DOI 10.1007/s00165-007-0067-y
BCS © 2009
Formal Aspects of Computing (2009) 21: 103–131

Formal Aspects of Computing

Graph transformations for object-oriented refinement[1]

Liang Zhao[1], Xiaojian Liu[1], Zhiming Liu[1] and Zongyan Qiu[2]

[1] UNU-IIST, P. O. Box 3058, Macao SAR, China. E-mail: z.liu@iist.unu.edu
[2] LMAM and Department of Informatics, School of Mathematics, Peking University, Beijing, China.

Abstract. An object-oriented program consists of a section of *class declarations* and a *main method*. The class declaration section represents the structure of an object-oriented program, that is the data, the classes and relations among them. The execution of the main method realizes the application by invoking methods of objects of the classes defined in the class declarations. Class declarations define the general properties of objects and how they collaborate with each other in realizing the application task programmed as the main method. Note that for one class declaration section, different main methods can be programmed for different applications, and this is an important feature of reuse in object-oriented programming. On the other hand, different class declaration sections may support the same applications, but these different class declaration sections can make significant difference with regards to understanding, reuse and maintainability of the applications. With a UML-like modeling language, the class declaration section of a program is represented as a *class diagram*, and the instances of the class diagram are represented by *object diagrams*, that form the *state space* of the program. In this paper, we define a class diagram and its object diagrams as *directed labeled graphs*, and investigate what changes in the class structure maintain the capability of providing *functionalities* (or *services*). We formalize such a structure change by the notion of *structure refinement*. 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 many, and as good, services (in terms of functional refinement) as the original graph. We then develop a *calculus of object-oriented refinement,* as an extension to the classical *theory of data refinement*, in which the refinement rules are classified into four categories according to their natures and uses in object-oriented software design. The soundness of the calculus is proved and the completeness of the refinement rules of each category is established with regard to normal forms defined for object-oriented programs. These completeness results show the power of the simple refinement rules. The normal forms and the completeness results together capture the essence of polymorphism, dynamic method binding and object sharing by references in object-oriented computation.

Keywords: Class graph; Object graph; Graph transformation; Normal form; Object-orientation; Structure refinement

Correspondence and offprint requests to: Z. Liu, E-mail: z.liu@iist.unu.edu
[1] This work is supported by the projects HighQSoftD and HTTS funded by Macao Science and Technology Fund, Chinese NSF project 60673114, Chinese NSF project 60573081 and 863 of China 2006AA01Z165.

1. Introduction

The research in the past half a century in the area of formal methods has been focused on procedural programs and languages. For this kind of programs, formalisms have been developed that define execution models of programs and provide useful techniques and tools for specifying what programs do, reasoning about and analyzing the correctness criteria of programs and verifying why a program does what it is required to do. There are also theories and calculi of refinement [BvW98, HHS86, Mor94] to support techniques of *correctness by construction* or *correctness preserving transformations*. We can confidently claim that for procedural programs, we have developed a good understanding of *what programs do*, *how they do it*, and *why they work*.

An analogous understanding for object-oriented programs and object-oriented languages has not yet been established. There exists a big body of research on formal semantics of object-oriented programs, e.g. [AC96, BSC03, JO04, AdB94, CN99, Nau94], but though rarely stated in the literature, the common feeling is that *it is difficult to understand and use the semantics of object-oriented programs*. One of the reasons is that it is hard to relate a denotational (or an axiomatic) semantics to an operational semantics. It is even harder to verify the refinement of object-oriented specifications and designs, and algebraic properties of object-oriented programs. This leads to the confusion that object-oriented programming can only be carried out in bottom-up and/or in an extreme programming process. However, this cannot be right as otherwise it would be contradictory to the fact that object-oriented development is effectively applied in large software projects.

Variants of λ-calculus, that is the fundamental underlying theory of functional and procedural languages, have been tested on object-oriented programs. However, there are always features of object-orientation that can be hardly treated in those calculi. For detailed discussion on this issue, we refer the reader to the book of Abadi and Cardelli [AC96]. The calculus developed in that book itself focuses on the characterization of the behavior of objects. This characterization helps the understanding and treatment of the functional behavior of object programs, but it largely ignores the design issues of structure of the data and classes of such a program. Hoare logic and the calculus of predicate transformers are also used in defining semantics of object-oriented languages, e.g. [AdB94, CN99, Nau94]. Again all these focus on functionalities of objects or methods of classes, without providing much help to the design of the class structure. However, experienced researchers and teachers in the area of object-oriented programming come to realize that the design of the class structure and the functionality are equally important and closely related.

In rCOS, a refinement calculus has been recently developed by He, Li and Liu [HLL06] based on a relational semantics given to object-oriented programs. There, an object-oriented program can be represented in the form of $Cdecls \bullet Main$, where the *class declaration section* $Cdecls$ is a sequence of class declarations with their attributes, methods, and inheritance relations; and $Main$ declares a *main class* with a *main method* [HLL06]. The main class declares, as its attributes, a set of main variables whose types are either primitive built-in data types or class types declared in $Cdecls$. The main method implements an application by calling some *public methods* of *public classes* in the declaration section. Given one class declaration section, different main methods can be programmed for different applications, and this is an important feature of reuse in object-oriented programming. On the other hand, different class declaration sections may support the same applications, but these different class declaration sections can make significant difference with regards to understanding, reuse and maintainability. 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 $Cdecls_1$ [HLL06]. Here, "behavior" is only about *functionality*, and the refined class structure $Cdecls_2$ supports all functionalities that the class structure $Cdecls_1$ supports. This notion of refinement combines the design of class structure and functionality of the classes. It is easy to see that a class declaration section can be represented as an UML *class diagram*, and the instances of the class diagram are represented by UML *object diagrams*. The class diagram determines the general properties of objects and how objects collaborate in realising the application task programmed as the main method.

1.1. Contribution

For an object-oriented program, we define its class declaration section by a *labeled direct graph* called a *class graph*, and a *state* of the program by a *rooted labeled directed graph* called an *object graph*. In a class graph, a node is either a *class node* that is labeled by a class name or a *leaf node* that is labeled by the name of a primitive data type. An outgoing edge of a class node is either labeled by an attribute of the class or the symbol $\triangleright$ representing the *inheritance* relation, and the target node of the edge is the name of the type (or class) of the attribute or the direct superclass of the source node, respectively. In an object graph, the root represents the unique instance of

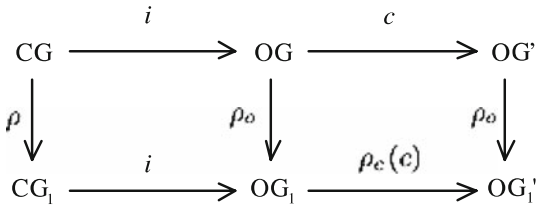


Fig. 1. Structure refinement

the main class. Any node different from the root is labeled by a *typed value* that is a pair (v, T), where v is either an *object reference* if T is a class, or an element of T if T is a primitive data type. For each attribute $x : T$ of the main class, there is an edge labeled by x from the root to a node whose type is T. An outgoing edge from (v_1, T_1) to (v_2, T_2) is labeled by an attribute a of T_1, representing that the value of attribute a of object v_1 is v_2. Therefore, there is an edge from (v_1, T_1) to (v_2, T_2) labeled with a if and only if one of the following conditions holds for the class graph

1. a is an edge from T_1 to T_2,
2. a is an edge from a superclass of T_1 to T_2,
3. a is an edge from T_1 to a superclass of T_2,
4. a is an edge from a superclass of T_1 to a superclass of T_2.

A class graph defines a set of object graphs for each given main class, and these object graphs form the potentially infinite *state space* of the program.

The execution semantics of a command in the program is then a state transition from one object graph OG to another object graph OG'. This allows us to relate the relational semantics of rCOS defined in [HLL06] to an operational semantics that can be easily defined in terms of graph transition systems [GRPPS98]. Based on this graph notation and the semantics, we study the theory of class graph transformations to understand the design of the class structure and its relation to the design of the functionality of the classes.

We then define the notion of structure refinements in terms of graph transformations so that given

1. an object-oriented program, let CG be its class graph and OG its set of object graphs,
2. a structure refinement ρ from CG to another class graph CG_1,

we are able to derive

1. a transformation ρ_o from an object graph OG of CG to an object graph OG_1 of CG_1, and
2. a transformation ρ_c from commands defined under CG to commands defined under CG_1,

such that the diagram in Fig. 1 commutes.

After formally defining the notion of structure refinement, we give four sets of structure refinement rules.

1. The first set of rules allow us to expand class graphs and they are used for object-oriented decomposition and incremental programming by adding classes. These rules do not depend on methods of classes.
2. The second group of rules are graph compression rules for combining classes, removing redundant classes, attributes, and collapsing an inheritance relation that does not involve method overriding.
3. The third group of rules are concerned with transforming definitions of methods of classes without changing the structure of the classes.
4. Two more rules for removing polymorphism by collapsing inheritance relations with method overriding.

The rules in the first group support object-oriented top down and incremental design, and the other rules are useful for program *refactoring*, *abstraction*, *analysis* and *reverse engineering*.

We will see that it is easy to establish the soundness of these rules with respect to the definition of the structure refinement. For their completeness we will prove that the first set of rules are complete for a restricted set of class graph transformations (called well-typed structure transformations). Furthermore, we will define two normal forms of object-oriented class graphs:

Normal Form I: Such a normal graph forms an inheritance tree that contains all classes and there is at least one private class. We will show that the second and third set of refinement rules, together with the attribute

renaming rule, are complete in that any class graph can be transformed by them to a graph of this normal form.

Normal Form II: Such a normal graph only has a set of public classes. These public classes may or may not be linked by associations and inheritance relations. We will prove that a graph of this form can be obtained from any class graph of the first normal form by using two rules for removing polymorphism.

These results of completeness capture the essence of polymorphism, dynamic method binding and data sharing via references in object-oriented computation. They show the difference and relation between object-oriented programming and procedure (and modular) programming.

The semantics of structure refinement is formally based on the denotational semantics in [HLL06]. However, for the understanding, an operational semantics based on the definition of object graphs as the program state space is informally proposed. It is straightforward to formalize this operational semantics and show its consistency with the denotational semantics. However, we will leave it out of this paper.

1.2. Related work

There exists a big body of research on formal semantics of object-oriented programs. Model-based formalisms have been used extensively in conjunction with object-oriented techniques, via languages such as Object-Z [Smi00], VDM++ [DD93], and methods such as Syntropy [CD94] which uses the Z notation and Fusion [Col94] that is based on VDM. Whilst these formalisms are effective at modeling data structures as sets and relations between sets, they are not designed for defining semantics of object-programs and thus do not deal with more sophisticated object-oriented mechanisms of object-oriented programming languages, such as dynamic binding and polymorphism. Moreover, we believe that a modeling notation using the format of classes with inheritance and abstract specifications of class methods are more directly related to object-oriented programming languages and appealing to practical engineers and programmers than the classical formal notations such as the Z-schemas and their operators.

Cavalcanti and Naumann defined an object-oriented programming language, called *ROOL*, with subtypes and polymorphism [CN99, Nau94] using predicate transformers. Sekerinski [Sek96, MS97] defined a rich object-oriented language by using a type system with subtyping and predicate transformers. However, neither reference types nor mutual dependency between classes are within the scope of these approaches. Because of complex flow of control, it is infeasible to calculate the weakest precondition of an object-oriented program for a given post condition. Thus semantic proofs of refinement rules in ROOL are quite hard and complex even without references. Without the inclusion of reference types, some interesting refinement rules can not be proved [BSC03]. America and de Boer have given a logic for the parallel language POOL [AdB94]. It applies to imperative programs with object sharing, but without subtyping and method overriding. Abadi and Leino have defined an axiomatic semantics for an imperative, object-oriented language with object sharing [AL97], but it does not permit recursive object types. Poetzsch–Heffter and Müller have defined a Hoare-style logic for object-oriented programs that relaxes many of the previous restrictions [PHM99]. As [Lei98] points out, the specification of a method in the Poetzsch–Heffter and Müller logic is derived from the method's known implementation, and therefore it does not support the notion of refinement. Leino has presented a logic in [Lei98] with imperative features, subtyping, and recursive types. It allows the specification of methods, but inheritance is restricted and visibility is not considered.

In addition to the limitations discussed above, there is a common feeling that these semantic definitions are difficult to understand. 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 is even harder. We attempt to improve the understanding and use of the formal calculus rCOS [HLL06] by using a graph theoretical approach. In rCOS, we provide a semantic model that gives a mathematical characterization of object-oriented concepts to ensure the correctness of programs. With rCOS, we define an object-oriented language with subtypes, visibility, reference types, inheritance, type casting, dynamic binding and polymorphism. The language is similar to Java and C++. It has been used to develop meaningful case studies and to capture some of the central difficulties in modeling object-oriented designs and programs [CHH+07]. The graph theoretical notation can be used as the basis for an operational semantics of the langauge that can improve the understandability of rCOS and the algebraic properties of object-oriented programs.

An algebraic calculus is established for rCOS [HLL06] based on its relational semantics defined in UTP [HH98]. Compared to the algebraic calculus of class structures given in [BSC03], the semantics of rCOS is easier

to understand and use for verification. The refinement rules in this paper agree with all the rules in [HLL06]. However, the graph calculus in this paper significantly improves the understanding of structure refinement and for future development of tool support. This paper also extends [HLL06] with the two normal forms of object-oriented programs and completeness of the rules.

The work in [MS97, BMvW00] handles class and interface refinements. However, there the focus is substitutability of individual classes in a class structure. Our work investigates the refinement of a class model as a whole and supports structure design at different stages of the system development. In [GMB04], a notion of equivalence between class diagrams 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.

There is a quite visible community working on graph transformations or transition systems [Ed97]. General studies of theories of graph transformations are carried in the model of *graph transition systems* or *graph processes* [CMR96, GRPPS98, EEPT06]. Our work in this paper focuses on particular graphs and transformations, that is graph transformations related to object-oriented program design. The results of those general studies provide the background for defining the operational semantics of object-oriented program in rCOS, where the object graphs are the graphs of transition systems and the executions of the methods are the corresponding graph transitions. The main focus of our work is the transformations of class graphs and their relation to the change of the methods.

Graph transformations have also been applied to software development and maintenance. General software architectures (or architecture styles) and their refinements are defined in [BHTV04] by graph transformations. However, only expanding transformations are allowed in a refinement. Furthermore, that work is based on a rather general model of graph transition systems and does not provide particular treatment of object-oriented programs. In particular, there are no normal forms and completeness results related to transformations of object-oriented programs. This is also true in paper [BHTV03]. Graph transformations are also used in [WF02]. There an algebraic framework is presented based on category theory where architectures are represented as graphs of CommUnity programs and superpositions. That allows ways to apply connectors to components restricted by an architectural style, given as a type graph. Dynamic reconfiguration is specified by graph transformation rules over architecture instances. Both styles and rules are used for modeling domain-specific restrictions rather than the underlying platform. Consequently, they do not deal with refinement relationships between different levels of abstraction.

