A Language-Theoretic View on Guidelines and Consistency Rules of UML

Zhe Chen[1] and Gilles Motet[1,2]

[1] Laboratory LATTIS, INSA, University of Toulouse,
135 Avenue de Rangueil, 31077 Toulouse, France
zchen@insa-toulouse.fr

[2] Foundation for an Industrial Safety Culture,
6 Allée Emile Monso, 31029 Toulouse, France
gilles.motet@insa-toulouse.fr

Abstract. Guidelines and consistency rules of UML are used to control the degrees of freedom provided by the language to prevent faults. Guidelines are used in specific domains (e.g., avionics) to recommend the proper use of technologies. Consistency rules are used to deal with inconsistencies in models. However, guidelines and consistency rules use informal restrictions on the uses of languages, which makes checking difficult. In this paper, we consider these problems from a language-theoretic view. We propose the formalism of C-Systems, short for "formal language control systems". A C-System consists of a controlled grammar and a controlling grammar. Guidelines and consistency rules are formalized as controlling grammars that control the uses of UML, i.e. the derivations using the grammar of UML. This approach can be implemented as a parser, which can automatically verify the rules on a UML user model in XMI format. A comparison to related work shows our contribution: a generic top-down and syntax-based approach that checks language level constraints at compile-time.

1 Introduction

The UML (Unified Modeling Language) is a graphic modeling language developed by OMG (Object Management Group), and defined by the specifications [1] and [2]. UML has emerged as the software industry's dominant modeling language for specifying, designing and documenting the artifacts of systems [3][4].

Evolving descriptions of software artifacts are frequently inconsistent, and tolerating this inconsistency is important [5][6]. Different developers construct and update these descriptions at different times during development [7], thus resulting in inconsistencies. They develop multiple views on a system providing pieces of information which are redundant or complementary. Constraints exist on these pieces of information whose violation leads to inconsistent models. Inconsistency problems of UML models have attracted great attention from both academic and industrial communities [8][9][10]. A list of 635 consistency rules are identified by [11][12].

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 66–81, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Guidelines, which also contain a set of rules, are often required on models which are specific to a given context. For instance, OOTiA (Object-Oriented Technology in Aviation) demands that "the length of an inheritance should be less than 6" [13]. This context is domain specific. If these constraints are not respected, the presence of faults is not sure but its risk is high. The context can also be technology specific. For instance, "multiple inheritance should be avoided in safety critical, certified systems" (IL #38 of [13]), if the UML models are implemented by Java code, as this language does not provide the multiple inheritance mechanism.

It seems that consistency rules and guidelines are irrelevant at first glance. However, in fact, they have the same origin from a language-theoretical view. We noticed that both of the two types of potential faults in models come from *the degrees of freedom* offered by languages. These degrees of freedom cannot be eliminated without reducing the language capabilities [14]. For instance, the multiple diagrams in UML are useful, as they describe various viewpoints on one system, even if they are at the origin of numerous inconsistencies. In the same way, multiple inheritance can be implemented in the C++ language.

To prevent these risks of faults, *the use of languages* must be controlled. To do it, guidelines are old and popular means in industry. However, their expression is informal and their checking is difficult. For instance, 6 months were needed to check 350 consistency rules on an avionics UML model including 116 class diagrams.

This paper aims at formalizing *the acceptable use of languages* and proposing a way to check *the use correctness*, by considering guidelines and consistency rules from a language-theoretical view. To achieve this goal, acceptable uses of a language are defined as a grammar handling the productions of the grammar of the language. To support this idea, UML must be specified by a formal language, or at least a language with precisely defined syntax, e.g., XMI in this paper. Thus, a graphic model can be serialized. This formalism also provides a deeper view on the origin of inconsistencies in models.

This paper is organized as follows. First, we introduce the grammar of UML in XMI in Section 2. Then in Section 3, we define the *C-System*, i.e. a formalism containing controlling grammars that restrict the use of the grammar of UML. We illustrate the formalism using examples in Section 4. Related work and implementation of this approach are discussed in Sections 5 and 6. Section 7 concludes the paper.

2 The Grammar of UML in XMI

XMI (XML Metadata Interchange) [15] is used to facilitate interchanging UML models between different modeling tools in XML format. Many tools implement the conversion, e.g., Altova UModel® can export UML models as XMI files.

A UML model in XMI is an XMI-compliant XML document that conforms to its XML schema, and is a derivative of the *XMI document productions* which is defined as a grammar. The XML schema is a derivative of the *XMI schema*

productions. The XMI specification defines both the XMI schema productions and the XMI document productions in [15].

XMI provides a mapping between a UML user model and an XML document, and a mapping between UML (also MOF) and an XML Schema. XMI generates an XML file using the XMI document productions, and generates an XML schema using the XMI schema productions. Each of the two sets of productions composes a context-free grammar in EBNF [16]. A UML user model can be expressed using an XMI-compliant XML document that conforms to the corresponding XML Schema, and is a derivative of the XMI document grammar.

