 Electronic Notes in Theoretical Computer Science 187 (2007) 145–159 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Object-Oriented Structure Refinement – A Graph Transformational Approach

# Xiaojian Liu†,‡, Zhiming Liu‡ and Liang Zhao‡

† *Computer School, Northwest Polytechnical University, Xi’an, China*

‡ *UNU-IIST, Macao SAR, China*

*Emails:* [*liuxiaojian@nwpu.edu.cn,*](mailto:liuxiaojian@nwpu.edu.cn){*z.liu, liang*}*@iist.unu.edu*

Abstract

In UML, the general structure of objects, their attributes and relations are modeled as a *class graph*, and an instance of a class graph is defined as an *object graph*. The class graph of a system determines the general properties of objects and how objects collaborate in realizing a use case. In this paper, we define class graphs and their object graphs as *directed labelled graphs*, and investigate in a graph theoretical approach what changes in the object structure maintain the capability of providing services. We define the general notion of *structure refinements*. A structure refinement is a transformation from one graph to another that preserves the *capability* of providing services, that is the resulting class graph should be able to provide at least as well as the original graph. We give a small set of structure refinement rules that is proved to be sound and complete for a kind of structure refinement.

*Keywords:* Object systems, class graphs, object graphs, labelled graphs, graph transformations, refinement

# Introduction

An object program can be represented in the form of *Cdecls* • *Main*, where the *class declaration section Cdecls* declares a sequences of classes with their attributes, meth- ods, and inheritance relations; and *Main* declares a *main class* and *main method* [[5](#_bookmark16)]. The main class declares, as its attributes, the global variables whose types are either primitive built-in data types or classes declared in *Cdecls*. The main method implements an application by calling some *public* methods of *public classes* in the declaration section. A class declaration section can be depicted by a UML class diagram [[6](#_bookmark17)]. Such a class diagram also contains the methods and their bodies. Otherwise, sequence diagrams and state diagrams are needed.

Different class declaration sections *Cdeclsi*, *i* = 1*,* 2, may support the same main class. Formally, if for any *Main*, *Cdecls*2 • *Main* behaves “at least as well as” (or *reﬁnes*) *Cdecls*1 • *Main*, we call *Cdecls*2 a *structure reﬁnement* of *Cdecls*1 [[5](#_bookmark16)]. Here, we are only concerned with functional correctness.

1571-0661 © 2007 Published by Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2006.08.049

Structure refinement is important for an object oriented design to be main- tainable, reusable and cohesive. In this paper, we propose a calculus of structure refinement by using graph transformations. We define a class declaration section as a directed labelled graph, called a *class graph*. The nodes are labelled with names of classes or primitive data types, such as *Int*, *Char* and *String*, and edges are labelled with attribute names (also represents UML associations) or symbol Q denoting the direct *inheritance relation*.

Given a class graph and a set of *public variables* declared in a main class, we define a *system state* as an *object graph*, a directed labelled graph with a *root node ε*. The root represents the reference of the instance of the main class [1](#_bookmark0) . Any node different from the root is labelled by a value which is a pair (*v, T* ), where *v* is either an *object reference* if *T* is a class, or an element of *T* otherwise. For each public variable *x* : *T* , there is an edge labelled by *x* from the root toa node whose type is *T* . There is an edge from (*v*1*, T*1) to (*v*2*, T*2) labelled with *a* if and only if *T*1 is a class and *a* is an edge from *T*1 to *T*2 in the class graph. Therefore, a class graph *CG* defines a set of object graphs for each set of public variables. An execution of a command *c* defined under *CG* from a given object graph (system state) *OG* will change *OG* to another object graph *OG*'.

Then for a structure refinement *ρ* from *CG* to *CG*1, we can derive

1. a transformation *ρo* from an object graph *OG* of *CG* to an object graph *OG*1 of

*CG*1, and

1. a transformation *ρc* from commands defined under *CG* to commands defined under *CG*1, such that
2. the diagram in Figure 1 commutes.

We will prove a small set of structure refinement rules, that is complete for a restricted definition of structure refinement.

*i*

CG



*i*CG

1

OG



OG1

*c*

 OG’



OG1 '

Fig. 1. Structure Refinement

There exists a big body of research on formal semantics of object-oriented pro- grams, e.g. [[3](#_bookmark14),[7](#_bookmark18),[10](#_bookmark21),[4](#_bookmark15),[5](#_bookmark16)]. There is a common feeling, though rarely stated in liter- atures, that *one does not (at least feel very difficult to) understand semantics of object-oriented programs deﬁned by another*. Except for a restricted class of static properties, the different semantic definitions do not seem to be effective for analysis and verification of object-oriented programs. Verification of refinement of object- oriented specifications and designs are even harder. The operational semantics (and its denotational counterpart) proposed in this paper based on class graphs and ob- ject graphs is promising to improve this situation.

1 An object system only has one instance of the main class.

An algebraic calculus of class structures is given in [[7](#_bookmark18)] based on predicate trans- formers. Structure refinement is studied in [[5](#_bookmark16)] in a denotational semantic model. The structure refinement rules in this paper agree all the rules in rCOS, except for

those that allow the removal of redundant classes and attributes and for compress- ing attributes. However, purpose of the graph calculus presented in this paper is to improve the understanding of structure refinement and for future development of tool support to graph transformations.

