

The cim22Grammar v0.2 Package

Pedro Assis
passis@dee.isep.ipp.pt

December, 2001

Abstract

This document intends to help the reader using the `cim22Grammar` package regarding both implementation issues and theoretical concepts. Background information is provided in sections 2 and 3 of this text. Section 2 provides an introduction to the Common Information Model (CIM) specification, and section 3 makes an overview of the ANother Tool for Language Recognition (ANTLR) fundamentals. The present version (0.2) of this package is composed by a main file—`cim22.java`—, a grammar file—`cim22Grammar.g`—, and some MS-DOS batch files. These files are described in section 4 and support the automation of some procedures, e.g. source files compilation and CIM Schema version 2.6 parsing execution.

Contents

1	Introduction	3
2	Common Information Model	3
2.1	CIM Specification	3
2.2	Metaschema	5
2.3	Naming	7
3	ANother Tool for Language Recognition	8
4	An Overview of the cim22Grammar Package	11
4.1	The <code>cim22Grammar.g</code> file	11
4.2	The <code>cim22.java</code> file	15
4.3	The batch files <code>run.bat</code> and <code>go.bat</code>	15

Revision History

- Revision 0.1, December, 2001, by Pedro Assis (*passis@dee.isep.ipp.pt*). Initial revision.

Acronyms

ABNF Augmented BNF
AST Abstract Syntax Tree
ANTLR ANother Tool for Language Recognition
BNF Backus-Naur Form
CIM Common Information Model
CIMOM CIM Object Manager
DMTF Distributed Management Task Force
DMTF MOF DMTF Managed Object Format
EBNF Extended BNF
IDL Interface Definition Language
IETF Internet Engineering Task Force
IM Information Model
IR Intermediate Representation
ISO International Organization for Standardization
JavaCC Java Compiler Compiler
LALR LookAhead Left Recursion
LL Left Lookahead
LR Left Recursion
MOF Managed Object Format
OMG Object Management Group
OO Object Oriented
PCCTS Purdue Compiler Construction Tool Set
RFC Request For Comments
UML Unified Modeling Language
WBEM Web-Based Enterprise Management
YACC Yet Another Compiler Compiler

1 Introduction

This document intends to help the reader using the `cim22Grammar` package regarding both implementation issues and theoretical concepts. Background information is provided in sections 2 and 3 of this text. Section 2 provides an introduction to the CIM specification, and section 3 makes an overview of the ANTLR fundamentals. The present version (0.2) of this package is composed by a main file—`cim22.java`—, a grammar file—`cim22Grammar.g`—, and some MS-DOS batch files. These files are described in section 4 and support the automation of some procedures, e.g. source files compilation and CIM Schema version 2.6 parsing execution.

With this package it is possible to parse all schemas based on the CIM Specification version 2.2(2) (i.e. CIM Specification version 2.2, June 14, 1999, plus Addenda 02) [4, 5], which at the present time means all CIM models available, notably the CIM Core, Common and Extension Models versions 2.4 up to 2.6.

The `cim22` Java class provides the means to implement lexical and syntactical analysis based on `cim22Grammar` rules. The `cim22Grammar.g` file is a complete CIM meta-model lexer and parser. This was developed using the ANTLR tool version 2.7.1. (It should be possible to run under earlier versions.) It makes use of the Augmented BNF (ABNF) grammar description of the DMTF Managed Object Format (DMTF MOF) syntax available in the Appendix A of [5]. This grammar was translated to Extended BNF (EBNF) notation (ANTLR compliant) and its implementation is LL(1) parseable ($k=1$).

2 Common Information Model

CIM can be characterized as an Object Oriented (OO) conceptual Information Model (IM) target to describe overall management information in the Internet, Intranets, and Extranets environments. With this initiative Distributed Management Task Force (DMTF) [13] underwent a major, and drastic, upheaval, trying to deliver a unified model, which enable the management data collection, storage and analysis using a common format. Being a conceptual model it is not bound to a particular implementation or vendor.

The main body of the CIM initiative is comprised of a specification plus a schema. Lets take a closer look at both parts.

