The C_2M project: a wrapper generator for chemistry and biology

Paul van der Vet and Eelco Mossel

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, the Netherlands
Phone +31 53 489 3694, fax +31 53 489 3503
Email {vet,mossel}@cs.utwente.nl

Abstract

Modern science relies on the availability of resources accessible over the web. Each resource uses its own format, among other things because science is highly dynamic and tasks change frequently. In other words, format multiplicity is a fact of life. Data interoperability relies on the presence of wrappers. The C_2M project aims to build a system that supports quick and easy generation of lightweight wrappers by providing a language in which formats can be specified. Because the project was originally aimed at chemical applications, the name "C_2M" is a chemical formula-like abbreviation of "chemical configurable middleware". For reasons of exposition, we will be looking at a simple chemical format in this paper. C_2M can be succesfully applied in other domains as well. The C_2M language has been designed to be easy to learn and use, yet it is sufficiently formal to allow unambiguous description of formats. There is a provision for including documentation, and in fact specification writers are encouraged to do so copiously. Underlying the design of the C_2M language is the intuition that there is such a thing as language ergonomics.

1 Introduction

The C_2M project[1] is concerned with the development of a user-friendly programming language dedicated to producing wrappers. A wrapper is a piece of software that interconverts between different data formats. The project has started by concentrating on (bio)chemical formats as examples, but the language is suited for other disciplines as well.

The problem of format interconversion is encountered in every project in which co-operation between multiple, heterogeneous resources is required. Data tend to come in formats that do not meet the requirements of the project at hand. Data coming from different sources almost always come in different formats. Co-operation between resources can only be achieved by interconversion. If there is a need to dynamically add resources to those already used, some internal uniform data format has to be chosen to ensure smooth interoperability. A pair of wrappers connects every resource to the user desktop (figure 1).

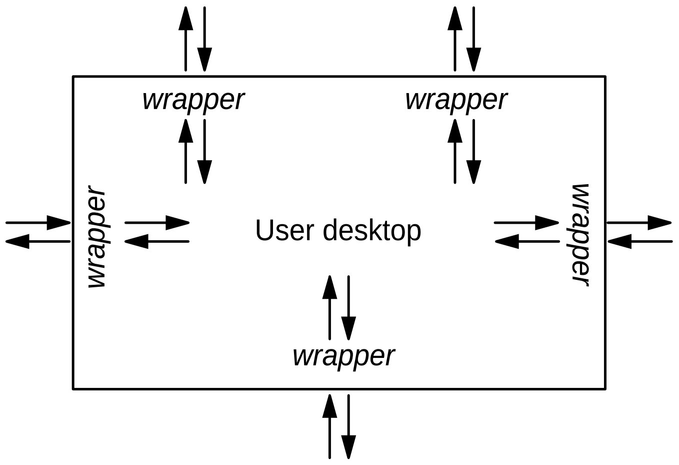

Figure 1. Providing the user desktop with access to multiple, heterogeneous resources.

The construction of wrappers has received enormous attention in the past decades. CORBA[2] is designed to be the ultimate solution to the problem of resource interoperability. A CORBA architecture is robust, portable across many platforms, and scalable. In practice, however, CORBA tends to be heavy-weight and static. Developing an interface in CORBA's IDL language is laborious. IDL code has procedural flavour and semantics of data are implicit. Perhaps for these reasons, the advent of CORBA has not stopped researchers from developing other approaches to resource interoperability.[3]