The work [[9](#_bookmark19),[12](#_bookmark22)] handles class and interface refinements. However, there the focus is substitutability of individual classes in a class structure. Our work, however, investigates the refinement of class models as a whole and supports structure design at different stages of the system development. In [[8](#_bookmark20)], a notion of equivalence between class graphs is proposed. There, the notion is defined according to properties of objects, instead of functionalities and object behavior. Thus, it does not address functional refinement.

The use of object graphs is influenced by notation of graphs for pointer structures in [[1](#_bookmark12)], and the idea of using paths of a graph comes from the trace model of pointers and objects with pointers [[2](#_bookmark13)].

Section 2 shows how a class declaration section can be defined as a directed labelled graph. In Section 3 we define object graphs for class graphs to represent system states. We also propose an informal, yet precise and obviously formalizable, operational semantics of programming commands based on class graphs and object graphs. In Section 4, we define structure refinements between class graphs and their derived relations between object graphs. Section 5 establishes a set of class graph refinement rules and prove that they are sound refinements. We also show that this set of rules are complete with respect to a restricted notion of structure refinement.

# Class Graphs

A class declaration section can be represented as a directed and labelled graph. We use names of *data types* and *classes* to label the nodes and names of attributes and an annotation of inheritance to label the edges. For this, we assume an infinite set CN of class names, an infinite set T of names of primitive data types, an infinite set A of attribute names, and a single name Q to annotate the inheritance relation. Let N be union of *types* in CN and T .

Definition 2.1 A class graph is a directed labelled graph Γ= ⟨*N, A, E*⟩, where

* *N* ⊆ N : is the set of *nodes* representing *types*, including both classes and data types
* *A* ⊆ A: a set of *attributes* names
* *E* ⊆ *N* × (*A* ∪ {Q}) × *N* : are the *edges* of the graph. An edge (*C, a, D*) ∈ *E* for *a* ∈ *A*

means that class *C* has an attribute *a* of type *D*, and (*C,* Q*, D*) ∈ *E* means that *C* is a direct subclass of *D*.

We use ≤ to denote the reflexive and transitive closure of the direct subclass

relation Q, and call *D* a superclass of *C* and *C* a subclass of *D* if *C* ≤ *D* holds. Ob- viously, not all analysis class graphs as defined above correspond *well-formed* class declarations.

Definition 2.2 A class graph Γ= ⟨*N, A, E*⟩ is well-formed if it satisfies the following conditions:

1. Data types are leaves of the graph: if (*C, a, D*) ∈ *E*, then *C* /∈T
2. The inheritance relation is only defined among classes: if (*C,* Q*, D*), then

*C, D* ∈ CN.

1. The inheritance relation is required to satisfy the following conditions
   1. There is at most one Q edge from each class, that is we assume no multiple inheritance.
   2. There is no cycle formed by Q edges.
   3. No attributes of the superclass can be redeclared in the subclasses: if *C*1 ≤ *C*, *C*1 /= *C* and (*C, a, D*) ∈ *E* then (*C*1*, a, C*2) /∈ *E* for any *a* and *D* and *C*2.

We simply call a well-formed class graph a class graph if there is no confusion.

Notice that a class graph has three disjoint sets of edges.

* *data attributes*: are those edges (*C, x, T* ) such that *T* is a primitive type.
* *relational attributes*: are those edges (*C, a, D*) such that *D* is a class. We can also call *a* an association between class *C* and *D*.
* *inheritances*: are the edges (*C,* Q*, D*) for all *C* and *D* in the graph.

We do not consider multiplicities of an association, as that will only introduce multi-objects (container objects) in the object graphs. Neither do we distinguish aggregation from general associations.

For a class node *C* of Γ, we define the following two sets.

* *attr*(*C*) = {*a* ∈ *A* | ∃*D* ∈ *N* · (*C, a, D*) ∈ *E*} denotes all labels of the outgoing edges from

*def*

*C*, i.e. the set of the attributes directly defined in class *C*.

* *Attr*(*C*) = {*a* | ∃*D* · *C* ≤ *D* ∧ *a* ∈ *attr*(*D*)} is the set of labels of the outgoing edges from

*def*

*C* and all its superclasses.

For a labelled graph Γ, we abuse the OO notation *v*0*.a*0*.* ·· · *.ak*−1 to denote a *path* [(*v*0*, a*0*, v*1)*,* (*v*1*, a*1*, v*2)*,* ··· *,* (*vk*−1*, ak*−1*, vk*)]; and use *dest*(*v*0*.a*0*.* ··· *.ak*−1) to denote the *desti- nation vk* of the path. And for two *p*1 = *C.α* and *p*2 = *D.β* such that *D* = *dest*(*C.α*), the concatenation *p*1*.p*2 of *p*1 and *p*2 is *C.α.β*.