2.1 CIM Specification

CIM Specification [4, 5] describes the DMTF MOF language, a naming mechanism, a metaschema, and mapping techniques to other management models. The approach taken was based on the development of an OO modeling meta-model inspired in the database world. Many researchers disagree with such hybrid approach, since it is not compliant with Object Management Group (OMG) modeling terminology. Although Unified Modeling Language (UML) notation

is used to describe CIM metamodel elements, CIM is not UML compliant¹. For that purpose distinct symbols are provided, in Appendix D of [5], for all metaschema elements except for the **Qualifier** construct (object semantic storage does not comply with UML). The metaschema (or in UML parlance, meta-model) is a formal description of the model itself. It builds on elements definition, syntactic and semantic issues, used to express: the schemas (or models), the details for integration with other management models, and the CIM naming mechanism (enterprise-wide object identification).

The schema part provides the actual models descriptions. CIM Schema [14] is structured in three layers.

- The bottom layer, or Core Schema, captures, in a unique model, general purpose concepts, mainly abstracts ones, that are common to all areas of management.
- The middle layer, or Common Schema, is a compound of different models each describing notions (concrete concepts) that are common to a specific management area, e.g. systems, applications, networks, etc.
- The top layer, or Extension Schemas, are otherwise technology and implementation dependent (developed by third parties), meaning that extensions must be provided concerning technology-specific management environment.

The CIM Specification and Schemas make use of an Interface Definition Language (IDL) oriented language called Managed Object Format (MOF). Actually the usage of IDL surname is rather overstate. IDL is somewhat more complete and powerful language that aims the Client-Object boundary description. Clearly there are some common points: both are textual declarative languages, purely used to make a specification description, and both provide platform independency.

The MOF syntax grammar is LL(1)-parsable², which description, using the ABNF [3], can be found in Appendix A of [5]. MOF syntax provides a way to describe object definitions in textual form. It establishes the syntax for writing, through a set of keywords and definitions based on a meta-construct set: **Class**, **Association**, **Property**, **Reference**, **Method** and **Instance** declarations, and their associated **Qualifiers**. Its parsing is case-insensitive, but in the construct naming context.

CIM Specification provides a naming mechanism to address enterprise-wide objects identification as well as sharing management information. Semantic analysis requires an explicit or implied namespace context against which the objects are validated. CIM naming uses CIM metamodel mechanisms (**KEY**) in a structured way enhancing this capability by introducing new qualifiers and

¹This rationale was, and still is, source of misunderstandings in the CIM world.

²LL(1) means a recursive-descent parser that processes, one symbol lookahead, the input rules from Left to right, and at each step makes a Left derivation of the constructed parse tree

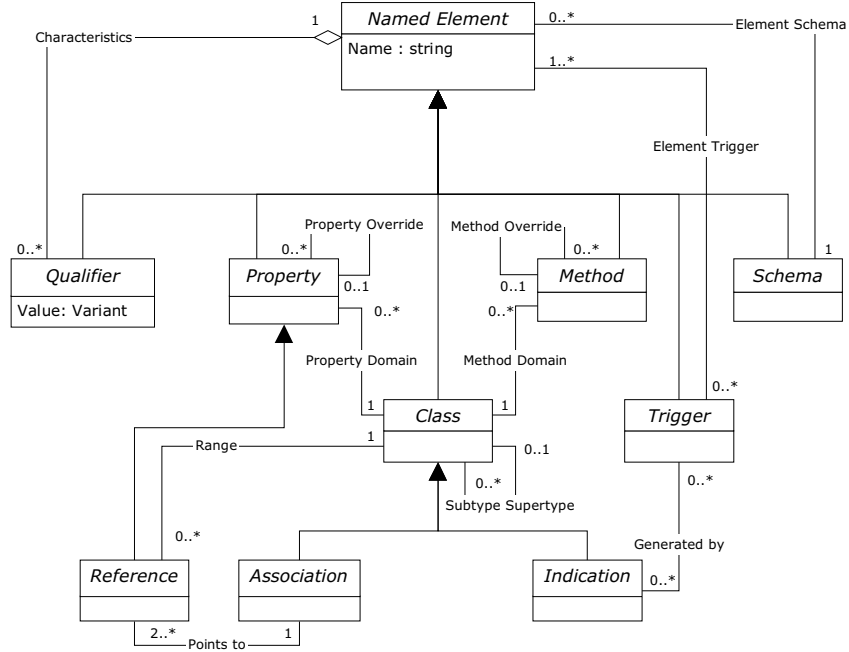


Figure 1: CIM metaschema structure.

pragmas: **WEAK** and **PROPAGATED** qualifiers, plus **SOURCE**, **SOURCETYPE**, **NONLOCAL** and **NONLOCALTYPE** pragmas/qualifiers.

