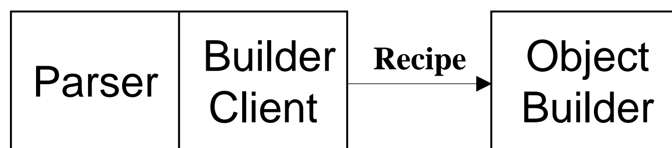
5 Recipe Encoding and Parsing

MONDO works in terms of recipes internally but we also need ways for people to easily describe and view these recipes. SGML has shown that using marked-up text works very well because it is highly-accessible to any user and it can encode arbitrarily complex information quite easily. MONDO chooses this same approach for encoding its recipes. Moreover, MONDO uses SGML/XML as one of its encoding formats so it can take advantage of the tools and familiarity available with SGML-type of markup. MONDO also offers a slightly different markup language called OML (Object Markup Language) which is very similar to SGML. These are expected to be the primary means of encoding recipes in human-readable files.

Before going into the encoding formats themselves and how they are interpreted, we will define what the Parser and BuilderClient are responsible for and how they fit into the MONDO architecture.

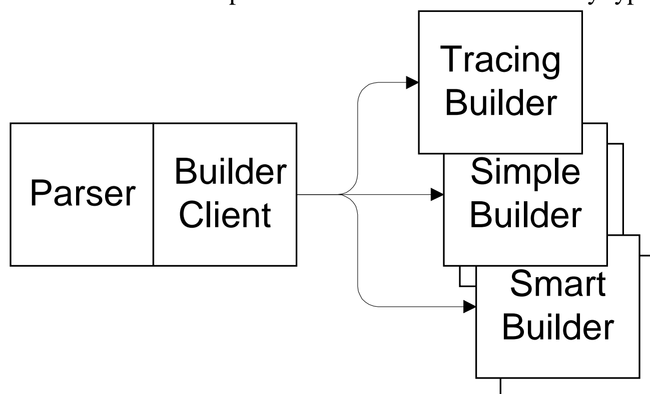
5.1 Parsers and BuilderClients

In the context of MONDO, a **Parser** is responsible for interpreting text and converting it to recipes. The **BuilderClient** then takes these recipes and tells them to the ObjectBuilder, which constructs the information model that the recipes encoded. The only purpose of the Parser-BuilderClient pair is to send recipes to the ObjectBuilder.



There is a conceptual flow of recipes from Parsers to ObjectBuilders, but rarely are recipes constructed and then passed. Instead, recipes are implicitly described in the communication through the three components.

BuilderClients are usually the back-end of a parser but they can also be found in other roles. They can be associated with decoding binary information or with an Encoder that talks to a Builder instead of writing the information to a file. A BuilderClient is simply a component that can talk the ObjectBuilder interface and protocol. A BuilderClient attached to a Parser has to understand the specifics of the Parser so it can produce recipes, but a Parser-BuilderClient pair will then be able to talk to any type of ObjectBuilder.



When the Parser-BuilderClient pair is completed, the parser's recipe encoding language is available to all of MONDO*.

---

* Except encoding a recipe back to the language. This is the easiest to implement component in MONDO and will be covered in the upcoming chapter on Encoding.

5.2 Recipe Encodings: SGML/XML & OML

MONDO supports both SGML/XML and OML because they each have important advantages. SGML and XML have the advantage of being well-established markup languages with a number of tools available for parsing and subsequent processing. Their disadvantage is that they have restrictions (especially 'valid' documents) on the content model that makes representing recipes (of general object models) cumbersome. OML is the embodiment of recipes in a textual form, so it is easier to understand the meaning of the text to MONDO and also easier to encode complex information. OML is also even simpler than XML[*]. OML's disadvantages are that it is new, is somewhat coupled with the recipe concept, and that there is already an existing standard. Ultimately there is no problem with having two formats: MONDO (or any tool) can easily convert between them. This document will mostly use OML because of its obvious translation to recipes.