Turning to the more specific issue of wrappers, many software packages come with built-in wrappers, but their range is restricted and including a new format is either impossible or very laborious. The other, very common alternative is to have project members write wrappers themselves. We will call these "Roll your own" (RYO) wrappers. RYO wrappers are typically written in procedural languages such as awk, Java, Perl, Python, or just straight C. Literature about the influence of notations and formalisms on human performance in a specification task, however, suggests a preference for languages that describe constraints on solutions rather than procedures for finding solutions.[4] In the wrapper domain, this preference translates into one for wrappers built from high-level, declarative descriptions of the formats involved. Indeed, RYO wrappers in a procedural language are generally hard to read and maintain, particularly so over intervals of a year or more. CORBA's IDL objects suffer from the same defect. On the positive side, RYO wrappers present a flexible, lightweight approach to resource interoperability, wholly in line with the dynamic nature of science on the Web. The way out is to use *wrapper generators*. A wrapper generator generates wrappers from high-level descriptions. Because implementation details are hidden from view, wrapper generators facilitate the task of building and maintaining RYO wrappers. The main fundaments of wrapper generators are known from compiler building.[5] An approach to wrapper generation has been published fourteen years ago by Mamrak and co-workers.[6] Since then, many approaches to wrapper generation have been published. The WWW Wrapper Factory (W4F) approach[7] resembles C_2M but is largely confined to XML formats. The C_2M language bears resemblance to the idea of token-templates.[8] Automated wrapper generation[9] and wrapper verification[10] have also received attention.

C_2M builds on these experiences to present a wrapper generator that facilitates the design and maintenance of RYO wrappers. It is not intended to supplant existing software for regular conversion jobs. Rather, C_2M supports irregular jobs such as extracting a table with protein-

protein interactions from the XHTML version of a journal article, extracting certain numbers from a file and output them for SPSS processing, turn metadata into a uniform format for the MPRESS project,[11] or, indeed, turn a high-level description of a resource into a piece of CORBA IDL code. Although originally intended for conversion of plaintext file formats, the design supports the conversion of any format because it can handle Unicode character sets and control codes; see also section 4.

2 An example: CT files

The C_2M language will be discussed by means of an example. A full description of the language will be given in the manual, which is under construction. The running example is the CT (connection table) format as exported by the ChemDraw package for drawing molecular structures. The CT format is one of the many available formats for representing chemical structure.[12] It is a very simple format; space restrictions forbid a more realistic example. A CT file just lists the graph: the atoms and the bonds between them. As usual in such files, all hydrogen atoms and the bonds connecting hydrogen atoms to other atoms are left out. The CT file for a popular molecule, CH_3CH_2OH or ethanol, is given in figure 2. The first line lists the filename but may contain any string. The second line lists the number of atoms and the number of bonds, respectively. Their presence dates from the Fortran era, when the reading program had to know in advance how many lines to read.

```
ethanol.ct
   3   2
    -0.8667  -0.2500  0.0000 C
     0.0000   0.2500  0.0000 C
     0.8667  -0.2500  0.0000 O
  1  2  1  1
  2  3  1  1
```

Figure 2. CT file for the ethanol molecule

The next three lines provide atom information, one atom per line. The three numbers in front are co-ordinates used by ChemDraw to reconstruct the drawing from which the file has been generated. The presence of such co-ordinates in chemical structure files is quite common. At the end of each atom line we find the chemical symbol. Implicitly each of the atoms is numbered, starting with 1. This becomes evident in the last two lines of the file, which list bonds. Each has four numbers: the first two are the numbers of the atoms the bond connects. Then follows the nature of the bond (single, double, …). The last number, called a bond order by ChemDraw, again is a drawing aid for ChemDraw.

3 The C_2M language and wrapper generator

3.1 Introduction

The C_2M project has concentrated from the start on the language in which the wrapper task is specified. The present section attempts to provide an impression of the C_2M language. It has to fulfill contradictory requirements: ease of use, sufficient freedom to express many formats, and sufficient formality to express unambiguous format descriptions. Because existing format specifications are often sloppy and imprecise, we consider it mandatory that the language can be used to publish and share format specifications even if no wrapper is intended. This gives the language a distinct advantage as communication medium over procedural languages, because in procedural languages format specifications are by necessity implicit in the procedures. In addition, there has to be a compiler that can produce an executable wrapper from format specifications. Finally, it would be nice if the wrapper runs in acceptable time and exhibits agreeable scaling behaviour. The result, no matter how we proceed, is inevitably a compromise.