2.2 Metaschema

The CIM metaschema is a formal definition of every meta-constructors used in the CIM-based schemas. These elements, displayed in Figure 1, reflect most of the OO design principles. As in the previous diagrams the UML notation is used to depict the structure of such metaschema.

The fundamental OO-based constructors described by the CIM metaschema are: **Schema**, **Class**, **Property** and **Method**. Also, specific mechanisms involved in the explicit mapping between these elements are supported, namely: **Association**, **Indication**, **Reference** and **Trigger**. Finally, the **Qualifier** directive provide further semantic to its associated element, it also offer the capability to extend the metaschema by the introduction of new qualifiers (e.g. user defined). This provides a flexible and unbounded mechanism to convey new semantic to the CIM models without compromising the compatibility with older versions.

In the following a brief introduction to the CIM meta-constructors is provided. Further information regarding this subject can be found in [2, 4, 5, 9, 14, 15].

- **Named Element** is the root (ancestor) of the CIM metaschema class hierarchy, meaning that all CIM meta-constructors descend from this class. Several qualifiers can be associated to every meta-constructor (represented by the **Characteristics** aggregation), providing information about its characteristics and consequently leveraging the semantic knowledge. This is a non-UML concept, but nevertheless a relevant one for its contribution towards a machine understandable management information schema. The **Element Trigger** association depicts the capability to create objects, named Indications, as a consequence of recognized events. And finally, the **Element Schema** association reflects the schema ownership over a particular named element.
- **Schema** is a collection of definitions in the context of a namespace and within an unit of ownership. It provides the class administration through a containment relationship.
- **Class** is a collection of instances, all of the same type (meaning the same properties and methods) which reflect a classification of a particular reality. CIM classes are always enumerable, but they do not have an algorithm to provide, at any time, its set members. The idea of a distinguish class as a set of its instances can be justified in light of the set theory³. Classes are arranged in a generalization hierarchy and do not support multiple inheritance, which means that a class can have only one superclass but many subclasses (this is emphasized by the **Subtype Supertype** reflexive association). Class is the domain, or containment, for property(ies) and method(s), represented in the metaschema diagram by the **Property/Method Domain** aggregations.
- **Property** denotes a characteristic of a class. More abstractly they are conventions that represent mappings between sets. As in the **Method** class, both have reflexive associations that establish override relationships (**Property/Method Override**) between properties (methods) from different classes. The overridden and overriding properties or methods classes (domains) must have a superclass/subclass relationship, otherwise it is forbidden.
- **Method** denotes a class behavior. Likewise property, a method has a signature and belongs to a specific domain (class container). Method inheritance only preserves its signature, it does not inherit its implementation. If a subclass does not provide a method implementation it is triggered a bottom-up search, towards the root class, seeking for an available implementation.
- **Association** is a class (with the **Association** meta qualifier) that describes a relationship between third party classes, making use of a variable

³This defines two ways to define a set: by extension, providing a comprehensive list of its members (class definition), or by intension, providing a clear criteria that enables the univocal identification of all its members (type definition).

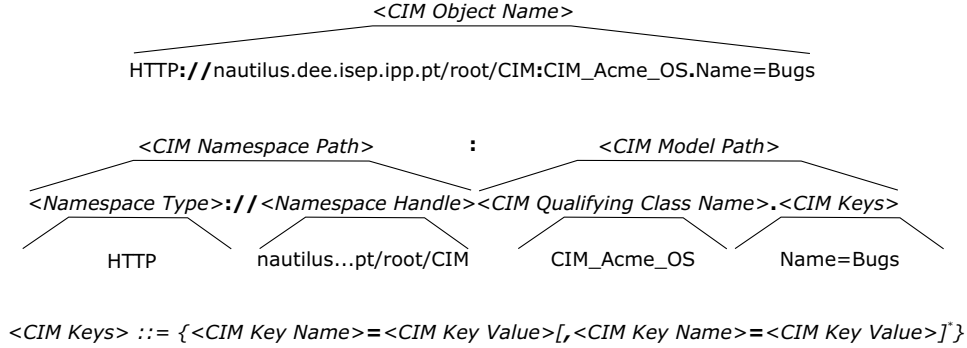


Figure 2: CIM object naming.

number (arity) of references (**Points to** aggregation). These references establish the roles of each party involved.

