



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# **Object-Oriented Structure Refinement – A Graph Transformational Approach**

---

**Xiaojian Liu, Zhiming Liu and Liang Zhao**

July 2006

## UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research **R**, Technical **T**, Compendia **C** or Administrative **A**. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

P.O. Box 3058  
Macau

---

# Object-Oriented Structure Refinement – A Graph Transformational Approach

---

**Xiaojian Liu, Zhiming Liu and Liang Zhao**

## **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*

**Xiaojian Liu** is a Fellow of UNU-IIST from Northwest Polytechnical University, Xi'an, China, where he is a lecturer of Computer School. His research areas include refinement calculus, formal methods and software engineering.

**Zhiming Liu** is a Research Fellow at UNU/IIST. His research interests include theory of computing systems, emphasising sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iist.unu.edu.

**Liang Zhao** is a fellow of UNU-IIST from Peking University, Beijing, China, where he is a master student. His research interests includes type systems and semantics of programming languages, formal methods in software development, OO programming and development and so on. E-mail:liang@iist.unu.edu.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Class Graphs</b>	<b>2</b>
<b>3</b>	<b>Object Graphs and Execution of Commands</b>	<b>4</b>
3.1	Object graphs as program states . . . . .	5
3.2	Execution of a Command . . . . .	6
<b>4</b>	<b>Structure Transformation</b>	<b>8</b>
4.1	Structure transformation . . . . .	8
4.2	Structure relation between object graphs . . . . .	10
<b>5</b>	<b>Structure Refinement Rules</b>	<b>11</b>
5.1	Soundness . . . . .	11
5.2	Validity of Proposition 1 . . . . .	11
<b>6</b>	<b>Conclusion</b>	<b>14</b>



## 1 Introduction

An object program can be represented in the form of  $Cdecls \bullet Main$ , where the *class declaration section*  $Cdecls$  declares a sequences of classes with their attributes, methods, and inheritance relations; and  $Main$  declares a *main class* and *main method* [5]. 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]. Such a class diagram also contains the methods and their bodies. Otherwise, sequence diagrams and state diagrams are needed.

Different class declaration sections  $Cdecls_i$ ,  $i = 1, 2$ , may support the same main class. Formally, if for any  $Main$ ,  $Cdecls_2 \bullet Main$  behaves “at least as well as” (or *refines*)  $Cdecls_1 \bullet Main$ , we call  $Cdecls_2$  a *structure refinement* of  $Cdecls_1$  [5]. Here, we are only concerned with functional correctness.

Structure refinement is important for an object oriented design to be maintainable, 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  $\triangleright$  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*  $\varepsilon$ . The root represents the reference of the instance of the main class <sup>1</sup>. 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 to a 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  $\rho$  from  $CG$  to  $CG_1$ , we can derive

1. a transformation  $\rho_o$  from an object graph  $OG$  of  $CG$  to an object graph  $OG_1$  of  $CG_1$ , and
2. a transformation  $\rho_c$  from commands defined under  $CG$  to commands defined under  $CG_1$ , such that
3. 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.

There exists a big body of research on formal semantics of object-oriented programs, e.g. [3, 7, 10, 4, 5]. There is a common feeling, though rarely stated in literatures, that *one does not (at least feel very difficult*

<sup>1</sup>An object system only has one instance of the main class.

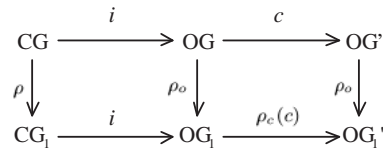


Figure 1: Structure Refinement

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 object graphs is promising to improve this situation.

An algebraic calculus of class structures is given in [7] based on predicate transformers. Structure refinement is studied in [5] 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 compressing 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, 12] 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], 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], and the idea of using paths of a graph comes from the trace model of pointers and objects with pointers [2].

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.

## 2 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  $\mathcal{CN}$  of class names, an infinite set  $\mathcal{T}$  of names of primitive data types, an infinite set  $\mathcal{A}$  of attribute names, and a single name  $\triangleright$  to annotate the inheritance relation. Let  $\mathcal{N}$  be union of types in  $\mathcal{CN}$  and  $\mathcal{T}$ .

