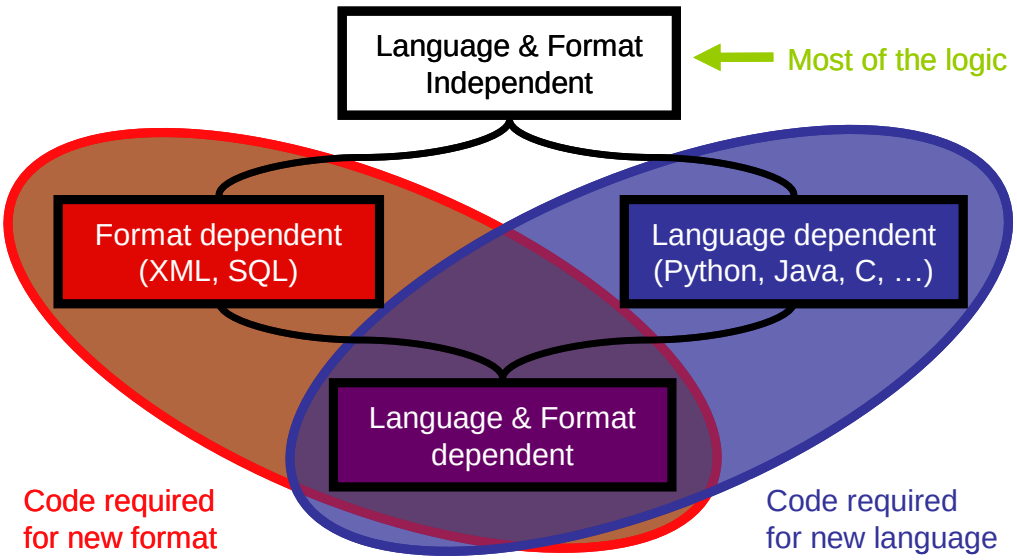


Supplementary Figure 1: Workflow for code generation. CCPN software is coloured in blue, while external software is coloured in orange. Boxes with thick black borders are files generated by the CCPN code.

The data model is edited in UML class diagrams augmented with tagged values. We are currently using ObjectDomain, a commercial editor, but in principle any UML editor could be used. The model description is transferred to an in-memory representation, specified as Python objects using the Memops metamodel (see below), and stored in a Memops XML format. Exporting the model description from the UML editor is the only part of the workflow that depends on the editor - all subsequent code generation steps work solely from the in-memory model description. The Memops metamodel is similar to the MOF 1.4 standard of the OMG.

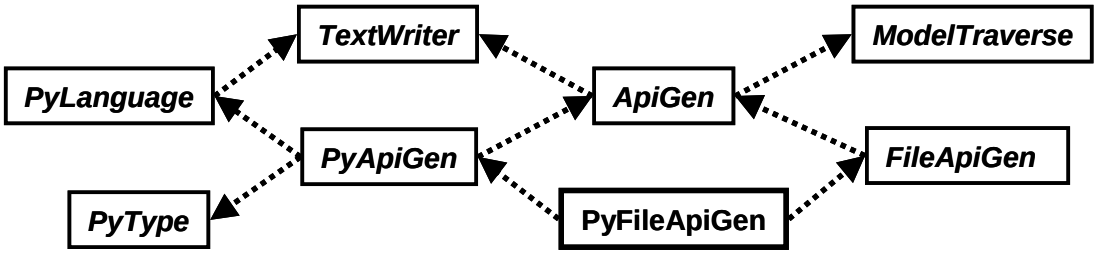
The abstract data model is language- and implementation- neutral. Language-specific information is stored as sets of tagged values, one for each language (and each storage implementation, if necessary). It consists mainly of code snippets for the body of manually coded methods or constraints, for integration into the code by the API generation machinery.

The Memops framework is written entirely in Python. We have found Python a good choice: flexible, object oriented, and well suited to the complex model queries and inheritance hierarchies of our generation scripts. First the model is read from XML and passed through an adaptation stage. Standard operations (get, set, etc.) and their parameters are added to the model at this stage, relieving the UML modellers of the work. The model adaptation also customizes the model to the specific implementation, *e.g.* by adding features that are specific for individual languages like Java.



Supplementary Figure 2: Organisation of API generator code. Many tasks are independent of the language or storage implementation being generated. By separating out the code, common actions need only be coded once, and new implementations can be added with minimum effort.

