01 Jan 2004

A Generic OO Architecture Language for Semantics Analysis of OO Specification

Xiaoqing Frank Liu
Missouri University of Science and Technology, fliu@mst.edu

Recommended Citation

X. F. Liu, "A Generic OO Architecture Language for Semantics Analysis of OO Specification," *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004*, Institute of Electrical and Electronics Engineers (IEEE), Jan 2004.
The definitive version is available at https://doi.org/10.1109/CMPSAC.2004.1342653

A Generic OO Architecture Language for Semantics Analysis of OO Specification

Franck Xia
Computer Science Dept., University of Missouri-Rolla, Rolla, MO 65409; xiaf@umr.edu

1. Introduction

Formal specification enables a rigorous analysis and model checking for ensuring the correctness of specification. However, most formal OO specification methods are of mathematical nature and the semantics of specification is purposely defined such that it is not related to the semantics of code. Yet when a specification is not semantically connected to its implementation, the correctness of the specification could be easily lost when it is converted to code in an artist and error-prone way [1]. This broken semantics link is a major obstacle which prevents a wide adoption of rigorous methods in engineering practice. Exceptions do exist. In [2], the semantics of class and collaboration diagrams in UML is decorated based on Eiffel. Yet using a programming language with implementation details for specification risks to lose abstraction. Alloy is a language designed with a precise and implementation independent semantics for analyzing the structural properties of OO specification [3]. Alloy ignores the dynamic interaction between objects and hence is not general enough for specification. We propose a new language which will lay a common semantics ground for both specification and code.

2. A Generic OO Architecture Language

Existing programming languages have many features based on implementation concerns such as variables vs. pointer, array vs. linked list, parameter passing mode, etc. As these features are irrelevant for specification, we propose a Generic Object-Oriented Architecture Language (GOOAL), free of any implementation concerns and being able to capture the essence of OO programming such as encapsulation, information hading, and inheritance. The elementary statement in GOOAL is method call but not basic statement. Hence GOOAL is intrinsically abstract and suitable for describing the architecture of OO systems. GOOAL has a relatively simple semantics, a subset of the semantics of OO programming languages, thus it lays a semantic ground for both specification and coding.

3. Syntax of GOOAL

The syntax of GOOAL can be easily defined using BNF. Here we provide an informal description of it. In GOOAL, attributes and methods can only be defined within classes. A method named *main* is the first method to be executed in an application. In the executive body of a method, the basic statement is method call, in combination with selection and looping. Classed may include invariants and method bodies contain pre and post conditions.

4. Semantics of GOOAL

4.1 Semantics Aspects

There are various semantics theories for programming languages, be axiomatic, operational or denotational. Overall, different aspects of program semantics such as data, control constructs, and subprograms are fairly well understood. The dynamic semantics of GOOAL can be simply defined using the known operational semantics which will be easy to interpret. The static semantics of any data can be fully described by a sextuple <name, type, value, address, scope, lifetime>. For data in specifications, address is irrelevant, so we only consider a quintuple for any data d = <name, type, value, scope, lifetime>. The semantics properties of name and lifetime are well known in the compiler literature. Type systems developed for OO programs is valid for OO specification [4]. In fact, only a subset of OO programs' type systems is needed for GOOAL. In the following, we propose three innovative ideas for defining the semantics of GOOAL: axiomatization of OO principles, scope theory, and architectural pipelining axioms.

4.2 Axiomatization of OO Principles

Interestingly, principles of encapsulation, information hiding, and inheritance mainly limit the scope of objects and methods. We propose to axiomatize the OO principles. The following shows two of them:

Axiom of Encapsulation: $\forall m \vDash \neg(\forall x \bullet m \notin x.op)$.

Axiom of Information Hiding: $\forall a \in A.at \vDash$ (a. visibility = Private $\vee$ a. visibility = Protected); $\forall m$, $\forall a \in m.obj\_ref$, $\forall x \in m.op\_ref \vDash$ (access(a, m) $\vee$ evoke(x, m)) $\Rightarrow$ a.visibility = Public.

Here, A.at and A.op denote the set of attributes and methods of A, respectively. The Boolean predicate access (o,m) stands for whether or not o is accessible in m; reference(o,m) for object o is referenced in m; and evoke(x,y) for method x can be evoked in y.

4.3 Scope Theory

In OO specifications, what is crucial at the architectural level is to know, for any method, what are the objects that can be accessed and what are the methods that can be evoked so that all objects of a system can interact with each other and the control flow can pass through a series of methods in an appropriate order. Because of encapsulation, information hiding, and inheritance, this task is much more complex in large OO software than in procedural programs. In procedural programs, we reason about it based on the notion of referencing environments,

Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)
0730-3157/04 $20.00 © 2004 **IEEE**

i.e. all the variables accessible in a statement. As there is no statement in specifications, the equivalent notion does not exists in the OO specification literature. So we propose several sets of innovative scope attributes that generalize the notion of referencing environments. As examples, the *direct object references of method m* (denoted as m.dir_obj_ref) of class A is the set of all the objects referenced in m without calling any method. It includes all the attributes of class A and all the input parameter objects of m. The *indirect object reference of m* (m.ind_obj_ref) is the set of all the objects referenced in m through calling method(s). The *object reference of m (m.obj_ref) is* the set of all the objects referenced in m. In parallel, the *direct operational references* (m.dir_op_ref) and the *indirect operational references* of m are defined as the set of all the methods that are directly and indirectly called in m, respectively, and their union is the *operational references of m*.