The grammar and its productions for deriving XMI-compliant XML documents of UML models are defined in [15]. The main part of the grammar is given here after. To make our presentation more concise, we omit declaration and version information of XML files (and the related productions whose names start with "1").

To make later reasoning easier, we modified some representations of the productions, but without changing the generative power of the grammar.

1. The choice operator "|" is used to compose several productions with the same left-hand side into a single line in [15]. We decomposed some of these productions into several productions without the choice operator. An original production n having k choices might be divided into a set of productions $\{n_i\}_{1\leq i\leq k}$. For example, the original production 2 with three choices was divided into the productions 2_1, 2_2 and 2_3.

2. The closure operator "*" is used to simplify the representation of the grammar in [15], but it also would make the representation of reasoning confusing. Thus, the productions whose names start with "3" were added to replace the productions with closure operators.

The **grammar** G of UML in XMI includes the following productions (each production is labeled by a name starting with a digit):

```
3_1:  XMIElements ::= 2:XMIElement
3_2:  XMIElements ::= 2:XMIElement 3:XMIElements

2_1:  XMIElement ::= 2a:XMIObjectElement
2_2:  XMIElement ::= 2b:XMIValueElement
2_3:  XMIElement ::= 2c:XMIReferenceElement

2a_1: XMIObjectElement ::= "<" 2k:QName 2d:XMIAttributes "/>"
2a_2: XMIObjectElement ::= "<" 2k:QName 2d:XMIAttributes ">"
                            3:XMIElements "</" 2k:QName ">"

2b_1: XMIValueElement ::= "<" xmiName ">" value "</" xmiName ">"
2b_2: XMIValueElement ::= "<" xmiName "nil='true'/>"

2c_1: XMIReferenceElement::= "<" xmiName 2l:LinkAttribs "/>"
2c_2: XMIReferenceElement::= "<" xmiName 2g:TypeAttrib
                              2l:LinkAttribs "/>"
```

```
2d_1: XMIAttributes ::= 2g:TypeAttrib  2e:IdentityAttribs
                        3h:FeatureAttribs
2d_2: XMIAttributes ::= 2e:IdentityAttribs  3h:FeatureAttribs

2e:   IdentityAttribs ::= 2f:IdAttribName "=‘" id "’"

2f_1: IdAttribName ::= "xmi:id"
2f_2: IdAttribName ::= xmiIdAttribName

2g:   TypeAttrib ::= "xmi:type=‘" 2k:QName "’"

3h_1: FeatureAttribs ::= 2h:FeatureAttrib
3h_2: FeatureAttribs ::= 2h:FeatureAttrib  3h:FeatureAttribs

2h_1: FeatureAttrib ::= 2i:XMIValueAttribute
2h_2: FeatureAttrib ::= 2j:XMIReferenceAttribute

2i:   XMIValueAttribute ::= xmiName "=‘" value "’"

2j:   XMIReferenceAttribute ::= xmiName "=‘" (refId | 2n:URIref)+"’"

2k:   QName ::= "uml:" xmiName  |  xmiName

2l:   LinkAttribs ::= "xmi:idref=‘" refId "’" | 2m:Link

2m:   Link ::= "href=‘" 2n:URIref "’"

2n:   URIref ::= (2k:QName)? uriReference
```

In the grammar, the symbol "::=" stands for the conventional rewriting symbol "→" in formal languages theory [17]. Each nonterminal starts with a capital letter, prefixing a label of the related production, e.g., "2:XMIElement" is a nonterminal with possible productions "2_1, 2_2, 2_3". Each terminal starts with a lowercase letter or is quoted.

As an example to illustrate the use of the grammar, Figure 1 represents a package *Root* which includes three classes, where the class *FaxMachine* is derived from *Scanner* and *Printer*. The core part of the exported XMI 2.1 compliant file (using Altova UModel®) is as follows:

```
<uml:Package xmi:id="U00000001-7510-11d9-86f2-000476a22f44"
             name="Root">
     <packagedElement xmi:type="uml:Class"
             xmi:id="U572b4953-ad35-496f-af6f-f2f048c163b1"
             name="Scanner" visibility="public">
          <ownedAttribute xmi:type="uml:Property"
             xmi:id="U46ec6e01-5510-43a2-80e9-89d9b780a60b"
          name="sid" visibility="protected"/>
```