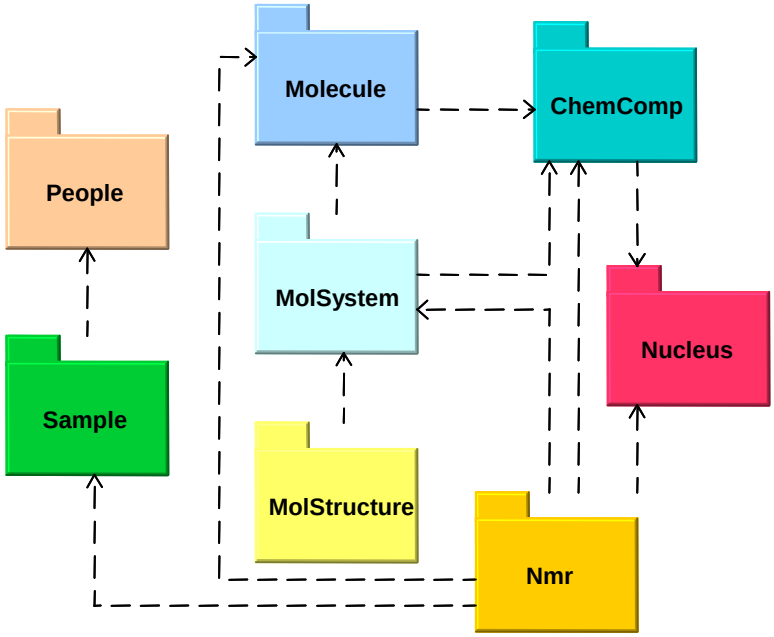
The adapted model is then passed to the code generator scripts, the main ones being those that generate the API and the XML I/O mappings. To simplify the writing and maintaining of generators, we have split the generation code into reusable modules. Figure 2 shows the organization, while Figure 3 shows the modules used in the Python+XML API generation. The logic for API generation resides mainly in ApiGen. Tasks that differ between implementations are delegated to functions that are called from ApiGen but defined lower down in the inheritance hierarchy. Particularly complex output code is written directly, e.g. from PyFileApiGen, but most code is produced by calling functions from PyLanguage, JavaLanguage etc. The level of organization should be evident from some sample function names: *setVar, newCollection, startLoop, endIf, callFunc, comparison, stringIsNotEmpty*, ... Formally the set of functions in the Language modules define a programming language, which the generators use to define the APIs. A more general solution would have been to implement automatic translation from an existing language like OCL. We settled for an internal *ad-hoc* solution because 1) it was easier to implement, 2) we could limit ourselves to constructs that occurred with some frequency in our APIs, 3) we could add special-case functions to optimize the generated code, like *e.g. reraiseException* or *collectionNotNoneAndNotEmpty*.



Supplementary Figure 3: API generator modules, specifically for the Python+XML API implementation. Each box represents a class. Inheritance is shown by dotted arrows from subtype to supertype – note that Python has no problems with multiple inheritance. The diagram for a Java+database implementation would have the same structure, with PyLanguage replaced by JavaLanguage, PyFileApiGen by JavaDbApiGen, etc. All classes are abstract, except for PyFileApiGen. The four 'ApiGen' classes contain the API generation logic – with ApiGen accounting for about half the code and FileApiGen for a third. ModelTraverse contains general looping and traversing code, while TextWriter contains code for writing to file. PyLanguage and PyType hold low-level language-specific code.

