Clone Detection in Source Code by Frequent Itemset Techniques

Vera Wahler, Dietmar Seipel, Jürgen Wolff v. Gudenberg, and Gregor Fischer
University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{wahler,seipel,wolff,fischer}@informatik.uni-wuerzburg.de

Abstract

In this paper we describe a new approach for the detection of clones in source code, which is inspired by the concept of frequent itemsets from data mining.

The source code is represented as an abstract syntax tree in XML*; we can use different kinds of* XML*–languages for different programming languages, such as Java, C++, or* PROLOG*. Our approach is very flexible; it can be configured easily to work with multiple languages.*

1. Introduction

A frequently observed phenomenon in larger software projects is that certain pieces of code are copied and slightly modified for reuse. A reason for this is the wish of the programmer to take advantage of existing software components. This is not a bad strategy, since components, that are already used, are well tested and known to work. Moreover, in most situations copying the required code fragments is the fastest way to software reuse. Methods for a better encapsulation of the code require a considerable amount of time. In most situations it is hard to produce a working version of the software in time and there is no time left for improving the code.

The handling of duplicated code can be very problematic in many respects. An error in one component is reproduced in every copy. Since it is not documented in which places duplicates can be found, it is extremely hard to find and remove such errors. The maintenance of existing software becomes much more complicated and costly. Moreover, blowing up the code reduces the level of abstraction of the code, which is highly undesirable, since it becomes much harder for others to get acquainted with the code, e.g. for adding new functionality.

It is very likely, that one cannot avoid the occurrence of clones due to the programming with copy and paste. Thus, it is a good strategy to provide tools for the program developer for finding duplicates and for preparing reports about them.

Various attempts have already been made to finding duplicates in software projects. However, most of the known techniques are restricted to analysing certain programming languages. Tests have shown [4] that none of the existing approaches produces optimal results in any case.

We have developed a new method based on the concept of frequent itemsets that works on an XML–representation of the program [6]. Hence, our method could be extended for searching clones in general tree structures, since we work on a more abstract level.

The rest of the paper is organized as follows: After presenting some basic definitions regarding clones in Section 2, we recapitulate some related work on clone detection in Section 3. In Sections 4 and 5 we introduce our new approach to detecting clones, which is inspired by the concept of frequent itemsets, including the XML–representations for the programs and the configuration data. Finally, two case studies are reported in Section 6.

2. Basic Definitions

Code Fragments. A code fragment is a contiguous piece of source code, i.e. one or more successive lines. In our approach one simple statement, which usually corresponds to one line of code, is the smallest unit of measurement.

Different Types of Clones. A clone is a copy of a code fragment. Usually, clones consisting out of more than 5 statements are considered interesting. Since the clone relation is symmetric we better say that the origin and the copy form a clone pair.

To further characterize different types of clones we adopt the notation of [4]: A textual copy where only formatting white space or comments may be changed is called a clone of type 1.

More interesting are clones of type 2, where parameters or variables may have different types or names.

We consider binary expressions as clones, if both operands form an expression clone pair. This recursive definition is anchored on the fact that two different variables or constants are taken as clones, respectively, whereas simple variables are distinguished from array accesses or constants. E.g., the following assignment statements `t = a[0]` and `max = a[2]` are clones of type 2.

Finally, in clones of type 3 one or more statements may be inserted or deleted. Although these are the most common clones, they are very hard to detect. Most of the current clone detection mechanisms including ours cannot deal with clones of type 3.

3. Related Work

According to [15] clone detection techniques can be divided into string–based, token based or parse tree–based.

Since *string–based* methods like the ones of Baker or Rieger [1, 5] usually work on the source code directly, they are quite general and may be applied to various languages. On the other hand, the semantics of the underlying programming language is completely ignored. These methods need no internal data structure, but [1] uses p–suffix–trees for optimizing the pattern matching.

Kamiyas approach [10] approach is *token–based*. The sequence of tokens is produced by a scanner and it becomes transformed to a new sequence of tokens by language specific transformation rules and by the replacement of parameters. The last step is the comparison of the possible substrings, and the result is a set of clone pairs.