Only attributes determine the navigation paths in an object graph. However, the inheritance relation defines the attributes that one class inherits from the others. A sequence *α* = {(*vi, ai, vi*+1) | *vi, vi*+1 ∈ *N, ai* ∈ *A, i* = 0*,. .., k*} of “edges” is called a *navigation path* of class graph Γ= ⟨*N, A, E*⟩ if for all *i* = 0*,. .. ,k*

∃*ui, ui*+1 · (*vi* ≤ *ui* ∧ *vi*+1 ≤ *ui*+1 ∧ (*ui, ai, ui*+1) ∈ *E*)

Notice that the above path is a navigation path iff for all *i* = 0*, ... , k*, *ai* ∈ *Attr*(*vi*) and

*vi*+1 is a subtype of the type declared for *ai* in Γ.

Example 2.3 The left part of Figure [2](#_bookmark1) is an analysis class graph which represents the UML class diagram illustrated in the right part of Figure [2](#_bookmark1).

Person



acct

Guest

Account

resv

stays amnt

trans

resvd

No

acct

Guest

Person

stays

resv

Reservation

|  |  |
| --- | --- |
| Account  amnt: Int | |
|  | |
|  | trans |

Transaction

Reservation

Room

Int

Transaction

Fig. 2. An example

|  |
| --- |
| Room |
| No: Int |
|  |

# Object Graphs and Execution of Commands

A class graph declares a family of types and itself can be understood as a “complex” type whose elements are object graphs. For a class graph Γ, we use *N*Γ to denote its nodes, *E*Γ the edges and *A*Γ the attribute names.

In general, we represent a *value v* of a type *T* as a pair (*r, T* ), where *r* is an element of *T* if *T* is a primitive type and a *reference* otherwise. We assume an infinite set *REF* of references, including a special value *null*. For a class graph, we use *V*Γ to denote the set of all values of types declared in Γ.

* 1. *Object graphs as program states*

An object graph is defined for a class graph Γ and a given finite set *X* of *global variables x*1 : *T*1*,. .., xn* : *Tn* such that all the types *Ti* are elements of the class graph. The variables are assumed to be used in the main method.

Definition 3.1 Let Γ be a class graph and *X* a set of global variable declarations. An object graph of Γ with variables *X*, is a rooted, directed and labelled graph Σ= ⟨*L, N, E, ε*⟩, where

* + - *L* = *X* ∪ *A*Γ is the set of names that will be used to label the edges.
    - A node in *N* is either the root node *ε* or an element in *V*Γ, that represents a data value or a reference with its type.
    - *E* ⊆ *N* × *L* × *N* are the edges of Σ.
    - The root node *ε* has no coming-in edges, and other node must have at least one coming-in edge.
    - For each node *v* in *N* , there is at least one path *p* from the root with *dest*(*p*)= *v*.

An object graph Σ of Γ is *complete and correctly typed* if every attributes of a class in Γ is assigned a value with its correct type. The type system is defined by the navigation paths of the class graph.

Definition 3.2 An object graph Σ of a class graph Γ is complete and correctly typed (CCT) with respect to Γ if the following conditions hold

1. Type correctness of nodes: if (*r, C*) ∈ *N* , then *C* must be node in Γ.
2. Type correctness of attributes: for any edge *e* ∈ *E*
   1. if *e* = (*ε, x,* (*r*2*, D*)) for *x* : *T* ∈ *X*, then *D* ≤ *T* ,
   2. if *e* = ((*r*1*,C*)*, a,* (*r*2*, D*)) then (*C, a, D*) is a navigation path of Γ.
3. Completeness: For each node *v* ∈ *N* ,
   1. if *v* = *ε*, it has one and only one outgoing edge for each *x* : *T* ∈ *X*,
   2. otherwise if *v* = (*r, T* ), then there exists an edge ((*r, T* )*, a,* (*r*1*, T*1)) in Σ iff *T* is a class name in Γ, *r* /= *null* and *a* ∈ *Attr*(*T* ).

We use M*X* (Γ) to denote all the CCT object graphs of Γ for variables *X*, and simply call a CCT object graph an object graph and omit the subscript *X*, when there is no confusion. Figure [3](#_bookmark3) is an example of an object graph of the class graph of Figure [2](#_bookmark1), with *X* = {*y*1 : *Room, y*2 : *Guest, y*3 : *Reservation*} as its global variables.

:Reservation



*y*1

:Room

stays

*y*2 *y*3

:Guest

No

acct

resv

:Account

0810

amnt

trans

1000

:Transaction

Fig. 3. An object graph

For an edge (*C, a, D*) of the class graph Γ, *dtype*(*C.a*) (or simply *dtype*(*a*) when there is no confusion) denotes the type *D*, called the *declared type* of *a* in Γ. For an edge ((*r*1*,C*)*, a,* (*r*2*, D*1)) in the object graph Σ, *type*(*r*1*.a*) denotes *D*1, called the *current type* of attribute *a* of object *r*1 in the object graph Σ. Also, *type*(*r, C*) denotes the *current type C* of the node (*r, C*) in the object graph. Definition [3.2](#_bookmark2) ensures that each object node in the object graph represents an object of a class declared in the class graph, and the current type of each attribute is a subtype of its declared type in the class graph.