Graph transformations are used in [EHHS00] to define the semantics of UML collaboration diagrams. A collaboration diagram is defined to be a transformation on the object graphs of a class graph. So transformations there correspond to the semantics of the commands in rCOS. The focus of our paper here, however, is transformations of class graphs and how they determine the transformations on commands to preserve functionality. In other words, we are treating and relating graph transformations at two levels of abstractions, the structural level and the program execution level.

In [KKR06], a theory of graph transformation is applied to a definition of an object-oriented execution semantics of a mini language. The simulation relation is then studied for programs in that language. The work in [TR05] formulates structural properties using graph constraints in type graphs with inheritance, and shows how to translate constrained type graphs with inheritance to equivalent constrained simple type graphs. It then follows that graph constraints can be translated into preconditions for productions of a typed graph transformation system which ensures those graph constraints. Our work in this paper goes beyond the concerns of both papers by looking at how transformations of the class graphs determine the transformations on program commands so that the functional behavior is preserved. This is an essential problem for program development and maintenance.

The use of object graphs is influenced by notation of graphs for pointer structures in [CJ06], and the idea of using paths of a graph comes from the trace model of pointers and objects with pointers [HH99]. The notion of object graphs can be seen as an extension to the notion of execution states in classical imperative procedural programs. Based on this understanding, we can claim that the theory of a *structure refinement* between object-oriented programs is a non-trivial extension to the theory of *data refinement* [HHS86] for the support of object-oriented software design. The calculus of object-oriented refinement is even more *workable* in the sense that, unlike in classical data refinement where a data mapping must be found for a refinement from one program to another, the refinement rules also give the data mappings. We believe that this is essential for model driven development of object-oriented software and for the development of tool support for correctness preserving model transformations. This extension advances the the classical refinement calculi to a *design method*[2] applicable in large scale system development with effective tool support for model transformations [CHH+07].

[2] The classical refinement calculi could hardly called a design method without effective tool support.

1.3. Overview

Section 2 shows how a class declaration section can be defined as a directed labeled graph. In Sect. 3, we define object graphs for a class graph to represent system states. We also propose an informal, yet precise and obviously formalizable, operational semantics of object-oriented programming commands based on class graphs and object graphs. Section 4 defines structure refinements between class graphs and their derived relations between object graphs. We also establish a set of basic graph transformation rules and prove that they are sound refinements and complete with respect to a set of conditions. These transformations can only expand class graphs. In Sect. 5, we introduce the concept of interface and extend the notion of structure refinements with respect to interfaces. We also study refinements for compressing a graph by removing redundant structure elements and combining classes. In Sect. 6, we provide the refinement rules for modifying methods in a class graph without changing the whole class structure. Section 7 defines the normal forms and proves completeness results. We will draw the conclusions in Sect. 8 with a discussion about future work.

2. Class graphs

We define a class declaration section as a directed and labeled 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, a set $\mathcal{T}$ of names of primitive data types, an infinite set $\mathcal{A}$ of attribute names, and a symbol $\triangleright$ to annotate the inheritance relation. Let $\mathcal{N}$ be the set of names of *types*, which is the union of $\mathcal{CN}$ and $\mathcal{T}$.

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

- $N \subseteq \mathcal{N}$: is a set of *nodes* representing *types*, including both classes and primitive data types
- $A \subseteq \mathcal{A}$: a set of *attribute* names
- $E \subseteq N \times (A \cup \{\triangleright\}) \times N$: a set of triples, called 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.

Please note that each node in the set N represents either a class or a type of data values. Similar to the definition of typed attribute graphs in [EEPT06], we can partition N into $N_1 \cup N_2$, where N_1 is a set of class names and N_2 is a set of names of data types.

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 class graphs as defined above correspond to *well-formed* class declarations, and we need to give the well-formedness conditions.

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

1. A node is a leaf of the graph if it is labeled by the name of a data type: $(C, a, D) \in E$, implies $C \in \mathcal{CN}$.
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. This implies that we assume there is no multiple inheritance.
   (b) there is no cycle formed by $\triangleright$ edges.
   (c) no attribute 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, D and C_2. That is, we assume no attribute overriding.
4. The names of the attributes of a class are all distinct, that is the labels of all outgoing edges from a class are different.

When there is no confusion, we simply use the term class graph for a well-formed class graph. Notice that a class graph has three disjoint sets of edges:

- *Data attributes*: *also called data edges*, are those edges (C, x, T) such that T is a primitive type.
- *Association attributes*: also called *association edges*, are those edges (C, a, D) such that D is a class. We also simply call an association attribute an association, and use the term attribute for either a data attribute or an association.

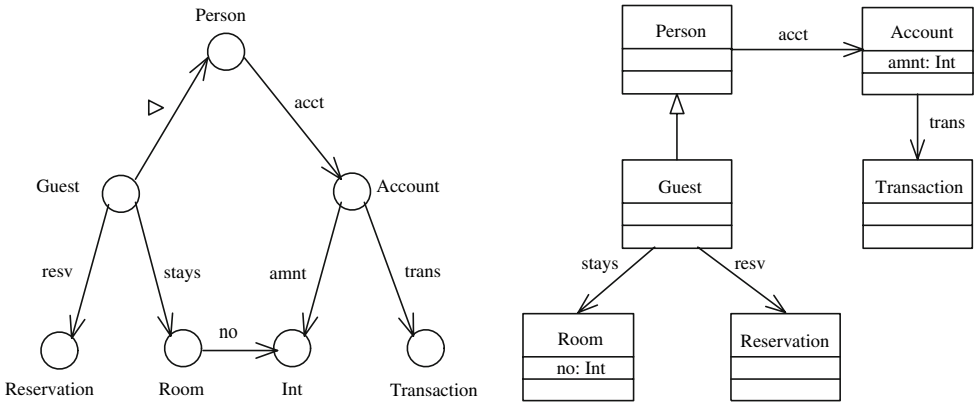


Fig. 2. An example

- *Inheritance relations*: also called *inheritance edges*, are the edges $(C, \triangleright, D)$ for some C and D in the graph.

An edge that is not a data edge is also called a *relational edge*.

We do not consider the notations of *multiplicity* and *aggregation* [BRJ99]. Multiplicities can be dealt with by introducing *container classes* whose instances are *multi-objects* and logic constraints on the numbers of objects in the multi-objects. Neither do we distinguish aggregations from general associations. As in our observation in [LHLC03], an aggregation is a special association and an association name is needed anyway when we talk about an invocation to a method of a "component objects" by the "aggregated" (or "whole") object. In other words, whether an association is an aggregation is a semantic property of the association.

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

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

For a class graph Γ, we abuse the OO notation $C_0.a_0.\ldots.a_{k-1}$ to denote a *path* $[(C_0, a_0, C_1), (C_1, a_1, C_2), \cdots, (C_{k-1}, a_{k-1}, C_k)]$; and use $dest(C_0.a_0.\ldots.a_{k-1})$ to denote the *destination* C_k of the path. For two paths $p_1 = C.\alpha$ and $p_2 = D.\beta$ such that $D = dest(C.\alpha)$, the *concatenation* $p_1.p_2$ of p_1 and p_2 is $C.\alpha.\beta$.

A sequence $\alpha = \{(C_i, a_i, C_{i+1}) \mid C_i, C_{i+1} \in N, a_i \in A, i = 0, \ldots, k\}$ of "edges" is called a *navigation path* of Γ if for all $i = 0, \ldots, k$, $\exists D_i, D_{i+1} \bullet (C_i \preceq D_i \wedge C_{i+1} \preceq D_{i+1} \wedge (D_i, a_i, D_{i+1}) \in E)$. In other words, for all $i = 0, \ldots, k$, $a_i \in Attr(C_i)$, and C_{i+1} is a subtype of the type declared for a_i in Γ. Only associations determine the navigation paths in a class graph. Notice that each path in the object graph, which will be introduced later, corresponds to a navigation path in the class graph.

Example 1 The left part of Fig. 2 is a class graph, representing the UML class diagram on the right of the figure.

3. Object graphs and execution of commands

A class graph declares a family of types, thus 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. We assume an infinite set REF of references including a special symbol $null$. We defined a *typed value* as a pair (r, T), where r is an element of T if T is a primitive type and a *reference* otherwise. For a class graph Γ, we use V_Γ to denote the set of all values of types declared in Γ.

3.1. Object graphs as program states

For a class graph Γ and a given finite set X of *main variables* $\{x_1 : T_1, \ldots, x_n : T_n\}$ such that each type T_i is a primitive type in $\mathcal{T}$ or a class type declared in Γ, we define the *state space* over X by the *object graphs* of Γ over X.

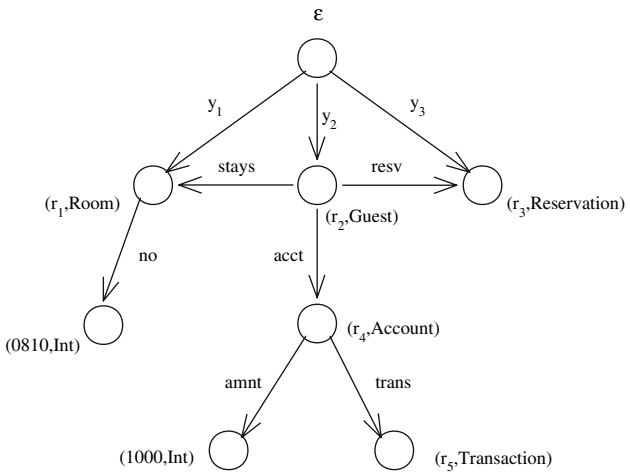


Fig. 3. An object graph

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

- N is the set of the nodes and each of them is either the **root node** ε or a typed value in V_Γ, that represents a data value or a reference with its type.
- $L = X \cup A_\Gamma$ is the set of the names used to label the edges, notice that A_Γ is the set of attribute names that label the edges of Γ.
- $E \subseteq N \times L \times N$ are the edges of Σ.
- The root node ε has no incoming edges.
- All nodes are reachable from the root, that is for each node $v \in N$, there is at least one path p from the root with $dest(p) = v$. This implies that all the nodes other than the root must have at least one incoming edge

The path in an object graph is defined similarly as in the class graph. An object graph Σ of Γ is *complete and correctly typed* if every attribute of a non-null object in Σ 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 $\Sigma = \langle N, L, E, \varepsilon \rangle$ 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 $(r, C) \in N$, then C must be a node in Γ.
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 Γ, that is there exists a node D_1 of Γ, $D \preceq D_1$ and (C, a, D_1) is an edge in Γ.
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 $v = (r, C)$, then there exists an edge $((r, C), a, (r_1, D))$ for some node (r_1, D) in Σ if and only if C is a class name in Γ, $r \neq null$ and $a \in Attr(C)$.

We use $\mathcal{M}_X(\Gamma)$ to denote the *state space* of X on Γ that is the set of the CCT object graphs of Γ with variable set X. we will simply call a CCT object graph an object graph and omit the subscript X when there is no confusion. Fig. 3 is an example of an object graph of the class graph in Fig. 2, with $X = \{y_1 : Room,\ y_2 : Guest,\ y_3 : Reservation\}$ as its main variables.

For an edge (C, a, D) of class graph Γ, $dtype(C.a)$ (or simply $dtype(a)$ when there is no confusion) denotes the type D, called the *declared type* of attribute a of class C in Γ. For an edge $((r_1, C), a, (r_2, D_1))$ in an object graph Σ, $type(r_1.a)$

denotes the type D_1, called the *current type* of attribute a of object r_1 in state Σ. Also, $type(r, C)$ denotes the *current type* C of the node (r, C) in the object graph (or state). 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. For a main variable $x : T$, T is called the declared type of x, denoted as $dtype(x)$. And for an edge $(\varepsilon, x, (r, T'))$ in the object graph Σ, $type(x)$ denotes the type T', called the current type of x in state Σ. Definition 4 also ensures that the current type of each main variable is a subtype of its declared type.

The root object, representing the instance of the main class, can access an object or a property of an object via different paths. We can thus use the set of all paths to a node to represent the object that the node intends to model. In Fig. 3, for example, the $(r_1, Room)$ instance can be represented by a set of two paths $\{\epsilon.y_1, \epsilon.y_2.stays\}$, and the data value $(1000, Int)$ by a single path $\{\epsilon.y_2.acct.amnt\}$.

3.2. Operational semantics

Let P be an object program specified by a class graph Γ and a main class *Main* with X as its main variables. In an operational semantic view, the execution of P from a state Σ_0, which is an object graph in $\mathcal{M}_X(\Gamma)$, is the execution of the command of the main method that changes the initial state Σ_0 to a final state, which is another object graph in $\mathcal{M}_X(\Gamma)$, if it terminates. Each step of the execution of the main method may call for the execution of a method of an object in the form $o.m(x; y)\{c\}$ from a state $\Sigma \in \mathcal{M}_X(\Gamma)$ and the execution step changes this state to another one (see Fig. 4 for examples), where x is the input parameter, y the output parameter, object o and x are nodes of Σ, though x can be a data value.

The syntax of command c is defined by the following syntactic rules:

$$\begin{aligned} c ::= \; & skip \mid chaos \mid le := e \mid C.new(le) \mid le_0.m(e; le) \mid m(e; le) \mid c_1; c_2 \mid \\ & c_1 \lhd b \rhd c_2 \mid c_1 \sqcap c_2 \mid b * c \mid var\ T\ x; c; end\ x \end{aligned}$$

Here $le := e$ assigns e to le; $C.new(le)$ creates an object of class C, and makes le refer to it afterwards; $le_0.m(e; le)$ or $m(e; le)$ denotes a method call, where e and le are input and output parameters respectively; $c_1; c_2$, $c_1 \lhd b \rhd c_2$, $c_1 \sqcap c_2$ and $b * c$ stands for sequential composition, conditional choice, non-deterministic choice and loop respectively; *var* and *end* together form a local scope.

The syntax of expression e and left-value expression le is:

$$\begin{aligned} e &::= x \mid l \mid null \mid a \mid e.a \mid (C)e \mid f(e_1, \cdots, e_n) \\ le &::= x \mid a \mid le.a \end{aligned}$$

where x represents a variable; l represents a literal of a primitive type; a is an attribute; $e.a$ denotes the attribute a of the object referred to by e; $(C)e$ casts the type of e to C; and f is a built-in operation for a built-in primitive types.