- **Indication** is a class (with the **Indication** meta qualifier) that describe the content, usually static data to fully describe its cause, delivered as a result of a trigger (**Generated by** association), i.e. registered event. A **Trigger** is an operation that is fired in recognition of specific event such as change of state (e.g. object creation, update, etc).
- **Qualifier** is a directive that is used to provide additional semantics for its associated element. There are four types of qualifiers: *meta*, *standard*, *optional* and *user-defined*. Qualifiers can be transmitted automatically between classes and their subclasses or instances. The transmission procedure is subject to certain rules know as flavors. The recognized flavors types are: **EnableOverride**, **DisableOverride**, **ToSubclass**, **Restricted**, and **Translatable**. The qualifiers flavors are not allowed to change within a namespace since this can lead to usage contradictions. They are established at the qualifier declaration (see `cim22Grammar qualifierDcl` rule).

2.3 Naming

CIM naming was defined to address enterprise-wide objects identification as well as sharing management information between applications. Semantic validation requires an explicit or implicit namespace context against which objects are validate. Regardless of the namespace implementation all classes instances must be distinguishable both within a single namespace and across namespace administrative boundaries. In general, it is necessary to establish mechanisms that provide object identity, application independency, referencing and integrating data from multiple sources, and also, data synchronization. These requirements justify the establishment of two component object identification based on the *namespace path* and *model path* concepts (Figure 2).

The *Namespace Path* provides access to a CIM implementation, i.e. CIM Object Manager (CIMOM) localization, based on two pieces of the namespace identifier: *Type*, type of implementation like access protocol; and *Handle*, which identifies a particular instance of the type of implementation. A namespace path can be specified for all the **Instance Of** statements using the **Source**, **SourceType**, **NonLocal** and **NonLocalType** *pragma*⁴ or by one of the several equivalent standard qualifiers (one for each statement). If used together the qualifier declaration overrides the general purpose *pragma*.

The *Model Path* provides full navigation within the CIM schema. This requires the usage of the object class name qualified by a unique combination of the *key* (also a non UML concept) properties values (**KEY** qualifier, see also 2.1). This mechanism enables the automated translation of object identification when it leaves its original space environment. How? The contents of the original namespace are exported creating a set of MOF files containing the desired view, subsequently these files are imported, loaded, into the target namespace, in which keys are guaranteed to be unique and retaining their original values. To deploy such mechanism it is required that the CIMOM is aware, properly understands, the key properties semantics.

CIM *weak association* implement a mechanism to name a particular instance within the context of another(s), but related, instance(s). **Weak** and **Propagated** standard qualifiers are the means to achieve key property propagation in the scope of a weak association (not UML compliant concept, this can be view as an UML extension) relationship. In this type of association one (and only one) of the roles is defined as being weak, which means that the keys of the scoping classes (other participants) are copied across to the weak side and marked as propagated. The key(s) property(ies) can be renamed but its contents remains the propagated from the originator instance. This situation is desirable when the reference class instances (weak side) identity depends on the identity of other classes instances.

3 ANother Tool for Language Recognition

Perhaps the simpler and probably most common way to write a formal language description is through the use of the Backus-Naur Form (BNF) standard. This notation is a formal meta-syntax used to express context-free grammars⁵, which is commonly known by its right trade-off between simplicity, compactness and description power. Over the time two modified versions have become increasingly popular—the IETF ABNF [3], and the ISO EBNF [7]—replacing in many cases the original form (although a BNF equivalent representation must be al-

⁴Keyword compiler directive, preceded by the hash (#) character.

⁵As described in section 2.2 of the “The Dragon Book” [1] a *context-free grammar* is a formal system that describes a language by specifying how any legal text can be derived from a distinguished symbol called the *axiom*, or sentence symbol. It consists of a set of *productions*, each of which states that a given symbol can be replaced by a given sequence of symbols; a set of tokens, known as terminal symbols; a set of nonterminals; and the designation of *start* symbol from the nonterminal set.

ways possible to achieve). The usage of EBNF or ABNF improves the grammar readability as strict BNF do not accept subrules, closures, nor optimal items.

The widespread application development based on the UNIX *Lex* and *Yacc* [8] programs contributed to the BNF popularity. Besides several flavors of the *Lex* and *Yacc* based toolkit (e.g. *Bison* and *Flex*), which usually generate C language code, it is worth mention the availability of some other free software tools like the Java Compiler Compiler (JavaCC) and ANTLR. The ANTLR (version 2.7.1) [12], former Purdue Compiler Construction Tool Set (PCCTS) component, is a language tool—predicated Left Lookahead (LL) parser generator—that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions (capable of automatically generating lexers and parsers for these languages). The ANTLR characteristics and parsing strategy offer adequate tools to generate recursive-descendent parsers, which tackle particularly difficult parsing problems, namely those that require context sensitive or large amounts of lookahead.