The root object, representing the instance of the main class, can access an object or a property of an object via different navigation paths. We can thus use the set of all paths to a node to represent the object that the node intends to model. In Figure [3](#_bookmark3), for example, {*є.y*1*, є.y*2*.stays*} represents the :Room instance, and *є.y*2*.acct.amnt* the value 1000.

* 1. *Execution of a Command*

Object graphs of a class graph can be seen as *states* of the object system. An execution of the object program is an execution of the main method command from an initial state to a final state, if terminates. The execution first calls a method *o.m*(*x*; *y*){*c*} of an object *o* of a class in the class graph Γ, where *x* is input parameter,

*y* the output parameter, and *c* the body command. This object *o* is a node in the initial state Σ0, and *x* is also a node of Σ0 though it can be a data value.

The execution of *o.m*(*x*; *y*) changes the object graph Σ0 according to the semantics of the body *c* of the method *m*(). A syntax and a formal denotational semantics of a OO language is defined in rCOS[[5](#_bookmark16)]. However, here we give it an informal

operational interpretation:

o

a

t v a l(e )

a :=e

(1 )

o

t  v a l(e )

a

o

a

b

o

s

b

a

s a . b := e

o

a :=C .ne w () t

o

a

(r,C )

a1 an

**...**

init(a 1) init(a n)

a

t v a l(e )

(2 )

t

t  v a l(e )

(3 )

Fig. 4. The change of object graphs

1. If *c* is a simple assignment *a* := *e*, where *a* is an attribute of *o*, then the execution changes the edge (*o, a, t*) in Σ0 to the edge (*o, a, val*(*e*)), where *val*(*e*) is the value of expression *e*, which can be an object or a data value. This is shown in Figure [4](#_bookmark4)(1).
2. If *c* is a complex assignment *a.b* := *e*, where *a* is an attribute of *o* of a class type and *b* is an attribute of *o.a*, then the execution changes the path (*o, a.b, t*) to the path (*o, a.b, val*(*e*)) as illustrated in Figure [4](#_bookmark4)(2). The general attribute assignment *a.b*1*.* ·· · *.bk* := *e* is performed by induction.
3. If *c* is an object creation *a* := *C.new*(), where *a* is an attribute of *o* and *C* is a class node of Γ and a subclass of *dtype*(*a*), the execution changes the edge (*o, a, t*) in Σ0 to a newly created rooted graph with (*r, C*) as the root and the initial values of the attributes of *C* (that we would like to ignore here) as nodes. This is shown in Figure [4](#_bookmark4)(3).
4. The meaning of compositions of commands can be defined inductively.

If we want to consider type casting, we just need to extend the node from a pair to a triple (*r, C, D*) to represent the reference, the current type and the casted type.

Notice that the execution of a command may cause an object in the object graph unreachable from the root. In this case, the node of object will be deleted from the object graph, just like what garbage collection does.

For an object graph Σ0 of Γ and an object node *o* in it, the set of all possible object graphs caused by the execution of *o.m*() for a method *m*(){*c*} defined in the type of *o* is denoted by [[*o.m*()]]Γ(Σ0).

We believe that it is easy to prove that the above operational semantics is consistent with the rCOS semantics defined in [[5](#_bookmark16)], though we have not done the proof. We allow nondeterministic choice and specification commands too. Then the execution of a method maps an object graph to a set of object graphs. Thus, commands refinement, as well as equivalence, can be defined for methods of class.

Definition 3.3 [Method refinement] Let

1. Γ be a class graph,
2. *X* be a set of variables with types in Γ,
3. *m*(*u* : *T*1; *v* : *T*2){*c*1} and *m*(*u* : *T*1; *v* : *T*2){*c*2} be two methods defined in a class *C* for some *x* : *C* ∈ *X*.

We say method *m*(*u* : *T*1; *v* : *T*2){*c*2} is a refinement of method *m*(*u* : *T*1; *v* : *T*2){*c*1}, de- noted by *m*(*u* : *T*1; *v* : *T*2){*c*1}± *m*(*u* : *T*1; *v* : *T*2){*c*2}, if for any object diagram Σ of Γ, any node *u* of Σ such that *type*(*u*) ≤ *T*1, the set [[*x.m*(){*c*2}]]Γ(Σ) is a subset of [[*x.m*(){*c*1}]]Γ(Σ), where *x* is the object that *x* points to in the object graph Σ.

However, to support design by stepwised refinements, we must be able to change the structure of the class graph, though the public class names are not changed.

Definition 3.4 Let PC be a set of class names, Γ1 and Γ2 be two class graphs both containing PC as some of their nodes. Γ2 is a structure refinement of Γ1, denoted as Γ1 ± Γ2, if for any *C* ∈ PC and a variable *x* : *C*, there exists a mapping *ρo* from

M(Γ1){*x*:*C*} to M(Γ2){*x*:*C*} such that for any method *m*(*u* : *T*1; *v* : *T*2){*c*1} defined in class *C*

of Γ1, we can define a correspondence method *m*(*u* : *T*1; *v* : *T*2){*c*2} in class *C* of Γ2 and

[[*x.m*(){*c*2}]]Γ2 (*ρo*(Σ)) ⊆ *ρo*([[*x.m*(){*c*1}]]Γ1 (Σ))

# Structure Transformation

* 1. *Structure transformation*

We now show that some refinement can be realized by certain class graph transfor- mations.