As we have explained elsewhere (ref. 1), C_2M conversion proceeds over an intermediate format. Having an intermediate format reduces the number of converters required from n^2—n to $2n$, which already pays off for $n > 3$. Also, it prepares for middleware tasks such as comparison of two or more sources and merging of sources. When C_2M has read data from a file, it stores these data in a format called C_2M's *native representation*. An ontology serves as a template for the native representation. Ontologies are specified by the user and can be tailored to the task at hand. Conversion of data from a source format *A* into a target format *B* thus proceeds in two steps. The first step takes its information from the specification of format *A* and an ontology as input. The specification of format *A* includes the specification of the accompanying ontology. The second step takes its information from the specification of format *B*. See figure 3.

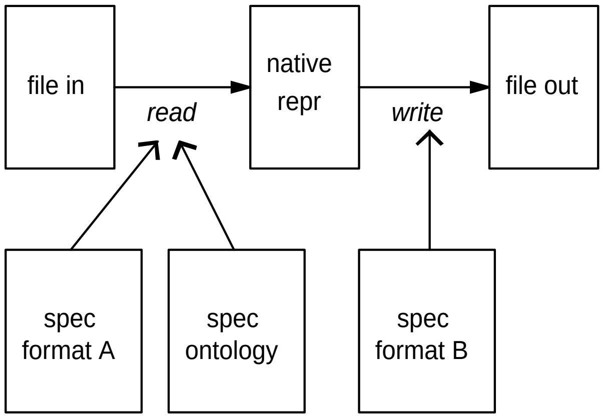

Figure 3. File conversion by the C_2M system.

C_2M views any file as a string of characters. The string can be analysed as consisting of substrings. These substrings fall into one of three categories: meaningful strings, landmarks, and redundant strings. The decision to view a particular string as meaningful or redundant depends somewhat on the application. *Meaningful strings* represent the information we want to read from the source file or write to the target file. In the CT example file of figure 2, for example, the meaningful strings are, on the atom lines, the co-ordinates, element symbols; and on the bond lines, the atom numbers, and the bond types. This means that we view the entire first and second lines and the ChemDraw bond order as *redundant strings*. The space characters in front of every line except the first are likewise redundant. *Landmarks* serve to delimit and identify records and fields. In the CT file, the space characters between data on the same line and the end-of-line characters serve as landmarks. In other formats, fields are identified by special strings. In the Unix refer bibliography format, for example, a two-letter string at the start of each field serves as field identifier.

The C_2M task is to extract the meaningful strings from the source file while preserving their mutual relations like order, and to store those strings, possibly after some modification, as the native representation. Modifications include simple calculations and character transformations; also, a table can be specified that for each input gives the corresponding native representation. To do this, we have to name those strings and the larger structures of which they form part. This can be done by parsing the source file. Next, the meaningful strings are extracted and possibly converted into the format required by the native representation according to conversion rules specified by the user. The conversion rules are known as semantic bindings. The parsers are generated from user-supplied grammars (compare, for example, ref. 5). This enforces a natural separation between form and content: the syntax of the source file is covered by the grammar while the semantics of meaningful strings are fixed by the conversion rules.

The overall view of the system is shown in figure 4, see ref. 1 for a more extensive discussion. The process starts with ontology and file format specifications supplied by end-users and/or content providers. The C_2M system consists of the C_2M compiler and a converter core. The C_2M compiler turns each specification into source code in some programming language, or into object code. The appropriate compiler and/or linker is then used to produce an executable that is able to convert from and into each format for which a specification is provided. Another program called the *documenter* turns the same specifications into XML or LaTeX sources, to produce documents for human consumption by the appropriate renderers. The documentation consists of the comments included in the specification and the specification itself. The principle followed here is that of literate programming:[13] always derive code and documentation from the same source.