Written in Java, it combines the lexical analyzer and parser specifications into a unique file, and it is capable of generating Abstract Syntax Tree (AST), tree walking classes, and interpreting semantic and syntactic predicates. The latest feature is very handy to achieve parsing fine-grained control since predicates explicitly direct production validation and are executed while guessing. ANTLR fully supports EBNF notation according to the “|”, “?”, “*”, and “+” operators. These characteristics, and others, are fully described in ANTLR’s on-line manual [12] and related publications [6, 10, 11].

Some of the ANTLR’s key features are associated with the grammar design easiness and flexibility, as the standardize syntax for specifying lexers, parsers, and tree parsers; the production allowed elements, and the actions statements placement.

Although lexical analysis is not required in ANTLR projects, it is common to have token identification within the lexer (scanner) and eventually actions associated with them. An ANTLR production can include lexical regular expressions⁶ placed in double-quotes (Listing 1) without being necessary its definition in the lexer specification.

ANTLR productionsTo increase the design expressiveness power the ANTLR’s rules support semantic and syntactic predicates. This means the possibility to use arbitrarily large values for lookahead with the assurance that ANTLR always generate the required minimum value. The actions in these rules are general purpose statements, written in the target language, which can be inserted almost anywhere in the ANTLR production. These actions describe the user program purpose and as such it does not make sense to classify its contents, but only if these actions initialize the production local variables (called in ANTLR parlance *init-action*). As common OO class methods (even specialized methods like constructors) and non-OO languages functions also productions can import

⁶I.e. tokens identified as non-protected lexer rules. The protected ANTLR concept is similar to the one applied in the Java class members access control context (i.e. public, protected and private control modifiers). These rules are helpers towards the non-protected ones and are referenced by the latest.

Listing 1: Lexical regular expressions usage in parser rules.

```
1 flavor returns [String cimFlavor]
2 {
3   cimFlavor = null;
4 }
5 : ( "enableoverride"
6     { cimFlavor = new String ("enableoverride"); }
7   | "disableoverride"
8     { cimFlavor = new String ("disableoverride"); }
9   | "tosubclass"
10    { cimFlavor = new String ("tosubclass"); }
11  | "restricted"
12    { cimFlavor = new String ("restricted"); }
13  | "translatable"
14    { cimFlavor = new String ("translatable"); }
15  )
16  {
17    ++cim22.parserTraceLine;
18    if (cim22.parserRuleTrace)
19      cim22.mofOut.println ("Parser rule trace\tt#"
20        + cim22.parserTraceLine + "\t: flavor ("
21        + cimFlavor + ")");
22  }
23 ;
```

(have arguments) and export (return) values (Listing 2).

Error reporting and recovery are the last of the ANTLR's key features mentioned here. Both lexer and parser exceptions are subclasses of the `ANTLRException` class, and comprehend three major exception handling classes related to: the lexer character input stream (`CharStreamException`); the token stream input (`TokenStreamException`), which can be thrown by both lexer and parser; and the `RecognitionException` that handles generic recognition problems.

Both error reporting and recovery can be fully customized by specifying the exception handlers to be used, although default handlers are provided automatically. In what concerns the reported messages (Listing 3), they can be redirected from the default output stream (see overloading ANTLR generated parser methods, e.g. `reportError()`), and its contents can be tailored according to the target rule semantics to provide richer debugging information (see `paraphrase` action option). The automatic error recovery strategy is to consume the tokens seeking the synchronization with the *FOLLOW*⁷ set of the specified parser rule. Specific exception handlers can be attached to rules, alternatives and labeled elements. Although this fine-grained exception handling control level is possible, errors in alternatives and labeled elements will not break the rule processing.

⁷The set of input symbols that may follow any reference, scoping the entire grammar, to a particular rule.