Definition 4.1 Let Γ1 and Γ2 be class graphs, *f* a subset of the class nodes of Γ1. A mapping *ρ* from Γ1 to Γ2 is a *f* -framed transformation, denoted by *ρ*[*f* ], if the following conditions hold

1. the restriction *ρ*|*N*1 of *ρ* to the nodes *N*1 of Γ1 maps each class name in the frame

*f* to itself, that is *ρ*(*C*)= *C* ∈ *N*2 for each *C* ∈ *f* .

1. the restriction *ρ*|*E*1 to the edges *E*1 of Γ1 maps each relational or inheritance edge (*C, a, D*) ∈ *E*1 to a nonempty path from *ρ*|*N*1 (*C*) to *ρ*|*N*1 (*D*) in Γ2; and maps each data attribute (*C, a, T* ) to a nonempty set of paths of Γ2 starting from *ρ*|*N*1 (*C*) with destinations that are data types in T .

We decompose the restriction *ρ*|*E*1 into two restrictions *ρ*|*r* and *ρ*|*d* to the relational (including inheritance) edges and data attributes, respectively.

 



C

*a*

D

*x*

Int



*init*(*D.x*)=*7*

*init*(*F.x1* )=2*, init*(*G.x2* )=*5*



F

*b*

G

*x1 x2*

*y*

C

*a*

E

Int

Fig. 5. An example of structure transformation

Obviously, not all framed structure transformations defined above are structure refinements. In what follows, we give a number of sufficient conditions for such a transformation to be a structure refinement.

Proposition 4.2 *A framed transformation ρ*[*f* ] *from* Γ1 *to* Γ2 *is a* structure refine- ment *if it satisﬁes the following properties:*

* 1. *If* (*C,* Q*, D*) *is an inheritance edge, then ρ*|*r* (*C,* Q*, D*) *is a path containing only inherence edges.*
  2. *If* (*C, a, D*) *is a relational edge, then the path ρ*|*r* (*C, a, D*) *contains at least one relational attribute.*
  3. *For two different relational or inheritance edges* (*C*1*, a*1*, D*1) *and* (*C*2*, a*2*, D*2)*, ρ*|*r*(*C*1*, a*1*, D*1) *is not a suffix of ρ*|*r*(*C*2*, a*2*, D*2)*.*
  4. *For any data edges* (*C*1*, a*1*, T*1) *and* (*C*2*, a*2*, T*2)*, a path p*1 *in ρ*|*d*(*C*1*, a*1*, T*1) *and a path p*2 *in ρ*|*d*(*C*2*, a*2*, T*2)*, p*1 *is not a suffix of p*2 *unless C*1 = *C*2*, a*1 = *a*2 *and p*1 = *p*2 *(obviously T*1 = *T*2 *too).*
  5. *For a data edge* (*C, a, T* )*, let ρ*|*d*(*C, a, T* )= {*C*1*.β*1*.a*1*,. .., C*1*.βn.an*} *such that dest*(*C*1*.βi.ai*)= *Ti for* 1 ≤ *i* ≤ *n, there exists a surjective operation g of the type T*1 × *.. .* × *Tn* → *T such that the initial value of C.a can be calculated from those of the target attributes: init*(*C.a*)= *g*(*init*(*D*1*.a*1)*, . . . , init*(*Dn.an*))*, where Di* = *dest*(*C*1*.βi*)*.*

A structure transformation from Γ1 to Γ2 in fact defines an *implementation* of the classes, their attributes and associations in Γ1 by those of Γ2. A single relational attribute (edge) in Γ1 can be realized by a path, and an data attribute can be a set of paths in Γ2. These are captured by conditions (ii)-(v).

The validity of the proposition is to be established in the next section in two steps:

* Soundness: prove a small set of rules that are structure refinements.
* Completeness: prove that any structure transformation that satisfies the condi- tions in Proposition [4.2](#_bookmark7) can be obtained by sequentially applying the proposed refinement rules.

Example 4.3 Figure [5](#_bookmark6) illustrates a structure transformation *ρ*[*f* ], where *f* = {*C*}, *ρ*|*N*1 (*C*)= *C*, *ρ*|*N*1 (*D*)= *F* , *ρ*|*r*(*C, a, D*)= *C.a.*Q, *ρ*|*d*(*D, x, Int*)= {*F.x*1*, F.b.x*2}, and the addition operation on integers preserves the initial values of attributes:

*init*(*D.x*)= *init*(*F.x*1)+ *init*(*G.x*2).

* 1. *Structure relation between object graphs*

Given variables *X*, a structure transformation *ρ* from class graph Γ1 to class graph

Γ2 determines a derived relation *ρo* between M*X* (Γ1) and M*X* (Γ2).