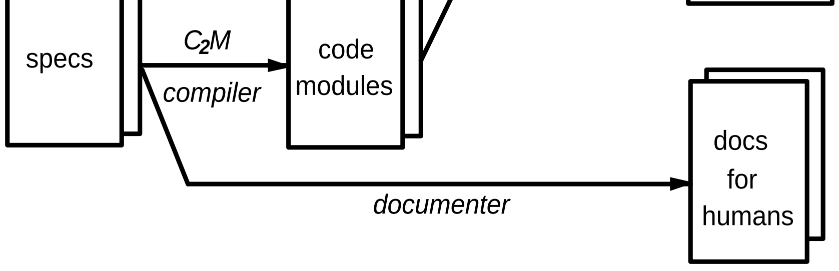

Figure 4. Operating the C_2M system.

In the following subsection, we will discuss a number of features of the C_2M language. A simple ontology of molecules and the format specification of CT files are given as appendices.

3.2 Basic structure of a C_2M specification

Ontology and file format specifications are written by the user to determine the way C_2M converts one file into another. Each specification comes in a file of its own. It includes both the specification itself and documentation to explain the choices made. A specification file is a plaintext file except that literal strings may consist of any character provided it is taken from some pre-defined character set such as Unicode.[14] The precise definition of a plaintext file is beyond the scope of the present paper; it roughly corresponds to the Unicode C0 character set[15] from which the control codes have been removed except for end-of-line. The plaintext requirement currently also applies to the documentation included in the file; this constraint will be relaxed in future versions.

The leading idea in designing the C_2M language has been to comply as much as sensible with well-known syntax practices without compromising the efficiency of the code generator. This has resulted in modest use of tags with or without attributes, as known from HTML syntax, and the use of well-known operators or their plaintext imitations, such as "::=" or "->" (imitation of "→") for grammar rules, "<-" (imitation of "←") for instantiation rules, and "=^" (imitation of "≙") for correspondence rules (the nature of these rules will be explained below).

Below, we will use the Arial font to refer to characters and strings that have to be present literally in the specification, and the *italic Arial* font is used to refer to variables that have to be replaced by literal strings in the final specification.

Every C_2M specification has the general structure:

```
<C2M-SPECIFICATION name=name
                         type=type>

blocks

</C2M-SPECIFICATION>
```

where *type* currently can be one of the two ontology or file-format; *name* is chosen by the user and is required to be unique within each C_2M implementation. The *blocks* are also enclosed in tags. What the blocks are depends on the nature of the specification: ontology or file-format.

Documentation can be included wherever a tag or rule is expected, but never within a tag or rule. Documentation is enclosed in tags <TEXT> and </TEXT> (case-insensitive). Specification writers are encouraged to insert copious documentation in their files.

3.3 Ontology specifications

Ontology specifications consist of a single block. The ontology formalism has been kept to an absolute minimum: it is expressed in a frame language in which concepts can have attributes. The relations between concepts and attributes are left unspecified. A simple ontology of molecules is used as illustration.

Concepts come in three sorts. *Complex concepts* have an arbitrary number of unique attributes. Attributes are concepts, and thus may have attributes themselves. An example is the concept of atom:

```
atom = chemical-symbol, id, dr-coord-x, dr-coord-y, dr-coord-z
```

where id is the number that identifies the atom in the molecule. The attributes are delimited by a comma followed by a space. In fact, any combination of comma's, space characters, and end-of-line characters, provided there is at least one of them, can be used to delimit tokens in a specification file.

Concepts with a repeated attribute have a single attribute that can, however, occur an arbitrary number of times. For example:

```
atom-list = repeated( atom )
```

In the native representation, the atom attribute will occur as many times as dictated by the source file: three times in the CT example of figure 2.

Primitive concepts have no attributes. They can hold an instantiation, normally a string read from a source file. In the rule above for atom, the concept chemical-symbol is primitive. If the CT file of figure 2 is read, it will hold the string "C" or "O", depending on from which line the instantiation is derived.