The next sections will cover OML and SGML. OML only has a single and simple translation to MONDO recipes so its description will be short. For SGML/XML, on the other hand, there are more variations in how a document could be interpreted. The SGML/XML section will discuss how SGML documents can be either:

1. Directly translated using a standard Element and Content oriented perspective.
2. Annotated to describe more general relationships. These annotations can be through:
   1. Architectural forms
   2. Defaulted attributes
   3. Tag naming conventions

5.3 OML: An Object Markup Language

OML is a markup language with somewhat simpler syntax than SGML/XML[†] and that works very well with the semantics of MONDO's recipes and building process. OML is more general[‡] than SGML/XML because ***any attribute of an element can have complex markup***, not just the 'content' attribute. This is a very important advantage because we should be able to easily create the complex relationships required for general object models.

OML is simpler because there are ***much fewer modes and variations*** handled by the parser: OML has less than 15 productions. This partially comes from the regularity of the notation, but it is primarily because OML does much less than SGML. OML is a notation designed for an ObjectBuilder client and the OML processor can rely on the rest of MONDO to provide sophisticated functionality. For example, OML delegates to the ObjectBuilder and the DomainModel the responsibilities for validation and metamodel to model relationships. These simplifications make converting between OML and recipes very easy for both humans and computers.

Elements and Parameters

The simplest example of OML is a single named element delimited by '<' and '>'.

```
<Date>
```

This causes the builder to be called with an identifier of 'Date' and no other parameters for the recipe. This will return a Date representing today from the ObjectBase.

To build a specific date you can add parameters by specifying a parameter name, an equals ('='), and the parameter value.

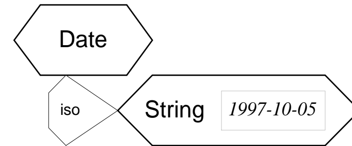
```
<Date iso='1997-10-5'>
```

This will cause the builder to be called twice: once to build the literal

---

[*] OML also has a side benefit (to some) of being very much like LISP with a little more structure.
[†] SGML's concrete syntax to be specific
[‡] In terms of basic concepts. Both (with a suitable application) are functionally equivalent.

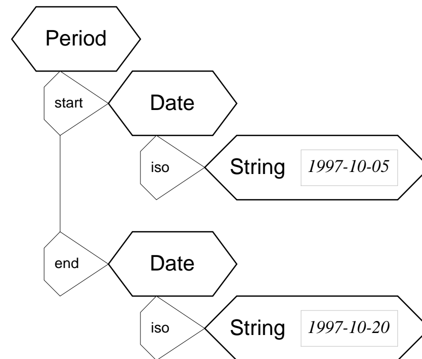
string and then once again to build the date with the string as a parameter called “iso”.



Elements as Attribute Values

To build a Period[*] you need two dates as attribute values:

```
<Period
    start = <Date iso='1997-10-5'>
    end   = <Date iso='1997-10-20'>
>
```



This shows the first major differences with SGML/XML: Elements can have other elements as attribute values. This regularity is important for easily encoding and understanding a general information model based on objects. A parameter has a value that is an Object whether it was built from a direct literal or from a more complex element encoding. There is no preferred parameter (i.e. ‘content’) or anything within an element other than parameters and their values (recipes).

At this point, we have enough functionality to be able to handle the information contained in almost any document. A recipe is formed from the hierarchy of elements and attributes, so we can already describe any recipe[†]. Everything we add from here is to make encoding certain types of information easier, and our major needs are for collections of objects (i.e. Lists) and mixed-content text.

Lists

We could build collections of values using numeric parameters with our List Factory:

```
<Person
    name = "Mark"
    favoriteColors = <List 1="green" 2="blue" 3="red">
>
```

This gets to be very tiresome when we have more than a very few elements. OML uses parentheses as a shorthand for the above construction, an ordered list of recipes/values:

```
<Person
    name = "Mark"
    favoriteColors = ("green" "blue" "red")
>
```

Lists use white-space as separators and can contain any type of value as a list item. In the following example, ‘stuff’ has a value of a List with four elements, one of which is a Date and another is an inner List.

```
<Person
    name = "Mark"
    stuff = (("green" "red") <Date> () 10.345)
>
```