2 API Implementations

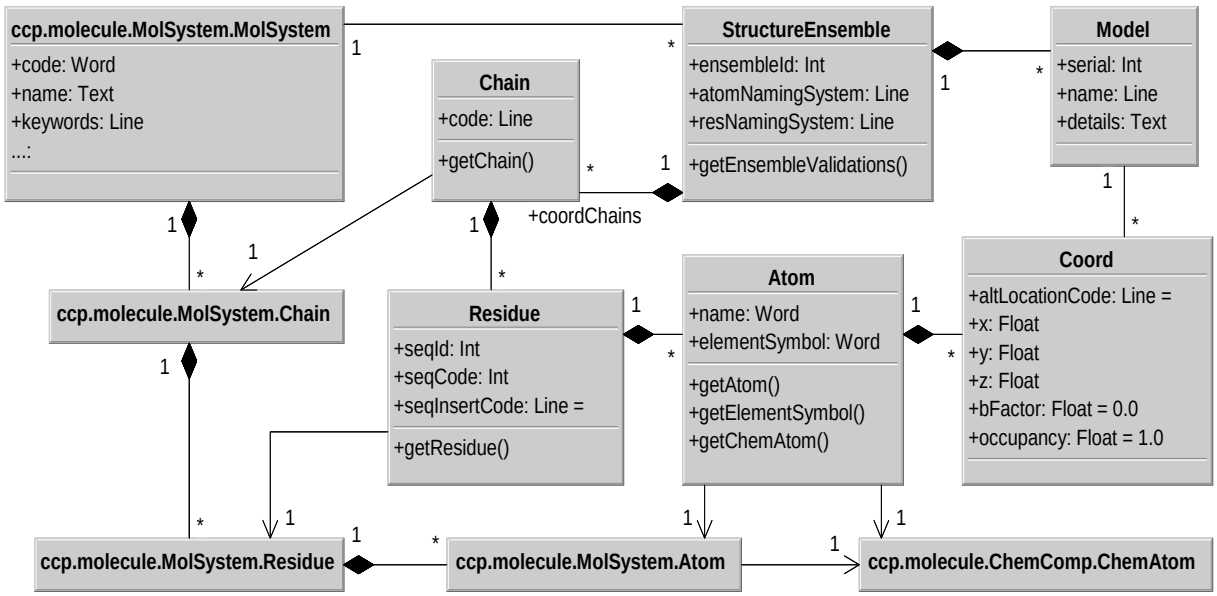
Data structure. The model consists of classes of DataObjects with attributes, links, operations, and constraints, DataTypes, and Complex Data Types. These are grouped in packages (see Figure 4). The model description is organised using the Memops metamodel (see below).



Supplementary Figure 4: Examples of packages from the CCPN data standard. Packages are used to partition both the model description, the API implementation code, and the data storage for file implementations. MolStructure contains coordinates, MolSystem atomic level description of molecular complexes, Molecule sequences, and ChemComp residue templates. Dotted arrows show dependency relationships; *e.g.* MolStructure depends on MolSystem and on ChemComp (not shown).

At run-time data are held in memory as linked objects with simple attributes or links (henceforth 'attributes'). The attributes may be single values or collections of various cardinalities, modifiable or frozen. Collections may be unique (sets), ordered (lists) or both (unique lists). The model may contain derived attributes, whose values are not stored but

(re)calculated when queried. Most objects are DataObjects, which are mutable, compare by identity and can appear in only one context in the model. As an alternative Complex Data Type objects are immutable, have simple attributes but no links, and compare by value. Complex Data Types are used for structured values that can appear in different contexts, such as Urls or orientation matrices. Finally, Constraints, which can be attached to attributes, classes, and simple or complex data types, are used to restrict which data values are allowed by the model. Constraints are evaluated at the beginning of attribute modifiers, at the end of object creation, and by the special checkValid function. Constraints are added as code snippets in all supported languages, and can be arbitrarily complex.



Supplementary Figure 5: Example package – 'MolStructure', used for storing molecular coordinate ensembles, with associated classes from other packages (names beginning with 'ccp.'). Classes are connected by 'parent-child' links (composition links - black diamonds). Each package has a TopObject class (here the StructureEnsemble) that is a child of the MemopsRoot (not shown). Links between classes are shown as dark lines – role names are derived from the names of the classes, except where explicitly overridden (e.g. 'coordChains'). One-way links are shown as arrows. Most methods are generated automatically. The few explicit operations in the diagram, like the Chain.getChain function, have non-standard code and are used to define derived attributes.

An example of a model package is shown in Figure 5. All (non-abstract) classes are required to have a mandatory, frozen 'parent' link to another class (in the same package), as well as a natural key - attributes and links that identify the object uniquely in the context of its parent. The parent links join all objects together in a tree, rooted in the *MemopsRoot* object. The tree is also used to define the containment relationships used in XML file storage. Each package must contain one single class (a *TopObject*) that is a child of MemopsRoot. The combination of links and keys ensures that there is a natural navigation path to each object and in general gives structure to the web of objects. Mostly parent links and keys reflect the logical structure of the data. In Figure 5, for instance, the *Residue* is a child of the *Chain* (key *seqId*) and the *Atom* is a child of the Residue (key *name*). An object of a given class can be uniquely identified by navigating from the MemospRoot through parent-child links to the object and keeping the natural key at each step. The resulting list of natural keys forms the *full key*.