3.4 File format specifications

File format specifications consist of two or four blocks, depending on whether the specification is intended for reading or writing only (two blocks) or for both reading and writing (four blocks). The read and write task each are specified in a syntax block that contains a grammar and a semantics block that contains so-called semantic bindings. Each block is enclosed in tags.

In specifications for both reading and writing, it may seem wasteful to specify the grammar and the semantic bindings for reading and for writing separately. We can only use the same grammar for reading and writing when it is reversible. A grammar is reversible when there is no syntactic indeterminacy. Unfortunately, syntactic indeterminacy in the kind of files addressed by C_2M is common. For example, in the CT file of figure 2, every line save the first starts with space characters, but whether they are there and, if so, how many are there does not matter. In file types with explicit field identifiers, field order within a record can be free. The grammar for reading allows for indeterminacy in these cases. The grammar for writing, however, needs explicit instructions that rule out syntactic indeterminacy. Semantic bindings, by contrast, can be reused to some extent. To be fit for reading and writing, the particular rule in the semantic bindings has to be bijective. In that case, the rule has to be specified in the semantic bindings for reading only.

3.5 File format specification files: reading

The grammar for reading follows the rules for BNF (Backus-Naur Form) grammars. A rule that describes any CT file in its entirety is:

```
ct-file -> ct-first-line ct-numbers-line ct-atom-lines ct-bond-lines
```

where the prefix ct is inserted to remind the human reader of the particular format we are describing. The four symbols at the right-hand side have to be defined elsewhere in the grammar such that ultimately we arrive at the level of strings. C_2M has a number of built-in character and string classes, for example the character classes letter, digit, and end-of-line; and the string classes word (only letters), spaces (any sequence of space characters), string (any sequence of non-space characters), line (any sequence of characters except end-of-line), integer, and float. Using these, we may write for example:

```
ct-first-line -> line end-of-line

ct-atom-line -> spaces  ct-coord-x  spaces  ct-coord-y  spaces
                ct-coord-z  spaces  ct-chem-symbol  end-of-line

ct-coord-x   -> float
[etc.]
```

The symbol ct-atom-lines stands for the series of consecutive lines with information on the atoms that constitute the molecule. Because the number of those lines is mentioned in the second line of the CT file, we could write a grammar rule that makes use of this information. This would be laborious, however; it is easier to write:

```
ct-atom-lines -> ct-atom-line+
```

where the BNF-operator + signals one or more occurrences. A rule of this kind always explicitly records the position information so that, in this case, we can identify the *n*th occurrence of ct-atom-line in the list ct-atom-lines.

At this point, the substrings of a CT file and groupings of those substrings have been named. We can rely on the parser to build a parse tree as it analyses the source file from which we can extract groupings and substrings by name. We now have to tell C_2M which meaningful strings are used to instantiate primitive concepts of the ontology and how to translate them in order to comply with the requirements of the native representation. This has to be done at two levels. First, we have to specify correspondences between groupings, for example:

```
atom-list =^ ct-atom-lines
```

Next, we have to specify conversions, if any. This is done by means of instantiation rules. We decide we want to store the strings as they are, for example:

chemical-symbol <- ct-chem-symb

This does not give us a way to instantiate the id concept, because there is no meaningful string in the source file that provides the required information. Instead, we make use of the fact that grammar rules for repeated occurrences record position information:

id <- position(ct-atom-line, ct-atom-lines)

which translates roughly as "for each atom, instantiate the id concept with the number that records the position of ct-atom-line in the list ct-atom-lines".

Instantiation rules can be left out of the format specification. The effect is that the corresponding primitive concept in the ontology will not be instantiated, which in turn means that the entire concept will be absent from the native representation. This feature can be used to incorporate a weak form of disjunction in ontologies. To promote reuse of ontologies, one can add all primitive concepts that might be needed. The decision which concepts to instantiate is taken in the file format specification.