Text

The final special construct is for marking up Strings of simple characters mixed with more general objects. This could be done with the structures we have but it is again cumbersome:

```
("This is the start of a " <EM content= "Beautiful"> " friendship")
```

The example requires remembering to put spaces inside the quote marks and requires closing and reopening the “main” String mode. To support entering general marked up text easily, OML uses curly braces (‘{}’)

---

[*] Where a Period is defined as a Range of Dates.
[†] It may already be obvious to you that the above syntax is a variation on LISP, which explains its generality.

to change to text mode. In text mode all characters and whitespace are interpreted as the content to Strings except the start of an element ('<') and the end of text mode ('}'). So the above recipe would be identically and more easily represented as:

```
{This is the start of a <EM content= "Beautiful"> friendship}
```

Unnamed Parameter

One last piece of behavior that makes document markup simpler and more SGML/XML-similar is the option to not name a parameter. If an element have a delimited value (a String, List, or Text) without a parameter name preceding it, the value is assumed to be part of the default parameter (usually called 'content'. This allows the above example to be simplified to:

```
{This is the start of a <EM "Beautiful"> friendship}
```

or more likely:

```
<P{This is the start of a <EM {Beautiful}> friendship}>
```

Closing Tags and Delimiters

OML has simple element closing rules because markup is not optional, attributes (even 'content') have their own delimiters, and delimiters always nest properly. Because of this, OML has no closing tags only the closing delimiters for currently open tags. Because a tag could have been opened several lines before, it can be useful to associate an end of an element with the beginning. You can do this by repeating the opening tag name at the end:

```
<P{This is the start of a <EM {Beautiful}> friendship}P>
```

Note that this does not require an SGML like '/' to precede it. If you have a closing name that does not match the currently open tag you will get an error.

Other Features

OML also support comments within elements, which are removed at the lexical stage and never visible to the parser itself. Comments are 'C++'-style and can be either line-terminated ('//') or fully delimited ('/*' and '*/'). XML style comments '<!--' will probably also be supported but they require changes to the parser instead of the lexer which makes them less desirable.

OML Non-Terminal Productions

The following summarizes OML's grammar. In the appendix is the full syntax including the terminals.

Element ::= <TAGO> <Name> (Parameter | UnnamedParameter)* <TAGC>

Parameter ::= <Name> (<EQ> Value)?

UnnamedParameter ::= DelimitedValue

Value ::= DirectValue | DelimitedValue

DirectValue ::= <DirectLiteral>

DelimitedValue ::= StringLiteral | Element | List | Text

StringLiteral ::= <StringSQ> | <StringDQ> | <StringLQ> | <StringVLQ>

List ::= <LSTO> ListItems <LSTC>

ListItems ::= (Value)*

Text ::= <TXTO> TextItems <TXTC>

TextItems ::= (<Chars> | Element)*

Summary

OML is just like SGML/XML only a little simpler. OML works with only three core concepts: Elements that have Parameters with Values. Values can either be literals or other Elements. The result of this is a tree of named Elements with named parameter branches connecting each Element to its value. This is exactly equivalent to a Recipe: a hierarchy of named Recipes with named parameters connecting each Recipe to its ingredients' recipe.

On top of this basic structure are two additional convenience constructors: Lists for sequences of elements and Text for combining sequences of Elements with character data (i.e. Strings). Finally, there is one unnamed, default parameter that allows for SGML-like markup for "content".

So although OML is new, it works extremely well for both structured information and textual information. OML is very simple and similar to SGML. But most importantly OML is the embodiment of Recipes and makes explaining, thinking about, and working with the recipe concept easier.

5.4 SGML-O and XML-O

SGML-O and **XML-O** are the MONDO-based object--recipe-perspectives on SGML and XML. As an international and increasingly popular standard, SGML and XML are the obvious choices for encoding information. MONDO applications can benefit from SGML/XML parsers and other tools. Although SGML/XML do not map as directly to MONDO's recipes as OML, the distance is not very far away. This short distance provides a few options: SGML/XML documents can be interpreted in a few different ways depending on your needs. This section will cover the conceptual translations that turn SGML/XML documents into MONDO recipes.