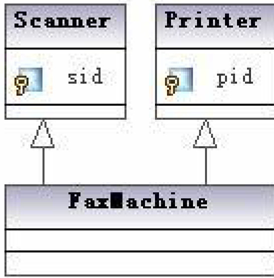

Fig. 1. A Class Diagram

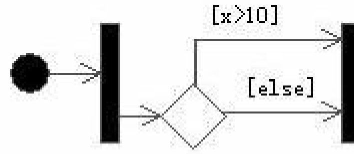

Fig. 2. An Activity Diagram

```
      </packagedElement>
      <packagedElement xmi:type="uml:Class"
              xmi:id="Ua9bd8252-0742-4b3e-9b4b-07a95f7d242e"
              name="Printer" visibility="public">
           <ownedAttribute xmi:type="uml:Property"
              xmi:id="U2ce0e4c8-88ee-445b-8169-f4c483ab9160"
           name="pid" visibility="protected"/>
      </packagedElement>
      <packagedElement xmi:type="uml:Class"
              xmi:id="U6dea1ea0-81d2-4b9c-aab7-a830765169f0"
              name="FaxMachine" visibility="public">
           <generalization xmi:type="uml:Generalization"
              xmi:id="U3b334927-5573-40cd-a82b-1ee065ada72c"
              general="U572b4953-ad35-496f-af6f-f2f048c163b1"/>
           <generalization xmi:type="uml:Generalization"
              xmi:id="U86a6818b-f7e7-42d9-a21b-c0e639a4f716"
              general="Ua9bd8252-0742-4b3e-9b4b-07a95f7d242e"/>
      </packagedElement>
 </uml:Package>
```

This text is a derivative of the XMI document productions, c.f. the previous grammar G. We may use the sequence of productions "2a_2, 2k(Package), 2d_2, 2e, 2f_1, 3h_1, 2h_1, 2i" to derive the following sentential form:

```
 <uml:Package xmi:id="U00000001-7510-11d9-86f2-000476a22f44"
              name="Root">
      3:XMIElements "</" 2k:QName ">"
```

Note that the production 2k has a parameter *xmiName*, i.e. the value of the terminal when apply the production. In a derivation, we specify a value of the parameter as "2k(value)". For example, "2k(Package)" is a derivation using 2k with *xmiName* = "Package". For simplicity, we consider "2k(value)" as a terminal as a whole. We continue to apply productions, and finally derive the XMI file previously presented.

Notice that the model of Fig. 1 (both in UML and XML) does not conform to the guidelines in OOTiA about multiple inheritance, since it uses multi-inheritance. The model of Fig. 2 has an inconsistency: "the number of outgoing edges of $ForkNode$ is not the same as the number of incoming edges of $JoinNode$". In particular, $JoinNode$ joins two *outgoing* edges from the same $DecicionNode$, This join transition will never be activated, since only one of the two *outgoing* edges will be fired.

We will define a formal model to check the conformance to these rules by controlling the use of the grammar of UML.

3 The C-System: A Formal Language Control System

In this section, we propose the formal model for controlling the use of grammars based on classical language theory [17].

Let $G = (N, T, P, S)$ be a grammar, where N is the set of nonterminals, T is the set of terminals, P is the set of productions of the form $l : A \rightarrow \alpha$ where l is the name of the production, $A \in N$, $\alpha \in (N \bigcup T)^*$, and S is the start symbol. A derivation using a specified production p is denoted by $\alpha \overset{p}{\Rightarrow} \beta$, and multiple derivations are denoted by $\alpha \Rightarrow^* \gamma$.

Definition 1. *A* ***controlling grammar*** $\hat{G}$ *over a* ***controlled grammar*** *(or simply grammar)* $G = (N, T, P, S)$ *is a quadruple* $\hat{G} = (\hat{N}, \hat{T}, \hat{P}, \hat{S})$*, where* $\hat{T} = P$*. The language* $L(\hat{G})$ *is called a* ***controlling language****.* □

The symbol $\hat{G}$ is read "control G" or "G hat". For making reading easier, we assume that $N \cap \hat{N} = \emptyset$. $\hat{T} = P$ means that the terminals of $\hat{G}$ are exactly the productions of G.

If we use an automaton A to process the input string, such that $L(A) = L(G)$, then we can also use a **controlling automaton** $\hat{A}$ to represent the controlling language.

As we know, each string $w \in L(G)$ has at least one *leftmost* derivation (denoted by "lm") using a sequence of productions from P, e.g. $p_1 p_2 ... p_k$. The controlling grammar restricts the derivation in the sense that the sequences of applied productions should be in the language it specifies, i.e., $p_1 p_2 ... p_k \in L(\hat{G})$. Formally, we have the following definition.

Definition 2. *Given a grammar* $G = (N, T, P, S)$*, the language of the grammar with a controlling grammar* $\hat{G}$ *is:*

$$L(G \overrightarrow{\cdot} \hat{G}) = \{w | S \underset{lm}{\overset{p_1}{\Rightarrow}} w_1 \cdots \underset{lm}{\overset{p_k}{\Rightarrow}} w_k = w,\ p_1, p_2, ..., p_k \in P \text{ and } p_1 p_2 ... p_k \in L(\hat{G})\}$$

We say that G *and* $\hat{G}$ *constitute a* ***C-System*** $C = G \overrightarrow{\cdot} \hat{G}$*, short for* ***formal language control system****. The language* $L(C) = L(G \overrightarrow{\cdot} \hat{G})$ *is called a* ***global system language****.* □

The symbol $\overrightarrow{\cdot}$ is called "meta composition". Its left operand is controlled by the right operand, which is a meta level grammar. If we use automata-based

notations, a string $w \in L(A \vec{\cdot} \hat{A})$ if and only if A accepts w, and $\hat{A}$ accepts the sequence of the labels of the transitions used.