TopObjects are used to divide data into separate extents. For each package, the children and descendants of a single TopObject are stored as a separate XML file (in file implementations). It is not possible to form links between objects from the same package belonging to different

TopObjects. This prevents e.g. forming a bond between atoms from different amino acid templates. TopObjects are given a globally unique identifier (GUID) on creation. This can be used to distinguish between objects created by different operators with accidentally / unavoidably overlapping key attributes.

Connections between objects that are not parent-child links are called *crosslinks*. It is the crosslinks that makes the object web differ from a tree. Most crosslinks are navigable from either side. An example would be the link between Model and Coord in Figure 5, where you could access both Model.coords and Coord.model. There may also be one-way links that can be navigated from one side only. These are often derived, like all the one-way links in Figure 5. Data defining two-way links are stored in the objects on both sides of the link. This allows rapid queries, at the cost of causing complications in modifying objects that would have been prohibitive in handwritten, non-generated code. The most complex parts of the code generation software tend to involve crosslinks. Modifying a crosslink requires changing the internal state of objects on both sides of the link, which makes the API classes strongly coupled. Pre-modification checks must likewise be carried out on objects on both sides of the link. The required code differs depending on the link cardinality (-to-one or -to-many, mandatory or optional) whether the link is modifiable, whether it is one-way or two-way, whether it is to an object in the same or another package, and which constraints, if any, apply.

API functions. Information in the objects is protected from casual modification by *encapsulation*; it can only be accessed by API function calls or through an explicit override mechanism. The Python API exposes the following functions:

- *get, set, add, remove.* For all attributes, depending on cardinality and modifiability
- *sorted*. For collections of DataObjects; returns the objects sorted by full key (see above).
- *findFirst* (*e.g.* Chain.findFirstResidue(seqCode=42, ...)) and *findAll*. For collections of DataObjects or Complex Data Types; returns first (all) object(s) with e.g. obj.seqCode == 42.
- *new* (*e.g.* Chain.newResidue(seqId=39, seqCode=42, ...)). For parent classes. Factory function, taking keyword arguments, that creates a new child object on the parent.
- *delete* (*e.g.* Model.delete()). For DataObjects (not Complex Data Types). Cascading delete. When an object is deleted, all other objects rendered invalid by the deletion are deleted recursively. For instance deleting an object of class Model will trigger the deletion of all Coord objects linked to it, since the Coord.model link is mandatory.
- *checkValid, checkAllValid*. Validity check of object resp. recursive check of object and all child objects.

The API further provides the normal object constructor for each language, and may provide extra functions for some languages, *e.g.* overloaded function forms for Java. For Python the normal attribute syntax (val = obj.attr; obj.attr = val) is also supported.

The Java API mirrors the Python API as far as possible. The only major difference arises because Java does not allow you to pass in keyword=value arguments to functions. To make up for this, findFirst, findAll and constructor functions come in several overloaded variants. In the standard variant, keyword=value arguments are packed into a dictionary that is passed in as an argument (which is what Python is doing 'under the hood'). Other variants are provided to avoid this rather cumbersome procedure. For the constructor, there is a variant where the function parameters are the mandatory attributes/links that do not have default

values (in alphabetic order). For findFirst and findAll there are variants for zero through four keyword/value pairs.