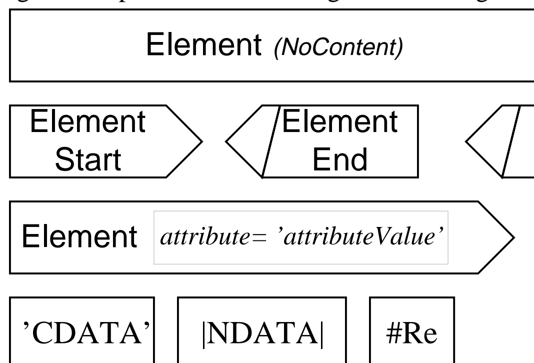
The primary decision is whether to use SGML/XML documents as they are or annotate them with extra information (similar to how HyTime works). If you use standard SGML/XML documents the Parser and BuilderClient will produce recipes that reflect the 'content', sequential, and containment oriented structure of standard SGML. You can annotate an SGML/XML document (or DTD) through any of the following:

1. Architectural forms
2. Defaulted attributes
3. Tag naming conventions

The annotations will allow the BuilderClient to produce more complex hierarchies where any number of parameters (attributes) can have complex markup within them.

SGML Notation

During this chapter we will be using the following notation for SGML/XML markup:



These are standard icons and are meant to reflect the element model of an SGML/XML parser. They also look somewhat like the characters (in the concrete syntax) that are used to enter SGML elements.

Recipe & Parameter Element Notation

To work with MONDO's builders we will need to distinguish elements that are 'recipes', elements that are parameters, and elements that function as both. By default, an element is assumed to be a recipe for building an object. Elements that are parameters are identified by shading:

An element that is both a parameter and a recipe is partially shaded:

The use of these notations will be discussed in the upcoming sections.

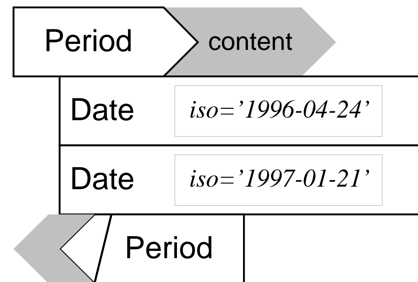
Building Grove-Oriented Recipes

Without any annotations, an SGML/XML document can be used to generate a ESIS or grove-oriented object-model through MONDO. All elements are assumed to have a default parameter of 'content' which is a List and sub-elements will automatically be placed into that List.

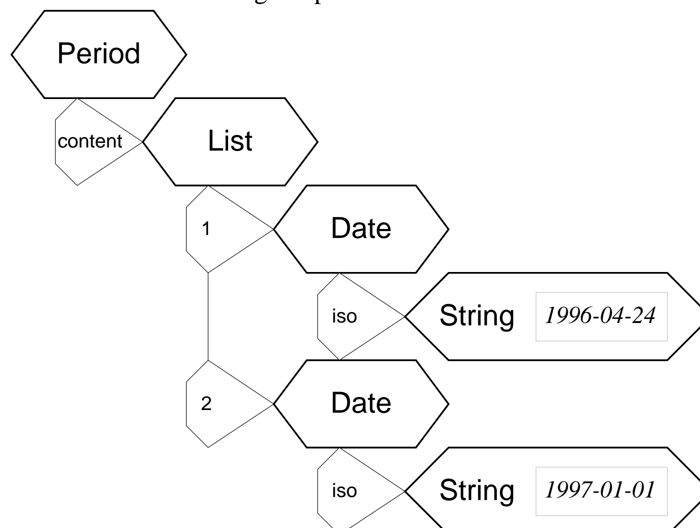
For example, with the following markup:

```
<Period>
    <Date iso='1996-04-24'>
    <Date iso='1997-01-01'>
</Period>
```

It would be assumed that '<Period>' has an automatic parameter of 'content' that would contain a collection of start-'<Date>' and end-'<Date>'. The conceptual markup would look something like the diagram:



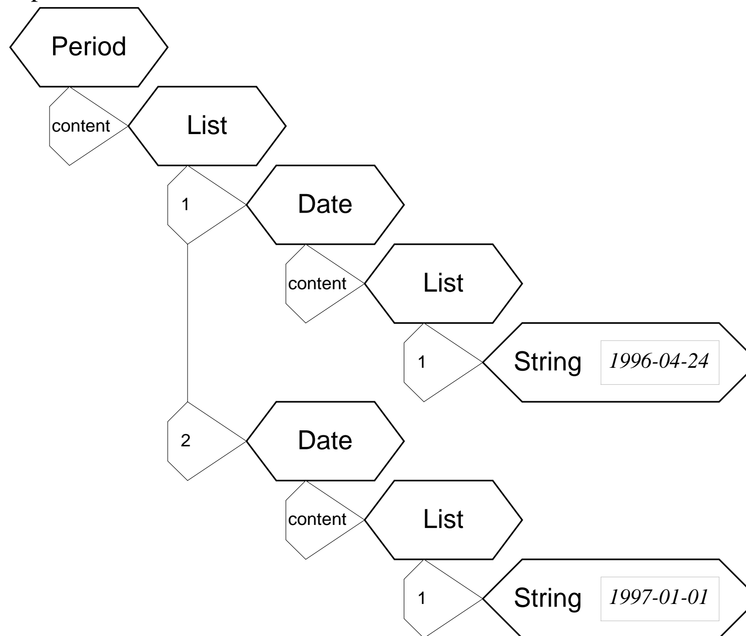
The diagram indicates that the 'content' parameter is implicit and will only apply if the children elements are not parameters themselves. The resulting recipe would be:



If Dates also used content for the date value:

```
<Period>
    <Date>1996-04-24</Date>
    <Date>1997-01-01</Date>
</Period>
```

You would get a recipe of:



This recipe shows the content-oriented containment hierarchy and the lack of more descriptive relationships and roles among the elements. Those relationships either have to be generated when building or processing the objects or may simply be unimportant for that processing.

More explicit information

We could more explicitly identify which date is the start and which is the end of the period by adding more tags:

```
<Period>
    <Start><Date iso="1996-04-24"></Start>
    <End><Date iso="1997-01-01"></End>
</Period>
```

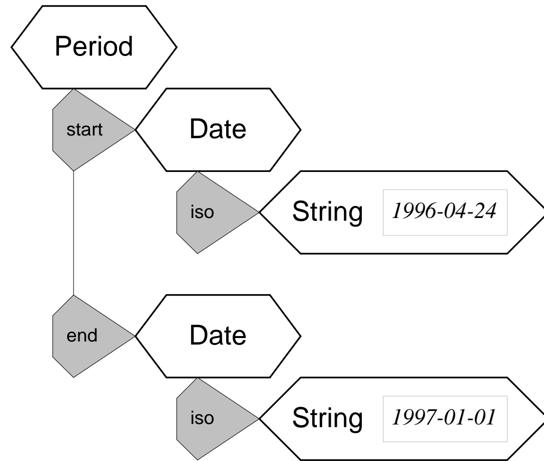
We could also have more explicit tags:

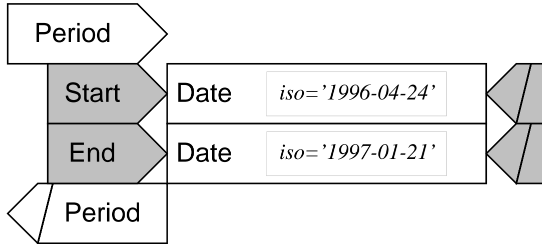
```
<Period>
    <StartDate iso="1996-04-24">
    <EndDate iso="1997-01-01">
</Period>
```

In either case the model is a bit more informative. This extra information is what can be used to transform the ‘content’ oriented data model into a more general object-oriented model. This is accomplished by annotating elements to identify them as parameters.