A denotational semantics of this OO language is defined in rCOS [HLL06]. With the class graphs and object graphs, that semantics can be further interpreted by replacing the heap representation of the state by the object-graph representation: the semantics of a command c under class graph Γ, denoted as $[\![c]\!]_\Gamma$, is a relation between object graphs of Γ. Furthermore, an operational semantics can be easily defined following the theory of graph transition (or transformation) systems [GRPPS98, KKR06]. In this paper, we outline informally the description of this operational semantics of the rCOS programs below, that is shown in Fig. 4:

1. If c is a simple assignment $a := e$, where a is an attribute of the current object node o, then the execution changes the edge (o, a, t) in Σ_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 Fig. 4(1).
2. If c is an assignment of the form $a.b := e$, where a is an attribute of the current object o 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 Fig. 4(2). The general attribute assignment $a.b_1 \ldots .b_k := e$ is defined by induction.
3. If c is an object creation $C.new(a)$, 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 Σ_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 Fig. 4(3).
4. The meaning of compositions of commands can be defined inductively.

We would like to note that the execution of a command may cause an object (i.e. a node) in the object graph to be unreachable from the root. In this case, the object and its outgoing links will be deleted from the object graph.

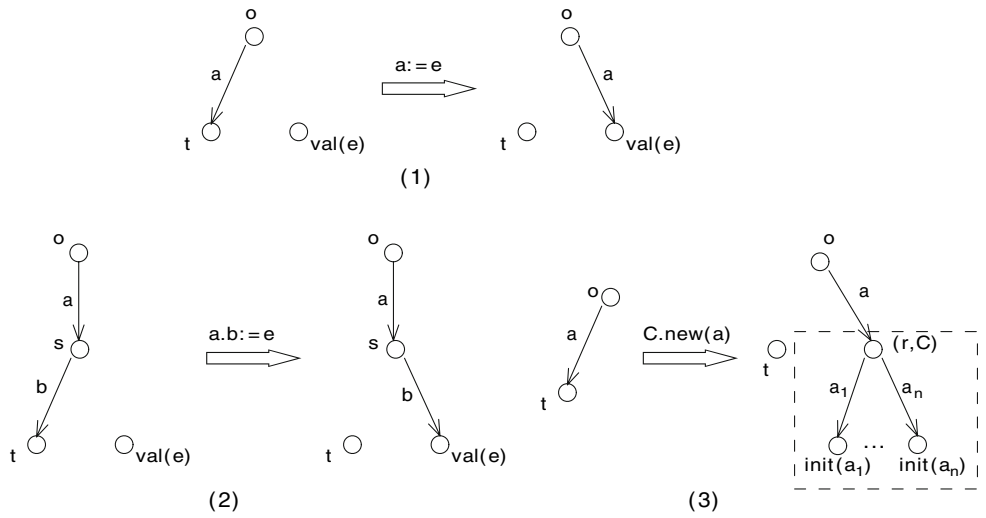


Fig. 4. Examples of object graph transformations

If this deletion makes some other objects unreachable, these objects and their outgoing edges will be deleted too. And this link-deletion should be done recursively, just as in garbage collection.

Since we allow nondeterministic choice, there is a set of possible final object graphs for an execution of a command from an initial object graph. Taking an object graph as a state of the program execution, the operational semantics agrees with denotational semantics defined in [HLL06] such that the condition defining the possible initial object graphs and the condition defining the possible final objects correspond to the pre and postconditions, respectively. For a command c we use $[\![c]\!]_\Gamma$ to denote the semantics of c under the structure defined by the class graph Γ.

To support design by stepwise refinement, we must be able to change the structure of the class graph, though we cannot change the names of the public classes and public methods.

Definition 5 Let F be a set of class names (called a **frame**), Γ_1 and Γ_2 be two class graphs both containing all names in F as their nodes. Γ_2 is a F-framed **structure refinement** of Γ_1, denoted as $\Gamma_1 \sqsubseteq_F \Gamma_2$ or simply $\Gamma_1 \sqsubseteq \Gamma_2$, if for any set X of main variables, there exists a relation ρ_o from $\mathcal{M}(\Gamma_1)_X$ to $\mathcal{M}(\Gamma_2)_X$ such that for any method $m(u : T_1;\ v : T_2)\{c_1\}$ defined in class C of Γ_1, where $(x : C) \in X$ for some variable x and $C \in F$, we can define a corresponding method $m(u : T_1;\ v : T_2)\{c_2\}$ in class C of Γ_2 and

$$\forall\, \Sigma \in \mathcal{M}(\Gamma_1)_X \bullet ([\![x.m(s;\ t)]\!]_{\Gamma_2}(\rho_o(\Sigma)) \subseteq \rho_o([\![x.m(s;\ t)]\!]_{\Gamma_1}(\Sigma)))$$

for any parameter values s, t.

The frame F in the definition in fact defines the *interface classes* of the class graph, and it is the interface classes that provide methods as *functional services* that can be invoked by the environment, e.g. the application program. Note that one can define different methods in the same frame of a class declaration section (represented by a class graph) to support different application programs. Therefore, the above definition says that when taking classes in F as the interface classes (or public classes), the refined graph Γ_2

1. Provides at least as many services to the environment as class graph Γ_1, that is for any method definable in Γ_1, there is a corresponding method definable in Γ_2, and
2. Provides as good services to the environment as class graph Γ_1, that is the execution of any method defined for class graph Γ_2 satisfies all properties of the execution of the corresponding method defined for Γ_1.

Notice that the set inclusion in the definition means that the execution of the command on the left is not more non-deterministic than the execution on the right. So, if Γ_1 is refined by Γ_2 with respect to frame F, then for any nonempty subset F' of F, Γ_2 is also a refinement of Γ_1 with respect to F'. We are interested in program refinement instead of *equivalence*, though most structural refinement rules are equivalences. This allows us to combine structural refinements with method refinements that can generally reduce non-determinism.

Later in the paper, we will define rules of transformations from a graph Γ_1 to another graph Γ_2 such that for each class graph transformation rule R, the "corresponding method" defined for Γ_2 can be obtained by a transformation R_c, which is derived from transformation R, on the command of the "original method".

4. Rules of structure refinement for structure expansion

We now study the class graph transformations that are used for object-oriented decomposition. Such a transformation shows how we can refine an object program by expanding its class structure, without weakening its capability of providing functional services.

Definition 6 Let $\Gamma_1 = \langle N_1, A_1, E_1\rangle$ and $\Gamma_2 = \langle N_2, A_2, E_2\rangle$ be class graphs, and a *frame* $F \subseteq N_1$. A mapping ρ from Γ_1 to Γ_2 is a F-framed **structure transformation**, denoted by $\rho_{[F]}$, if the following conditions hold:

1. The restriction $\hat{\rho}$ of ρ to the nodes is an injective mapping from N_1 to N_2, satisfying $\hat{\rho}(C) = C$ for each $C \in F$.
2. The restriction $\bar{\rho}$ to the edges maps each association attribute or inheritance relation (C, a, D) in E_1 to a path from $\hat{\rho}(C)$ to $\hat{\rho}(D)$ in Γ_2; and maps each data attribute (C, a, T) in E_1, where $T \in \mathcal{T}$, to a nonempty set of paths in Γ_2, each of which starts from $\hat{\rho}(C)$ and ends at a data type in $\mathcal{T}$.

   With Condition 2 in the above definition, we can decompose the restriction $\bar{\rho}$ of ρ to E into its two restrictions

   - the restriction of $\bar{\rho}$ to the relational edges, denoted by ρ_r, and
   - the restriction of $\bar{\rho}$ to the data attributes, denoted by ρ_d.

For a structural transformation ρ, let $\hat{\rho}_+$ denote the nodes of Γ_2 that are not in the range of $\hat{\rho}$ and $\bar{\rho}_+$ the sets of edges in Γ_2 that are not in any path in the range of $\bar{\rho}$. With these notations, we can represent a structure transformation ρ by the tuple $\langle F, \hat{\rho}, \rho_r, \rho_d, \hat{\rho}_+, \bar{\rho}_+\rangle$. Obviously, not all structure transformations defined above are structure refinements. They need to satisfy certain typing conditions [EEPT06] or algebraic conditions [CMR96].

Definition 7 A structure transformation $\rho_{[F]}$ from Γ_1 to Γ_2 is **well-typed** if it satisfies the following conditions:

1. If $(C, \triangleright, D)$ is an inheritance relation in Γ_1, then $\rho_r(C, \triangleright, D)$ is a path in Γ_2 containing only inheritance relations.
2. If (C, a, D) is an association attribute in Γ_1, then the last edge in the path $\rho_r(C, a, D)$ is also an association attribute in Γ_2.
3. For two different association attributes or inheritance relations (C_1, a_1, D_1) and (C_2, a_2, D_2) in Γ_1, $\rho_r(C_1, a_1, D_1)$ is not a suffix of $\rho_r(C_2, a_2, D_2)$ in Γ_2.
4. For any data attributes (C_1, a_1, T_1) and (C_2, a_2, T_2) in Γ_1, a path p_1 in $\rho_d(C_1, a_1, T_1)$ and a path p_2 in $\rho_d(C_2, a_2, T_2)$ in Γ_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 attribute (C, a, T) in Γ_1, let $\rho_d(C, a, T) = \{C_1.\beta_1.a_1, \ldots, C_1.\beta_n.a_n\}$ where $dest(C_1.\beta_i.a_i) = T_i \in \mathcal{T}$ in Γ_2 for $i : 1 \leqslant i \leqslant n$, there exists a surjective operation $g : T_1 \times \cdots \times T_n \to T$ such that the initial value of $C.a$ can be calculated from those of the target attributes: $init(C.a) = g(init(D_1.a_1), \ldots, init(D_n.a_n))$, where $D_i = dest(C_1.\beta_i)$ for $i : 1 \leqslant i \leqslant n$.

A structure transformation from Γ_1 to Γ_2 in fact defines an *implementation* of the classes, their attributes and associations in Γ_1 by those of Γ_2. A single inheritance relation is implemented by a number of steps of inheritance, single association attribute or edge in Γ_1 can be realized by a path, and a data attribute can be a set of paths in Γ_2. These are captured by Conditions (1)–(5) of the well-typed structure transformations. Condition 1 requires an inheritance should not be replaced by associations, and Condition 2 implies that association should not be implemented by inherence either. The falsification of Condition 3 (similarly for Condition 4) implies that D_1 and D_2 are the same class. In this case if $\rho_r(C_1, a_1, D_1)$ is a suffix of $\rho_r(C_2, a_2, D_2)$ in Γ_2, it would limit the functionality since an instance of $\hat{\rho}(C_2)$ can only access its associated instance of $\hat{\rho}(D_2)$ via the instance of $\hat{\rho}(C_1)$ linked to the instance of $\hat{\rho}(C_2)$. However, it is not necessary for an instance of C_2 to access its associated instance of D_2 via a link from an instance of C_1 to the instance of D_2. Finally, Condition 5 requires that any data attribute of a class in Γ_1 can be "computed" by an expression of the data attributes of the classes in Γ_2 that are the decomposition of the original class by the transformation. Conditions (4)–(5) allow the decomposition of a single attribute into a tree of classes and attributes. What we would like to show is that a well-typed transformation is a structure refinement.

Proposition 1 A well-typed structure transformation $\rho_{[F]}$ from Γ_1 to Γ_2 is a *structure refinement*.

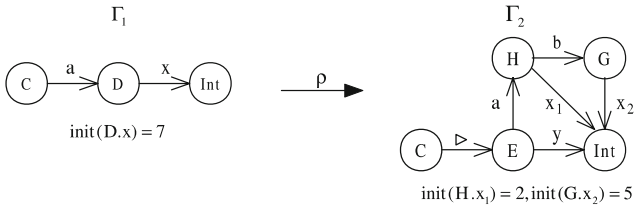


Fig. 5. An example of structure transformation

We only consider the well-typed structure transformations in the rest of this section, and thus simply call them *structure transformations*. The validity of the proposition is to be established in the following subsections in two steps, which are to be proved in Sect. 4.4:

1. Soundness: provide a small set of rules that are structure refinements.
2. Completeness: prove that any well-typed structure transformation can be obtained by applying a sequence of these refinement rules.

Example 2 Figure 5 illustrates a structure transformation $\rho_{[F]}$ in which class D corresponds to a class H, association (C, a, D) is realized by two relational edges, and data attribute (D, x, Int) is decomposed into two paths from class H to data types in the resulting class graph, formally

1. $F = \{C\}$,
2. $\hat{\rho}(C) = C$, $\hat{\rho}(D) = H$,
3. $\rho_r(C, a, D) = C.\triangleright.a$, $\rho_d(D, x, Int) = \{H.x_1, H.b.x_2\}$,
4. $\hat{\rho}_+ = \emptyset$, $\bar{\rho}_+ = \{(E, y, Int)\}$,
5. The addition operation on integers preserves the initial values of attributes: $init(D.x) = init(H.x_1) + init(G.x_2)$.

4.1. Object graph transformations derived from structure transformations

In this subsection, we show, for a set X of variables, how a structure transformation ρ from class graph Γ_1 to class graph Γ_2 determines a transformation ρ_o from the object graphs $\mathcal{M}_X(\Gamma_1)$ to the object graphs $\mathcal{M}_X(\Gamma_2)$. This is in fact to show that the left part of the diagram in Fig. 1 commutes.