A ***regular C-System*** is a C-System of which the controlled grammar G is a regular grammar (or A is a finite automaton). Some variants of regular C-Systems are proposed for ensuring system safety requirements, e.g. Input/Output C-Systems [18], Interface C-Systems [19][20]. We denote by $\mathcal{C}_R$ the family of regular C-Systems.

A ***context-free C-System*** is a C-System of which the controlled grammar G is a context-free grammar (or A is a pushdown automaton). We denote by $\mathcal{C}_{CF}$ the family of context-free C-Systems.

Generally, we denote by $\mathcal{C}_X^Y$ the family of C-Systems that consist of X-type controlled grammar and Y-type controlling grammar, where $X, Y \in \{R, CF\}$. Although X, Y could be also other types in Chomsky hierarchy, e.g. context-sensitive, this is beyond the scope of this paper.

Obviously, the set of accepted inputs is a subset of the controlled language, such that the sequence of the applied productions belongs to the controlling language. Consider a simple example as follows.

Example 1. Given a regular grammar G and a regular controlling grammar $\hat{G}$:

$$G \begin{cases} p_1 : S \to aS \\ p_2 : S \to bS \\ p_3 : S \to \epsilon \end{cases} \qquad \hat{G} \begin{cases} \hat{S} \to p_1\hat{S} | p_3\hat{S} | p_2 A \\ A \to p_2 A | p_3 A | \epsilon \end{cases}$$

$L(G)$ accepts the language $(a|b)^*$, e.g., aab, $abab$. $L(\hat{G})$ accepts the language $(p_1|p_3)^* p_2 (p_2|p_3)^*$. The trivial grammar G is considered to provide a simple illustration of the introduced principles.

The grammars G and $\hat{G}$ constitute a regular C-System $C = G \vec{\cdot} \hat{G} \in \mathcal{C}_R^R$.

Given the string $aab \in L(G)$, we conclude that $aab \in L(G \vec{\cdot} \hat{G})$, because we have the leftmost derivations $S \overset{p_1}{\Rightarrow} aS \overset{p_1}{\Rightarrow} aaS \overset{p_2}{\Rightarrow} aabS \overset{p_3}{\Rightarrow} aab$, where $p_1p_1p_2p_3 \in L(\hat{G})$ as $\hat{S} \Rightarrow p_1\hat{S} \Rightarrow p_1p_1\hat{S} \Rightarrow p_1p_1p_2A \Rightarrow p_1p_1p_2p_3A \Rightarrow p_1p_1p_2p_3$. On the contrary, we have $abab \notin L(G \vec{\cdot} \hat{G})$. Although we have the leftmost derivation $S \overset{p_1}{\Rightarrow} aS \overset{p_2}{\Rightarrow} abS \overset{p_1}{\Rightarrow} abaS \overset{p_2}{\Rightarrow} ababS \overset{p_3}{\Rightarrow} abab$, $p_1p_2p_1p_2p_3 \notin L(\hat{G})$.

In fact, the language $L(C) = L(G \vec{\cdot} \hat{G})$ is equivalent to the language a^*b^+, which is the subset of $(a|b)^*$ satisfying the constraints: "every a should appear before b" and "at least one b". □

We remark here that our model is different from regularly controlled grammars [21][22], in the sense that we restrict derivations to be *leftmost* and allow *context-free controlling grammars*. These differences result in different theoretical results, which are beyond the scope of this paper.

4 Examples

In this section we use some practical examples to illustrate the idea of the previous section. We denote the grammar of UML by $G = (N, T, P, S)$, where P is

the set of productions listed in Section 2, and each production $p \in P$ is labeled by a name starting with a digit.

Example 2. Consider two rules on class diagrams:

Rule 1: Each class can have at most one generalization. This rule is a guideline, as we mentioned in Section 1 and at the end of Section 2. This rule is also a consistency rule in the context of Java, since Java does not allow multiple inheritance. However we may derive a class from multiple classes in the context of C++.

Rule 2: Each class can have at most 30 attributes. This rule may be adopted by software authorities as a guideline in avionics, in order to increase the safety of software systems by minimizing the complexity of classes.

Note that these rules cannot be explicitly integrated into the grammar of UML, but only recommended as guidelines or consistency rules. We cannot put rule 1 into the standard of UML, since UML models can be implemented with both C++ and Java programming languages. Rule 2 is a restriction for a specific domain, and we should not require all programmers to use limited attributes by specifying the UML language.

We aim to specify the rules from the meta-language level, thus control the use of the language. Consider the example of Fig. 1, to obtain the associated XMI text, the sequence of applied productions of G in the leftmost derivation is as follow ("..." stands for some omitted productions, to save space):

```
2a_2, 2k(Package), 2d_2, 2e, 2f_1, 3h_1, 2h_1, 2i,
..., 2k(packagedElement), ..., 2k(Class),
     ..., 2k(ownedAttribute), ..., 2k(Property),
..., 2k(packagedElement),
..., 2k(packagedElement), ..., 2k(Class),
     ..., 2k(ownedAttribute), ..., 2k(Property),
..., 2k(packagedElement),
..., 2k(packagedElement), ..., 2k(Class),
     ..., 2k(generalization), ..., 2k(Generalization),
     ..., 2k(generalization), ..., 2k(Generalization),
..., 2k(packagedElement),
..., 2k(Package)
```

Let c, g stand for $2k(Class), 2k(Generalization)$, respectively. Note that the occurrence of two g after the third c violates Rule 1. In fact, all the sequences of productions in the pattern "...c...g...g..." are not allowed by the rule (there is no c between the two g), indicating that the class has two generalizations.