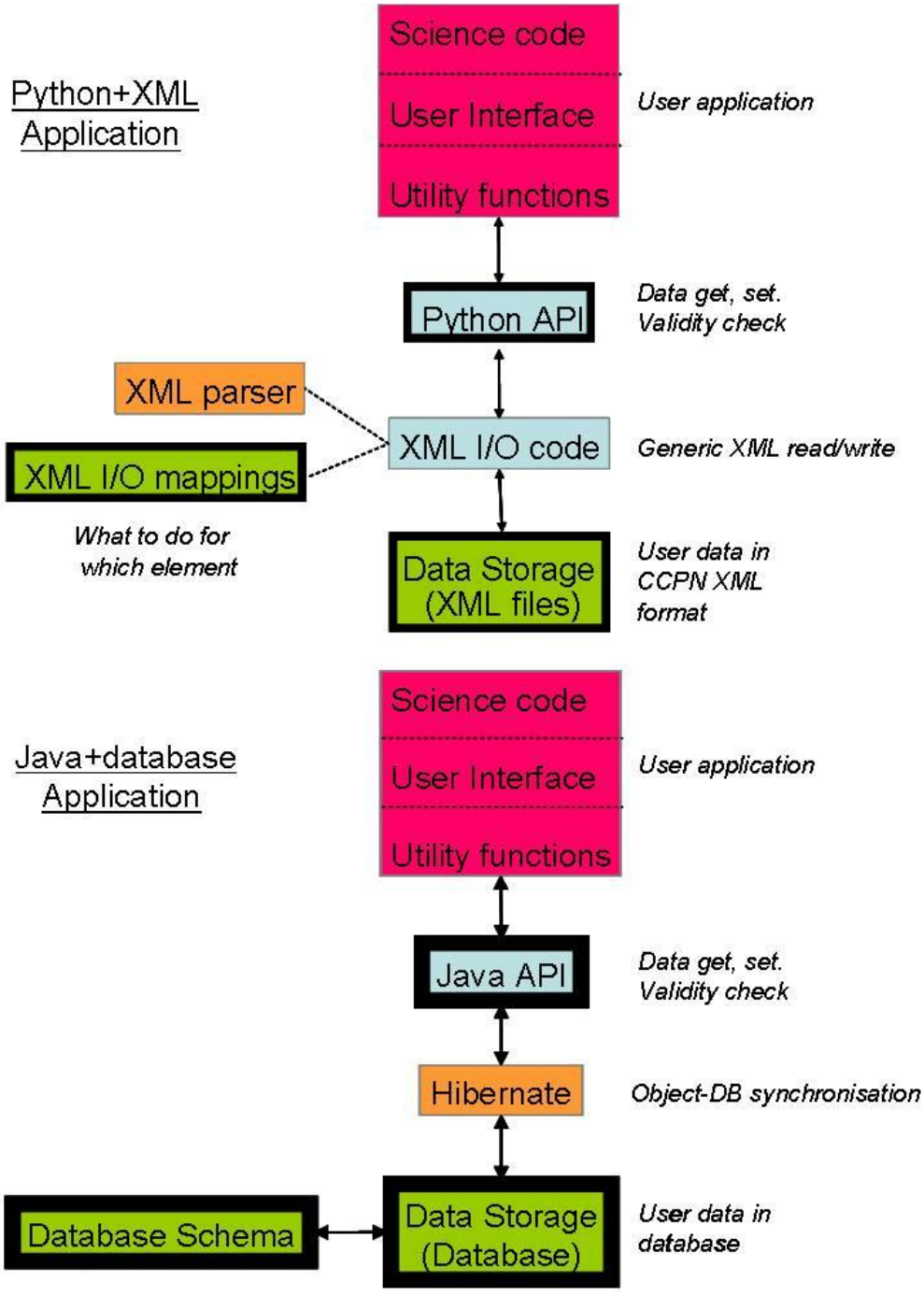
The C API faces the same problems as the Java API, so again there are variant forms for the findFirst, findAll and constructor functions. Since C does not allow function overloading as in Java, the variant forms in C each have slightly differing names. A more important problem is that C has a global namespace. To avoid possible name conflicts the short version of the package name and the class name is prepended to all function names, making them globally unique. The current version of the C API piggybacks off the Python API. Python itself is coded in C, and there is a well specified way of accessing Python objects from C. The C API just calls the corresponding functions in the Python API. C is not object oriented so structures are used for the classes instead, in the same way as the C subroutines that underlie Python.

File storage implementation. We have chosen to handle data persistence transparently – the implementation fetches data into memory as required. For file implementations the directory structure and file names are standardized. Only the topmost directory locations are under user control; the relevant information being kept in the *Implementation* package together with the MemopsRoot. In file implementations data are split into multiple files (one file per MolSystem and one file per StructureEnsemble, in the example from Figure 5) that are saved and loaded independently. This allows the use of lazy loading. The actual file I/O is handled by a generic routine based on a standard parser, with an autogenerated I/O map storing the correspondence between XML tags and model operations. The organisation of the Python+XML implementation is shown in Figure 6 (top).

Database storage implementation. Some applications require concurrent multi-user access to the data and in such cases a database persistence layer has considerable advantages over storing data in XML files. Currently Memops provides database storage for the Java version of the API. The database version of the API is functionally identical to the XML version.