**Definition 1** A class graph is a directed labelled graph  $\Gamma = \langle N, A, E \rangle$ , where

- $N \subseteq \mathcal{N}$ : is the set of nodes representing types, including both classes and data types
- $A \subseteq \mathcal{A}$ : a set of attributes names
- $E \subseteq N \times (A \cup \{\triangleright\}) \times N$ : are the edges of the graph. An edge  $(C, a, D) \in E$  for  $a \in A$  means that class  $C$  has an attribute  $a$  of type  $D$ , and  $(C, \triangleright, D) \in E$  means that  $C$  is a direct subclass of  $D$ .

We use  $\preceq$  to denote the reflexive and transitive closure of the direct subclass relation  $\triangleright$ , and call  $D$  a superclass of  $C$  and  $C$  a subclass of  $D$  if  $C \preceq D$  holds. Obviously, not all analysis class graphs as defined above correspond *well-formed* class declarations.

**Definition 2** A class graph  $\Gamma = \langle N, A, E \rangle$  is **well-formed** if it satisfies the following conditions:

1. Data types are leaves of the graph: if  $(C, a, D) \in E$ , then  $C \notin \mathcal{T}$
2. The inheritance relation is only defined among classes: if  $(C, \triangleright, D)$ , then  $C, D \in \mathcal{CN}$ .
3. The inheritance relation is required to satisfy the following conditions
  - (a) There is at most one  $\triangleright$  edge from each class, that is we assume no multiple inheritance.
  - (b) There is no cycle formed by  $\triangleright$  edges.
  - (c) No attributes of the superclass can be redeclared in the subclasses: if  $C_1 \preceq C$ ,  $C_1 \neq C$  and  $(C, a, D) \in E$  then  $(C_1, a, C_2) \notin E$  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  $(C, \triangleright, D)$  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  $\Gamma$ , we define the following two sets.

- $attr(C) \stackrel{def}{=} \{a \in A \mid \exists D \in N \cdot (C, a, D) \in E\}$  denotes all labels of the outgoing edges from  $C$ , i.e. the set of the attributes directly defined in class  $C$ .
- $Attr(C) \stackrel{def}{=} \{a \mid \exists D \cdot C \preceq D \wedge a \in attr(D)\}$  is the set of labels of the outgoing edges from  $C$  and all its superclasses.

For a labelled graph  $\Gamma$ , we abuse the OO notation  $v_0.a_0 \cdots .a_{k-1}$  to denote a *path*

$$[(v_0, a_0, v_1), (v_1, a_1, v_2), \cdots, (v_{k-1}, a_{k-1}, v_k)]$$

and use  $dest(v_0.a_0 \cdots .a_{k-1})$  to denote the *destination*  $v_k$  of the path. And for two  $p_1 = C.\alpha$  and  $p_2 = D.\beta$  such that  $D = dest(C.\alpha)$ , the concatenation  $p_1.p_2$  of  $p_1$  and  $p_2$  is  $C.\alpha.\beta$ .

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

$$\alpha = \{(v_i, a_i, v_{i+1}) \mid v_i, v_{i+1} \in N, a_i \in A, i = 0, \dots, k\}$$

of “edges” is called a *navigation path* of class graph  $\Gamma = \langle N, A, E \rangle$  if for all  $i = 0, \dots, k$

$$\exists u_i, u_{i+1} \cdot (v_i \preceq u_i \wedge v_{i+1} \preceq u_{i+1} \wedge (u_i, a_i, u_{i+1}) \in E)$$

Notice that the above path is a navigation path iff for all  $i = 0, \dots, k$ ,  $a_i \in Attr(v_i)$  and  $v_{i+1}$  is a subtype of the type declared for  $a_i$  in  $\Gamma$ .

**Example 1** Figure 2(a) is an analysis class graph representing UML class diagram Figure 2(b).

### 3 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  $\Gamma$ , we use  $N_\Gamma$  to denote its nodes,  $E_\Gamma$  the edges and  $A_\Gamma$  the attribute names.

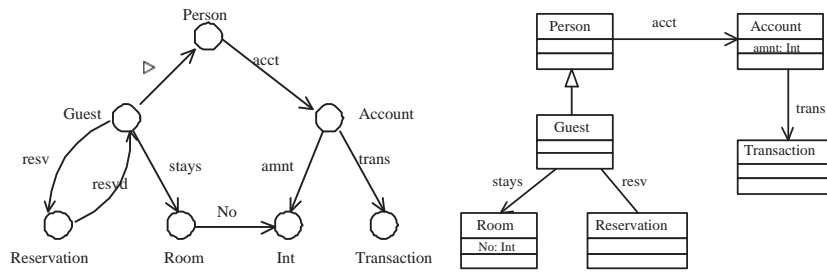


Figure 2: An example

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  $V_\Gamma$  to denote the set of all values of types declared in  $\Gamma$ .