The *parse tree* or *abstract syntax tree* (AST) contains the complete information about the source code. Hence more sophisticated methods for the detection of clones can be applied. Baxter [3] generates an annotated parse tree by a compiler generator and then searches for clones. Krinke [11] has chosen program dependency graphs (an extension of ASTs) as the internal format. He uses an iterative approach for detecting maximal similar subgraphs. Additionally, Merlo uses metrics, which are applied to an AST–representation to find clones. His unit for measurement are the bodies of functions. A similar technique is described in [14]. Our approach is also parse tree–based. However, we use an XML–representation, and thus we add one level of abstraction.

A comparison of the methods known from literature has shown that so far there exists no single method that is superior to all other methods in all situations [4, 15]. All approaches have certain advantages and disadvantages. Techniques that detect many clones (high recall) also return many code fragments which are no clones (lower precision). In turn, techniques with a high precision will usually have a lower recall.

A major deficiency in all approaches is that too few clones are detected [4]. Most approaches can find clones of type 1 more easily than clones of type 2. Kamiya, Krinke, Merlo, and Rieger mention that their approaches can also find clones of type 3, but in practise only Krinke's approach does. In his approach, however, the clones of the other types are found with a very low recall, and the running time is exponential. Thus, clones of type 3 cannot be larger than a certain size.

4. Finding Clones as Frequent Itemsets

Starting with the XML representation of the AST we generate our initial database and we employ a link structure and a hash table to speed up the data access. The second step is the application of the algorithm for finding frequent itemsets. Figure 1 shows the overall view of the approach.

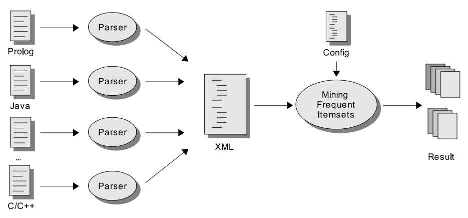

Figure 1. Overall Architecture

4.1. Frequent Itemsets in Data Mining

In data mining [8] frequent itemsets are used to illustrate relationships within large amounts of data. The classical example is the analysis of the buying–behaviour of customers. The database consists of a set of transactions, and each transaction is a set of items (individual articels) from a universal itemset I^*.

The goal is to find itemsets I that are subsets of many transactions T in the database D ($I \subseteq T$). An itemset is called *frequent*, if it occurs in a percentage that exceeds a certain given support count σ:

$$\sigma(I) = \frac{|\{\, T \in D \mid I \subseteq T \,\}|}{|D|} \geq \sigma.$$

An itemset consisting of k items is called k–itemset.

The method for finding frequent itemsets is iterative. First, the set $L_1 = \{\, I \subseteq I^* \mid |I| = 1 \wedge \sigma(I) \geq \sigma \,\}$ of all frequent 1–itemsets is constructed. The generation of L_k from L_{k-1} is divided into two steps: In the *join step* the set C_k of all the possible candidates for L_k is build by combining itemsets from L_{k-1} that overlap in $k-2$ items:

$$C'_k = \{\, I_1 \cup I_2 \mid I_1, I_2 \in L_{k-1} \wedge \; |I_1 \cap I_2| = k-2 \,\}.$$

C_k contains all itemsets $I \in C'_k$, such that all $k-1$–subsets I' of I are in L_{k-1}. This construction is allowed due to the so–called apriori–property, which says that all subsets of frequent itemsets have to be frequent, too. During the *prune step* the frequent k–itemsets are selected from C_k:

$$L_k = \{ I \in C_k \mid \sigma(I) \geq \sigma \}.$$

This process is repeated until $L_k = \emptyset$, i.e. until no frequent k–itemsets are found. Every itemset $I \in L_k$, such that no extension of I is in L_{k+1} is a maximal frequent itemset.

4.2. Frequent Itemsets for Clone Detection

A program consists of statements, which may be structured; they form the items in the database D. Structured statements contain additional statements. XML–configuration files define how to proceed with those structured statements; in the following section these configuration files are explained in more detail.

As a refinement of the original algorithm in clone detection a k–itemset can only consist of k consecutive statements. Based on the link structure in the initial database consecutive statements which represent itemsets can be found very fast. Obviously, clones are sequences which occur in at least two places in the program. Thus, clones correspond to frequent itemsets for the support count $\sigma = 2/|D|$.