Thus, we propose the following controlling grammar $\hat{G}_c$ to restrict the use of the language to satisfy Rule 1:

$$\hat{G}_c \begin{cases} S \to c \;\; Q_c \mid D\; S \mid D \\ Q_c \to c \;\; Q_c \mid g \;\; Q_g \mid D\; Q_c \mid D \\ Q_g \to c \;\; Q_c \mid D\; Q_g \mid D \\ D \to \{p \mid p \in P \;\wedge\; p \notin \{c, g\}\} \end{cases} \tag{1}$$

where S, Q_c, Q_g, D are nonterminals, D includes all productions except c, g. $L(\hat{G}_c)$ accepts the sequences of productions satisfying Rule 1.

Implicitly, the controlling grammar specifies an automaton $\hat{A}_c$ in Fig. 3, where $\lightning$ is an implicit error state (the dashed circle). Strings of the pattern $D^*cD^*gD^*gD^*$ will lead $\hat{A}_c$ to the error state.

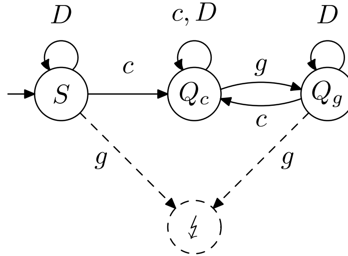

Fig. 3. The Automaton $\hat{A}_c$

If the sequence of productions applied to derive a model is accepted by the language $L(\hat{G}_c)$, then the model conforms to Rule 1. In Fig. 1, the derivation of the class *FaxMachine* uses the pattern $D^*cD^*gD^*gD^* \notin L(\hat{G}_c)$, which leads to $\lightning$ of the automaton, thus it violates Rule 1. On the contrary, the derivations of *Scanner* and *Printer* are accepted by $L(\hat{G}_c)$, thus satisfy Rule 1.

Now consider Rule 2. Let c, pr, pe stand for $2k(Class)$, $2k(Property)$, $2k(PackagedElement)$, respectively. Note that the occurrence of more than 30 pr after a c violates Rule 2. In fact, all the sequences of productions in the pattern "$...c...(pr...)^n, n > 30$" are not allowed by the rule (there is no c between any two pr), indicating that the class has more than 30 attributes.

To satisfy Rule 2, we propose the following controlling grammar $\hat{G}_p$ to restrict the use of the language:

$$\hat{G}_p \begin{cases} S \to pe \;\; S \mid c \;\; Q_c \mid D\; S \mid D \\ Q_c \to pe\; S \mid c \;\; Q_c \mid pr \;\; Q_1 \mid D\; Q_c \mid D \\ Q_i \to pe\; S \mid c \;\; Q_c \mid pr \;\; Q_{i+1} \mid D\; Q_i \mid D \;\; (1 \le i < 30) \\ Q_{30} \to pe \;\; S \mid c \;\; Q_c \mid D\; Q_{30} \mid D \\ D \to \{p \mid p \in P \;\wedge\; p \notin \{c, pr, pe\}\} \end{cases} \tag{2}$$

where S, Q_c, Q_i are nonterminals, D includes all productions except c, pr, pe. $L(\hat{G}_p)$ accepts the sequences of productions satisfying Rule 2.

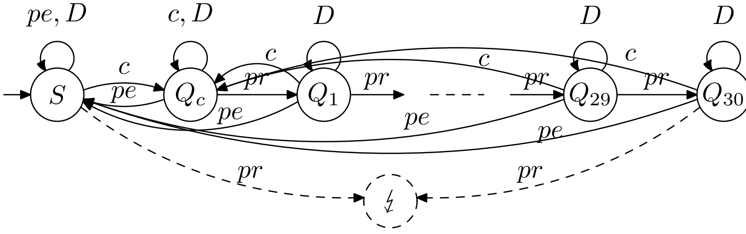

Fig. 4. The Automaton $\hat{A}_p$

Implicitly, the controlling grammar specifies an automaton $\hat{A}_p$ in Fig. 4. Strings of the pattern "$D^*cD^*(pr\ D^*)^n, n > 30$" will lead $\hat{A}_p$ to the error state.

If the sequence of productions applied to derive a model is accepted by the language $L(\hat{G}_p)$, then the model conforms to Rule 2. In Fig. 1, the derivations of the classes *Scanner* and *Printer* use the pattern $D^*cD^*prD^* \in L(\hat{G}_p)$, thus satisfy Rule 2.