Listing 2: *ANTLR* parser rule for discriminating the country locale.

```
1 cLocale returns [String cimCountryLocale] {
2   // Default value
3   cimCountryLocale = new String ("en_US");
4 }
5 : ( lbl_1:STRING.VALUE
6   {
7     cimCountryLocale = lbl_1.getText ();
8   }
9 )
10 {
11   // Global variables declared in the
12   // cim22 main file
13   ++cim22.parserTraceLine;
14   if (cim22.parserRuleTrace)
15     cim22.mofOut.println ("Parser rule trace\t#\"
16       + cim22.parserTraceLine + "\t: cLocale (\"
17       + cimCountryLocale + "\")");
18 }
19 ;
```

Listing 3: *ANTLR*'s error reporting customization.

```
1 public void reportError(RecognitionException ex) {
2   ++cim22.parserErrorLine;
3   cim22.mofOut.println ("Parsing ERROR trace\t#\"
4     + cim22.parserErrorLine + "\t: Source file \" + ex);
5 } // public void reportError(RecognitionException)
```

4 An Overview of the cim22Grammar Package

At present time (version 0.2 of December, 2001) the `cim22` package contents include: a grammar file named `cim22Grammar.g` (*ANTLR* based), a Java main file named `cim22.java`, and two MS-DOS batch files named respectively `run.bat` and `go.bat`.

The following points provide a brief overview of each of these files, highlighting some of its relevant issues.

4.1 The `cim22Grammar.g` file

This grammar provide a full compliant CIM version 2.2(2) (CIM Specification version 2.2, June 14, 1999, plus Addenda 02) [4, 5] meta-model lexer and parser, based on an *ANTLR* 2.7.1 grammar. It should be possible to run this package under *ANTLR*'s earlier versions. For this purpose edit the Java main file (`cim22` class) and remove the comments under 2.6.0 version related blocks and comment the 2.7.1 version blocks. These code blocks are strictly related to exception handling aspects in both versions.

Listing 4: ANTLR’s lexer options section.

```

1 class cim22Lexer extends Lexer;
2 options {
3     k=2;
4     caseSensitive=false;
5     caseSensitiveLiterals=false;
6     charVocabulary='\3'..'\'377';
7 }

```

The lexer and parser productions are preceded by a *header section* containing source code that is placed before any ANTLR generated code. Common statements in this section are the traditional Java import package declarations. Note that comments are not allowed before the *header section*, which is required to be on the top of the grammar file.

Each parser, lexer, and tree-parser class definitions can be customized by command-line arguments specified in the source code—*options section*—rather than on ANTLR invocation (`java antlr.Tool cim22Grammar.g -D<name> = <value>`). Well, as a matter of fact they can be specified for any element reference, i.e. within the production rules. Examples of such customizations are described in the followings listings. Listing 4 shows the specified lexer lookahead (line 3); line 4 and 5 customizes the case handling for characters and literals respectively, setting both options to **false** means that character and literal handling is case insensitive, but in the first case it also means that the case is preserved (this is required by the CIM specification); line 6 describes the allowed character set: ASCII 3 to 255 in octal notation (lexer rules can specify additional valid characters). Listing 5 describes the usage of options declaration inside production rules. This particular case is a classical example that turns off the ANTLR warning issues related with the **protected ESC** rule alternatives. Others typical examples are available in the ANTLR on-line manual (e.g. greedy/nongreedy rules).

The *tokens section* is used to both declare literals and define the so called “imaginary”⁸ tokens. Whether the literals, present in both lexer and parser, are declared within the *tokens section* or referenced in the productions they all must be stored in the lexer hash table. In order to be able check the input stream symbols against the hash table literals it is required to have specified a matching pattern (see lexer rules **IDENTIFIER**, **STRING_VALUE**, etc) according to the literal type. In what concerns the “imaginary” tokens usage they are mainly related to the tree construction, e.g. mark subtree boundaries (this point is described in detail in the AST chapter of the ANTLR’s on-line manual).

The **cim22Grammar** validation was made in two steps, first by the individual test of all MOF meta-constructor (CIM Specification version 2.2(2)) grammar rules, this stand-alone test pattern enabled the individual parsing detailed anal-

⁸Well, this is the ANTLR author—Terence Parr—definition. *Virtual* seems, to the present author, to be a good candidate.

Listing 5: ANTLR’s rule alternate options usage.

```

1 protected
2 ESC
3   : '\\',
4     ( 'n',
5       | 't',
6       | 'v',
7       | 'b',
8       | 'r',
9       | 'f',
10      | 'a',
11      | '\\',
12      | '?'
13      | SINGLE_QUOTE
14      | DOUBLE_QUOTE
15      | ('0' | '1' | '2' | '3')
16      | ( options {warnWhenFollowAmbig = false;}
17        : OCTAL_DIGIT
18          ( options {warnWhenFollowAmbig = false;}
19            : OCTAL_DIGIT)?
20          )?
21      | 'x' HEX_DIGIT
22      | ( options {warnWhenFollowAmbig = false;}
23        : HEX_DIGIT)?
24    )
25 ;

```

ysis, and second by a full compliance validation against the current version of the CIM model (2.6). Actually, tests were carried a little further to check the syntax of DMTF MOF files comprising all the CIM models from version 2.4 up to the 2.6 version.