Given k consecutive statements $b_1, \ldots, b_k$ in our program. The straighforward *join step* would combine two frequent $k-1$–itemsets of the form

$$I_1 = \{ b_1, \ldots, b_{k-1} \},\ I_2 = \{ b_2, \ldots, b_k \}.$$

For clone detection we found out that it is much more efficient to use a modified join step, which combines a frequent 1–itemset $I_1 = \{ b_1 \}$ with a frequent $k-1$–itemset $I_2 = \{ b_2, \ldots, b_k \}$. If b_1 is the header of a nested expression, then we allow this join only if I_2 contains the complete body of this nested expression.

As an extension, we can compute clones of type 2 which occur with a given minimal support count $\sigma > 2/|D|$.

Example. Consider the following code fragment, which determines the maximum `max` of an array `a[0..2]` with three values by conditionally exchanging pairs of values:

```
 1   if (a[0] > a[1]) {
 2      t = a[0];
 3      a[0] = a[1];
 4      a[1] = t;
 5   }
 6   if (a[1] > a[2]) {
 7      t = a[1];
 8      a[1] = a[2];
 9      a[2] = t;
10   }
11   max = a[2];
```

The statements in lines 1, 2, 3, 4, 6, 7, 8, 9, and 11 form frequent 1–itemsets. They form 4 clone classes: $\{ 1, 6 \}$, $\{ 2, 7, 11 \}$, $\{ 3, 8 \}$, and $\{ 4, 9 \}$. Observe, e.g., that 2 and 3 are no clones of each other, since the variable `t` is no clone of `a[0]`. The frequent 2–itemsets are $\{1, 6\}$, $\{2, 3\}$, $\{3, 4\}$, $\{7, 8\}$, and $\{8, 9\}$. The frequent 3–itemsets are $\{2, 3, 4\}$ and $\{7, 8, 9\}$. E.g., $I = \{2, 3, 4\}$ is built by combining the frequent 1–itemset $I_1 = \{2\}$ with the frequent 2–itemset $I_2 = \{3, 4\}$. The frequent 3–itemsets subsume the frequent 1– and 2–itemsets, i.e. the contained 1– and 2–itemsets are removed from the result. Finally, the frequent 4–itemsets are $\{1, 2, 3, 4\}$ and $\{6, 7, 8, 9\}$; they are the result of the algorithm, since they subsume the 3–itemsets.

4.3. Implementational Aspects

We have also investigated to what extent XML–query languages such as XQuery can be used in our algorithm for clone detection, since we store Java source code in XML–files in JAML format [6]. In one version of our tool the generation of relevant subtrees from the DOM–tree is done by using the tool Galax [7] for posing XML–queries to JAML–files. However, subsequent test cases have shown that this solution is not fast enough. Moreover, it turned out that we did not need to select subelements of an XML–document using complex path expressions.

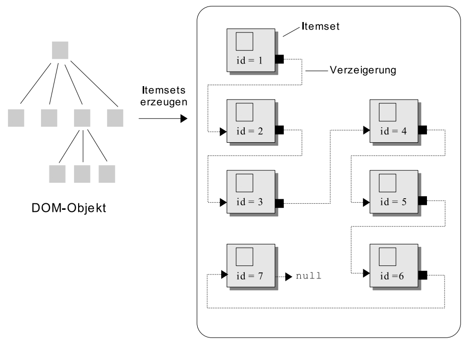

Figure 2. Link Structure

In the current version of our implementation the inital database was used as the basis, and we created an efficient link structure and a hash table for speeding up the data access. This revised version turned out to be relatively efficient for our practical applications. Figure 2 visualizes the link structure in more detail. The identifiers in the picture become the keys for the global hash table.

5. Source Code and Meta Data in XML

The program works on an XML–representation of the source code, and it can be configured by an XML–file containing meta data about how statements are nested and how they may be considered as clones.

So far we have worked with Java and PROLOG source code. A clever adaption of the configuration files opens the application for other languages and other XML–representations of source code.