Thanks to the controlling grammars, when a model violates required rules, the controlling language will reject the model (an implicit error state ↯ will be activated). Some error handling method may be called to process the error, e.g., printing an error message indicating the position and the cause.

We can also use controlling grammar to handle a consistency rule concerning activity diagrams.

Example 3. In an activity diagram, the number of outgoing edges of $ForkNode$ should be the same as the number of incoming edges of its pairwise $JoinNode$.

Let n, f, j, i, o stand for $2k(node)$, $2k(ForkNode)$, $2k(JoinNode)$, $2k(incoming)$, $2k(outgoing)$, respectively. We propose the following controlling grammar $\hat{G}_a$ to restrict the use of the language to satisfy the rule:

$$\hat{G}_a \begin{cases} S \rightarrow N \;\; F \;\; I^* \;\; Q \;\; O^* \;\; N \mid N \;\; I^* \;\; O^* \;\; N \mid D^* \\ Q \rightarrow O \;\; Q \;\; I \mid N \;\; S \;\; N \;\; J \\ N \rightarrow n \;\; D^* \\ F \rightarrow f \;\; D^* \\ J \rightarrow j \;\; D^* \\ I \rightarrow i \;\; D^* \\ O \rightarrow o \;\; D^* \\ D \rightarrow \{p \mid p \in P \;\wedge\; p \notin \{n, f, j, i, o\}\} \end{cases} \tag{3}$$

$L(\hat{G}_a)$ accepts all the sequences of productions of the pattern $NFI^*O^nNSNJI^nO^*N$, which leads to models respecting the rule. This context-free grammar implicitly specifies a PDA (Pushdown Automaton [17]), which is more complex than the automata in Figures 3 and 4.

Globally, any UML user model M derived from the C-System $C = G \vec{\cdot} \hat{G}_a \in \mathcal{C}^{CF}_{CF}$, i.e. $M \in L(C)$, conforms to the rule in Example 3.

As a more concrete instance, we consider the model in Fig. 2. The XMI-compliant document of the model in Fig. 2 is the follows:

```
<packagedElement xmi:type="uml:Activity"
             xmi:id="U937506ed-af64-44c6-9b4c-e735bb6d8cc6"
             name="Activity1" visibility="public">
<node xmi:type="uml:InitialNode" xmi:id="U16aa15e8-0e5d-
       4fd1-930a-725073ece9f0">
   <outgoing xmi:idref="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"/>
</node>
<node xmi:type="uml:ForkNode" xmi:id="U26768518-a40c-
                  4713-b35e-c267cc660508" name="ForkNode">
   <incoming xmi:idref="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"/>
   <outgoing xmi:idref="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"/>
</node>
<node xmi:type="uml:DecisionNode" xmi:id="Uc9e4f0de-8da6-
       4c98-9b95-b4cde30ccfc0" name="DecisionNode">
   <incoming xmi:idref="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"/>
   <outgoing xmi:idref="Ua4a2b313-13d6-4d69-9617-4803560731ef"/>
   <outgoing xmi:idref="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"/>
</node>
<node xmi:type="uml:JoinNode" xmi:id="Ud304ce3c-ebe4-
                  4b06-b75a-fa2321f8a151" name="JoinNode">
   <incoming xmi:idref="Ua4a2b313-13d6-4d69-9617-4803560731ef"/>
   <incoming xmi:idref="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"/>
</node>
<edge xmi:type="uml:ControlFlow"
       xmi:id="Ua4a2b313-13d6-4d69-9617-4803560731ef"
       source="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"
       target="Ud304ce3c-ebe4-4b06-b75a-fa2321f8a151">
   <guard xmi:type="uml:LiteralString"
       xmi:id="U6872f3b3-680c-430e-bdb3-21c0a317d290"
       visibility="public" value="x>10"/>
</edge>
<edge xmi:type="uml:ControlFlow"
       xmi:id="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"
       source="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"
       target="Ud304ce3c-ebe4-4b06-b75a-fa2321f8a151">
   <guard xmi:type="uml:LiteralString"
       xmi:id="Ub853080d-481c-46ff-9f7c-92a31ac24349"
       visibility="public" value="else"/>
</edge>
<edge xmi:type="uml:ControlFlow"
       xmi:id="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"
       source="U26768518-a40c-4713-b35e-c267cc660508"
       target="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"/>
```

```
<edge
      xmi:type="uml:ControlFlow"
      xmi:id="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"
      source="U16aa15e8-0e5d-4fd1-930a-725073ece9f0"
      target="U26768518-a40c-4713-b35e-c267cc660508"/>
</packagedElement>
```

It is easy to detect that the sequence of applied productions "...$nD^*fD^*iD^*oD^*nD^*$ ... $nD^*jD^*iD^*i$..." is not accepted by $L(\hat{G}_a)$ (one o follows f, while two i follow j), thus there is an inconsistency.

We remark here that there are two preconditions of using the controlling grammar concerning the sequences of the model elements in the XML document: 1. $ForkNode$ must appear before its pairwise $JoinNode$; 2. *incoming* edges must appear before *outcoming* edges in a node. The two conditions are trivial, since it is easy to control their positions in the exporting XMI documents in implementing such a transformation.