Definition 8 Let ρ be a structure transformation from class graph Γ_1 to class graph Γ_2 with frame F, and X be a set of main variables. The **derived object graph transformation** of ρ, denoted by ρ_o, is a relation between object graphs $\mathcal{M}_X(\Gamma_1)$ and $\mathcal{M}_X(\Gamma_2)$ such that for any $\Sigma_1 \in \mathcal{M}_X(\Gamma_1)$ and $\Sigma_2 \in \mathcal{M}_X(\Gamma_2)$, $\rho_o(\Sigma_1, \Sigma_2)$ if the following conditions hold

1. (**Type consistency**) The restriction $\hat{\rho}_o$ of ρ_o to the set of nodes of the object graph Σ_1 is an injective mapping to the set of nodes of Σ_2 such that $\hat{\rho}_o$ maps each object node (r, C) in Σ_1 to a node (r', C') in Σ_2 with $C' = \hat{\rho}(C)$.
2. (**Global variables preservation**) For each association edge $(\varepsilon, x, (r, C))$ from the root in Σ_1 where $x : C \in X$ and C is a class, the restriction $\bar{\rho}_o$ of ρ_o to the edges maps it to an association edge $(\varepsilon, x, \hat{\rho}_o(r, C))$ in Σ_2 (note that $\hat{\rho}_o(r, C) = (r', C)$ for some r'), and $\bar{\rho}_o$ maps each data edge $(\varepsilon, x, (r, T))$ in Σ_1, where $x : T \in X$ and $T \in \mathcal{T}$, to a data edge $(\varepsilon, x, (r, T))$ in Σ_2.
3. (**Association consistency**) For each association attribute $((r_1, C), a, (r_2, D))$ in Σ_1, $\bar{\rho}_o((r_1, C), a, (r_2, D))$ is a path in Σ_2 such that
   (a) it starts from the node $\hat{\rho}_o(r_1, C)$ and ends with $\hat{\rho}_o(r_2, D)$, and
   (b) $hideRef(\bar{\rho}_o((r_1, C), a, (r_2, D))) = \rho_r(C, a, D)\backslash\triangleright$, where $hideRef(\alpha)$ denotes the sequence obtained by removing all references from the path α, and $\rho_r(C, a, D)\backslash\triangleright$ is the path obtained from the path $\rho_r(C, a, D)$ by removing all appearances of $\triangleright$.
4. (**Data consistency**) For each data attribute edge $((r, C), a, (v, T))$ in the object graph Σ_1, $\bar{\rho}_o((r, C), a, (v, T))$ is a set of paths $\{(r', C').\beta_1.a_1, \ldots, (r', C').\beta_n.a_n\}$ such that
   (a) $(r', C') = \hat{\rho}_o(r, C)$,

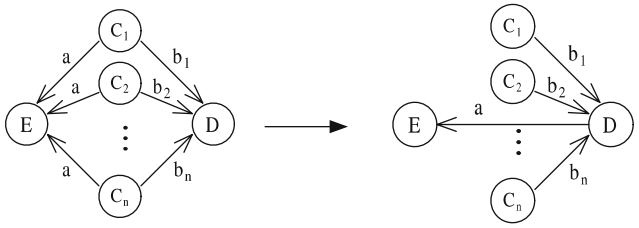


Fig. 6. R5 Forward attributes

(b) $hideRef(\beta_i) = \gamma_i \setminus \rhd$, where $\rho_d(C, a, T) = \{C'.\gamma_i.a_i \mid 1 \leqslant i \leqslant n\}$, and

(c) $v = g(v_1, \ldots, v_n)$, where $(v_i, T_i) = dest((r', C').\beta_i.a_i)$ for $i : 1 \leqslant i \leqslant n$ and g is the function given in the class graph transformation ρ that calculates the data attribute to (C, a, T) in Γ_1 from those at the destinations of $\rho_d(C, a, T)$.

4.2. Refinement rules for structure expansion

We give a set of rules in Fig. 7 which transform a class graph $\Gamma_1 = \langle N_1, A_1, E_1 \rangle$ to another $\Gamma_2 = \langle N_2, A_2, E_2 \rangle$. In this table, the first and second columns are the names and descriptions of the rules. The precondition for each rule in the third column ensures that the class graph after transforming is a well-formed one. Notice that each rule has a frame, which is depicted in the last column, representing the unchanged class names before and after the transformation. Here, we use $\mathcal{C}$ to denote the set of class names declared in Γ_1, which equals $N_1 \setminus \mathcal{T}$.

In fact, **R5.1** and **R5.2** are two special cases of the general rule **R5** for moving edges depicted in Fig. 6, where each edge (C_i, b_i, D) could be either an inheritance relation or an association attribute for $i : 1 \leqslant i \leqslant n$. However, these two special rules can be used to handle edge moving in most cases.

4.3. Soundness of the rules for structure expansion

It is straightforward to prove that each rule defines a well-typed structure transformation on class graphs. Thus each rule R determines a structure relation R_o between the object graphs of the corresponding class graphs.

Furthermore, each rule R derives a transformation R_c that transforms a command c well-formed with respect to class graph Γ_1 to a command $R_c(c)$ well-formed with respect to class graph Γ_2. That is, the variables and types in the command are all defined in the graphs. More precisely, statements and expressions are correctly typed [HLL06]. The derived command transformations are given in Fig. 8, where notation $[D/C]$ denotes a substitution for each occurrence of 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 static type of e is C or a subclass of C. The definitions of $[C.b.a/C.a]$ and $[g(C.x_1, \cdots, C.x_n)/C.x]$ are given similarly. Notice that in the command transformation of **R6**, we used the non-deterministic multiple assignment

$$g(C.x_1, \ldots, C.x_n) := e \text{ or } (C.x_1, \ldots, C.x_n) := (v_1, \ldots, v_n).g(v_1, \ldots, v_n) = e$$

to denote that $(C.x_1, \ldots, C.x_n)$ are assigned with values $(v_1, \ldots, v_n)$, respectively, such that $g(v_1, \ldots, v_n) = e$ holds. This might be of unbounded nondeterminism, but this does not disturb the theory of UTP.

The structure refinement rules must be used together with their derived transforms on commands when we develop an object-oriented software in an incremental and iterative development process such as RUP [Kru00]. This implies that the decomposition of low cohesive classes into simpler and more reusable classes [Lar01] and the decomposition of functionality by delegation, must be consistently combined [HLL06].

Theorem 1 (Soundness of the rules for expansion) If rule $R_{[F]}$ transforms Γ_1 to Γ_2, then $\Gamma_1 \sqsubseteq_I \Gamma_2$. Here, we use $R_{[F]}$ to denote a rule R taken from **R1** to **R7** with frame F.

Proof. For any set of main variables X, any variable $(x : T) \in X$ such that $T \in F$, we can construct a relation ρ_o from $\mathcal{M}(\Gamma_1)_X$ to $\mathcal{M}(\Gamma_2)_X$ for each rule such that for any method $m(u : T_1;\ v : T_2)\{c\}$ defined in class P of Γ_1 we can also define a corresponding method $m(u : T_1;\ v : T_2)\{R_c(c)\}$ in class T of Γ_2, and that the condition given in Definition 5 holds:

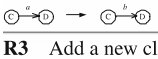
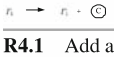
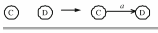
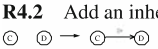
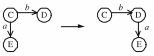
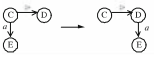
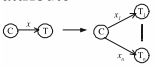
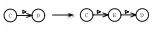
Rules	Description	Precondition	Frame
R1 Rename a class 	class name C is changed to a different name D	$C \in N_1$; $D \notin N_1$	$\mathcal{C} \setminus \{C\}$
R2 Rename an attribute 	the name of an attribute (C, a, D) is changed to (C, b, D)	$(C, a, D) \in E_1$; $a \neq b$; for any $C' \preceq C$, $b \notin Attr(C')$	$\mathcal{C}$
R3 Add a new class 	add a class C	$C \notin N_1$	$\mathcal{C}$
R4.1 Add an attribute 	add an attribute (C, a, D)	$C, D \in N_1$; for any $C' \preceq C$, $a \notin Attr(C')$	$\mathcal{C}$
R4.2 Add an inheritance 	add an inheritance relation $(C, \triangleright, D)$	$C, D \in N_1$; for each $D' \in N_1$, $(C, \triangleright, D') \notin E_1$; $D \not\preceq C$; for each $C' \preceq C$, $attr(C') \cap Attr(D) = \emptyset$	$\mathcal{C}$
R5.1 Forward an attribute through an association 	an attribute (C, a, E) is forwarded through an association attribute (C, b, D) to form another attribute (D, a, E)	$(C, a, E), (C, b, D) \in E_1$; for any $D' \preceq D$, $a \notin Attr(D')$	$\mathcal{C}$
R5.2 Forward an attribute through an inheritance 	an attribute (C, a, E) is forwarded through an inheritance relation $(C, \triangleright, D)$ to form another attribute (D, a, E)	$(C, a, E), (C, \triangleright, D) \in E_1$; for any $D' \preceq D$, $a \notin Attr(D')$ unless $D' \preceq C$	$\mathcal{C}$
R6 Decompose a data attribute 	a data attribute (C, x, T) is decomposed to a set of data attributes $(C, x_1, T_1), \ldots, (C, x_n, T_n)$	$(C, x, T) \in E_1$; there exists a surjective primitive operation $g : T_1 \times \ldots \times T_n \to T$ preserving the initial values of attributes; for each $i : 1 \leq i \leq n$ and $C' \preceq C$, $x_i \notin Attr(C')$	$\mathcal{C}$
R7 Decompose an inheritance 	an inheritance relation $(C, \triangleright, D)$ is decomposed into two inheritance relations $(C, \triangleright, E)$ and $(E, \triangleright, D)$	$(C, \triangleright, D) \in E_1$; $E \in N_1$; $D \not\preceq E$; for each $F \in N_1$, $(E, \triangleright, F) \notin E_1$; for each $E' \preceq E$, $attr(E') \cap Attr(D) = \emptyset$	$\mathcal{C}$

Fig. 7. Basic rules

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

Fig. 8. Command transformation

- **Case R1.** Construct the relation ρ_o such that $\rho_o(\Sigma, \Sigma')$ if and only if Σ' is obtained from Σ by substituting each node of the form (r, C) in Σ with (r, D). Notice that $R_c(c) = c[D/C]$, so $[\![x.m(s;\ t)]\!]_{\Gamma_2} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma_1}$, where $\circ$ denotes the composition of two relations.
- **Case R2.** Construct relation ρ_o such that $\rho_o(\Sigma, \Sigma')$ if and only if the object graph Σ' is obtained from Σ by substituting each edge of the form $((r, C'), a, v)$, where $C' \preceq C$, with $((r, C'), b, v)$. Since $R_c(c) = c[C.b/C.a]$, we have $[\![x.m(s;\ t)]\!]_{\Gamma_2} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma_1}$.
- **Case R3, R4, R5.2 and R7.** For each of these, the corresponding transformation is the identity transformation, that is $\rho_o(\Sigma, \Sigma')$ if and only if $\Sigma' = \Sigma$.

- **Case R5.1.** For the class graph transformation ρ of this rule, we construct ρ_o such that $\rho_o(\Sigma, \Sigma')$ if and only if the object graph Σ' is obtained from Σ by substituting each edge of the form $((r, C'), a, v)$, where $C' \preceq C$ thus there exists another edge $((r, C'), b, v')$ in Σ, with the edge (v', b, v). For this, we also have $R_c(c) = c[C.b.a/C.a]$, so $[\![x.m(s;\ t)]\!]_{\Gamma_2} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma_1}$.
- **Case R6.** For this case, we can construct the transformation ρ_o such that $\rho_o(\Sigma, \Sigma')$ if and only if the object graph Σ' is obtained from Σ by substituting each edge of the form $((r, C'), x, (k, T))$, where $C' \preceq C$, with a set of edges $\{((r, C'), x_1, (k_1, T_1)), \ldots, ((r, C'), x_n, (k_n, T_n))\}$, satisfying $g(k_1, \ldots, k_n) = k$. Since $R_c(c) = c[g(C.x_1, \ldots, C.x_n)/C.x]$, we have $[\![x.m(s;\ t)]\!]_{\Gamma_2} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma_1}$.

□

This theorem implies that all rules given in Fig. 7 are structure refinements. It also shows the commutativity of the diagram of Fig. 1 in Sect. 1. Obviously, the structure refinement relation defined in Definition 5 is transitive.

Corollary 1 If Γ_1 is transformed to Γ_2 by a sequential applications of rules $R_{1[F_1]}, \ldots, R_{k[F_k]}$, then $\Gamma_1 \sqsubseteq_I \Gamma_2$, provided $F = F_1 \cap \cdots \cap F_k \neq \emptyset$.

4.4. Completeness of rules R1–R7 and validity of Proposition 1

We now prove the completeness of the rules **R1**–**R7** for well-typed structure refinements. This together with the soundness in Theorem 1 implies the validity of Proposition 1.

Theorem 2 (Completeness result I) If $\rho_{[F]}$ is a well-typed structure transformation from Γ_1 to Γ_2, there exists a finite sequence of applications of rules $R_{1[F_1]}, \ldots, R_{k[F_k]}$ taken from **R1** to **R7** that transform Γ_1 to Γ_2 where $F \subseteq F_i$ for $i : 1 \leqslant i \leqslant k$.

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

1. Change each class C to $\hat{\rho}(C)$, by applications of **R1**.
2. Using Rule **R6**, decompose each data attribute (C, x, T) to a set of data attributes:

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

   provided $\rho_d(C, x, T) = \{(C.\beta_1.x_1), \ldots, (C.\beta_n.x_n)\}$ and $dest(C.\beta_i.x_i) = T_i$.
3. For edges, there are two cases
   (a) using **R3, R4** and **R5**, change each data attribute (C, x, T) to a path $C.\beta.x$ in $\rho_d(C, x, T)$, or
   (b) using applications of **R2, R3, R4, R5** and **R7**, change each association or inheritance relation (C, a, D) to a path $\rho_r(C, a, D)$
4. Add additional nodes and edges according to $\hat{\rho}_+$ and $\bar{\rho}_+$ by using **R3** and **R4**.

□

From this completeness theorem and the soundness theorem, we have proved the validity of Proposition 1.

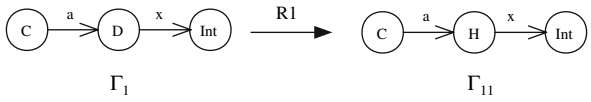
Corollary 2 (**Proposition 1 holds**) A well-typed structure transformation $\rho_{[F]}$ from Γ_1 to Γ_2 is a structure refinement.

Example 3 For the structure transformation illustrated in Example 2, Fig. 9 shows the applications of the rules that transform Γ_1 to Γ_2.

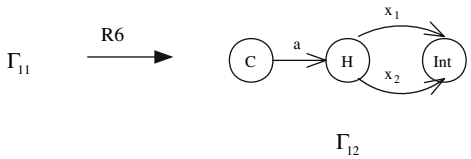
5. Structure refinement for graph compression

In previous sections, refinement is achieved by expanding the class graph and the refinement rules are independent of what methods would be declared in classes of the refined graph. We now consider how to compress a class graph while preserving the provided services. Generally, a class structure can not be arbitrarily compressed since after removing or combining classes and attributes we may lose important services determined by methods. Therefore, the configuration of methods does matter when we want to compress a class graph, and we now extend the definition of class graphs with methods of classes.

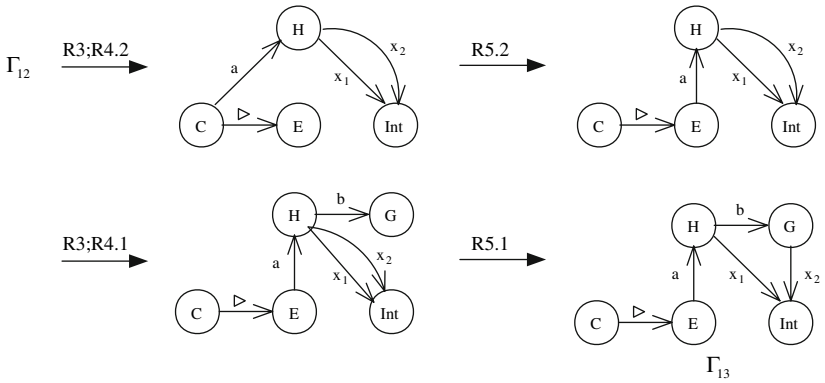
Step 1: Rename classes



Step 2: Decompose data attributes



Step 3: Transform edges



Step 4: Add extra nodes and edges

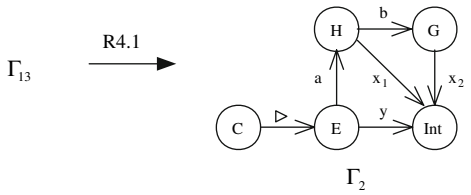


Fig. 9. An example

Definition 9 (Class graph with methods) A class graph with methods is represented as $\Gamma = \langle N, A, E, M \rangle$ where M is a function that labels a class node with a set of methods of the form $m(u : T_1;\ v : T_2)\{c\}$, or $m(T_1\ u;\ T_2\ v)\{c\}$, where m is the method name, $u : T_1$ and $v : T_2$ are the input and output parameters, and command c is the method body.