4.4 Architectural Pipelining Axioms

Hoare's axiomatic semantics focuses on effect of statements on the value attribute of data. The pre-condition of a statement is not defined but calculated from its post-condition. An implicit assumption with Hoare Logic is that the post-condition of any statement is always equal to the pre-condition of its following statement. This assumption, however, is no longer valid for specification and design, since we usually define both pre- and post-condition for each component. Thus a critical issue is to determine, at the architectural level, how different components can interface correctly. We use a pipeline network metaphor to illustrate a fundamental requirement for software architecture: Consider the whole architecture as a network of pipes with flux passing through, diverging and merging in it, from the entry through the entire pipeline network and leaving it without any leaking. As each method in GOOAL is specified with a pre- and post-condition, which can be interpreted as two joints of a pipe in the network, *the post condition of one method x and the pre-condition of method y following x must guarantee that all data coming out of x completely enter y for processing*. The following illustrates some of architectural pipelining axioms we propose:

$$\frac{\{P_1\}m_1();\{Q_1\} \quad \{P_2\}m_2();\{Q_2\} \quad Q_1 \Rightarrow P_2}{\{P_1\}m_1();m_2();\{Q_2\}},$$

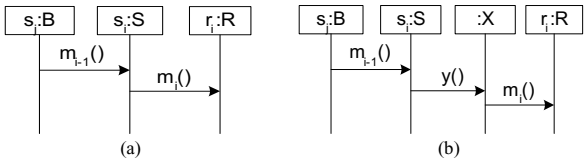
$$\frac{\{P_1\}m_1();\{Q_1\} \quad \{P_2\}m_2();\{Q_2\}}{\{(C \wedge P_1) \vee (\neg C \wedge P_2)\}m_1() < C > m_2()\{Q_1 \vee Q_2\}},$$

$$\frac{\{P_n\}m_n();\{Q_n\} \quad \{P_i\}m_i();\{Q_i\} \quad P_n \Rightarrow P_i \quad Q_i \Rightarrow Q_n}{\{P_n\}m_n(m_i(););\{Q_n\}}.$$

Note that $X < C > Y$ stands for if C then X else Y.

5. Application of GOOAL to UML

A set of scope and lifetime properties of OO specification can be proven illustrating the usefulness of GOOAL for ensuring the consistency of UML diagrams. For example, $\forall m_{i-1} \in NM$, ($life(m_{i-1},r_i) \wedge (r_i \in m_{i-1}.dir\_obj\_ref \vee r_i \in m_{i-1}.ind\_obj\_ref\_acc) \wedge (m_i \in r_i.op \vee (\exists x \in r_i.op \mid x \in AC \bullet m_i \in x.op\_ref\_acc)) \wedge (\forall x \in m_i.sig, life(m_{i-1},x) \wedge (x \in m_{i-1}.dir\_obj\_ref \vee x \in m_{i-1}.ind\_dir\_obj\_ref\_acc))$.



Here NM denotes the set of nested methods, and life(x,y) means that object x is within its lifetime in method y. For UML interaction diagrams, we require that accessors, i.e. methods with only one return statement, should not be shown in the diagrams to keep the diagrams abstract. The theorem has four parts: 1) $life(m_{i-1},r_i)$ and $life(m_{i-1},x)$ ensure that all the objects involved in the call, i.e. target object r_i and actual parameter objects of the called method m_i ($\forall x \in m_i.sig$), are within their lifetime when m_i is evoked; 2) ($r_i \in m_{i-1}.dir\_obj\_ref \vee r_i \in m_{i-1}.ind\_obj\_ref\_acc$) states that the target object r_i of message $m_i()$ must be either directly referenced in m_{i-1}, which immediately precedes $m_i()$ ($m_{i-1}.dir\_obj\_ref$), or indirectly referenced from m_{i-1} only via a sequence of calls of accessors ($m_{i-1}.ind\_obj\_ref\_acc$). Because if $m_i()$ can only be evoked through an intermediate method, say y() of class X, and y() is not an accessor, then the calling sequence would be $m_{i-1}()$ { y(); … } and y() { $r_i.m_i()$; … }. Hence the equivalent sequence diagram cannot be (a) but (b), for non accessor y() must be shown but it is not in (a); 3) likewise, ($m_i \in r_i.op \vee (\exists x \in r_i.op \mid x \in AC \bullet m_i \in x.op\_ref\_acc)$) states that $m_i()$ must be either a method of target r_i ($r_i.op$) or can be evoked only through a sequence of calls of accessors before evoking $m_i()$; 4) ($\forall x \in m_i.sig, (x \in m_{i-1}.dir\_obj\_ref \vee x \in m_{i-1}.ind\_obj\_ref\_acc)$) is about the accessibility of actual parameter objects of method m_i, $m_i.sig$ standing for the signature of m_i.

Our class experience shows an immediate benefit of our approach: As the semantics properties derived based on GOOAL are valid for both specification and code and no additional formal notation is needed, even undergraduate students with an ordinary mathematics background can learn and apply them for UML consistency checking.

Reference:

1. Bunse C., Atkinson C., The normal object form: bridge the gap from models to code, UML'99, 691-705
2. Paige R., Ostroff J., Brooke P.J., Checking the consistency of collaboration and class diagrams using PVS, ROOM'2002
3. Jackson D., Alloy: A lightweight object-oriented modeling notation, ACM T-SEM, 11(2), 2002, 256-290
4. Abadi M. & Leino K.R.M., A Logic of object-oriented programs, LNCS 1214, 1997