Rather than develop our own object-relational mapping infrastructure, we use Hibernate for the Java+database API implementation. Hibernate mapping and configuration files are generated from the data model and the database DDL is generated from the mapping files using standard Hibernate tools. The database version of the API contains some subtle differences in the way that data are organized. In particular, we wanted to support sharing reference data among projects, and loading more than one project into the database in order to facilitate mining of information across a collection of projects. When searching across multiple projects, care has to be taken not to create Hibernate proxy objects corresponding to the entire database, since this would be prohibitively slow. To mitigate this problem we support the use of special-purpose queries in Hibernate Query Language (HQL), as discussed for the EUROCarbDB application in the main text. As an additional advantage, these can be customized at individual sites. To further potentiate this option the machinery includes hooks to add special denormalized search tables to the database.

The database dialects supported are essentially limited to those supported by Hibernate. All applications to date have used Postgres. However, it is anticipated that Oracle and MySQL will be actively supported soon, in addition to Postgres. Additional I/O calls to support laboratory information management system applications (LIMS) are also being developed**.** The organisation of the Java database implementation is shown ion Figure 6 (bottom).



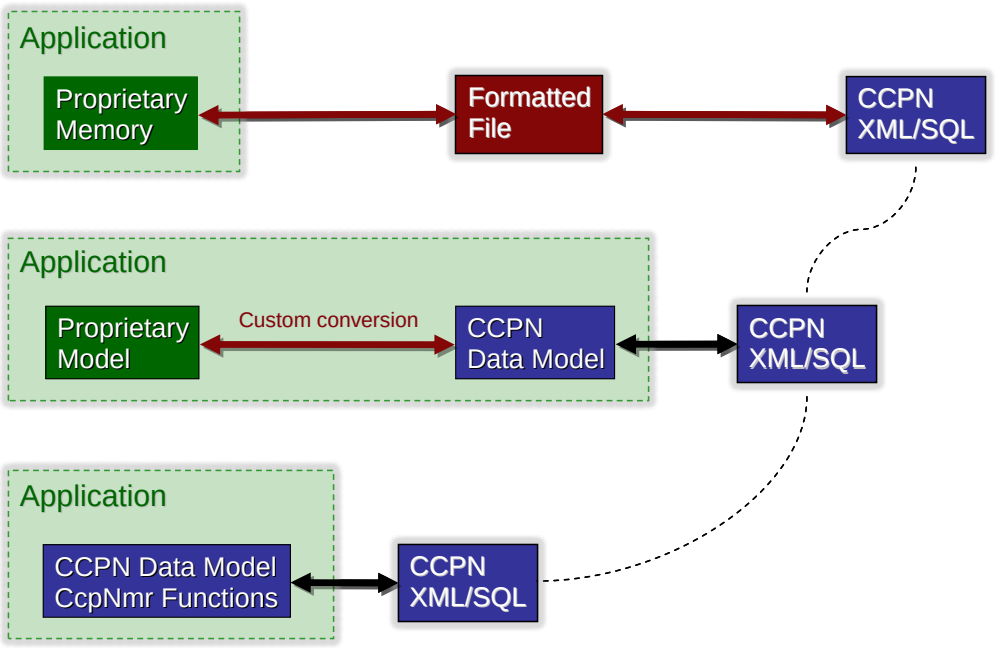
Supplementary Figure 6: Runtime code organization for an application using a Python+XML Implementation (top) or a Java+database implementation (bottom). CCPN software is in blue, external software is in orange, and code specific for the application is in red. Boxes with thick black borders are files generated by CCPN code. The application is connected to the API through an optional layer of utility functions. These execute common tasks that require extensive data model manipulations, e.g. 'Create a molecule from a string of one-letter codes'.

3 Data exchange and software integration

The purpose of a data exchange standard is to support the flow of data between unrelated programs within a single field. At the lowest level even a reliable conversion of data between

different file formats (translation, as it were), is a great improvement. This is the function of CcpNmr FormatConverter, as discussed in the main text. Efficient integration requires smooth, lossless transfer of data between the actual applications, rather than their output files. For ease of use, data transfer must become an integral part of running the programs, without a specific transfer step. This has the further advantage that there is then no separate format conversion code that needs to be updated following program changes.