5 Related Work

The most popular technique for verifying software correctness is *model checking* [23]. In this framework, we have three steps in verifying a system. First, we formalize system behavior as a model (e.g., a transition system, a Kripke model [24]). Second, we specify the properties that we aim at validating using temporal logics. Third, we use a certain checking algorithm to search for a counterexample which is an execution trace violating the specified properties. If the algorithm finds such a counterexample, we have to correct the original design.

Most checking tools use specific semantics of UML diagrams. They have the flavor of model checking, e.g., Egyed's UML/Analyzer [25][26] and OCL (Object Constraint Language) [27]. At first, developers design UML diagrams as a model. Then, we specify the consistency rules as OCL or similar expressions. Certain algorithms are executed to detect counterexamples that violate the rules [28]. Note that these techniques do not discriminate the rules on the model level and those concerning the language level features.

Unlike these techniques, our framework takes another way of ensuring correctness. It consists of the following steps:

1. Specifying the grammar G of a language. It specifies an *operational semantics*, which defines what a language is able to model. Developing the grammar is mainly performed by *language designers*.
2. Modeling correctness rules of the use of languages as a controlling grammar $\hat{G}$. It specifies a *correctness semantics*, which defines what a language is authorized to derive. This process is the duty of *safety engineers* whose responsibility is to assure the correct use of the language.

3. The two grammars constitute a consistent language as a whole, that is, any derivations of the global system language is a correct and consistent use of the language.

In particular, our work differs from model checking in the following aspects:

1. Our work has different objectives, and uses different approaches to those of model checking. As we show in Fig. 5, model checking techniques use a **bottom-up approach** — they verify execution traces T^* at the lower level L_1 to prove the correct use of the grammar G at the middle level L_2. Whereas our proposal uses a **top-down approach** — we model correctness rules as acceptable sequences of productions (P^*) at the higher level L_3 to ensure the correct use of G. Then any derivatives (at L_1) that conform to the C-System $C = G \overrightarrow{\cdot} \hat{G}$ are definitely a correct use. So the two techniques are complementary.

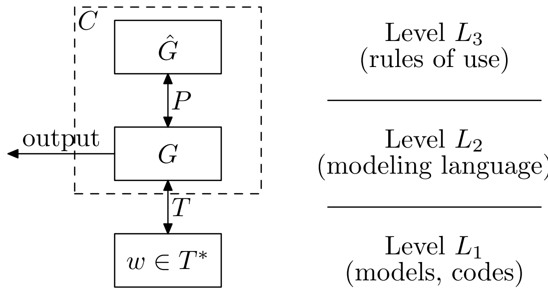

Fig. 5. Three Levels of the Framework

2. Our work and model checking express language-level and model-level constraints, respectively. Language-level constraints are more effective, because they implicitly have reusability. That is, we only need to develop one language-level constraint and apply it to all the models in the language. However, using model checking, we need to replicate model-level constraints for each model. Additionally, model checking can process model-specific constraints.

3. Our work and model checking use syntax-based and semantics-based approaches (or static and dynamic analysis), respectively. As a result, our approach is generic and metamodel-independent, and concerns little about semantics. So it can be applied to all MOF-compliant languages, not only to UML. However, model checking techniques depends on the semantics of a language, thus specific algorithms should be developed for different models.

4. Our work and model checking catch errors at compile-time and runtime, respectively. As a result, our approach implements membership checking of context-free languages, which is decidable. That is, it searches in a limited space, which is defined by grammars. Model checking may search in a larger, even infinite space, so we have to limit the space of computing, and introduce the risk of missing solutions.

6 Discussion

In this section, we would like to shortly discuss some issues which are beyond the scope of this paper.

The first issue concerns the implementation of controlling grammars. The controlled and controlling grammars can be implemented using two parsers separately. The technique for constructing a parser from a context-free grammar is rather mature [29][30]. Some tools provide automated generation of parsers from a grammar specification, such as Yacc, Bison.

Notice that the inputs of controlling parsers are the sequences of productions applied in the parsing of $L(G)$. So there are communications between the two parsers. Once module G uses a production p_i, then the name of the production is sent to $\hat{G}$ as an input. If $L(\hat{G})$ accepts the sequence of productions and $L(G)$ accepts the model, then $L(G \overrightarrow{\cdot} \hat{G})$ accepts the model.

The second issue deals with multiple rules. If we have multiple guidelines or consistency rules, each rule is formalized using a grammar. We can develop an automated tool that converts the grammars into automata, and then combine these automata to compute an intersection, i.e., an automaton A' [17]. The intersection A' can be used as a controlling automaton, which specifies a controlling language $L(A')$ that includes all the semantics of the rules.