Aiming to study alternatives rules to implement MOF error pattern detection some variations of the grammar rules (which are not conformant with the DMTF MOF standardized ABNF syntax grammar) were implemented with some interesting results. Diverse types of errors were found, and in some cases the same error have been propagated, at least, since the earlier tested CIM model (version 2.4). With this approach it were possible to attain some useful results, as several errors were identified and reported back to the DMTF steering groups.

Although the majority of the detected non-conformances were probably due to editing mistakes⁹, their detection was not as trivial as the reader may think (take in consideration that, in some cases, these models are the final DMTF specifications, and they successfully passed the DMTF people and their validation tools).

To give the reader a simple and paradigmatic example just considerer the array initializer rule parsing (`arrayInitializer`). According to the CIM spec-

⁹The CIM models MOF source files tend to be a little verbose: 1.6 Mbyte for the 2.6 version.

Listing 6: `arrayInitializer` rule.

```

1 arrayInitializer
2   : LCURLY
3     ( constantValue
4       | (STRING_VALUE)+
5     )
6     (COMMA ( constantValue
7             | (STRING_VALUE)+
8           )
9     )*
10    RCURLY
11    {
12      ++cim22.parserTraceLine;
13      if (cim22.parserRuleTrace)
14        cim22.mofOut.println (" Parser rule trace\t#"
15      + cim22.parserTraceLine + "\t: arrayInitializer");
16    }
17  ;

```

ification its parsing is quite straightforward:

“Arrays can be defined to be of type *Bag*, *Ordered* or *Indexed*, and can be initialized by specifying their values in a comma-separated list (as in the C programming language). The list of the array elements is delimited with curly brackets.” (From CIM Specification v2.2 [5], page 45.)

Running a somewhat modified version the `cim22Grammar` v0.2 package against the `CIM.Device24` MOF description, the following error was issue in the context of the `PaperSizesSupported` property of the `CIM.Printer` class:

Parsing ERROR trace #1: Source file line 1337: expecting RCURLY, found “A10”

The reason for this error message is the lack of a comma between the “A9” and the “A10” array initializer values. This is quite difficult to detect as this could also mean the valid “A9A10” array initializer value. The regular parser would try to match the subrule (STRING)+ (Listing 6, lines 4 and 7), inside the `arrayInitializer` rule, leading to the second form (“A9A10”) recognition. In the present CIM specification this is (with automatic syntactic/semantic analysis) hard to differentiate from the desirable situation. (Probably this can be the reason for this systematic error in all tested models.)

Since the forms—“A9”, “A10” and “A9”_“A10”— are both valid entries, concerning the array initializer rule, maybe it would be advisable to use another delimiter to support this case in particular. Although the author is not in favor of special cases, the following could help to detect malformed expressions for this particular case.

Values {“Unknown”, “Other”, {“A very long sentence”
“spanning multiple lines”}}, {“or just for” “fun!”}, “B” “C”, “D”}

This would solve the present problem as the lack of comma, outside the second level of {}, would be easily detected as a violation. The absence of a { or } is a classical error addressed by many grammars (e.g. C block delimiters).

The present grammar does not include any of studied MOF error pattern detection variations (this is foreseen for the next version). In what concerns the reported error the present version fully supports the presence of multiple strings in the array initializer values, so this error in particular will not be detected. In order to run tests against these type of error simply make the changes suggested in the header of the `arrayInitializer` rule (line 960 of the `cim22Grammar.g` file).

4.2 The `cim22.java` file

The `cim22` class provides general purpose methods, e.g. `programUsage` and `doFile`, initialization of local and global variables, e.g. lexer and parser trace lines and the output print stream, and the invocation of the generated ANTLR lexer and parser in the `scannerFile` method.

The `cim22` `main` method checks the input arguments syntax and in case of error the `programUsage` method is called; initializes some of the control variables, and feeds the `doFile` method with the name of the target file or directory (if it exists). At program exiting the parsing statistics are displayed on the screen.

The `doFile` recursive method manages the invocation of the `scannerFile` method for each valid MOF file implied in the passed argument. From the original `cim22` input list arguments either the name of the target MOF file or the name of the directory top level tree is feed to this method. If the name matches a directory then the entire hierarchy is scanned seeking for the presence of MOF files. (No doubt that the method recursive nature is of great advantage in this particular.) The recursive ending point condition is related to the directory file search completion regardless of its success (each directory contents are stored in a array, hence the termination point is well defined).