5.1. Source Code in XML

The Java representation JAML is relatively verbous, since it stores all information – including white spaces and line breaks. E.g., a `while`–statement

```
while (t < a[i]) {
   i++;
}
```

is represented as

```
<while-statement>
  <keyword>while</keyword>
  <symbol>(</symbol>
  <condition>...</condition>
  <symbol>)</symbol>
  <block>
    <symbol>{</symbol>
    ...
    <symbol>}</symbol>
  </block>
</while-statement>
```

where the attributes of the element `symbol` and the JAML–representation of the condition `t < a[i]` and the statement `i++;` have been omitted.

PROLOG–rules are represented in another suitable XML–language, which was developed in [16]; an example is given in the appendix.

5.2. Meta Data in XML

There exist several XML–files for flexibly configuring our algorithm.

The configuration files for Java are based on the Java syntax definition. For instance the following part of the JAML DTD specifies a `while`–construct:

```
<!ELEMENT while-statement
   (keyword,
    symbol, condition, symbol,
    %statement;)>
```

The corresponding part in the configuration file looks like:

```
<recursive_node
   tagname="while-statement">
   <startnode
      necessary="no"
      lookup="yes">
      statement
   </startnode>
</recursive_node>
```

The tag `recursive_node` is used for statements which have a more complex structure and can contain additional statements such as `while`, `for`, `if`, etc. The body of a `while`–statement has additional statements.

Figure 3 shows the tree structure of a `while`–statement in JAML. The node `statement` is not a real XML–node, but it represents a parameter entity (an abbreviation). The tagname `startnode` in the configuration file has several attributes: `necessary="no"` is used to express that there need not exist nested statements (a `while`–expression does not need a body). If the startnode is a parameter entity (such as `%statement;`), then the `lookup`–attribute has the value `yes`; in that case we have to look up the meaning of the possible parameter entities in an additional configuration file.

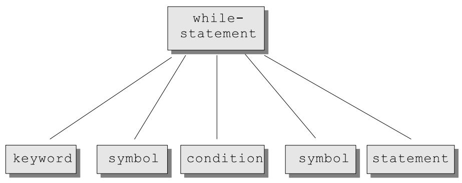

Figure 3. Tree Structure of while–Statement

For PROLOG the configuration file contains only two – almost identical – elements. The first element specifies how the head of a PROLOG rule is processed:

```
<recursive_node
   tagname="head">
   <startnode
      necessary="no"
      lookup="no"
      firstNodeBelow="yes"/>
</recursive_node>
```

The second element – which differs from the first one only in the attribute value `body` for `tagname` – deals with the rule bodies.

6. Case Studies

We have evaluated the Java core API (JDK), another sofware called HagerROM, which is developed at Würzburg University, our own CloneDetection tool, and some PROLOG programs. In this section we report about the case study with the JDK and with PROLOG, respectively.

6.1. The Java Development Kit (JDK)

The JDK is an integrated development environment for Java–applications, –applets, and –components, which was developed by Sun Microsystems. The standard version, which can be downloaded free of charge, is accompanied by a huge portion of the sources, which form the Java core API. It can be split into several packages (cf. Figure 4).

Package	Size
com	4.6 MB
java	14.4 MB
javax	11.5 MB
org	9.2 MB

Figure 4. Package Sizes

It was remarkable that the package java contributed a large portion of the overall running time, whereas the package javax could be handled relatively fast although its size was about the same. The reason are the smaller sizes of the sets $|L_k|$ of frequent k–itemsets in the first three iterations, which dominate the run time, cf. Figures 5 and 6.

For javax much fewer frequent 1–itemsets and much fewer clones are found than for the other packages.

The largest clone which we have detected is a 42–itemset; it has the frequency 7. The clone is the body of the method `int java.util.Arrays`:

```
private static void
    sort1(TYPE x[], int off, len),
```

where `TYPE` is one of the 7 primitve types (int, float, etc.). This clone is an example for necessary duplication.