Definition 4.4 Let *ρ* be a structure transformation from class graph Γ1 to class graph Γ2 with a frame {*C*1*,. .., Cn*} satisfying the conditions in Proposition [4.2](#_bookmark7), and

*X* = {*x*1 : *C*1*,. .., xn* : *Cn*} be variables. Σ1 and Σ2 be object graphs with variables *X* for Γ1

and Γ2 respectively. The derived structure relation of *ρ*, denoted by *ρo*(Σ1*,* Σ2), is a relation between object graphs M(Γ1) and M(Γ2) such that the following conditions hold

1. for each edge *e* = ((*r*1*,C*)*, a,* (*r*2*, D*)) in Σ1 such that *a* is a relational attribute, there is a path *ρo*(*e*)= ((*r*' *,C*')*, α,* (*r*' *, D*')) in Σ2 such that

1 2

* 1. *C*' = *ρ*|*N* (*C*), *D*' = *ρ*|*N* (*D*), and

1 1

* 1. (*C*'*, α, D*')= *ρ*|*r* (*C, a, D*)\Q that is the path obtained from *ρ*|*r* (*C, a, D*) after re- moving the inheritance edges.

1. for each data attribute edge *e* = ((*r, C*)*, a,* (*v, T* )) in Σ1, there is a set of paths

*ρo*(*e*)= {((*r*'*,C*')*, β*1*.a*1*,* (*v*1*, T*1))*,.. .,* ((*r*'*,C*')*, βn.an,* (*vn, Tn*))}

such that *v* = *g*(*v*1*,. .. , vn*), *C*' = *ρ*|*N* (*C*) and *ρ*|*d*(*C, a, T* )= {*C*'*.γi.ai*}, where *βi* = *γi*\Q for

1

1 ≤ *i* ≤ *n* and *g* is the primitive operation corresponding to (*C, a, T* ) in Γ1.

# Structure Refinement Rules

We give a set of rules in Figure [6](#_bookmark9) which transform a class graph Γ1 = ⟨*A*1*, N*1*, E*1⟩ to an- other Γ2 = ⟨*A*2*, N*2*, E*2⟩. Notice that each rule has a frame representing the unchanged class names before and after the transformation. The purpose of introducing pre- condition for each rule is to ensure that the class graph after transforming is a well-formed one. Here, we use C to denote the set of class names declared in Γ1.

* 1. *Soundness*

It is straightforward to prove that each rule defines a structure transformation on class graphs, which satisfies the conditions of Proposition [4.2](#_bookmark7). Thus each rule *R* determines a structure relation *Ro* between the object graphs of the corresponding class graphs.

Also, each rule *R* derives a transformation *Rc*. *Rc* transforms a command *c* that is *syntactically well-formed* under Γ1 to a command *Rc*(*c*) that is *syntactically well- formed* under Γ2. This means that the variables and types in the command are all defined in the graph. Statements and expressions are correctly typed [[5](#_bookmark16)]. The command transformations are given in Figure [7](#_bookmark10), where notation [*D/C*] denotes a substitution for each class name *C* by *D*, and notation [*C.b/C.a*] denotes a substitution for each expression of the form *e.a* by another expression *e.b* if the declared type of *e* is

*C*. The meaning of notations [*C.b.a/C.a*] and [*g*(*C.x*1*,* ·· · *, C.xn*)*/C.x*] is defined inductively.

Theorem 5.1 (Soundness of Rules) *If rule R*[*f* ] *transforms* Γ1 *to* Γ2*, then*