We assume that the names of methods declared in the same class are distinct. Based on this assumption, we can prefix a method with the name of its class so that we can use M as a set of methods, and then simply use $C :: m \in M$ to denote the case that there exists a method $m(u : T_1;\ v : T_2)\{c\} \in M(C)$. Of course, each method declared in a class graph should be well-typed [HLL06, ZZLQ06], and we only consider well-typed class graphs in the following discussion.

5.1. Interfaces and extended structure refinement

Generally, the main method of an object-oriented program can only directly access the public methods of public classes in the classes declaration section. We call these classes and methods that are directly accessible by the main method the *interface*.

Definition 10 (Interface) Let $\Gamma = \langle N, A, E, M\rangle$. An **interface** I is a pair $\langle PC, PM\rangle$, where

- $PC \subseteq N$ is a set of class names
- $PM \subseteq M$ is a set of method names, each of which has the form $C :: m$ such that for each $C :: m \in PM$, $C \in PC$.

For interfaces $I = \langle PC, PM\rangle$ and $I' = \langle PC', PM'\rangle$ of Γ, $I \cap I'$ denotes the interface $\langle PC \cap PC', PM \cap PM'\rangle$, and the predicate $I \subseteq I'$ denotes $(PC \subseteq PC') \wedge (PM \subseteq PM')$. We also assume the methods in each interface have distinct names except those with overriding relations. That is, $C :: m, D :: m \in PM$ implies m is a polymorphic method declared in both class C and D and maybe some other classes in Γ. It is only a technical assumption which could simplify our discussion without losing generality, since we can rename some methods to meet this assumption without changing the behavior of the program. The notion of interfaces extends that of frames in the previous sections. An interface of a class graph defines which classes and which methods in these classes are public ones and thus accessible from outside the class graph. With the concept of interfaces, we can naturally extend the definition of structure refinement.

Definition 11 (Extended structure refinement) Let $I = \langle PC, PM\rangle$ be an interface of both class graphs $\Gamma_1 = \langle N_1, A_1, E_1, M_1\rangle$ and $\Gamma_2 = \langle N_2, A_2, E_2, M_2\rangle$. Γ_2 is an I-framed structure refinement of Γ_1, denoted as $\Gamma_1 \sqsubseteq_I \Gamma_2$, if for any set X of main variables, there exists a relation ρ_o from $\mathcal{M}(\Gamma_1)_X$ to $\mathcal{M}(\Gamma_2)_X$ such that for any method $m(u : T_1;\ v : T_2)\{c_1\} \in M_1(C)$ of Γ_1 with $C :: m \in PM$, there is a correspondence method $m(u : T_1;\ v : T_2)\{c_2\} \in M_2(C)$ of Γ_2 such that

$$\forall\, \Sigma \in \mathcal{M}(\Gamma_1)_X \bullet ([\![x.m(s;\ t)]\!]_{\Gamma_2}(\rho_o(\Sigma)) \subseteq \rho_o([\![x.m(s;\ t)]\!]_{\Gamma_1}(\Sigma)))$$

for any $(x : C) \in X$ and parameters s, t.

Obviously, if class graph Γ_1 is refined by class graph Γ_2 with respect to interface I, then Γ_2 is also a refinement of Γ_1 with respect to any non-empty interface $I' \subseteq I$. Besides, it is easy to see that the refinement relation defined above is a reasonable extension of that in Definition 5. In fact, the former relation corresponds to the special case of the latter when PM includes all methods that are "definable" in classes in PC. We use $\equiv$ to denote the equivalence relation between class graphs with respect to refinement.

5.2. Refinement rules for removing redundant structure elements

Intuitively, we can remove classes, attributes and methods that are not related to (or referred to), either directly or indirectly, in the interface. In other words, elements of a class graph that are of no contribution to the provision of services can be deleted. We first formalize the definitions of the accessible nodes, edges and methods of a class graph. For this, the following two sets are first defined. Let $\Gamma = \langle N, A, E, M\rangle$ be a class graph and $C_1, C_2 \in N$. The set $cb(C_1, C_2)$ contains the nodes that are superclasses of C_1 and subclasses of C_2, and $eb(C_1, C_2)$ denotes the set of inheritance relations between C_1 and C_2:

$$cb(C_1, C_2) \stackrel{def}{=} \{C \in N \mid C \notin \mathcal{T}, C_1 \preceq C \preceq C_2\}$$
$$eb(C_1, C_2) \stackrel{def}{=} \{(C, \rhd, D) \in E \mid C_1 \preceq C, D \preceq C_2\}$$

Obviously, $cb(C_1, C_2)$ and $eb(C_1, C_2)$ are empty if C_1 is not a subclass of C_2.

Definition 12 (Directly accessible elements of expressions) Given a class graph $\Gamma = \langle N, A, E, M\rangle$ and a well-typed expression e, we use $nodes(e)$ and $edges(e)$ to denote respectively the set of class nodes and the set of edges that expression e *directly accesses* or refers to, defined as follows:

1. if e is variable x, then $nodes(e) = \{dtype(x)\} \setminus \mathcal{T}$, $edges(e) = \emptyset$, that is a simple variable only refers to its type if it is a class.
2. if e is *null* or a literal, then $nodes(e) = edges(e) = \emptyset$, that is a *null* object or a constant does not refer to any class or edge.
3. if e is an attribute reference a of the current class C', and C'' is the superclass of C' such that $a \in attr(C'')$, then $nodes(e) = cb(C', C'') \cup \{dtype(a)\} \setminus \mathcal{T}$, $edges(e) = eb(C', C'') \cup \{(C'', a, dtype(a))\}$; this means a refers to its declared type, as well as the class where it declared, the current class and other classes and inheritance edges between them.
4. if e is $e_0.a$, then $nodes(e) = nodes(e_0) \cup cb(C', C'') \cup \{dtype(e_0.a)\} \setminus \mathcal{T}$, $edges(e) = edges(e_0) \cup eb(C', C'') \cup \{(C'', a, dtype(e_0.a))\}$, where $C' = dtype(e_0)$ and C'' is the superclass of C' such that $a \in attr(C'')$; that is, besides what e_0 can access, expression $e_0.a$ refers to its declared type, as well as the class where it declared, the declared type of e_0 and other classes and inheritance edges between them.

5. if e is $(C_0)e_0$, then $nodes(e) = nodes(e_0) \cup cb(C', C'')$, $edges(e) = edges(e_0) \cup eb(C', C'')$, where $C' = C_0 \wedge C'' = dtype(e_0)$ if $C_0 \preceq dtype(e_0)$, or $C' = dtype(e_0) \wedge C'' = C_0$ otherwise, this says that, besides what e_0 can access, expression $(C_0)e_0$ refers to all classes and inheritance edges between C_0 and the declared type of e_0.
6. if e is $f(e_1, \ldots, e_n)$, then $nodes(e) = nodes(e_1) \cup \ldots \cup nodes(e_n)$, and $edges(e) = edges(e_1) \cup \ldots \cup edges(e_n)$; this says that the expression $f(e_1, \ldots, e_n)$ refers to all classes and edges its sub-expressions can access.

Using the sets of nodes and edges that are directly referred to in expressions, we can inductively define the sets of nodes and edges that are referred to by a command of a method, and the set of methods that are called in a command.

Definition 13 (Directly accessible elements of commands) Given a class graph $\Gamma = \langle N, A, E, M \rangle$ and a well-typed command c, we use $nodes(c)$, $edges(c)$ and $meths(c)$ to denote respectively the set of class nodes, edges and methods[3] c *accesses* or *refers to directly*, defined as follows:

1. *skip* and *chaos* command do not refer to any class, edge or method: if c is *skip* or *chaos*, then $nodes(c) = \emptyset$, $edges(c) = \emptyset$ and $meths(c) = \emptyset$;
2. an assignment command refers to what the two expressions can access, and all classes and inheritance edges between the declared types of them: if c is $e_1 := e_2$, then $nodes(c) = nodes(e_1) \cup nodes(e_2) \cup cb(dtype(e_2), dtype(e_1))$, $edges(c) = edges(e_1) \cup edges(e_2) \cup eb(dtype(e_2), dtype(e_1))$ and $meths(c) = \emptyset$;
3. command $C.new(e)$ refers directly to all classes and inheritance edges between C and the declared type of e plus those that e accesses: if c is $C.new(e)$, then $nodes(c) = nodes(e) \cup cb(C, dtype(e))$, $edges(c) = edges(e) \cup eb(C, dtype(e))$ and $meths(c) = \emptyset$;
4. a loop command refers directly to what the loop condition and loop body command can access: if c is $b * c_1$, then $nodes(c) = nodes(c_1) \cup nodes(b)$, $edges(c) = edges(c_1) \cup edges(b)$ and $meths(c) = meths(c_1)$;
5. similarly if c is $c_1 \lhd b \rhd c_2$, then $nodes(c) = nodes(c_1) \cup nodes(c_2) \cup nodes(b)$, $edges(c) = edges(c_1) \cup edges(c_2) \cup edges(b)$ and $meths(c) = meths(c_1) \cup meths(c_2)$;
6. inductively for sequential composition and nondeterministic choice, i.e. when c is of the form $c_1; c_2$ or $c_1 \sqcap c_2$, $nodes(c) = nodes(c_1) \cup nodes(c_2)$, $edges(c) = edges(c_1) \cup edges(c_2)$ and $meths(c) = meths(c_1) \cup meths(c_2)$;
7. for a local variable declaration statement, i.e. when c is of the form $var\ T\ x;\ c_1;\ end\ x$, $nodes(c) = nodes(c_1) \cup \{T\} \setminus \mathcal{T}$, $edges(c) = edges(c_1)$ and $meths(c) = meths(c_1)$;
8. when c is a method invocation $e.m(e_1;\ e_2)$, where the type of formal parameters e_1 and e_2 are respectively T_1 and T_2, then
   (a) the nodes that c accesses include those e, e_1 or e_2 can access, all classes between the types of formal and actual parameters, or between C', the declared type of e, and C'' where m is declared.
   $$nodes(c) = nodes(e) \cup nodes(e_1) \cup nodes(e_2) \cup cb(C_1, T_1) \cup cb(C_2, T_2) \cup cb(C', C'')$$
   (b) c accesses one single method $C'' :: m$.
   $$meths(c) = \{C'' :: m\}$$
   (c) the edges that c accesses include those e, e_1 or e_2 can access, all inheritance edges between the types of formal and actual parameters, or between class C' and class C''.
   $$edges(c) = edges(e) \cup edges(e_1) \cup edges(e_2) \cup eb(C', C'') \cup eb(C_1, T_1) \cup eb(C_2, T_2)$$
   where $C_1 = dtype(e_1)$, $C_2 = dtype(e_2)$, $C' = dtype(e)$ and C'' is the *least* superclass of C' such that m is declared in C''.

Using the above definitions we can calculate the sets of nodes and edges that are directly referred to in a method and the set of methods that can be called in a method.

Definition 14 (Directly accessible elements of methods) Given a class graph $\Gamma = \langle N, A, E, M \rangle$ and any well-typed method mth, we use $nodes(mth)$, $edges(mth)$ and $meths(mth)$ to denote respectively the sets of class nodes, edges and

[3] A method is accessible by a command if the command calls that method.

methods that mth accesses directly, they are defined as: for a method mth that is declared as $m(u : T_1;\ v : T_2)\{c\}$, then $nodes(mth) = nodes(c)$, $edges(mth) = edges(c)$ and $meths(mth) = meths(c)$.

Now we use the structure of the body command of a method to define inductively the sets of nodes and edges that can be referred to by that method.

Definition 15 (Accessible elements of methods) Given a class graph $\Gamma = \langle N, A, E, M\rangle$ and a well-typed method mth, we use $Nodes(mth)$, $Edges(mth)$ and $Meths(mth)$ to denote the sets of accessible class nodes, edges and methods of mth, respectively. These are defined as follows:

1. for any class node $C \in N$, $C \in Nodes(mth)$ if there exists a finite sequence of methods $[mth_0, \ldots, mth_n](n \geqslant 0)$ such that $mth_0 = mth$, $mth_i \in meths(mth_{i-1})(1 \leqslant i \leqslant n)$ and $C \in nodes(mth_n)$.
2. for any edge $edg \in E$, $edg \in Edges(mth)$ if there exists a finite sequence of methods $[mth_0, \ldots, mth_n](n \geqslant 0)$ such that $mth_0 = mth$, $mth_i \in meths(mth_{i-1})(1 \leqslant i \leqslant n)$ and $edg \in edges(mth_n)$.
3. for any method $mth' \in M$, $mth' \in Meths(mth)$ if there exists a finite sequence of methods $[mth_0, \ldots, mth_n](n \geqslant 0)$ such that $mth_0 = mth$, $mth_i \in meths(mth_{i-1})(1 \leqslant i \leqslant n)$ and $mth' \in meths(mth_n)$.

The nodes and edges that can be referred to in an interface are those that are related to the nodes of the interfaces plus those that can be referred to in the methods of the interface. The idea is that nodes and edges that cannot be referred to by the interface of a class graph are redundant and thus can be removed by structure transformation without affecting the functionality of the graph.

Definition 16 (Accessible elements of interfaces) Given a class graph $\Gamma = \langle N, A, E, M\rangle$ and an interface $I = \langle PC, PM\rangle$ of Γ, we use notations $Nodes(I)$, $Edges(I)$ and $Meths(I)$ to denote the sets of accessible class nodes, edges and methods of I, respectively. These are defined as follows:

1. $Nodes(I) = \bigcup\{cb(C, D) \mid C, D \in PC\} \cup \bigcup\{Nodes(C :: m) \mid C :: m \in PM\}$.
2. $Edges(I) = \bigcup\{eb(C, D) \mid C, D \in PC\} \cup \bigcup\{Edges(C :: m) \mid C :: m \in PM\}$.
3. $Meths(I) = \{C :: m \mid \exists D \succeq C, eb(C, D) \subseteq Edges(I), D :: m \in (PM \cup Meths(PM))\}$.

where $Meths(PM)$ denotes all methods accessed by some public method, that is $\bigcup\{Meths(P :: m) \mid P :: m \in PM\}$.