### 3.1 Object graphs as program states

An object graph is defined for a class graph  $\Gamma$  and a given finite set  $X$  of *global variables*  $x_1 : T_1, \dots, x_n : T_n$  such that all the types  $T_i$  are elements of the class graph. The variables are assumed to be used in the main method.

**Definition 3** Let  $\Gamma$  be a class graph and  $X$  a set of global variable declarations. An **object graph** of  $\Gamma$  with variables  $X$ , is a rooted, directed and labelled graph  $\Sigma = \langle L, N, E, \varepsilon \rangle$ , where

- $L = X \cup A_\Gamma$  is the set of names that will be used to label the edges.
- A node in  $N$  is either the **root node**  $\varepsilon$  or an element in  $V_\Gamma$ , that represents a data value or a reference with its type.
- $E \subseteq N \times L \times N$  are the edges of  $\Sigma$ .
- The root node  $\varepsilon$  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  $\Sigma$  of  $\Gamma$  is *complete and correctly typed* if every attributes of a class in  $\Gamma$  is assigned a value with its correct type. The type system is defined by the navigation paths of the class graph.

**Definition 4** An object graph  $\Sigma$  of a class graph  $\Gamma$  is **complete and correctly typed (CCT)** with respect to  $\Gamma$  if the following conditions hold

1. **Type correctness of nodes:** if  $(r, C) \in N$ , then  $C$  must be node in  $\Gamma$ .
2. **Type correctness of attributes:** for any edge  $e \in E$ 
  - (a) if  $e = (\varepsilon, x, (r_2, D))$  for  $x : T \in X$ , then  $D \preceq T$ ,
  - (b) if  $e = ((r_1, C), a, (r_2, D))$  then  $(C, a, D)$  is a navigation path of  $\Gamma$ .
3. **Completeness:** For each node  $v \in N$ ,
  - (a) if  $v = \varepsilon$ , it has one and only one outgoing edge for each  $x : T \in X$ ,
  - (b) otherwise if  $v = (r, T)$ , then there exists an edge  $((r, T), a, (r_1, T_1))$  in  $\Sigma$  iff  $T$  is a class name in  $\Gamma$ ,  $r \neq \text{null}$  and  $a \in \text{Attr}(T)$ .

We use  $\mathcal{M}_X(\Gamma)$  to denote all the CCT object graphs of  $\Gamma$  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 is an example of an object graph of the class graph of Figure 2, with  $X = \{y_1 : \text{Room}, y_2 : \text{Guest}, y_3 : \text{Reservation}\}$  as its global variables.

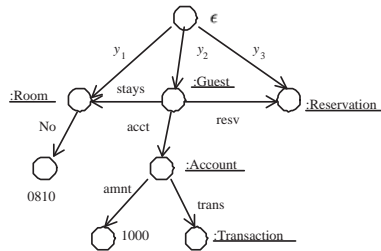


Figure 3: An object graph

For an edge  $(C, a, D)$  of the class graph  $\Gamma$ ,  $dtype(C.a)$  (or simply  $dtype(a)$  when there is no confusion) denotes the type  $D$ , called the *declared type* of  $a$  in  $\Gamma$ . For an edge  $((r_1, C), a, (r_2, D_1))$  in the object graph  $\Sigma$ ,  $type(r_1.a)$  denotes  $D_1$ , called the *current type* of attribute  $a$  of object  $r_1$  in the object graph  $\Sigma$ . Also,  $type(r, C)$  denotes the *current type*  $C$  of the node  $(r, C)$  in the object graph. Definition 4 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, for example,  $\{\varepsilon.y_1, \varepsilon.y_2.stays\}$  represents the :ROOM instance, and  $\varepsilon.y_2.acct.amnt$  the value 1000.

### 3.2 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  $\Gamma$ , where  $x$  is input parameter,  $y$  the output parameter, and  $c$  the body command. This object  $o$  is a node in the initial state  $\Sigma_0$ , and  $x$  is also a node of  $\Sigma_0$  though it can be a data value.