3.6 File format specification files: writing

The grammar for writing is much like the grammar for reading, except that indeterminacy is not allowed. This means, among other things, that pre-defined string classes such as spaces or line cannot be used. In writing, every grammar symbol will have to produce output. In addition to a grammar, a format specification for writing has semantic bindings (correspondences and instantiations) to translate the native representation into the target format. If a format specification is used for both reading and writing, bijective semantic bindings already specified in the read part do not have to be repeated in the write part. Correspondences are always bijective.

We could in principle reuse the first rule of the grammar of a CT file for reading:

ct-file -> ct-first-line ct-numbers-line ct-atom-lines ct-bond-lines

but we cannot use other read rules for writing. In the first place, we have to say what should come at the first line, for example:

ct-first-line -> "File generated by C2M" end-of-line

which puts the string at the right-hand side on the first line of the target file and ends it with an end-of-line character. Next, at the second line called ct-numbers-line, the numbers of atom lines and bond lines have to be filled in. In the native representation, the numbers of atoms and bonds are not present explicitly. Therefore we have to establish them in another way. The C_2M language has the construct:

fl-text(*Supersymbol*, *Symbol*, *NrOccs*)

that is a canned version of the grammar rule

Supersymbol -> *Symbol*+

except that the number of occurrences of *Symbol* is fixed to be *NrOccs*. This can be either an explicit number or a grammar symbol that writes that number at the appropriate place. In the latter case, there are two occurrences of the *NrOccs* symbol that should be at the right-hand side of the same grammar rule. Using this construct, we may choose to write the entire CT file using the rule:

```
ct-file -> "File generated by C2M" end-of-line
           space space nr-atom-lines space space nr-bond-lines end-of-line
           fl-text(ct-atom-lines, ct-atom-line, nr-atom-lines)
           fl-text(ct-bond-lines, ct-bond-line, nr-bond-lines)
```

which will write a correct CT file provided there are correct rules for ct-atom-line and ct-bond-line elsewhere in the grammar for reading.

4 Implementation

The current version of C_2M is implemented in Prolog.[16] We believe this has considerably eased the development of the system, but we also want to stress that the same functionality can be realised in any other programming language. Indeed, sharing and reuse of C_2M specifications should be independent of the programming language or languages used.

The implementation has been inspired by work in natural-language processing. For example, the two read steps are syntactic and semantic analysis, precisely as in many natural-language understanding systems. For syntactic analysis, we use the parser generator built into Prolog. Semantic analysis is based on knowledge-based system technology. The converter core of C_2M incorporates a special-purpose inference engine that uses the semantic parts of a file format specification as knowledge base.

A few points in the implementation merit attention. Using grammars in specification files to generate parsers is a very general way to handle the syntax of formats. Because tokens can be delimited by any character, a separate tokenisation step is not foreseen. Because the parser analyses the source file on a character-by-character basis, C_2M can handle any type of file but conversion will proceed slower than in the presence of a tokenisation step. Even control codes can be defined as characters and handled appropriately. Molecular structure files for not too large molecules (say, with 1000 atoms or less) are processed within a second. Preliminary experiments have shown that the scaling behaviour of the system approaches linear behaviour.

The C_2M compiler in fact converts a specification file into a Prolog file. In other words, the compile task can in principle also be done by the C_2M converter, given the appropriate specification files. One reason not to do so is efficiency. In specification files we do know what the token delimiters are, and therefore we can insert a tokenisation step before the parsing step. Also, the C_2M compiler optimises to some extent. Code optimisation is beyond the capabilities of a format converter such as C_2M.

Finally, the Prolog implementation we used is Quintus Prolog[17] because it is robust and very fast. Although most constructs are standard Prolog,[18] we sometimes made use of Quintus built-ins because they are more efficient. Quintus Prolog is proprietary software. We are currently contemplating to implement future versions in Java.