From the above definitions, the nodes, edges and methods that are accessible of an interface satisfy some good properties (also called *healthiness conditions*) formulated in the following theorem.

Theorem 3 Let $\Gamma = \langle N, A, E, M\rangle$ be a class graph and $I = \langle PC, PM\rangle$ an interface of Γ, then

1. Each accessible edge of the interface starts from and ends at accessible nodes, that is for each edge $(C, a, D) \in E$, if $(C, a, D) \in Edges(I)$, then $C \in Nodes(I)$, and $D \in Nodes(I)$ unless D is a primitive type.
2. A class is accessible by the interface if it declares a method accessible by the interface, that is each method $C :: m \in M$, if $C :: m \in Meths(I)$, then $C \in Nodes(I)$.
3. If a method can be called from the interface, then any method that overrides it through accessible inheritance edges can also be called from the interface, that is for methods $C :: m, D :: m \in M$ such that $C \succeq D$ and $eb(D, C) \subseteq Edges(I)$, $C :: m \in Meths(I)$ implies $D :: m \in Meths(I)$.

Proof. Straightforward from the above definitions.

1. If $(C, a, D) \in Edges(I)$ which equals to $\bigcup\{eb(P_1, P_2) \mid P_1, P_2 \in PC\} \cup \bigcup\{Edges(P :: m) \mid P :: m \in PM\}$, then either $(C, a, D) \in eb(P_1, P_2)$ for some $P_1, P_2 \in PC$ or $(C, a, D) \in Edges(P :: m)$ for some $P :: m \in PM$. So, either $C, D \in cb(P_1, P_2)$ holds, or $C \in Nodes(P :: m)$ and $D \in Nodes(P :: m)$ hold unless D is a primitive type. In each case, we have $C \in Nodes(I)$, and $D \in Nodes(I)$ if D is not a primitive type.
2. If $C :: m \in Meths(I)$, then there exists a superclass C' of C such that $eb(C, C') \subseteq Edges(I)$ and $C' :: m \in (PM \cup Meths(PM))$. If $C' \neq C$, then $eb(C, C') \subseteq Edges(I)$ implies $C \in Nodes(I)$. If $C' = C$, then $C :: m \in PM$ or $C :: m \in Meths(mth)$ holds for some public method $mth \in PM$. So, either $C \in PC$ or $C \in Nodes(mth)$. In each case, we also have $C \in Nodes(I)$.
3. For methods $C :: m, D :: m \in M$ such that $C \succeq D$ and $eb(D, C) \subseteq Edges(I)$, if $C :: m \in Meths(I)$, then there exists a superclass C' of C such that $eb(C, C') \subseteq Edges(I)$ and $C' :: m \in (PM \cup Meths(PM))$. So, C' is also a superclass of D and $eb(D, C') \subseteq Edges(I)$ holds. As a result, we have $D :: m \in Meths(I)$.

□

Rule R	**R8.1** Remove a class	**R8.2** Remove an edge	**R8.3** Remove methods
Description	Remove a class C from N	Remove an edge (C, a, D) from E	Remove all the redundant methods from M
Precondition	$C \in N$; $M(C) = \emptyset$; class C is not associated with any edge; $C \notin Nodes(I)$	$(C, a, D) \in E$; $(C, a, D) \notin edges(C' :: m)$ for each method $C' :: m \in M$	$true$
Command Transformation	$R_c(c) = c$	$R_c(c) = c$	$R_c(c) = c$
Frame	I	I	I

Fig. 10. Rules for removing redundant elements

An interface I is used to represent the public classes and public methods of a class graph Γ. Thus, $Nodes(I)$, $Edges(I)$ and $Meths(I)$ that are accessible by the interface denote the set of nodes, edges and methods that contribute to the provision of services of Γ. In contrary, the other class nodes, edges and methods are *redundant* in that they do not take part in the services of Γ with respect to I. As a result, they can be simply removed from Γ without losing any functionality. This leads to the following theorem.

Theorem 4 Let $\Gamma = \langle N, A, E, M\rangle$ be a class graph and $I = \langle PC, PM\rangle$ an interface of Γ. If a well-typed class graph $\Gamma' = \langle N', A', E', M'\rangle$ is obtained from Γ after removing any redundant class nodes, edges and methods, that is if Γ' contains all the accessible elements of the interface: $Nodes(I) \subseteq N' \subseteq N$, $Edges(I) \subseteq E' \subseteq E$ and $Meths(I) \subseteq M' \subseteq M$, then Γ' is equivalent to Γ, that is $\Gamma' \equiv_I \Gamma$.

Proof. Since Γ' is a subgraph of Γ, $\Gamma' \sqsubseteq_I \Gamma$. We only need to prove $\Gamma \sqsubseteq_I \Gamma'$.

For any set of main variables X, any variable $(x, P) \in X$ such that $P \in PC$, we construct a relation ρ_o from $\mathcal{M}(\Gamma)_X$ to $\mathcal{M}(\Gamma')_X$ such that $\rho_o(\Sigma, \Sigma')$ if and only if object graph Σ' is obtained from Σ by deleting

- all object nodes of the form (r, C), where $C \in N \setminus N'$,
- all edges of the form $((r_1, D_1), a, (r_2, D_2))$ such that there exists $(C_1, a, C_2) \in E \setminus E'$ and $C_1 \preceq D_1$ and $C_2 \preceq D_2$, and
- all data nodes related to these edges, from object graph Σ.

For any method $m(u : T_1;\ v : T_2)\{c\}$ defined in class C of Γ such that $C :: m \in PM$, $P :: m \in Meth(I)$ is not a redundant method, thus $P :: m \in M'$. This means there is a corresponding method $m(u : T_1;\ v : T_2)\{c\}$ in class C of Γ', and $[\![x.m(s;\ t)]\!]_{\Gamma'} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma}$ holds. So, $\Gamma \sqsubseteq_I \Gamma'$. □

Note that for a well-typed class graph Γ, the class graph Γ' obtained by removing all the inaccessible class nodes, edges and methods from Γ is also well-typed. This is because those redundant elements are not referred to by the rest part of the graph. From Theorem 4, we know that Γ' is equivalent with Γ.

5.2.1. Rules for removing redundant elements

Let $\Gamma = \langle N, A, E, M\rangle$ be a class graph and $I = \langle PC, PM\rangle$ be an interface of Γ. We provide a set of rules in Fig. 10 to remove redundant nodes, edges and methods from Γ. Notice that each rule has a frame, representing those class names and method names which should remain unchanged. We use $R_{[I]}$ to represent a rule R with frame I. The soundness of Rule **R8** is ensured by Theorem 4. The theorem also ensures that Rule **R8** is an equivalence rule.

5.3. Rules for combining classes

It is not difficult to understand that, in a class graph $\Gamma = \langle N, A, E, M\rangle$, a class D can be merged into another class C in the following steps:

1. turn each outgoing (incoming) edge from (to) D into an outgoing (incoming) edge from (to) C,
2. turn each edge between C and D to a self loop edge of C, and
3. move each method declared in D to C.

We can name the combined class by C, D or any fresh class name, but we just use C in the following discussions.

Rules for merging classes are provided in Fig. 11. When merging two classes, we should discriminate two different cases: (a) if one class is a subclass of the other, we use **R9.1**, **R9.2**, and (b) if there is no such subclass

Rule R	**R9.1** merge to superclass	**R9.2** merge to subclass	**R9.3** merge to another class
Description	Merge a class D to its direct superclass C	Merge a class D to its direct subclass C	Merge a class D to another class C, where no subclass relation exists between C and D
Precondition	$C, D \in N$; $(D, \triangleright, C) \in E$; for each $C' \preceq C$ but $C' \neq D$, $attr(C') \cap attr(D) = \emptyset$; if $C' :: m \in M$ for some $C' \succeq C$, then $D :: m \notin M$	$C, D \in N$; $(C, \triangleright, D) \in E$; for each $D' \preceq D$ but $D' \neq C$, $attr(D') \cap attr(C) = \emptyset$; if $D' :: m \in M$ for some $D' \succeq D$, then $C :: m \notin M$	$C, D \in N$; $C \not\preceq D$; $D \not\preceq C$; $(C, \triangleright, C_1), (D, \triangleright, D_1) \in E$ implies $C_1 = D_1$; for each $C' \preceq C$, $attr(C') \cap Attr(D) = \emptyset$; for each $D' \preceq D$, $attr(D') \cap Attr(C) = \emptyset$; $\{m \mid C' :: m \in M\} \cap \{m \mid D' :: m \in M\} = \emptyset$ if $C' \preceq C \wedge D \preceq D'$ or $C' \succeq C \wedge D \succeq D'$
Command Transformation	$R_c(c) = c[C/D]$	$R_c(c) = c[C/D]$	$R_c(c) = c[C/D]$
Frame	$I = \langle N \setminus \{D\}, \{G :: m \in M \mid G \neq D\}\rangle$	$I = \langle N \setminus \{D\}, \{G :: m \in M \mid G \neq D\}\rangle$	$I = \langle N \setminus \{D\}, \{G :: m \in M \mid G \neq D\}\rangle$

Fig. 11. Rules for combining classes

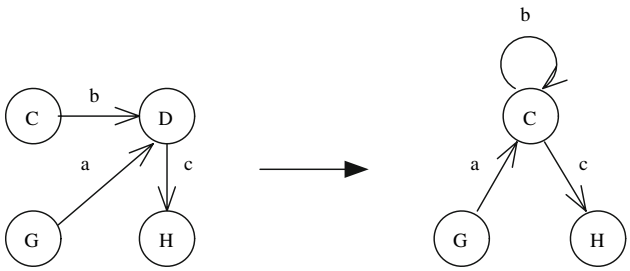


Fig. 12. Class combination

relation between them, we then use **R9.3**. We combine more than two classes to one by a number of applications of these rules. The soundness of **R9** is straightforward for each case. Also notice that **R9** is an equivalence rule, since the merging of classes does not influence the classes and methods in the interface thus does not change the global functionality of the class graph.

6. Refinement rules for changing methods

In addition to refining the structure of class graph, we sometimes need to change a method while keeping the class structure unaltered. In principle, this kind of refinement corresponds to the simple procedural refinement without changing the data space [Mor94, BvW98].

For a class graph $\Gamma = \langle N, A, E, M\rangle$, each of the rules in Fig. 13 allows us to transform a class graph Γ to a graph $\Gamma' = \langle N, A, E, M'\rangle$ without changing the set of nodes and edges. Notice that a special case of **R10.1** is to rename a method which is not polymorphic. That is, if a method $C :: m$ does not override or is not overridden by any other method, Rule **R10.1** allows us to rename it to fresh name $C :: m'$ while substituting each invocation to m with an invocation to m' in the bodies of other methods.

It is easy to see the soundness of rules **R10.1**, **R10.2** and **R10.3**. Also any class graph Γ' obtained from a class graph Γ by these rules is equivalent to Γ. The reason lies in the fact that when renaming, adding or moving up methods according to these rules, we do not change the functionality of methods in the interface.

Theorem 5 (Soundness and equivalence) If Γ_1 is transformed to Γ_2 by a sequence $R_{1[I_1]}$, ..., $R_{k[I_k]}$ of applications of rules **R8**, **R9** and **R10**, then $\Gamma_1 \sqsubseteq_I \Gamma_2$ and $\Gamma_2 \sqsubseteq_I \Gamma_1$, Thus $\Gamma_1 \equiv_I \Gamma_2$, provided the interface $I = I_1 \cap \cdots \cap I_k$ is not empty.

Proof. Since the refinement relation is transitive, and the soundness of **R8** is ensured by Theorem 4, we only need to prove that each rule in **R9** and **R10** is a sound equivalence rule. This is to prove that for each of these rules with frame $I = \langle PC, PM\rangle$ that transforms $\Gamma = \langle N, A, E, M\rangle$ to $\Gamma' = \langle N', A', E', M'\rangle$, any set of main variables X, and any variable $(x, C) \in X$ such that $C \in PC$, we can construct a transformation ρ_o from $\mathcal{M}(\Gamma)_X$ to $\mathcal{M}(\Gamma')_X$ such that for any

Rule R	**R10.1** rename methods	**R10.2** add a method	**R10.3** move up a method
Description	Rename a whole set of polymorphic methods $\{C_0 :: m, C_1 :: m, \cdots, C_k :: m\}$ to $\{C_0 :: m', C_1 :: m', \cdots, C_k :: m'\}$ simultaneously, where $k \geq 0$	Add a well-typed new method m, where m is a fresh name, to class C	Move a method m form class C to its direct superclass D
Precondition	for each $i : 0 \leq i \leq k$, $C_i :: m \in M$; m' is a fresh method name; for each $i : 1 \leq i \leq k$, $C_i \preceq C_0$; for each class $C \in N$ satisfying $C \preceq C_0$ or $C_0 \preceq C$, $C :: m \notin M$ unless $C \in \{C_0, C_1, \cdots, C_n\}$	for each class D such that $C \preceq D$ or $D \preceq C$, $D :: m \notin M$	$C :: m \in M$; $(C, \triangleright, D) \in E$; for each class $D' \succeq D$, $D' :: m \notin M$; and $C :: m$ only accesses attributes and methods declared in class D
Command Transformation	$R_c(c) = c[C_0.m'(s,t)/C_0.m(s,t)]$	$R_c(c) = c$	$R_c(c) = c$
Frame I	$\langle N, M \setminus \{C_0 :: m, C_1 :: m, \cdots, C_k :: m\}\rangle$	$\langle N, M\rangle$	$\langle N, M \setminus \{C :: m\}\rangle$

Fig. 13. Rules for changing methods

method $m(u : T_1;\ v : T_2)\{c\} \in M(C)$ satisfying $C :: m \in PM$, there is a corresponding method $m(u : T_1;\ v : T_2)\{\rho_c(c)\} \in M'(C)$, and that the condition given in Definition 11 holds. We prove this for each of **R9** and **R10.1–R10.3**.

- For **R9**, we define the transformation such that $\rho_o(\Sigma, \Sigma')$ if and only if Σ' is obtained from Σ by substituting each node of the form (r, D) with (r, C). With the command transformation $R_c(c) = c[C/D]$, we have $[\![x.m(s;\ t)]\!]_{\Gamma'} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma}$.
- For **R10.1**, we define ρ_o to be the identy transformation and thus $\rho_o(\Sigma, \Sigma')$ if and only if $\Sigma' = \Sigma$. Obviously, the semantics of methods can not be modified by consistent method renaming in a method declaration and method invocations, so $[\![x.m(s;\ t)]\!]_{\Gamma'} = [\![x.m(s;\ t)]\!]_{\Gamma}$. Thus $[\![x.m(s;\ t)]\!]_{\Gamma'} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma}$.
- For **R10.2** and **R10.3**, we define ρ_o as the identity transformation such that $\rho_o(\Sigma, \Sigma')$ if and only if $\Sigma' = \Sigma$. Notice that $R_c(c) = c$, so we have $[\![x.m(s;\ t)]\!]_{\Gamma'} \circ \rho_o = \rho_o \circ [\![x.m(s;\ t)]\!]_{\Gamma}$ since no method in the frame is modified.