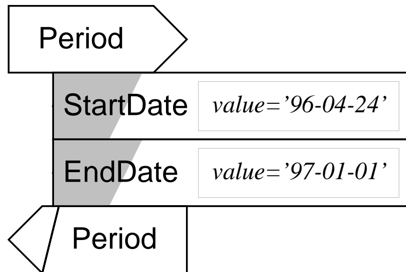
Parameter Annotations to enhance the Recipe

In the <Period> example with <Start> and <End> tags, we would like to specify that those two elements do not start building a new object (e.g. a 'Start' object), they just describe the role of the ingredient that will go into building the 'Period' object. We want to have the recipe to the right, which would require identifying the 'Start' and 'End' elements as parameters:





In a similar manner we would like the <Period> example which used 'StartDate' and 'EndDate' to produce exactly the same recipe, but for these cases we will need a single element to function as both a parameter identifier (e.g. 'start') and an object recipe (e.g. 'Date').



In both cases we need to provide enough information to the BuilderClient (the SGML application) so that it can distinguish the different types of elements. This can be done by attributes or naming conventions.

Annotation by Attribute

The simplest way to annotate an element is by adding an attribute. For example, we can specify the "parameter-ness" of the element with:

```
<Period>
    <Start mondoElementType="Parameter"><Date iso="1996-04-24"></Start>
    <End mondoElementType="Parameter"><Date iso="1997-01-01"></End>
</Period>
```

or much more conveniently, we can make it part of the document type:

```
<!ATTLIST (Start | End) %mondoParameter;>
```

Where we have predefined:

```
<!ENTITY % mondoElementTypes  "Object | Parameter | ParameterAndObject">
<!ENTITY % mondoParameter
           "mondoElementType (%mondoElementTypes) #FIXED Parameter"
>
```

And we do not need to modify the document at all. The information was already present, but we needed a way for the document to describe itself well enough that the application knew about that information*. Because we are annotating by attribute we can also use the approach of an architectural form.

---

* The name of the parameter is implicitly coming from the name of the element. This is acceptable for the moment but will need to be enhanced to support the possibility of element name conflicts for validation against a DTD.

Dual-Role Attribute Annotation

In the second, more explicit example we combined the parameter-ness and the object creation into a single element.

```
<Period>
    <StartDate iso="1996-04-24">
    <EndDate iso="1997-01-01">
</Period>
```

We need to provide the same identifying information:

```
<!ATTLIST (StartDate | EndDate) %mondoParameterAndObject;>
```

```
<!ENTITY % mondoParameterAndObject
           "mondoElementType (%mondoElementTypes) #FIXED ParameterAndObject"
>
```

but we also probably need to specify either the parameter name or the object type name. In this case we need to specify both.

```
<!ATTLIST (StartDate | EndDate)
    %mondoParameterAndObject;
    mondoClassName CDATA  #FIXED "Date"
>
<!ATTLIST StartDate  mondoParameterName CDATA  #FIXED "start">
<!ATTLIST EndDate  mondoParameterName CDATA  #FIXED "end">
```

Annotation by Naming Convention

Another approach is to use Element naming conventions to distinguish between Object and Parameter elements. This actually works very well now that XML has standardized on case-sensitive processing. Without case-sensitivity, we must distinguish parameters with something like an initial 'p.':

```
<Period>
    <p.start><Date iso="1996-04-24"></p.start>
    <p.end><Date iso="1997-01-01"></p.end>
</Period>
```

But now we can use a convention like Object tags are initial-uppercase and parameter tags are initial-lowercase. This is a well-established naming convention for OO information modeling and programming so it should be very familiar and is easy to remember. The above example would then become:

```
<Period>
    <start><Date iso="1996-04-24"></start>
    <end><Date iso="1997-01-01"></end>
</Period>
```

Note that unfortunately XML does not allow the empty end tag so you can not really shorten the above any further without combining tags.