The overall running time for our case study was about 60 minutes.

size	com	java	javax	org	sum
1	7053	13830	2101	6840	29824
2	2999	6592	716	2921	13228
3	1278	2817	340	1762	6197
4	620	1139	131	501	2391
5	325	606	78	207	1216
6	174	363	81	134	752
7	128	206	33	63	430
8	101	191	25	40	357
9	38	46	21	28	133
10	37	49	16	30	132
11	19	35	2	10	66
12	9	28	4	21	62
13	20	37	2	12	71
14	4	57	2	3	66
15	15	24	2	5	46
16	6	22	2	6	36
17	0	4	0	0	4
18	2	18	0	2	22
19	5	5	0	6	16
20	0	5	0	4	9
21	2	4	0	4	10
22	2	10	0	2	14
23	2	0	0	2	4
24	0	4	0	4	8
…	…	…	…	…	…
42	0	1	0	0	1

Figure 5. Maximal Clones for the Different Parts of the JDK

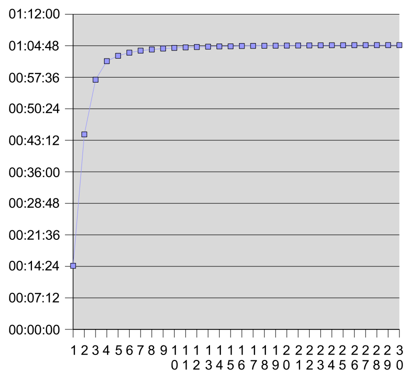

Figure 7. Runtime for the JDK

6.2. The DISLOG Development Kit

We are developing a tool kit called DISLOG Development Kit (DDK) under XPCE/SWI–PROLOG; the functionality ranges from (non–monotonic) reasoning in disjunctive deductive databases to various PROLOG applications, such as a PROLOG software engineering tool and a tool for the management and the visualization of stock information.

Currently, the DDK has about 90.000 lines of PROLOG code, which could be analyzed using our CloneDetection tool in about 90 minutes. Several interesting results about DDK were obtained during our case study.

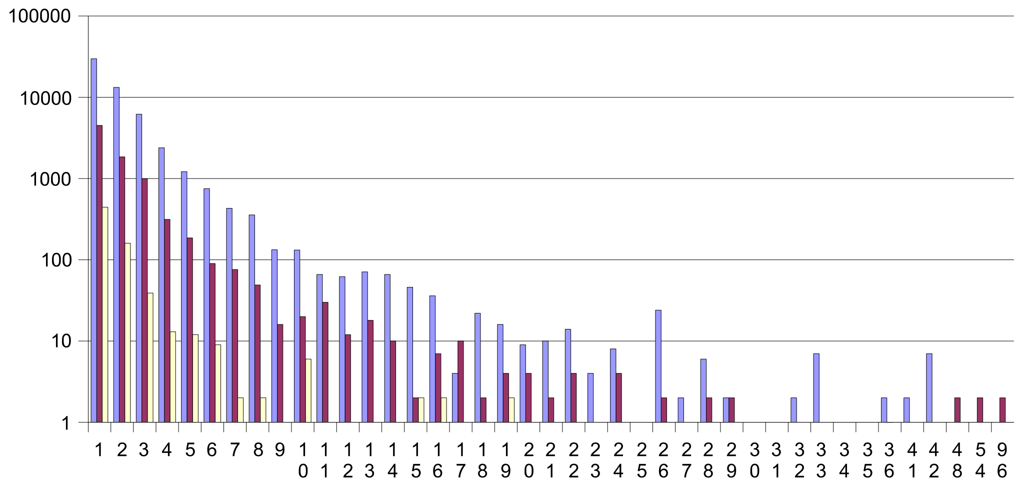

Figure 6. Maximal Clones for Several Case Studies: JDK, HagerROM, CloneDetection

7. Conclusions and Future Work

We have developed a new algorithm for the detection of clones of the types 1 and 2 based on the finding of frequent itemsets. The evaluation has demonstrated the feasability of our approach.

The program works on an XML–representation of the source code, and it can be configured by an XML–file containing meta data about how statements are nested and how they may be considered as clones. Thus, alternative definitions of type 2 clones can easily be obtained. A clever adaption of these configuration files opens the application for *other languages* and other XML–representations of the source code. For representing Java source code we currently use the XML–language JAML [6], but an alternative source language independent format such as srcML [12] would also be possible.

We see two major applications of clone detection. Firstly, it signals weak points in the program and encourages the *restructuring* and *refactoring*. A fully automatic replacement of clones by higher order structures, however, is certainly not the best choice. But in this aspect an integration with an interactive program development environment would be very helpful. A second application is the detection of copies (clones) in a *teaching environment*. We will pursue these two applications in the future

Furthermore, we will extend the algorithm to detect clones of type 3 and clones that occur with a frequency greater than 2 as well. The latter will be possible in our approach, since we can simply lift the support count for frequent itemsets.