5 Conclusion

We have given a very cursory overview of the C_2M language. The language is designed with ease of use as primary concern, where “ease of use” has been operationalised mainly as “familiar” in a number of ways. User surveys will have to bear out whether we succeeded in this respect. A wide scope has been another design criterion. It has been operationalised by analysing foreign files at the level of individual characters. C_2M, however, does not aim to make existing software superfluous. For example, the provision of extensive options for calculations has not been considered because there is well-known and widely available software that can do this. A desktop system built around C_2M will perform calculations be having C_2M convert some intermediate result into a query to a calculation package, and having C_2M convert the package’s output into a suitable form for further processing.

C_2M is now tested on a large number of formats to detect weaknesses and omissions. Future work will be directed at an extension of the possibilities of the system in two directions: more instantiation functions and more special grammar constructs will widen the range of formats that can be described by specification files; and possibilities will be added for using C_2M as middleware system. We view the project as a first step toward a more empirical approach to language design, which will involve (among other things) user surveys.

Acknowledgements

The authors wish to acknowledge the many helpful comments from Judith Plümer (University of Osnabrück) and Andrei Malchanau (University of Twente).

Appendix A: Simple-chem ontology

```
<C2M-SPECIFICATION name=”simple-chem”
                   type=”ontology”>

<TEXT>This example lacks documentation because the choices have been
explained in the main text, section 3.3.</TEXT>

<ONTOLOGY>

molecule = atom-list, bond-list

atom-list = repeated( atom )
bond-list = repeated( bond )

atom = chemical-symbol, id, dr-coord-x, dr-coord-y, dr-coord-z

bond = id1, id2, bond-type

</ONTOLOGY>

</C2M-SPECIFICATION>
```

Appendix B: CT file format specification

```
<C2M-SPECIFICATION  name=”ct”
                    filetype=”plaintext”
                    type=”file-format>

<TEXT>Here, too, documentation has not been inserted because the
main text provides sufficient background information to understand the
choices made. See sections 3.4—3.6.</TEXT>

<READGRAM  startsymbol=”ct-file”>

ct-file -> ct-first-line ct-numbers-line ct-atom-lines ct-bond-lines

ct-first-line -> line end-of-line

ct-numbers-line -> spaces nr-atom-lines spaces nr-bond-lines end-of-line

ct-atom-lines -> ct-atom-line+
ct-bond-lines -> ct-bond-line+

ct-atom-line -> spaces ct-coord-x spaces ct-coord-y spaces ct-coord-z spaces
                ct-chem-symbol end-of-line

ct-bond-line -> spaces ct-id1 spaces ct-id2 spaces ct-bond-type spaces
                ct-bond-order end-of-line

ct-coord-x -> float
ct-coord-y -> float
ct-coord-z -> float

ct-chem-symbol -> upper-case-letter
ct-chem-symbol -> upper-case-letter lower-case-letter

ct-id1 -> integer
ct-id2 -> integer

ct-bond-type -> integer
ct-bond-order -> integer

</READGRAM>

<SBREAD ontology=”simple-chem”
        top-concept=”molecule”>

molecule =^ ct-file

atom-list =^ ct-atom-lines
```

```
bond-list =^ ct-bond-lines

dr-coord-x <- ct-coord-x
dr-coord-y <- ct-coord-y
dr-coord-z <- ct-coord-z

chemical-symbol <- ct-chem-symbol
id <- position(ct-atom-line, ct-atom-lines)

id1 <- ct-id1
id2 <- ct-id2
bond-type <- ct-bond-type

</SBREAD>

<WRITEGRAM start-symbol="ct-file">

ct-file -> "File generated by C2M" end-of-line
           space space nr-atom-lines space space nr-bond-lines end-of-line
           fl-text(ct-atom-lines, ct-atom-line, nr-atom-lines)
           fl-text(ct-bond-lines, ct-bond-line, nr-bond-lines)

ct-atom-line -> space space ct-coord-x space space ct-coord-y
                space space ct-coord-z space space ct-chem-symbol
                end-of-line

ct-bond-line -> space space ct-id1 space space ct-id2
                space space ct-bond-type space space ct-bond-type
                end-of-line

   <TEXT>We just fill in the bond type where ChemDraw expects bond order:
   this will not harm</TEXT>

</WRITEGRAM>

<SBWRITE>

<TEXT>This block is empty because as a matter of fact all semantic bindings in
the SBREAD-block are bijective.</TEXT>

</SBWRITE>

</C2M-SPECIFICATION>
```