The execution of  $o.m(x; y)$  changes the object graph  $\Sigma_0$  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]. However, here we give it an informal operational interpretation:

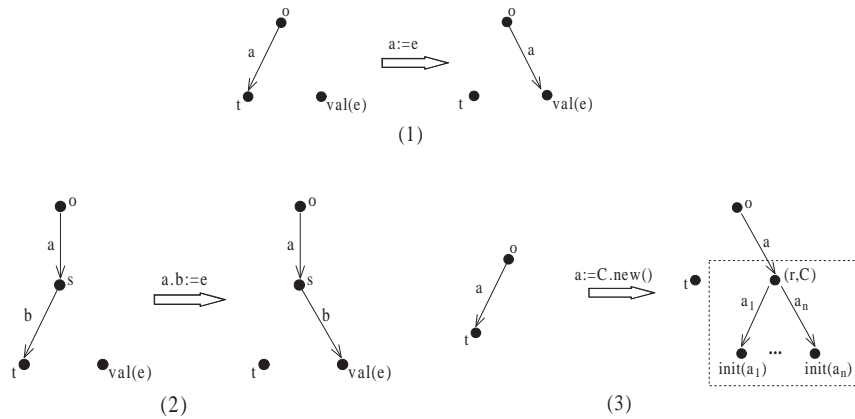


Figure 4: The change of object graphs

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  $\Sigma_0$  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(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(2). The general attribute assignment  $a.b_1 \dots b_k := e$  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  $\Gamma$  and a subclass of  $dtype(a)$ , the execution changes the edge  $(o, a, t)$  in  $\Sigma_0$  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(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  $\Sigma_0$  of  $\Gamma$  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  $\llbracket o.m()\rrbracket_{\Gamma}(\Sigma_0)$ .

We believe that it is easy to prove that the above operational semantics is consistent with the rCOS semantics defined in [5], 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 5 (Method refinement)** *Let*

1.  $\Gamma$  be a class graph,
2.  $X$  be a set of variables with types in  $\Gamma$ ,
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  $x : C \in X$ .

We say method  $m(u : T_1; v : T_2)\{c_2\}$  is a **refinement** of method  $m(u : T_1; v : T_2)\{c_1\}$ , denoted by  $m(u : T_1; v : T_2)\{c_1\} \sqsubseteq m(u : T_1; v : T_2)\{c_2\}$ , if for any object diagram  $\Sigma$  of  $\Gamma$ , any node  $u$  of  $\Sigma$  such that  $\text{type}(u) \preceq T_1$ , the set  $\llbracket x.m()\{c_2\}\rrbracket_{\Gamma}(\Sigma)$  is a subset of  $\llbracket x.m()\{c_1\}\rrbracket_{\Gamma}(\Sigma)$ , where  $x$  is the object that  $x$  points to in the object graph  $\Sigma$ .

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

**Definition 6** *Let  $\mathcal{PC}$  be a set of class names,  $\Gamma_1$  and  $\Gamma_2$  be two class graphs both containing  $\mathcal{PC}$  as some of their nodes.  $\Gamma_2$  is a **structure refinement** of  $\Gamma_1$ , denoted as  $\Gamma_1 \sqsubseteq \Gamma_2$ , if for any  $C \in \mathcal{PC}$  and a variable  $x : C$ , there exists a mapping  $\rho_o$  from  $\mathcal{M}(\Gamma_1)_{\{x:C\}}$  to  $\mathcal{M}(\Gamma_2)_{\{x:C\}}$  such that for any method  $m(u : T_1; v : T_2)\{c_1\}$  defined in class  $C$  of  $\Gamma_1$ , we can define a correspondence method  $m(u : T_1; v : T_2)\{c_2\}$  in class  $C$  of  $\Gamma_2$  and*

$$\llbracket x.m()\{c_2\}\rrbracket_{\Gamma_2}(\rho_o(\Sigma)) \subseteq \rho_o(\llbracket x.m()\{c_1\}\rrbracket_{\Gamma_1}(\Sigma))$$

## 4 Structure Transformation

### 4.1 Structure transformation

We now show that some refinement can be realized by certain class graph transformations.