□

7. Normal forms and completeness results

This section studies the completeness of the structure refinement rules. It is difficult to establish a set of rules that is *complete* in the sense that any refinement of Γ can be obtained by applying the rules in the set on Γ. However, to demonstrate the power of the refinement rules that we have given, we show that any class graph can be transformed into certain normal forms. We study two normal forms and show that the rules are powerful enough for transforming any class graph to a normal form.

7.1. Two normal forms

Definition 17 (Normal forms) Let $\Gamma = \langle N, A, E, M\rangle$ be a class graph with an interface $I = \langle PC, PM\rangle$. We say Γ is of **Normal Form I** if the following conditions hold.

1. Γ contains at least one class not in the interface I, that is a private class.
2. All classes of Γ belong to one inheritance tree, that is there exists a class C of Γ which is the root of the inheritance tree such that $D \preceq C$ for any class D of Γ.
3. For any inheritance relation $(D, \triangleright, C) \in E$, if C or D is not in the interface, this inheritance is *polymorphic* meaning that some methods of the superclass C are overridden in the subclass D: $(\{m \mid C :: m \in M\} \cap \{m \mid D :: m \in M\} \neq \emptyset)$.
4. Γ contains no redundant nodes, edges or methods, i.e. $N \setminus \mathcal{T} = Nodes(I), E = Edges(I), M = Meths(I)$.

We say that Γ is of **Normal Form II**, if Γ satisfies both the following conditions.

1. All class nodes of Γ are public, that is $N \setminus \mathcal{T} = PC$, and
2. Γ contains no redundant edges or methods: $E = Edges(I), M = Meths(I)$.

We now show that the graph compression rules are complete in that they can transform each class graph to an equivalent graph of either of the normal forms.

Theorem 6 (Completeness result II) For each class graph $\Gamma = \langle N, A, E, M\rangle$ with interface $I = \langle PC, PM\rangle$, there exists a class graph Γ' with interface I such that

1. Γ' is of either Normal Form I or Normal Form II,
2. $\Gamma' \equiv_I \Gamma$, and
3. Γ can be transformed to Γ' by applications of rules **R2, R8, R9** and **R10**.

Proof. For an arbitrary class graph Γ, we can have the following transformations.

1. First use **R8** to eliminate inaccessible nodes, edges and methods.
2. For each pair of classes, if the precondition of **R9** holds, then apply **R9** to combine them into one class. If it is needed, use **R2, R10.1** to rename their attributes and methods to avoid name conflicts before the combination.
3. Repeat Step 2 until each pair of classes could not be further merged even with possible renaming to avoid name conflicts. Then we have a class graph Γ'.

If Γ' is not Normal Form II, we need to prove Γ' is in Normal Form I as the other two conditions of the theorem obviously hold. For this we need to prove that Γ' satisfies the four conditions of Normal Form I.

1. There is at least one private class in Γ'. Otherwise, Γ' is of Normal Form II.
2. To prove the condition that all classes of Γ' belong to one inheritance tree, assume Γ' contains at least two inheritance trees. Let D be a private class in Γ', then there exists at least one inheritance tree that does not contain class D. Let class C be the root of such an inheritance tree. We can use rules **R2, R10.1** to rename attributes and methods of class D (if needed) and then **R9.3** to merge it into class C, since D and C are not related by inheritance relation. This however contradicts with the fact that no pair of classes in Γ' could be further merged.
3. For the condition that inheritance relation in Γ' is polymorphic unless it is associated with two public classes, assume $(C, \triangleright, D)$ is an inheritance relation in Γ' and either class C or class D is a private class. If C is private, then we can merge C into its direct superclass D by Rule **R9.1**. Similarly, if D is private, we could merge D into its direct subclass C by Rule **R9.2**. This also contradicts with the condition that no pair of classes in Γ' could be further merged.
4. Finally, the condition that Γ' contains no redundant nodes, edges or methods holds, since Γ' is obtained after applying Rule **R8** to remove all those elements inaccessible from the interface.

□

This completeness result shows that the essential difference between object-oriented programming and "procedural" programming is the mechanisms of polymorphism and dynamic procedure call binding in object-oriented programs. We use the following example to show how a class graph can be transformed into a graph of Normal Form I.

Example 4 Let Γ be the class graph in Fig. 14, with interface $I = \langle\{P_1, P_2, P_3\}, \{P_1 :: get, P_2 :: set\}\rangle$, where the methods set are

$P_1 :: get(;\ Int\ v)\{v := a\}$ $\quad C :: get(;\ Int\ v)\{var\ D\ o;\ D.new(o);\ o.set(1;\ );\ v := a + o.d;\ end\ o\}$
$D :: set(Int\ u;\ )\{d := u\}$ $\quad P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$K :: get(;\ Int\ v)\{v := k\}$ $\quad G :: get(;\ Int\ v)\{v := g\}$
$P_3 :: get(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ C.new(o);\ o.get(;\ v);\ end\ o)\}$

Γ is transformed to a graph of **Normal Form I** by following steps, that are depicted in Fig. 15.

1. *Remove redundant nodes and edges:* It is easy to verify that $(H, h, Bool) \notin Edges(I)$ and $H \notin Nodes(I)$, we thus remove them by Rule **R8.2** and **R8.1**, respectively. The other nodes, edges and methods are accessible from the interface thus could not be further deleted.

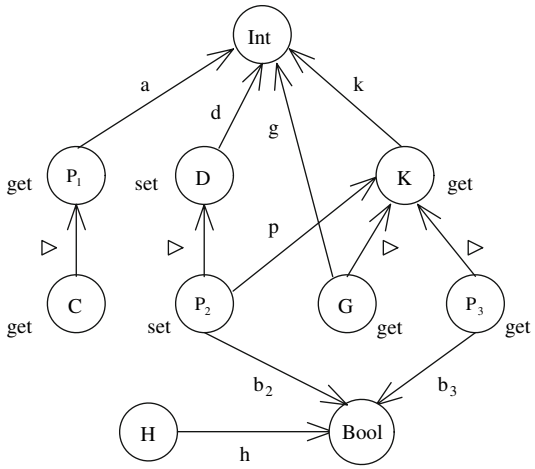


Fig. 14. A class graph Γ

2. *Merge classes:* First we merge class D into class P_1, since D is a private class $D \notin PC$. The set of methods become

$P_1 :: get(;\ Int\ v)\{v := a\}$ $\quad$ $P_1 :: set(Int\ u;\ )\{d := u\}$
$C :: get(;\ Int\ v)\{var\ P_1\ o;\ P_1.new(o);\ o.set(1;\ );\ v := a + o.d;\ end\ o\}$
$P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$K :: get(;\ Int\ v)\{v := k\}$ $\quad$ $G :: get(;\ Int\ v)\{v := g\}$
$P_3 :: get(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ C.new(o);\ o.get(;\ v);\ end\ o)\}$

Use Rule **R10.1** to rename methods $K :: get$, $G :: get$ and $P_3 :: get$ to $K :: get'$, $G :: get'$ and $P_3 :: get'$. After that, we merge class G, which is also a private class, into P_1. We now have a graph Γ' that is of Normal Form I. The methods of final graph are

$K :: get'(;\ Int\ v)\{v := k\}$
$P_1 :: get(;\ Int\ v)\{v := a\}$ $\quad$ $P_1 :: set(Int\ u;\ )\{d := u\}$ $\quad$ $P_1 :: get'(;\ Int\ v)\{v := g\}\}$
$C :: get(;\ Int\ v)\{var\ P_1\ o;\ P_1.new(o);\ o.set(1;\ );\ v := a + o.d;\ end\ o\}$
$P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get'(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$P_3 :: get'(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ C.new(o);\ o.get(;\ v);\ end\ o)\}$

7.2. Eliminating polymorphism

With the rules of structure refinement given so far, we cannot eliminate all the private classes from any graph since these classes may contain polymorphic methods which are not allowed to be merged or removed by any refinement rule. As for this, a normal form may consist of either only public classes or an inheritance tree with at least one private class.

We now consider the elimination of polymorphic inheritance relations such that every private class in a class graph could be merged into public classes. For this purpose, we prove a pair of rules in Fig. 16 to combine classes with overriding methods. The soundness of each rule is easy, and it is clear that each rule is an equivalence rule.

Theorem 7 (Completeness result III) For each class graph $\Gamma = \langle N, A, E, M\rangle$ with interface $I = \langle PC, PM\rangle$, there exists a class graph Γ' with interface I such that

1. Γ' is in Normal Form II,
2. $\Gamma' \equiv_I \Gamma$, and
3. Γ could be transformed to Γ' by sequential applications of rules **R2, R8, R9, R10** and **R11**.

Proof. From Theorem 6, we can apply rules **R2, R8, R9** and **R10** to transform Γ to a class graph Γ'' that is of either Normal Form I or Normal Form II. If Γ'' is not of Normal Form II, then each private class D in Γ'' is associated with another class C by a polymorphic inheritance relation. We could further merge D into C by Rule **R11**. Repeat

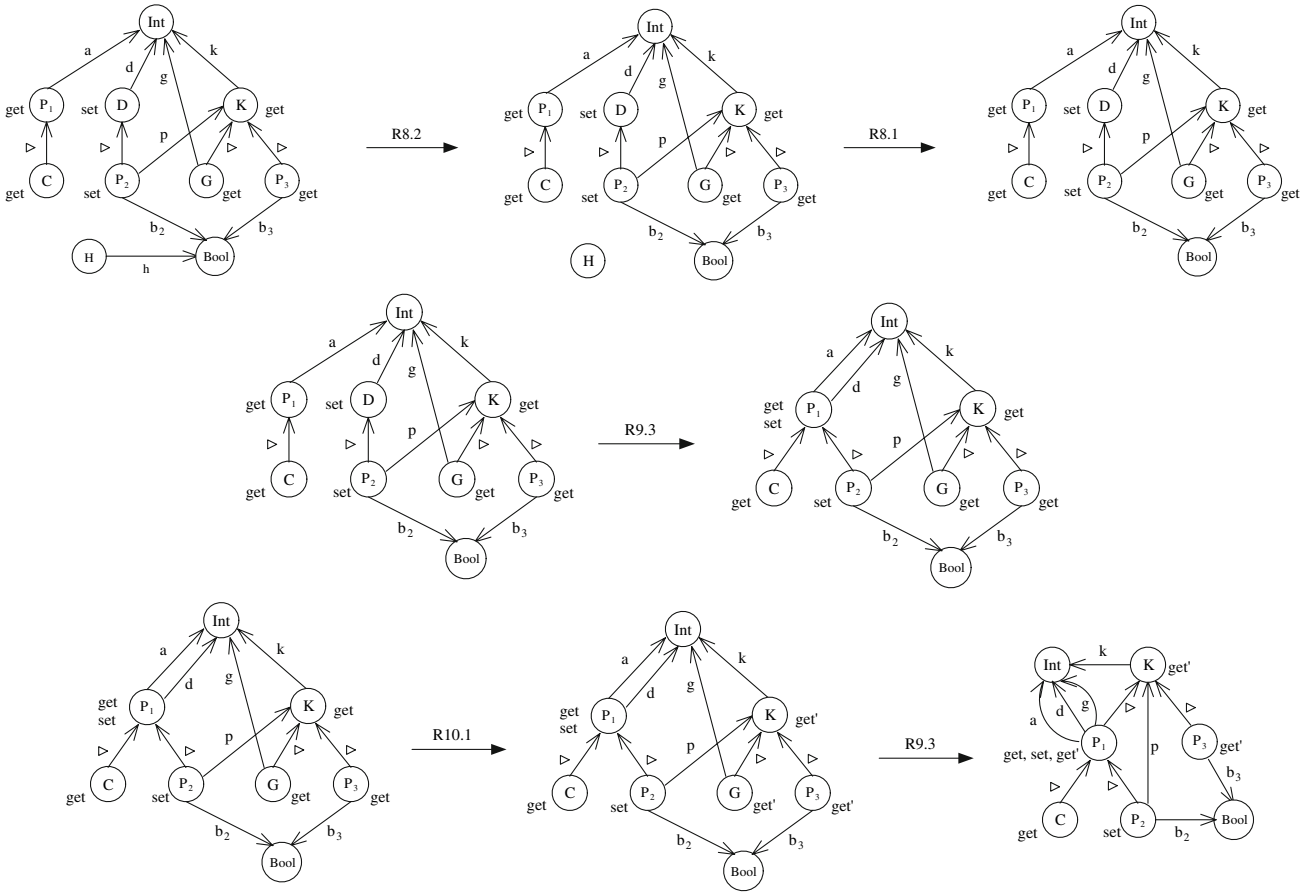


Fig. 15. Transform to Normal Form I

Rule R	**R11.1** Eliminate Polymorphism - **merge to superclass**	**R11.2** Eliminate Polymorphism - **merge to subclass**
Description	merge class D to its direct superclass C with polymorphic methods $m_0, \cdots, m_k$, where $k \geq 0$. A new boolean attribute b is added to class C and each pair of polymorphic method is combined to one whose method body is a conditional choice, according to the postcondition. Edges and other methods associated with class D simply turn to those associated with class C	merge class D to its direct subclass C with polymorphic methods $m_0, \cdots, m_k$, where $k \geq 0$. A new boolean attribute b is added to class C and each pair of polymorphic method is combined to one whose method body is a conditional choice, according to the postcondition. Edges and other methods associated with class D simply turn to those associated with class C
Precondition	$C, D \in N$; $(D, \triangleright, C) \in E$; for each $C' \preceq C$ but $C' \neq D$, $attr(C') \cap attr(D) = \emptyset$; $\{m \mid C :: m \in M \wedge D :: m \in M\} = \{m_0, \cdots, m_k\}$; $m_0(u_0 : S_0; v_0 : T_0)\{c_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c_k\} \in M(C)$; $m_0(u_0 : S_0; v_0 : T_0)\{c'_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c'_k\} \in M(D)$	$C, D \in N$; $(C, \triangleright, D) \in E$; for each $D' \preceq D$ but $D' \neq C$, $attr(D') \cap attr(C) = \emptyset$; $\{m \mid C :: m \in M \wedge D :: m \in M\} = \{m_0, \cdots, m_k\}$; $m_0(u_0 : S_0; v_0 : T_0)\{c_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c_k\} \in M(C)$; $m_0(u_0 : S_0; v_0 : T_0)\{c'_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c'_k\} \in M(D)$
Postcondition	$(C, b, Bool) \in E'$, where b is a fresh name; $m_0(u_0 : S_0; v_0 : T_0)\{c_0 \lhd b \rhd c'_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c_k \lhd b \rhd c'_k\} \in M'(C)$	$(C, b, Bool) \in E'$, where b is a fresh name; $m_0(u_0 : S_0; v_0 : T_0)\{c_0 \lhd b \rhd c'_0\}, \cdots,$ $m_k(u_k : S_k; v_k : T_k)\{c_k \lhd b \rhd c'_k\} \in M'(C)$
Command Transformation	$R_c(c) = c[C.new(le); le.b := true / C.new(le)]$ $[D.new(le); le.b := false / D.new(le)]$ $[C/D]$	$R_c(c) = c[C.new(le); le.b := true / C.new(le)]$ $[D.new(le); le.b := false / D.new(le)]$ $[C/D]$
Frame	$I = \langle N \setminus \{D\}, \{G :: m \in M \mid G \neq D\}\rangle$	$I = \langle N \setminus \{D\}, \{G :: m \in M \mid G \neq D\}\rangle$

Fig. 16. Rules for eliminating polymorphism

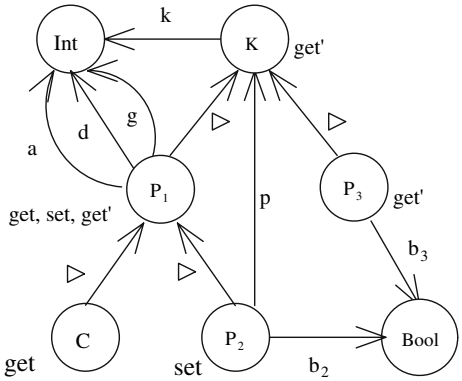


Fig. 17. Γ' in Normal Form I

this step and we can get a class graph Γ' without private classes, thus it is of Normal Form II. Obviously, Γ' also satisfies both the second and the third conditions in this theorem. □

This completeness result means that any object-orient program can be transformed to a "procedural" program with data sharing via references. We use the following example to show how a class graph can be transformed into a Normal Form II.