Further automatic annotation

We could add the convention that a '.' can be used in a name to combine sequential "parameter.Object" tags. This would allow the above example to shrink to:

```
<Period>
    <start.Date iso="1996-04-24">
    <end.Date iso="1997-01-01">
</Period>
```

This has the advantage that it can avoid content model contentions over the name space within a DTD. For example, say one 'start' element takes a Date and another takes a position in a document. On the other hand, this has the disadvantage that it is connecting two logically separated (orthogonal) concepts, the naming of a parameter and the type/recipe of the ingredient.

The biggest advantage to using a naming convention is that it does not require a DTD to understand. Simple parsers can easily identify which tags are Objects/Recipes and which tags are parameters. The structure of the information is also more visible to the readers of the document. The biggest disadvantage to naming convention annotation is that most documents do not currently have all the tags necessary or the correct naming of all the tags. This requires modifying the documents themselves and that is far, far, less

desirable than modifying the DTD for those documents. However, this drawback does not exist for new documents.

Attributes merged with Parameters Elements

In SGML, content is supposed to contain the core information of a document and attributes should contain auxiliary information[*]. This is a difficult separation to make cleanly and recipes do not distinguish between types of parameters[†]. Normally an SGML BuilderClient will treat attributes as parameters just like parameter elements. Attributes are simply parameters that must have simple values/recipes. This causes the possibility of conflict, but the resolution is simple: the BuilderClient can warn about the conflict but will then add both parameter values to a list of values for the particular parameter name. Multiple parameters indicate a List of parameter values for that parameter.

Issues

Although the above approaches work well to allow SGML/XML be a front-end to MONDO, they have a couple difficulties relative to the simpler OML.

Parameter Element Names

Foremost is that specifying parameters is difficult because of DTD restrictions. The content model of an element is fixed independent of its context, so if we define a '<Start>' parameter element to contain a <Date>, we can not use 'Start' elements anywhere else they may be appropriate. This is very inconvenient and basically requires using a fuller path name: 'Period.Start' instead of just 'Start'. For non-validated (just well formed) XML documents this is not a problem. It may also be less of an issue with namespace proposals.

Validation

Another issue for using SGML with MONDO is DTD validation. DTDs provide rules for how elements can be put together. This effectively constrains and validates a 'recipe' to certain forms. The problem with validating a recipe is that the recipe can not be evaluated against the semantics of the information model. The later is what we really care about: Is the information valid? We only care whether the encoding/recipe is valid because it helps predict (earlier on) whether the information is valid.

By validating a recipe we will either over-constrain or under-constrain the possible information models. Under-constraining is not a major problem but it can lead to confusion of what the validation meant. Even after validating we will need to do a semantic validation within the ObjectBase. Over constraining[‡] is more problematic. Perfectly valid recipes to MONDO will be rejected. This can cause users to distort the information modeling to satisfy the parser. The result is a model and recipes (e.g. documents) which are harder to understand and use.

The only suggestion is to be careful with this. Validation of the parse/recipe is useful in its immediacy but is much less important than the correctness and expressiveness of the information itself.

Summary

SGML/XML works well as a recipe encoding language and front-end to the MONDO building process -- in spite of having some problematic restrictions. MONDO can take existing SGML documents as they are and generate content-oriented recipes. With a bit of annotation to the DTD or a standard naming convention, SGML documents can also produce arbitrarily complex recipes. The biggest issues are DTD validation vs.

---

[*] This is, at least, the most common way to distinguish content and attributes.

[†] The meta-information about a recipe could distinguish any number of properties about a recipe/object. But the stage this occurs is after parsing. "Understanding" a recipe is left up to the ObjectBuilder and the ObjectBase.

[‡] Either because the parser can not understand the information model (being separate from it) or because the parser has limited capabilities to express rules that the information model can. Generally both of these will occur for the Parser.

contextual reuse of parameter tags and some overhead associated with the markup, especially with XML-required named end-tags.