The experience of building a software pipeline for macromolecular NMR spectroscopy, chiefly through the Extend-NMR collaboration, has provided a number of illustrative examples of how to achieve integration. In general we find that there is a trade-off, so that more indirect integration is easier to set up but less complete and harder to maintain, especially for applications that are still developing. More direct integration with the data standard requires more work to set up for existing applications, but is easier to maintain in the long term, and comes with less potential for information loss and less need for user intervention. Figure 7 shows the typical integration approaches used in the macromolecular NMR software pipeline.



Supplementary Figure 7: Software integration approaches:

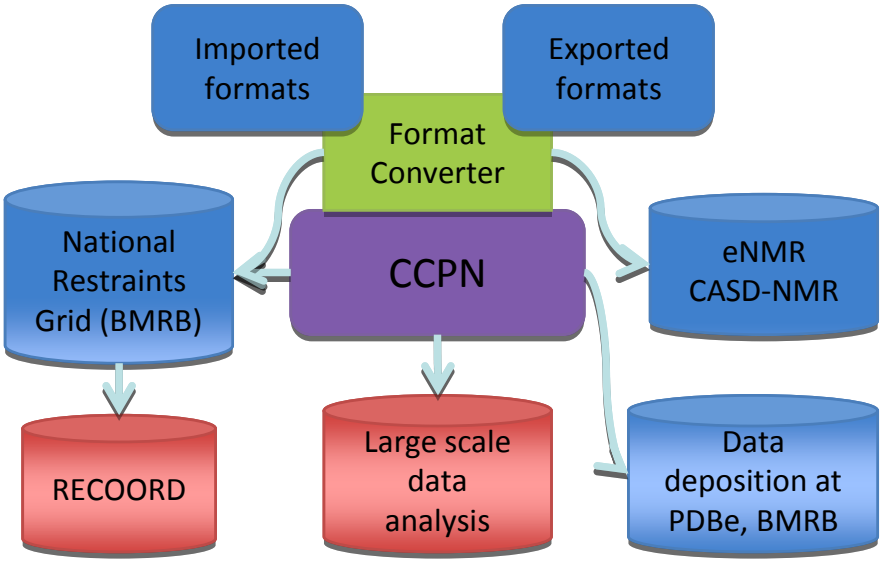
Top: Data exchange through proprietary files. This approach is possible without modifying the application code, but requires continued maintenance of program-specific I/O and mappings. Integration of application output is often problematical, as output data items can be hard to match with input data items, let alone with the CCPN objects that gave rise to them.

Middle: In-memory data conversion. Requires a program-specific conversion layer, but relies on existing CCPN (and often application) I/O. Integration is easier as all information relative to data flow remains available.

Bottom: Direct data access through CCPN API. Does away with the need for conversion or separate I/O code. All information is connected to the data standard throughout. Requires the application to be (re)written specifically for the data standard.

The CCPN data standard is also at the centre of a series of applications built around the deposition databases for macromolecular NMR data (BMRB) and structures (PDBe) – see Figure 8. The eNMR project forms a distributed facility that makes NMR analysis and