Γ1 ± Γ2*.*

|  |  |  |  |
| --- | --- | --- | --- |
| Rules | Description | Precondition | Frame |
| R1 Rename a class  C D | class name *C* is changed to a different name *D* | *C* ∈ *N*1 *,D* ∈*/ N*1 | C\ {*C*} |
| R2 Merge classes  *a a*  *c c*  C D D  *b b* | merge class *C* to another | *C, D* ∈ *N*1 , (*C, a, E*)*,* (*D, a, F* ) ∈ *E*1 | C\ {*C*} |
| class *D* such that all the | implies *E* = *F* and |  |
| outgoing (incoming) edges | *init*(*C.a*)= *init*(*D.a*) |  |
| from (to) *C* become those |  |  |
| from (to) *D*, and the edges |  |  |
| between *C* and *D* become |  |  |
| self loop edges |  |  |
| R3 Rename  an attribute  *a b*  C D C D | the name of an attribute  (*C, a, D*) is changed to (*C, b, D*) | (*C, a, D*) ∈ *E*1, *a* /= *b* and  (*E, b, F* ) ∈*/ E*1 for any *F* ∈ *N*1  if *E* is a superclass or subclass of *C* | C |
| R4 Add a new class  + C | add a class *C* | *C* ∈*/ N*1 | C |
| R5.1 Add an attribute  *a*  C D C D | add an attribute (*C, a, D*) | *C, D* ∈ *N*1 , (*E, a, F* ) ∈*/ E*1 for any *F* ∈ *N*1 if *E* is a superclass or subclass of *C* | C |
| R5.2 Add  an inheritance  C D C D | add an inheritance edge  (*C,* Q*,D*) | *C, D* ∈ *N*1 , for each *C*' ∈ *N*1 (*C,* Q*,C*') ∈*/ E*1, *D* /≤ *C*, and for each pair of edges (*E, a, E*')*,* (*F, b, F* ') ∈ *E*1 where *E* ≤ *C* and *F* ≥ *D*, we have *a* /= *b* | C |
| R6.1 Forward an attribute through a relational attribute  *b b*  C D C D  *a a*  E E | an attribute (*C, a, E*) is for- | (*C, a, E*)*,* (*C, b, D*) ∈ *E*1, | C |
| warded through a relational | (*F, a, F* ') ∈*/ E*1 for any *F* ' ∈ *N*1 if |  |
| attribute (*C, b, D*) to form | *F* is a superclass or subclass |  |
| another attribute (*D, a, E*) | of *D* |  |
| R6.2 Forward an | an attribute (*C, a, E*) is for- | (*C, a, E*)*,* (*C,* Q*,D*) ∈ *E*1 , | C |
| attribute through | warded through an inheri- | (*F, a, F* ') ∈*/ E*1 for any *F* ' ∈ *N*1 if |  |
| an inheritance  C D C D  *a a* | tance edge (*C,* Q*,D*) to form  another attribute (*D, a, E*) | *F* is a superclass or subclass  of *D* |  |
| E E |  |  |  |
| R7 Decompose  a data attribute  *x* T1  *1*  *x*  C T C  *xn* Tn | a data attribute (*C, x, T* ) | (*C, x, T* ) ∈ *E*1 , there exists a | C |
| is decomposed to a | surjective primitive operation |  |
| set of data attributes | *g* : *T*1 × *.. .* × *Tn* → *T* preserving |  |
| (*C, x*1*, T*1)*, . ..,* (*C, xn, Tn*) | the initial values of at- |  |
|  | tributes, and for each 1 ≤ *i* ≤ *n*, |  |
|  | (*D, xi,D*') ∈*/ E*1 for any *D*' ∈ *N*1 if |  |
|  | *D* is a superclass or subclass |  |
|  | of *C* |  |
| R8 Decompose an edge  *a a*  C D C E D | a relational attribute or in- heritance edge (*C, a, D*) is decomposed to two edges  (*C, a, E*) and (*E,* Q*,D*) | (*C, a, D*) ∈ *E*1, *E* ∈ *N*1 | C |

Fig. 6. Basic Rules

|  |  |
| --- | --- |
| Rule *R* | Command Transformation *Rc* |
| R1 Rename a class | *Rc*(*c*)= *c*[*D/C*] |
| R2 Merge classes | *Rc*(*c*)= *c*[*D/C*] |
| R3 Rename an attribute | *Rc*(*c*)= *c*[*C.b/C.a*] |
| R4 Add a new class | *Rc*(*c*)= *c* |
| R5.1 Add an attribute | *Rc*(*c*)= *c* |
| R5.2 Add an inheritance | *Rc*(*c*)= *c* |
| R6.1 Forward an attribute through a relational attribute | *Rc*(*c*)= *c*[*C.b.a/C.a*] |
| R6.2 Forward an attribute through an inheritance | *Rc*(*c*)= *c* |
| R7 Decompose a data attribute | *Rc*(*c*)= *c*[*g*(*C.x*1*,* ·· · *, C.xn*)*/C.x*] |
| R8 Decompose an edge | *Rc*(*c*)= *c* |

Fig. 7. Command transformation

Proof (Outline) If a method *m*(*u* : *T*1; *v* : *T*2){*c*} is defined in class *C* (*C* ∈ *f* ) of Γ1, then we can also define a corresponding method *m*(*u* : *T*1; *v* : *T*2){*Rc*(*c*)} in class *C* of Γ2 such that the condition given in Definition [3.4](#_bookmark5) holds.

This theorem implies that all rules given in Figure [6](#_bookmark9) are structure refinements. It also shows the commutativity of the diagram of Figure 1 in the Section 1. Obviously, the structure refinement relation defined in Definition [3.4](#_bookmark5) is transitive.

Corollary 5.2 *If* Γ1 *is transformed to* Γ2 *by a sequential applications of rules*

*R*1[*f*1 ]*, ... , Rk*[*fk* ] *, then* Γ1 ± Γ2*, provided f*1 ∩· · · ∩ *fk* /= ∅*.*

* 1. *Validity of Proposition* [*4.2*](#_bookmark7)

We now establish the validity of Proposition [4.2](#_bookmark7) as the following completeness theorem.

Theorem 5.3 (Completeness of Rules) *If ρ*[*f* ] *is a structure transformation from* Γ1 *to* Γ2 *that satisﬁes the conditions of* Proposition [*4.2*](#_bookmark7)*, then there exist a ﬁnite number of sequential applications of rules R*1[*f*1 ] *,.. ., Rk*[*fk* ] *that transforms* Γ1 *to* Γ2 *and f* ⊆ *fi for* 1 ≤ *i* ≤ *k.*

Proof (Outline) Given a structure transformation *ρ*[*f* ], we can identify a sequence of applications of refinement rules as follows.