[1] P.E. van der Vet, H.E. Roosendaal, and P.A.T.M. Geurts, "C_2M: configurable chemical middleware", *Comparative and Functional Genomics* 2 (2001), 371—375.

[2] http://www.corba.org

[3] See, for example, S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano, "Semantic integration of heterogeneous information sources", *Data and Knowledge Engineering* 36 (2001), 215—249, and C.A. Goble *et al.*, "Transparent access to multiple bioinformatics information sources", *IBM Systems Journal* 40 (2001), 532—551.

[4] For example, B.W. van Schooten, *Development and specification of virtual environments*, Ph.D. thesis, Parlevink group,University of Twente, Enschede, the Netherlands, 2003, chapter 5; also available as http://wwwhome.cs.utwente.nl/~schooten/proefschrift.pdf; B. Khazaei and C. Roast, "The usability of formal specification representations", in: G. Kadoda (ed.), *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group*, Bournemouth UK, April 2001, pp. 305—310.

[5] The classic text on compiler building is A.V. Aho and J.D. Ullman, *Principles of compiler design*, Reading MA: Addison-Wesley, 1979.

[6] S.A. Mamrak, M.J. Kaelbling, C.K. Nicholas, and M. Share, "Chameleon: a system for solving the data-translation problem", *IEEE Transactions on Software Engineering*, 15 (1989), 1090—1108; S.A. Mamrak, C.S. O'Connell, and J. Barnes, *The integrated Chameleon architectures*, Englewood Cliffs NJ: Prentice Hall, 1994.

[7] A. Sahuguet and F. Azevant, "Building intelligent Web applications using lightweight wrappers", *Data and Knowledge Engineering* 36 (2001), 283—316.

[8] B. Thomas, "Token-templates and logic programs for intelligent web search", *Journal of Intelligent Information Systems* 14 (2000), 241—261.

[9] N. Kushmerick, "Wrapper induction: efficiency and expressiveness", *Artificial Intelligence* 118 (2000), 15—68; P.B. Golgher, A.H.F. Laender, A.S. da Silva, and B. Ribeiro-Neto, "An example-based environment for wrapper generation", in: S.W. Liddle, H.C. Mayr, and B. Thalheim (eds.), *Conceptual Modeling for E-Business and the Web*, Berlin: Springer, 2000, pp. 152—164.

[10] N. Kushmerick, "Wrapper verification", *World Wide Web* 3 (2000), 79—94.

[11] J. Plümer, "MPRESS – Transformation von Metadaten Formaten", *This volume.*

[12] T. Engel and J. Gasteiger, "Chemical structure representation for information exchange", *Online Information Review* 26 (2002), 139—145.

[13] D.E. Knuth, *Literate programming*, Palo Alto CA: Center for the Study of Language and Information of Stanford University, 1992.

[14] The Unicode Consortium, *The Unicode standard, version 3.0*, Reading MA: Addison-Wesley, 2000.

[15] Ref. 14, pp. 336—340.

[16] L. Sterling and E. Shapiro, *The art of Prolog*, Cambridge MA: MIT Press, 1994 (2nd edition); I. Bratko, *Prolog programming for artificial intelligence*, Harlow: Addison-Wesley, 2001 (3rd edition).

[17] http://www.sics.se/isl/quintus/

[18] P. Deransart, A.A. Ed-Dbali, and L. Cervoni, *Prolog: the standard*, Berlin: Springer, 1996.