**Definition 7** Let  $\Gamma_1$  and  $\Gamma_2$  be class graphs,  $f$  a subset of the class nodes of  $\Gamma_1$ . A mapping  $\rho$  from  $\Gamma_1$  to  $\Gamma_2$  is a  **$f$ -framed transformation**, denoted by  $\rho_{[f]}$ , if the following conditions hold

1. the restriction  $\rho|_{N_1}$  of  $\rho$  to the nodes  $N_1$  of  $\Gamma_1$  maps each class name in the frame  $f$  to itself, that is  $\rho(C) = C \in N_2$  for each  $C \in f$ .
2. the restriction  $\rho|_{E_1}$  to the edges  $E_1$  of  $\Gamma_1$  maps each relational or inheritance edge  $(C, a, D) \in E_1$  to a nonempty path from  $\rho|_{N_1}(C)$  to  $\rho|_{N_1}(D)$  in  $\Gamma_2$ ; and maps each data attribute  $(C, a, T)$  to a nonempty set of paths of  $\Gamma_2$  starting from  $\rho|_{N_1}(C)$  with destinations that are data types in  $T$ .

We decompose the restriction  $\rho|_{E_1}$  into two restrictions  $\rho|_r$  and  $\rho|_d$  to the relational (including inheritance) edges and data attributes, respectively.

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 1** A framed transformation  $\rho_{[f]}$  from  $\Gamma_1$  to  $\Gamma_2$  is a structure refinement if it satisfies the following properties:

1. If  $(C, \triangleright, D)$  is an inheritance edge, then  $\rho|_r(C, \triangleright, D)$  is a path containing only inheritance edges.
2. If  $(C, a, D)$  is a relational edge, then the path  $\rho|_r(C, a, D)$  contains at least one relational attribute.
3. For two different relational or inheritance edges  $(C_1, a_1, D_1)$  and  $(C_2, a_2, D_2)$ ,  $\rho|_r(C_1, a_1, D_1)$  is not a suffix of  $\rho|_r(C_2, a_2, D_2)$ .
4. For any data edges  $(C_1, a_1, T_1)$  and  $(C_2, a_2, T_2)$ , a path  $p_1$  in  $\rho|_d(C_1, a_1, T_1)$  and a path  $p_2$  in  $\rho|_d(C_2, a_2, T_2)$ ,  $p_1$  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  $\rho|_d(C, a, T) = \{C_1.\beta_1.a_1, \dots, C_1.\beta_n.a_n\}$  such that  $\text{dest}(C_1.\beta_i.a_i) = T_i$  for  $1 \leq i \leq n$ , there exists a surjective operation  $g$  of the type  $T_1 \times \dots \times T_n \rightarrow T$  such that the initial value of  $C.a$  can be calculated from those of the target attributes:  $\text{init}(C.a) = g(\text{init}(D_1.a_1), \dots, \text{init}(D_n.a_n))$ , where  $D_i = \text{dest}(C_1.\beta_i)$ .

A structure transformation from  $\Gamma_1$  to  $\Gamma_2$  in fact defines an *implementation* of the classes, their attributes and associations in  $\Gamma_1$  by those of  $\Gamma_2$ . A single relational attribute (edge) in  $\Gamma_1$  can be realized by a path, and an data attribute can be a set of paths in  $\Gamma_2$ . 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.

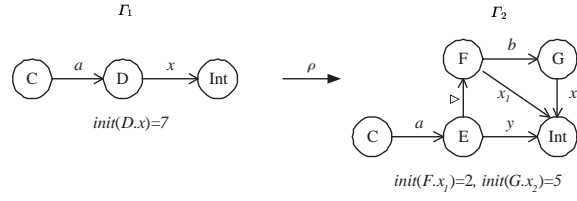


Figure 5: An example of structure transformation

- **Completeness:** prove that any structure transformation that satisfies the conditions in Proposition 1 can be obtained by sequentially applying the proposed refinement rules.

**Example 2** Figure 5 illustrates a structure transformation  $\rho|_{[f]}$ , where  $f = \{C\}$ ,  $\rho|_{N_1}(C) = C$ ,  $\rho|_{N_1}(D) = F$ ,  $\rho|_r(C, a, D) = C.a.\triangleright$ ,  $\rho|_d(D, x, Int) = \{F.x_1, F.b.x_2\}$ , and the addition operation on integers preserves the initial values of attributes:

$$init(D.x) = init(F.x_1) + init(G.x_2).$$

## 4.2 Structure relation between object graphs

Given variables  $X$ , a structure transformation  $\rho$  from class graph  $\Gamma_1$  to class graph  $\Gamma_2$  determines a derived relation  $\rho_o$  between  $\mathcal{M}_X(\Gamma_1)$  and  $\mathcal{M}_X(\Gamma_2)$ .