5.5 Example translations between OML and SGML

OML notation is used within most MONDO examples that require showing an encoding (this is rare because MONDO cares about the Recipe not the encoding). Because SGML-like notation is the standard and OML is slightly different, the following table gives example translations between the two. As mentioned above, there are multiple ways to annotate SGML documents to work with MONDO. The following translations will use the most directly equivalent version of SGML/XML, it will use full end tags (e.g. XML compliant), and it will use a naming convention of parameter elements starting with a lowercase letter. Finally, the translation will not take advantage of using attributes for simple parameter types (e.g. Strings) except in one example. This is just for consistency and to remind the user that any parameter can have complex data.

OML	**SGML/XML**
<Date>	<Date>
<Date iso=”97-10-27”>	<Date><iso>97-10-27</iso></Date> <Date iso=”97-10-27”> [alternative]
<Period start=<Date iso=”97-10-27”> end =<Date iso=”97-11-15”> Period>	<Period> <start><Date><iso>97-10-27</iso></Date></start> <end><Date><iso>97-11-15</iso></Date></end> </Period>
<P{This is some text with <EM{some important}> content}P>	<P>This is some text with <EM>some important</EM> content</P>
<Section title={Simple syntax comparison} body={OML Notation is used within most of the <&MONDO> examples…} >	<Section> <title>Simple syntax comparison</title> <body>OML Notation is used within most of the &MONDO; examples…</body> </Section>

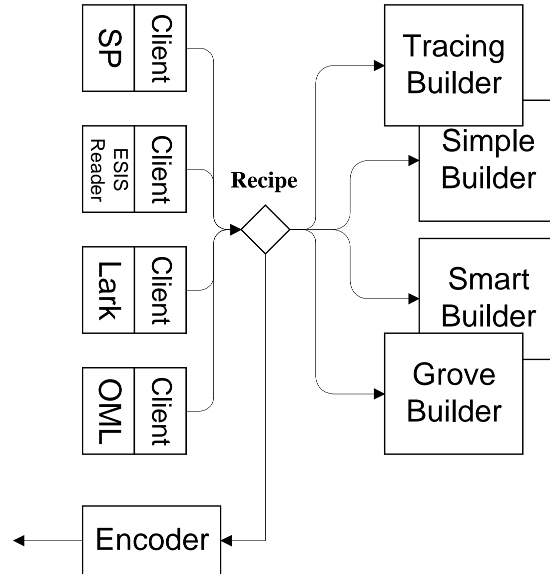
Note that the OML parser is just a parser that produces recipes and does no manipulation or replacements of the text or recipes. So the last OML example with the “<&MONDO>” reference is a little different from the SGML “&MONDO;”. The OML version will go to the ObjectBuilder to determine how to build a “&MONDO” recipe. The SGML version will be handled by the Parser and the ENTITY expanded before getting to the ObjectBuilder. So SGML for MONDO provides a textual macro expansion capability that OML does not provide. OML instead focuses on symbolic/recipe expansions that can only be handled by the ObjectBuilder.

A final difference is that the parameters in the SGML examples always have Lists as values. The OML parameters sometimes have just elements (“=<”) and sometimes have lists (“={“ or “=(“). Intelligence on the part of the BuilderClient or the ObjectBuilder would have to determine which SGML elements are Lists with a single value and which are really just single values.

5.6 Summary: Encoding Languages and Parsing

MONDO uses textual encoding as the primary mechanism to create, modify, and view recipes. Both OML and SGML/XML can easily encode recipes and the result of parsing either encoding (or any other encoding) will be exactly the same: a recipe. This uniformity allows many parsers, multiple encodings, and many builders to all be interchangeable. The recipe and ObjectBuilder interface provide the common bridge.

SGML/XML provides the more standardized approach to working with MONDO. SGML also has much more powerful parsers. OML provides a simpler and more direct approach but fewer tools and alternatives. Because all encodings are encodings of recipes, translating between any two encodings is trivial: just hook up a recipe builder (Parser and BuilderClient) of one type with a recipe encoder of the other type. So you should use whatever encoding you are comfortable with and suits the needs of the information you are working on.



6 Describing

7 Encoding

8 MONDO-L1

Beans, Introspection based factories and Pens,

9 MONDO-L2

Hypertext: (Addressing and Linking)
OQL support
Searching and filtering
DSSSL