The third issue is about the tradeoff between cost and benefits of applying the proposed approach. It seems that writing a controlling grammar is expensive, because it involves formal methods. However, it is probably not the case. As we mentioned, a controlling grammar specify language-level constraints, and can be reused by all the models derived from the controlled grammar. Thus the controlling grammar can be identified and formalized by the organizations who define the language or its authorized usage, e.g., OMG and FAA (Federal Aviation Administration), respectively. Developers and software companies can use the published standard controlling grammar for checking inconsistencies in their models. By contraries, if every user writes their own checking algorithms and codes, e.g., in OCL or other programming languages, the codes will be hard to be reused by other users who have different models to check. Thus the total cost of all the users may be higher. Of course, more empirical results on the tradeoff is a good direction for future work.

7 Conclusion

We provided a language-theoretic view on guidelines and consistency rules of UML. We proposed the formalism of C-Systems, short for "formal language control systems". To the best of our knowledge, none related work proposed similar methodologies. Rules are considered as controlling grammars which control the use of modeling languages. This methodology is generic, syntax-based and metamodel-independent. It provides a top-down approach that checks and reports violations of language level constraints at compile-time. It can be also applied to all MOF-compliant languages, not only to UML, since it does not depend on the specific semantics of languages.

Since we focused on the methodological foundation, one of the future work is to develop an automated checking tool implementing the presented principles. We will also examine instant checking techniques of our method. One feature of UML/Analyzer is instant checking, which only verifies the small portion where the model changes, in order to save the cost of checking [31]. Intuitively, our approach is also easy to be extended to instant checking. We only need to generate the XMI document of the changed part of diagrams (e.g. a class in a class diagram), and verify it. However, this calls for more works in detail.

References

1. OMG: Unified Modeling Language: Infrastructure, version 2.1.1 (07-02-06). Object Management Group (2007)
2. OMG: Unified Modeling Language: Superstructure, version 2.1.1 (07-02-05). Object Management Group (2007)
3. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. 11(1), 2–57 (2002)
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
5. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering (ICSE 1991), pp. 158–165. IEEE Computer Society, Los Alamitos (1991)
6. Easterbrook, S.M., Chechik, M.: 2nd international workshop on living with inconsistency. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 749–750. IEEE Computer Society, Los Alamitos (2001)
7. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Leveraging inconsistency in software development. IEEE Computer 33(4), 24–29 (2000)
8. Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z. (eds.): Workshop on consistency problems in UML-based software development I, co-located with UML 2002 (2002), `http://www.ipd.bth.se/consistencyUML/`
9. Kuzniarz, L., Huzar, Z., Reggio, G., Sourrouille, J.L. (eds.): Workshop on consistency problems in UML-based software development II, co-located with UML 2003 (2003), `http://www.ipd.bth.se/consistencyUML/`
10. Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L. (eds.): Workshop on consistency problems in UML-based software development III, co-located with UML 2004 (2004), `http://www.ipd.bth.se/consistencyUML/`
11. Vidal, J.-P.S., Malgouyres, H., Motet, G.: UML 2.0 Consistency Rules (2005), `http://www.lattis.univ-toulouse.fr/UML/`
12. Vidal, J.P.S., Malgouyres, H., Motet, G.: UML 2.0 consistency rules identification. In: Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP 2005). CSREA Press (2005)
13. Federal Aviation Administration: Handbook for Object-Oriented Technology in Aviation (OOTiA), vol. 2.1, Considerations and issues. U.S. Department of Transportation (October 2004)
14. Motet, G.: Risks of faults intrinsic to software languages: Trade-off between design performance and application safety. Safety Science (2009)
15. OMG: MOF 2.0 / XMI Mapping, version 2.1.1 (07-12-01). Object Management Group (2007)

16. ISO/IEC: ISO/IEC 14977:1996(E): Extended BNF (1996)
17. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
18. Chen, Z., Motet, G.: Modeling system safety requirements using input/output constraint meta-automata. In: Proceedings of the 4th International Conference on Systems (ICONS 2009), pp. 228–233. IEEE Computer Society, Los Alamitos (2009)
19. Chen, Z., Motet, G.: System safety requirements as control structures. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009). IEEE Computer Society, Los Alamitos (2009)
20. Chen, Z., Motet, G.: Formalizing safety requirements using controlling automata. In: Proceedings of the Second International Conference on Dependability (DEPEND 2009). IEEE Computer Society, Los Alamitos (2009)
21. Ginsburg, S., Spanier, E.H.: Control sets on grammars. Mathematical Systems Theory 2(2), 159–177 (1968)
22. Dassow, J., Paun, G., Salomaa, A.: Grammars with controlled derivations. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 101–154. Springer, Heidelberg (1997)
23. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
24. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University Press, Cambridge (2004)
25. Egyed, A.: Fixing inconsistencies in UML design models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 292–301. IEEE Computer Society, Los Alamitos (2007)
26. Egyed, A.: UML/Analyzer: A tool for the instant consistency checking of UML models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 793–796. IEEE Computer Society, Los Alamitos (2007)
27. OMG: Object Constraint Language, version 2.0 (06-05-01). Object Management Group (2006)
28. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL environment. Electr. Notes Theor. Comput. Sci. 102, 99–110 (2004)
29. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)
30. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
31. Egyed, A.: Instant consistency checking for the UML. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 381–390. ACM, New York (2006)