**Definition 8** Let  $\rho$  be a structure transformation from class graph  $\Gamma_1$  to class graph  $\Gamma_2$  with a frame  $\{C_1, \dots, C_n\}$  satisfying the conditions in **Proposition 1**, and  $X = \{x_1 : C_1, \dots, x_n : C_n\}$  be variables.  $\Sigma_1$  and  $\Sigma_2$  be object graphs with variables  $X$  for  $\Gamma_1$  and  $\Gamma_2$  respectively. The **derived structure relation** of  $\rho$ , denoted by  $\rho_o(\Sigma_1, \Sigma_2)$ , is a relation between object graphs  $\mathcal{M}(\Gamma_1)$  and  $\mathcal{M}(\Gamma_2)$  such that the following conditions hold

1. for each edge  $e = ((r_1, C), a, (r_2, D))$  in  $\Sigma_1$  such that  $a$  is a relational attribute, there is a path  $\rho_o(e) = ((r'_1, C'), \alpha, (r'_2, D'))$  in  $\Sigma_2$  such that
  - (a)  $C' = \rho|_{N_1}(C)$ ,  $D' = \rho|_{N_1}(D)$ , and
  - (b)  $(C', \alpha, D') = \rho|_r(C, a, D) \setminus \triangleright$  that is the path obtained from  $\rho|_r(C, a, D)$  after removing the inheritance edges.
2. for each data attribute edge  $e = ((r, C), a, (v, T))$  in  $\Sigma_1$ , there is a set of paths

$$\rho_o(e) = \{((r', C'), \beta_1.a_1, (v_1, T_1)), \dots, ((r', C'), \beta_n.a_n, (v_n, T_n))\}$$

such that  $v = g(v_1, \dots, v_n)$ ,  $C' = \rho|_{N_1}(C)$  and  $\rho|_d(C, a, T) = \{C'.\gamma_i.a_i\}$ , where  $\beta_i = \gamma_i \setminus \triangleright$  for  $1 \leq i \leq n$  and  $g$  is the primitive operation corresponding to  $(C, a, T)$  in  $\Gamma_1$ .

## 5 Structure Refinement Rules

We give a set of rules in Figure 6 which transform a class graph  $\Gamma_1 = \langle A_1, N_1, E_1 \rangle$  to another  $\Gamma_2 = \langle A_2, N_2, E_2 \rangle$ . Notice that each rule has a frame representing the unchanged class names before and after the transformation. The purpose of introducing precondition for each rule is to ensure that the class graph after transforming is a well-formed one. Here, we use  $\mathcal{C}$  to denote the set of class names declared in  $\Gamma_1$ .

### 5.1 Soundness

It is straightforward to prove that each rule defines a structure transformation on class graphs, which satisfies the conditions of **Proposition 1**. Thus each rule  $R$  determines a structure relation  $R_o$  between the object graphs of the corresponding class graphs.

Also, each rule  $R$  derives a transformation  $R_c$ .  $R_c$  transforms a command  $c$  that is *syntactically well-formed* under  $\Gamma_1$  to a command  $R_c(c)$  that is *syntactically well-formed* under  $\Gamma_2$ . This means that the variables and types in the command are all defined in the graph. Statements and expressions are correctly typed [5]. The command transformations are given in Figure 7, 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  $[g(C.x_1, \dots, C.x_n)/C.x]$  is defined inductively.

**Theorem 1 (Soundness of Rules)** *If rule  $R_{[f]}$  transforms  $\Gamma_1$  to  $\Gamma_2$ , then  $\Gamma_1 \sqsubseteq \Gamma_2$ .*

**Proof (Outline)** If a method  $m(u : T_1; v : T_2)\{c\}$  is defined in class  $C$  ( $C \in f$ ) of  $\Gamma_1$ , then we can also define a corresponding method  $m(u : T_1; v : T_2)\{R_c(c)\}$  in class  $C$  of  $\Gamma_2$  such that the condition given in Definition 6 holds.

This theorem implies that all rules given in Figure 6 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 6 is transitive.

**Corollary 1** *If  $\Gamma_1$  is transformed to  $\Gamma_2$  by a sequential applications of rules  $R_{[f_1]}, \dots, R_{[f_k]}$ , then  $\Gamma_1 \sqsubseteq \Gamma_2$ , provided  $f_1 \cap \dots \cap f_k \neq \emptyset$ .*

### 5.2 Validity of Proposition 1

We now establish the validity of **Proposition 1** as the following completeness theorem.