Finally, note that a preprocessing step of the XML representation of the abstract syntax tree using XSLT transformations can enable our algorithm to work on general XML files. It is remarkable that this extension will actually mean a simplification of the algorithm.

References

[1] Brenda S. Baker: On Finding Duplication and Near–Duplication in Large Software Systems, Second Working Conference on Reverse Engineering 1995, IEEE Computer Society Press.

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, Kostas Kontogiannis: Measuring Clone Based Reengineering Opportunities, Proceedings of 6th IEEE International Symposium on Software Metrics, 1999.

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant' Anna, Lorraine Bier: Clone Detection Using Abstract Syntax Trees, Proceedings International Conference on Software Maintenance ICSM 1998.

[4] Stefan Bellon, Daniel Simon: Vergleich von Klonerkennungstechniken, 5th Workshop on Software Reengineering, 2003.

[5] Stephane Ducasse, Matthias Rieger, Serge Demeyer: A Language Independent Approach for Detecting Duplicated Code, Proceedings of the International Conference on Software Maintenance ICSM 1999.

[6] Gregor Fischer, Jürgen Wolff von Gudenberg: JaML – An Xml–Representation of Java Source Code, University of Würzburg, Institute for Computer Science, Technical Report, 2004, to appear.

[7] Galax – A Software for XQuery: http://db.bell-labs.com/galax/

[8] Jiawei Han, Micheline Kamber: Data Mining - Concepts and Techniques, Kaufmann, 2001.

[9] Marbod Hopfner, Dietmar Seipel, Jürgen Wolff von Gudenberg: Comprehending and Visualising Source Code based on Xml–Representations and Call Graphs, Proceedings of International Conference on Program Comprehension IWPC 2003.

[10] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: CCFinder: A Multi–Linguistic Token–based Code Clone Detection for Large Scale Source Code, IEEE Transactions on Software Engineering, to appear.

[11] Jens Krinke: Identifying Similar Code with Program Dependence Graphs, Proceedings of the Eighth Working Conference On Reverse Engineering WCRE 2001.

[12] Jonathan I. Maletic, Michael L. Collard, Andrian Marcus: Source code Files as Structured Documents, Proceedings of 10th IEEE International Workshop on Program Comprehension IWPC 2002.

[13] Jeat Mayrand, Claude Leblanc, Ettore M. Merlo: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, Proceedings of International Conference on Software Maintenance ICSM 1996.

[14] Filip van Rysselberghe: Detecting Duplicated Code Using Metric Fingerprints. Master's thesis, University of Antwerp, 2002.

[15] Filip van Rysselberghe, Serge Demeyer: Evaluating Clone Detection Techniques, Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications ELISA 2003.

[16] Dietmar Seipel, Marbod Hopfner, Bernd Heumesser: Analyzing and Visualizing Prolog–Programs based on Xml–Representations, Proceedings of Workshop on Logic Programming Environments WLPE 2003.

Appendix

The following Xml–representation for Prolog–rules was developed in [16]. E.g., the transitive closure rule

```
tc(U1, U2) :-
  arc(U1, U3), tc(U3, U2).
```

is represented as

```
<rule file="transitive_closure">
  <head>
    <atom predicate="tc/2">
      <var name="U1"/>
      <var name="U2"/>
    </atom>
  </head>
  <body>
    <atom predicate="arc/2">
      ...
    </atom>
    <atom predicate="tc/2">
      ...
    </atom>
  </body>
 </rule>
```

We have used the following DTD:

```
<!ELEMENT program (rule*)>
<!ELEMENT rule (head, body)>
<!ELEMENT head (atom*)>
<!ELEMENT body (atom*)>
<!ELEMENT atom ((term|var)*)>
<!ELEMENT term (term*)>

<!ATTLIST rule
    file CDATA #required>
<!ATTLIST atom
    predicate CDATA #required>
<!ATTLIST term
    functor CDATA #implied>
<!ATTLIST var
    name CDATA #implied>
```