The `scannerFile` method invokes the ANTLR generated lexer and parser for the MOF file specified. In case of a parsing error occur an output file is created at the same location as the input file and named after this one with the “p_” prefix and “.txt” suffix. This method catches most of the ANTLR specialized exceptions and throws the general purpose ones.

4.3 The batch files `run.bat` and `go.bat`

Make use of the `run.bat` MS-DOS batch file to compile the `cim22Grammar` package, and the `go.bat` to execute some parsing trials. Both batch files implement some integrity tests before running to ensure the package proper usage. The `go.bat` batch file offers two kinds of tests based on CIM Schema version 2.6,

namely the stand alone file and directory hierarchy parsing. The user selects the desired option at the MS-DOS command line shell.

References

- [1] A. Aho, R. Sethi, and J. Ulman, *Compilers: Principles, Techniques, and Tools*, first ed., World Student Series, Addison-Wesley, 1986.
- [2] A. Keller, H. Kreger, and K. Schopmeyer, *Towards a CIM Schema for RunTime Application Management*, 12th International Workshop on Distributed Systems: Operations and Management (DSOM'2001) (Nancy, France) (O. Festor and A. Pras, eds.), IEEE, October 15-17 2001.
- [3] D. Crocker and P. Overell, Eds., *Augmented BNF for Syntax Specifications: ABNF*, IETF Network Working Group, RFC 2234, November 1997.
- [4] DMTF, *Common Information Model (CIM) Specification*, Version 2.2, Addenda (Draft Version 2), DSP0007.
- [5] DMTF, *Common Information Model (CIM) Specification*, Version 2.2, June 14 1999.
- [6] G. Schaps, *Compiler Construction with ANTLR and Java: Tools for building tools*, Dr. Dobbs' Journal (1999), no. 3, 84–89.
- [7] ISO/IEC, *Information technology—Syntactic metalanguage—Extended BNF*, ISO/IEC 14977:1996, 1996.
- [8] J. Levine, T. Mason, and D. Brown, *lex & yacc*, second ed., A Nutshell Handbook, O'Reilly & Associates, Inc., February 1995.
- [9] J. Strassner, *Directory Enabled Networks*, first ed., Technology Series, Macmillan Technical Publishing, 1999.
- [10] T. Parr, *An Overview of SORCERER: A Simple Tree-Parser Generator*, International conference on Compiler Construction, April 1994.
- [11] T. Parr and R. Quong, *ANTLR: A Predicated-LL(k) Parser Generator*, Software-Practice and Experience **25** (1995), no. 7, 789–810.
- [12] The ANOther Tool for Language Recognition Home Page, <http://www.antlr.org/index.html>.
- [13] The Distributed Management Task Force Inc. Home Page, <http://www.dmtf.org/>.
- [14] The DMTF Common Information Model (CIM) Standards Home Page, http://www.dmtf.org/standards/standard_cim.php.

- [15] W. Bumpus, J. Sweitzer, P. Thompson, A. Westerinen, and R. Williams, *Common Information Model: Implementing the Object Model for Enterprise Management*, first ed., Wiley Computer Publishing, John Wiley & Sons, 2000.

Index

- ABNF, 4
- ANTLR error recovery, 10
- ANTLR error reporting, 10
- ANTLR header section, 11
- ANTLR init-action, 9
- ANTLR lexical analysis, 9
- ANTLR options section, 12
- ANTLR tokens section, 12

- BNF, 8

- CIM, 3
- CIM association, 6
- CIM class, 6
- CIM Common Schema, 4
- CIM Core Schema, 4
- CIM Extension Schemas, 4
- CIM indication, 7
- CIM Metaschema, 5
- CIM method, 6
- CIM model path, 8
- CIM named element, 6
- CIM namespace path, 8
- CIM object naming, 7
- CIM property, 6
- CIM qualifier, 7
- CIM Schema, 4, 6
- CIM Specification, 3
- CIM weak association, 8
- cin22 `doFile` method, 15
- cin22 `scannerFile` method, 15
- cin22 main, 15
- cin22Grammar error detection, 12
- cin22Grammar.g description, 11

- EBNF, 3

- go MS-DOS batch file, 15

- IDL, 4
- IETF ABNF, 8
- IM, 3
- ISO EBNF, 8

- LL(1) grammar, 4

- run MS-DOS batch file, 15