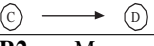
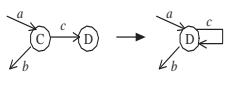
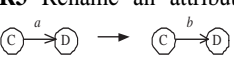
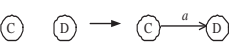
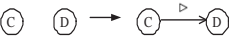
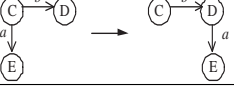
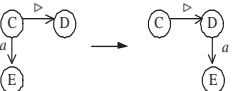
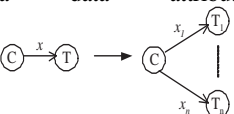
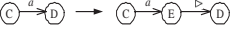
Rules	Description	Precondition	Frame
<b>R1</b> Rename a class 	class name $C$ is changed to a different name $D$	$C \in N_1, D \notin N_1$	$\mathcal{C} \setminus \{C\}$
<b>R2</b> Merge classes 	merge class $C$ to another class $D$ such that all the outgoing (incoming) edges from (to) $C$ become those from (to) $D$ , and the edges between $C$ and $D$ become self loop edges	$C, D \in N_1,$ $(C, a, E), (D, a, F) \in E_1$ implies $E = F$ and $init(C.a) = init(D.a)$	$\mathcal{C} \setminus \{C\}$
<b>R3</b> Rename an attribute 	the name of an attribute $(C, a, D)$ is changed to $(C, b, D)$	$(C, a, D) \in E_1,$ $a \neq b$ and $(E, b, F) \notin E_1$ for any $F \in N_1$ if $E$ is a superclass or subclass of $C$	$\mathcal{C}$
<b>R4</b> Add a new class $r_1 \rightarrow r_1 + \textcircled{C}$	add a class $C$	$C \notin N_1$	$\mathcal{C}$
<b>R5.1</b> Add an attribute 	add an attribute $(C, a, D)$	$C, D \in N_1, (E, a, F) \notin E_1$ for any $F \in N_1$ if $E$ is a superclass or subclass of $C$	$\mathcal{C}$
<b>R5.2</b> Add an inheritance 	add an inheritance edge $(C, \triangleright, D)$	$C, D \in N_1,$ for each $C' \in N_1$ $(C, \triangleright, C') \notin E_1,$ $D \not\leq C,$ and for each pair of edges $(E, a, E'), (F, b, F') \in E_1$ where $E \preceq C$ and $F \succeq D,$ we have $a \neq b$	$\mathcal{C}$
<b>R6.1</b> Forward an attribute through a relational attribute 	an attribute $(C, a, E)$ is forwarded through a relational attribute $(C, b, D)$ to form another attribute $(D, a, E)$	$(C, a, E), (C, b, D) \in E_1,$ $(F, a, F') \notin E_1$ for any $F' \in N_1$ if $F$ is a superclass or subclass of $D$	$\mathcal{C}$
<b>R6.2</b> Forward an attribute through an inheritance 	an attribute $(C, a, E)$ is forwarded through an inheritance edge $(C, \triangleright, D)$ to form another attribute $(D, a, E)$	$(C, a, E), (C, \triangleright, D) \in E_1,$ $(F, a, F') \notin E_1$ for any $F' \in N_1$ if $F$ is a superclass or subclass of $D$	$\mathcal{C}$
<b>R7</b> Decompose a data attribute 	a data attribute $(C, x, T)$ is decomposed to a set of data attributes $(C, x_1, T_1), \dots, (C, x_n, T_n)$	$(C, x, T) \in E_1,$ there exists a surjective primitive operation $g : T_1 \times \dots \times T_n \rightarrow T$ preserving the initial values of attributes, and for each $1 \leq i \leq n,$ $(D, x_i, D') \notin E_1$ for any $D' \in N_1$ if $D$ is a superclass or subclass of $C$	$\mathcal{C}$
<b>R8</b> Decompose an edge 	a relational attribute or inheritance edge $(C, a, D)$ is decomposed to two edges $(C, a, E)$ and $(E, \triangleright, D)$	$(C, a, D) \in E_1, E \in N_1$	$\mathcal{C}$