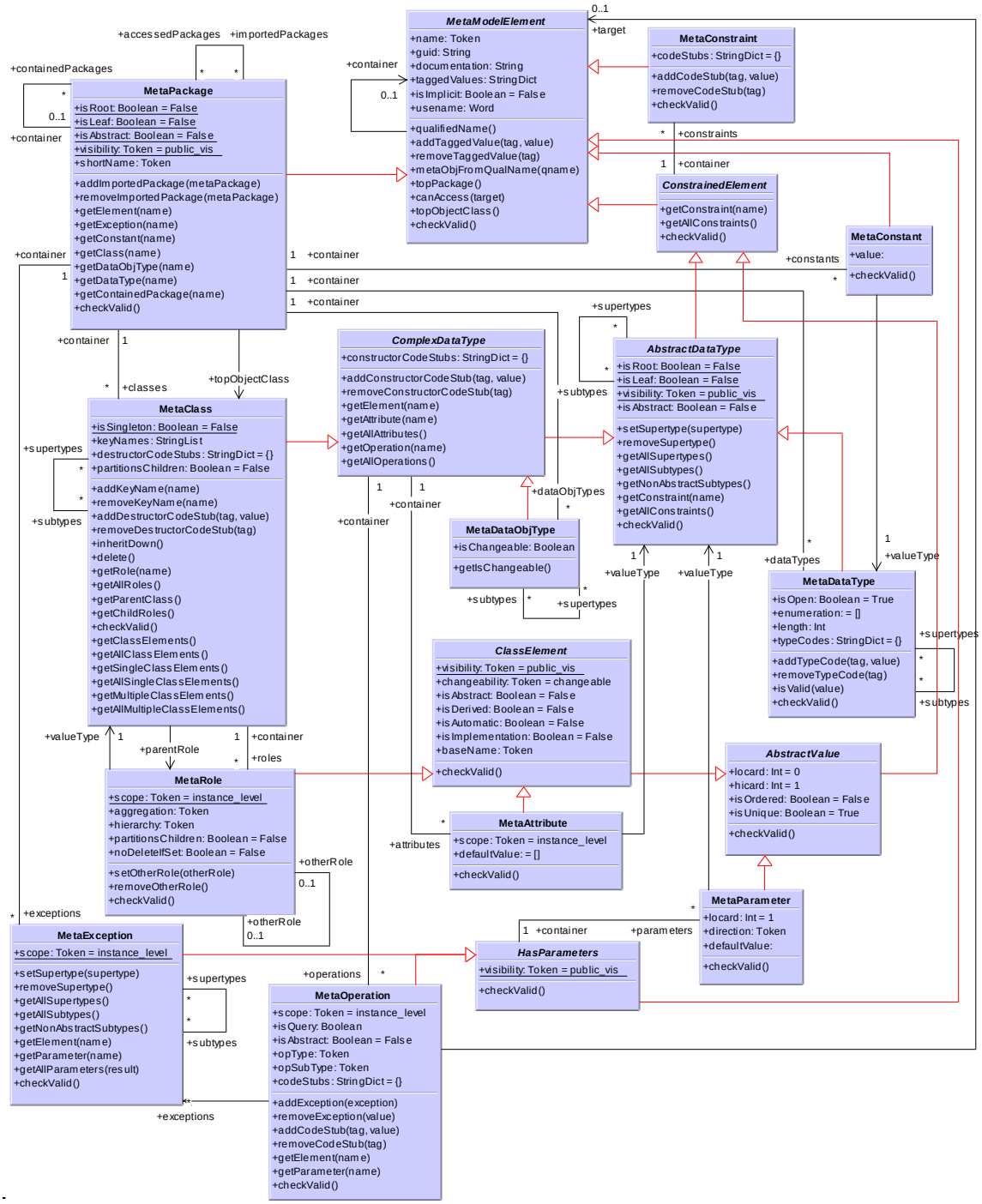
structure generation software available on grid servers. It includes many of the programs also integrated in the Extend-NMR pipeline, such as ARIA, HADDOCK, and CING. CASD-NMR is a part of the eNMR project that sets up blind tests for NMR assignment and structure generation programs along the lines of the CASP protein structure prediction project. An interesting resource is the RECOORD database of NMR structures recalculated from the original with standardized calculation methods. The input for the RECOORD calculation was the FRED database of cleaned-up original deposited restraints, which was generated using the CcpNmr FormatConverter.



Supplementary Figure 8 showing how the FormatConverter and CCPN project files are used in ongoing projects in NMR. The FormatConverter deals with import of NMR file formats, often from archives such as the BMRB or PDB. The data is then made available or analysed as CCPN projects or as files in other file formats exported through the FormatConverter.

4 Memops Metamodel

The in-memory representation of the data model uses the Metamodel shown in Figure 9. For code generation purposes this is implemented as a series of Python classes, with modelled entities as objects.



Supplementary Figure 9: shows the metamodel used for storing the data model description. Broad-headed red arrows are used for inheritance, black lines for object-to-object links. In addition to the operations shown in the diagram the model includes get and set functions for all attributes and roles (not shown). All MetaPackages must be contained within another MetaPackage, except for the topmost one which serves as root. Underlined attributes are not currently used and are fixed at the given default values – they are implemented as class (static) attributes.

For portability the MetaModel has been limited to single inheritance, which means that some type constraints could not be represented properly in the diagram – these are enforced separately by the underlying code. Specifically, MetaAttribute.valueType must be a MetaDataObjType or MetaDataType, and MetaOperation.target must be an AbstractDataType, a ClassElement, or a MetaOperation.