1. change each class *C* to *ρ*|*N*1 (*C*), by applications of R1 and R2. When mapping

*ρ*|*N*1 is injective, R2 is not needed.

1. decompose each data attribute (*C, x, T* ) to a set of data attributes:

{(*C, x*1*, T*1)*,. .. ,* (*C, xn, Tn*)}*,*

provided *ρ*|*d*(*C, x, T* )= {(*C.β*1*.x*1)*, ... ,* (*C.βn.xn*)} and *dest*(*C.βi.xi*)= *Ti*, using rule R7.

1. for edges there are two cases
   1. change each data edge (*C, x, T* ) to a path *C.β.x* in *ρ*|*d*(*C, x, T* ), using R4, R5 and

R6.

* 1. change each relational or inheritance edge (*C, a, D*) to a path *ρ*|*r* (*C, a, D*), by applications of R3, R4, R5, R6 and R8.

1. add additional nodes and edges by using rules R4 and R5 if necessary.

Example 5.4 For the structure transformation illustrated in Example [4.3](#_bookmark8), Figure [8](#_bookmark11) shows the applications of the rules that transform Γ1 to Γ2.

step 1: rename classes

R1



C

*a*

D

*x*

Int



C

*a*

F

*x*

Int

step 2: decompose data attributes

*x1*

R7



C

*a*

F

Int

*x2*



step 3: transform edges



F *b*

G

*x*

C

*a*

*x1*

E

Int

R4;R5.1



F

*x2*

R4;R8

*x*

C

*a*

*1*

E

Int

F *b* G

*x2*



*x*

R6.1



step 4: add extra nodes and edges

*1 2*

C *a* E Int







F

*b*

G

R5.1

*x1 y*

*x2*

C

*a*

E

Int



Fig. 8. An example

Now we have established the validity of Proposition [4.2](#_bookmark7).

Corollary 5.5 Proposition [*4.2*](#_bookmark7) *holds.*

# Conclusion

We have proposed a graph theoretical approach to studying the relation between changes in class declarations and changes in method definitions. The main purpose is to make the semantics and refinement of object oriented programs easier to un- derstand and more operational. We believe this is important for development of tool support to object system development by transformations [[11](#_bookmark23)].

Another contribution of this paper is the proposal of an operational semantics for object oriented programs in the graph theoretical notation. This allows us to un- derstand the execution of an object program in the same as an imperative program by taking graphs as the states. In our future work, we will study this operational semantics together with the study of operations and properties of graphs. This will lead to the development of a Hoare-logic for object-oriented programs with predicates of graphs.

The approach presented here suggests a design method of object oriented systems that allows the automatic derivation of methods definitions from their specifications and structure transformation.

This work is still at its early stage in that the structure refinements are restricted to only expanding the graph. No rules are provided for removing classes and at- tributes or compressing long paths to shorter paths. Therefore, some refinement laws proved in rCOS [[5](#_bookmark16)] have not been established. In other words, only “true” refinements are treated, but not the “abstractions” that preserve functionality. The difficulty in establishing this kind of abstraction rules is due to the fact that we con- sider arbitrary methods definable in a class graph. In further work, we will consider rules of class refinement for fixed methods in the public methods.

# References

1. Y. Chen and J. Sanders. Compositional reasoning for pointer structures. *Proc. of 8th International Conference on Mathematics of Program Construction (MPC06)*. Springer. 2006.
2. C.A.R. Hoare and J. He. A trace model for pointers and objects. In *ECOOP’99, LNCS1628*, pages 1-17. Springer, 1999.
3. M.Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
4. P. America and F.de Boer. Reasoning about dynamically eveloving process structures. *Formal Aspects of Computing*, 6(3):269-316,1994.
5. J. He, X. Li and Z. Liu. rCOS: A Refinement Calculus for Object-oriented Systems. Accepted for publication in *Theoretical Computer Science*. Also available as Technical Report 322. UNU/IIST, P.O.Box 3058, Macao SAR China ([http://www.iist.unu.edu](http://www.iist.unu.edu/) ).
6. Z.Liu, J.He, X.Li and Y.Chen. A relational model for formal requirements analysis in UML. Proc. of ICFEM03, LNCS 2885, 641-664, 2003.
7. P. Borba, A. Sampaio, and M. Cornelio. A Refinement Algebra for Object-Oriented Programming. In

*ECOOP 2003, LNCS 2743*, pages 457-482. Springer, 2003.

1. R. Gheyi, T. Massoni and P. Borba. An abstract equivalence notation for object models. *Proc. SBMF2004*, pages 1-15. 2004.
2. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proc of FME’97, LNCS 1313*. Springer, 1997.
3. EB Johnsen and O. Owe. Object-Oriented Specification and Open Distributed Systems. In *LNCS 2635, pages 137-164*. Springer, 2004.
4. Tata Consultancy Services. MasterCraft.(<http://www.tata-mastercraft.com/>)
5. R. Back, A.Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 2:18-40, 2000.