Figure 6: Basic Rules

Rule $R$	Command Transformation $R_c$
<b>R1</b> Rename a class	$R_c(c) = c[D/C]$
<b>R2</b> Merge classes	$R_c(c) = c[D/C]$
<b>R3</b> Rename an attribute	$R_c(c) = c[C.b/C.a]$
<b>R4</b> Add a new class	$R_c(c) = c$
<b>R5.1</b> Add an attribute	$R_c(c) = c$
<b>R5.2</b> Add an inheritance	$R_c(c) = c$
<b>R6.1</b> Forward an attribute through a relational attribute	$R_c(c) = c[C.b.a/C.a]$
<b>R6.2</b> Forward an attribute through an inheritance	$R_c(c) = c$
<b>R7</b> Decompose a data attribute	$R_c(c) = c[g(C.x_1, \dots, C.x_n)/C.x]$
<b>R8</b> Decompose an edge	$R_c(c) = c$

Figure 7: Command transformation

**Theorem 2 (Completeness of Rules)** *If  $\rho_{[f]}$  is a structure transformation from  $\Gamma_1$  to  $\Gamma_2$  that satisfies the conditions of **Proposition 1**, then there exist a finite number of sequential applications of rules  $R_{1[f_1]}, \dots, R_{k[f_k]}$  that transforms  $\Gamma_1$  to  $\Gamma_2$  and  $f \subseteq f_i$  for  $1 \leq i \leq k$ .*

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

1. change each class  $C$  to  $\rho|_{N_1}(C)$ , by applications of R1 and R2. When mapping  $\rho|_{N_1}$  is injective, R2 is not needed.
2. decompose each data attribute  $(C, x, T)$  to a set of data attributes:

$$\{(C, x_1, T_1), \dots, (C, x_n, T_n)\},$$

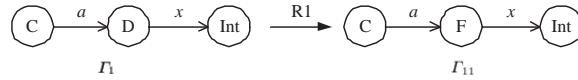
provided  $\rho|_d(C, x, T) = \{(C.\beta_1.x_1), \dots, (C.\beta_n.x_n)\}$  and  $\text{dest}(C.\beta_i.x_i) = T_i$ , using rule R7.

3. for edges there are two cases
  - (a) change each data edge  $(C, x, T)$  to a path  $C.\beta.x$  in  $\rho|_d(C, x, T)$ , using R4, R5 and R6.
  - (b) change each relational or inheritance edge  $(C, a, D)$  to a path  $\rho|_r(C, a, D)$ , by applications of R3, R4, R5, R6 and R8.
4. add additional nodes and edges by using rules R4 and R5 if necessary.

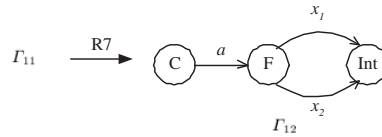
**Example 3** *For the structure transformation illustrated in Example 2, Figure 8 shows the applications of the rules that transform  $\Gamma_1$  to  $\Gamma_2$ .*

Now we have established the validity of **Proposition 1**.

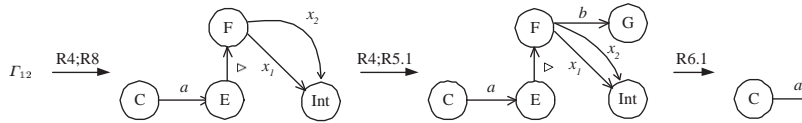
step 1: rename classes



step 2: decompose data attributes



step 3: transform edges



step 4: add extra nodes and edges

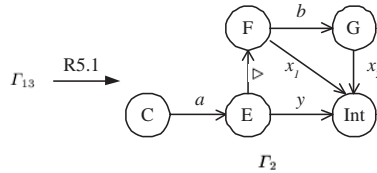


Figure 8: An example

**Corollary 2** Proposition 1 holds.

## 6 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 understand and more operational. We believe this is important for development of tool support to object system development by transformations [11].

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 understand 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 attributes or compressing long paths to shorter paths. Therefore, some refinement laws proved in rCOS [5] 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 consider 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. Cardeli. *A Theory of Objects*. Springer-Verlag, 1996.
- [4] P. America and F.de Boer. Reasoning about dynamically evolving 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>).
- [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.
- [8] R. Gheyi, T. Massoni and P. Borba. An abstract equivalence notation for object models. *Proc. SBMF2004*, pages 1-15. 2004.
- [9] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proc of FME'97, LNCS 1313*. Springer, 1997.
- [10] EB Johnsen and O. Owe. Object-Oriented Specification and Open Distributed Systems. In *LNCS 2635, pages 137-164*. Springer, 2004.
- [11] Tata Consultancy Services. MasterCraft.(<http://www.tata-mastercraft.com/>)
- [12] R. Back, A.Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 2:18-40, 2000.