Example 5 Let Γ be the class graph in Fig. 14, with interface $I = \langle\{P_1, P_2, P_3\}, \{P_1 :: get, P_2 :: set\}\rangle$. In Example 4, we transformed Γ to Γ' in Normal Form I by applications of rules **R8, R9** and **R10**. From Fig. 17, the methods of Γ' are

$K :: get'(;\ Int\ v)\{v := k\}$
$P_1 :: get(;\ Int\ v)\{v := a\}$ $P_1 :: set(Int\ u;\ )\{d := u\}$ $P_1 :: get'(;\ Int\ v)\{v := g\}$
$C :: get(;\ Int\ v)\{var\ P_1\ o;\ P_1.new(o);\ o.set(1;\ );\ v := a + o.d;\ end\ o\}$
$P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get'(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$P_3 :: get'(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ C.new(o);\ o.get(;\ v);\ end\ o)\}$

Γ' could be further transformed to a normal form II by the following steps, depicted in Fig. 18.

1. *Eliminate polymorphism by merging private class C into* P_1. A new boolean attribute b is declared in class P_1. The methods become

$K :: get'(;\ Int\ v)\{v := k\}$
$P_1 :: get(;\ Int\ v)\{v := a \lhd b \rhd (var\ P_1\ o;\ P_1.new(o);\ o.b := true;\ o.set(1;\ );\ v := a + o.d;\ end\ o)\}$
$P_1 :: set(Int\ u;\ )\{d := u\}$ $P_1 :: get'(;\ Int\ v)\{v := g\}$
$P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get'(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$P_3 :: get'(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ P_1.new(o);\ o.b := false;\ o.get(;\ v);\ end\ o)\}$

2. *Eliminate polymorphism by merging private class K into* P_1. A new boolean attribute b' is declared in class P_1. Thus we get a class graph Γ'' in Normal Form II containing no private classes. The methods of Γ'' are

$P_1 :: get(;\ Int\ v)\{v := a \lhd b \rhd (var\ P_1\ o;\ P_1.new(o);\ o.b := true;\ o.set(1;\ );\ v := a + o.d;\ end\ o)\}$
$P_1 :: set(Int\ u;\ )\{d := u\}$ $P_1 :: get'(;\ Int\ v)\{v := g \lhd b' \rhd v := k\}$
$P_2 :: set(Int\ u;\ )\{(var\ Int\ x;\ p.get'(;\ x);\ d := x;\ end\ x) \lhd b_2 \rhd d := u\}$
$P_3 :: get'(;\ Int\ v)\{v := k \lhd b_3 \rhd (var\ P_1\ o;\ P_1.new(o);\ o.b := false;\ o.get(;\ v);\ end\ o)\}$

Corollary 3 Let Γ be a class graph, *Main* the main class with the main variables X and $main()\{c\}$ the main method. We use $\Gamma \bullet Main$ to denote the class graph with interface $I = \langle\{Main\}, \{Main::main\}\rangle$. Then $\Gamma \bullet Main$ can be transformed to a single class with name *Main*, with $main()\{c\}$ as the only public (i.e. observable) method.

This corollary implies that object-oriented programming and procedural programming have equal expressive power in regards to computability. The point here is not to show this equivalence, but to show the power of the structure refinement rules.

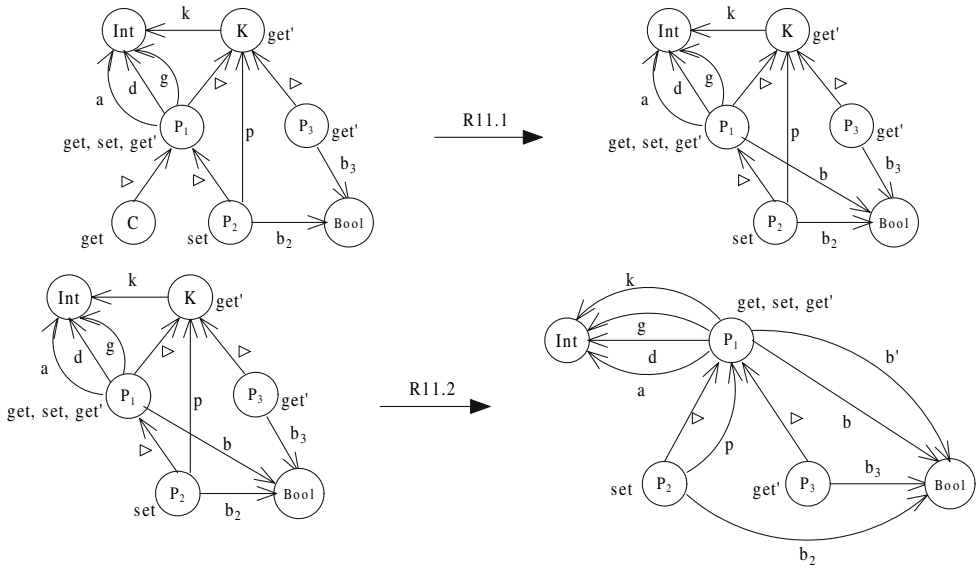


Fig. 18. Transform to Normal Form II

8. Conclusions

We have proposed a graph theoretical approach to studying the relation between transformations 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 of object-oriented system development by model transformations [Ser].

This paper has substantially extended the previous workshop version [LLZ06] by providing refinement rules for removing classes, attributes and inheritance, and rules for compressing long paths to shorter paths, combining classes. Hence not only "true" refinements are treated, but also "abstractions" that preserve functionality. We also provide the notion of interface which represents the services a class graph provides to outside. Based on this concept, a refinement is defined with respect to a certain interface, illustrating not only what classes but also what methods are public ones. A crucial progress made in this version is the introduction to the two normal forms and completeness results.

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 way as we understand 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 on graphs.

Acknowledgments

We would like to thank the three anonymous referees for their spirited and detailed comments to bring this paper to its present form, and the editors of this issue for their effort in ensuring this paper to come out as its right form. We wish to thank Zhenbang Chen, Volker Stolz and Naijun Zhan for their careful reading and comments that helped us to remove some errors and improve the presentation. We also owe thanks to our colleagues Xin Chen, Zhenbang Chen, He Jifeng, Wei Ke, Xiaoshan Li, Joseph Okika, Anders P. Ravn, Volker Stolz, Lu Yang and Naijun Zhan for the collaboration in the research on rCOS.

References

[AC96] Abadi M, Cardeli L (1996) A theory of objects. Springer, Heidelberg
[AdB94] America P, de Boer F (1994) Reasoning about dynamically evolving process structures. Form Aspects Comput 6(3):269–316

[AL97] Abadi M, Leino R (1997) A logic of object-oriented programs. In: Bidoit M, Dauchet M (eds) TAPSOFT '97: theory and practice of software development, 7th international joint conference. Springer, Heidelberg, pp 682–696
[BHTV03] Baresi L, Heckel R, Thöne S, Varró D (2003) Modeling and validation of service-oriented architectures: Application vs. style. In: Proceedings of ESEC/FSE 03 European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering. ACM Press, pp 68–77
[BHTV04] Baresi L, Heckel R, Thöne S, Varró D (2004) Style-based refinement of dynamic software architectures. In: WICSA '04: Proceedings of the fourth working IEEE/IFIP conference on software architecture (WICSA'04). IEEE Computer Society, Washington DC, pp 155–164
[BMvW00] Back R, Mikhajlova A, von Wright J (2000) Class refinement as semantics of correct object substitutability. Form Aspects Comput 2:18–40
[BRJ99] Booch G, Rumbaugh J, Jacobson I (1999) The unified modelling language user guide. Addison-Wesley, Reading
[BSC03] Borba P, Sampaio A, Cornelio M (2003) A refinement algebra for object-oriented programming. In: Cardelli L (ed) Proceedings of ECOOP 2003, Lecture Notes in Computer Science 2743. Springer, Heidelberg, pp 457–482
[BvW98] Back R, von Wright LJ (1998) Refinement calculus. Springer, Heidelberg
[CD94] Cook S, Daniels J (1994) Designing object systems: object-oriented modelling with syntropy. Prentice-Hall, Englewood Cliffs
[CHH+07] Chen Z, Hannousse AH, Van Hung D, Knoll I, Li X, Liu Y, Liu Z, Nan Q, Okika J, Ravn AP, Stolz V, Yang L, Zhan N (2007) Modelling with relational calculus of object and component systems-rCOS. In: The common component modeling example: comparing software component models. Springer, Heidelberg (To be published as a Chapter of a Volume of Lecture Notes in Computer Science)
[CJ06] Chen Y, Sanders J (2006) Compositional reasoning for pointer structures. In: Proceedings of 8th international conference on mathematics of program construction (MPC06), Lecture Notes in Computer Science, vol 4014. Springer, Heidelberg, pp 115–139
[CMR96] Corradini A, Montanari U, Rossi F (1996) Graph processes. Fundam Inf 26(3,4):241–265
[CN99] Cavalcanti A, Naumann D (1999) A weakest precondition semantics for an object-oriented language of refinement. In: World congress on formal methods (2), Lecture Notes in Computer Science, vol 1709. Springer, Heidelberg, pp 1439–1460
[Col94] Coleman D et al (1994) Object-oriented development: the FUSION method. Prentice-Hall, Englewood Cliffs
[DD93] Dürr E, Dusink EM (1993) The role of VDM^{++} in the development of a real-time tracking and tracing system. In: Woodcock J, Larsen P (eds) Proceedings of FME'93, Lecture Notes in Computer Science, vol 670. Springer, Heidelberg
[Ed97] Rozenberg G (ed) (1997) Handbook of graph grammars and computing by graph transformation, vol 1, Foundations World Scientific
[EEPT06] Ehrig H, Ehrig K, Prange U, Taentzer G (2006) Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. Fundam Inf 74(1):31–61
[EHHS00] Engels G, Hausmann JH, Heckel R, Sauer S (2000) Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in uml. In: Proceedings of UML 2000, The unified modeling language, Lecture Notes in Computer SCience, vol 1939. Springer, Heidelberg, pp 323–337
[GMB04] Gheyi R, Massoni T, Borba P (2004) An abstract equivalence notation for object models. Electron Note Theor Comput Sci 130:3–21
[GRPPS98] Große-Rhode M, Parisi-Presicce F, Simeoni M (1998) Spatial and temporal refinement of typed graph transformation systems. In: Proceedings of mathematical foundations of computer science, Lecture Notes in Computer Science, vol 1450. Springer, Heidelberg, pp 553–561
[HH98] Hoare CAR, He J (1998) Unifying theories of programming. Prentice-Hall, Englewood Cliffs
[HH99] Hoare CAR, He J (1999) A trace model for pointers and objects. In: Proceedings of ECOOP'99, Lecture Notes in Computer Science, vol 1628. Springer, Heidelberg, pp 1–17
[HHS86] He J, Hoare CAR, Sanders JW (1986) Data refinement refined. In: Proceedings of ESOP 86, Lecture Notes in Computer Science, vol 213. Springer, Heidelberg, pp 187–196
[HLL06] He J, Li X, Liu Z (2006) rCOS: A refinement calculus for object-oriented systems. Theor Comput Sci 365(1–2):109–142
[JO04] Johnsen EB, Owe O (2004) Object-oriented specification and open distributed systems. In: From object-orientation to formal method, Lecture Notes in Computer Science, vol 2635. Springer, Heidelberg, pp 137–164
[KKR06] Kastenberg H, Kleppe A, Rensink A (2006) Defining object-oriented execution semantics using graph transformations. In: Proceedings of the 8th IFIP international conference on formal methods for open object-based distributed systems (FMOODS'06), Lecture Notes in Computer Science, vol 4037. Springer, Heidelberg, pp 186–201
[Kru00] Kruchten P (2000) The rational unified process: an introduction. Addison-Wesly, Reading
[Lar01] Larman C (2001) Applying UML and patterns. Prentice-Hall International, Englewood Cliffs
[Lei98] Rustan K, Leino M (1998) Recursive object types in a logic of object-oriented programming. Nordic J Comput 5(4):330–360
[LHLC03] Liu Z, He J, Li X, Chen Y (2003) A relational model for formal requirements analysis in UML. In: Proceedings of ICFEM03, Lecture Notes in Computer Science, vol 2885. Springer, Heidelberg, pp 641–664
[LLZ06] Liu X, Liu Z, Zhao L (2006) Object-oriented structure refinement—a graph transformational approach. Electron Notes Theor Comput Sci 187:145–159
[Mor94] Morgan CC (1994) Programming from specifications. Prentice Hall, Englewood Cliffs
[MS97] Mikhajlova A, Sekerinski E (1997) Class refinement and interface refinement in object-oriented programs. In: Proceedings of FME'97, Lecture Notes in Computer Science, vol 1313. Springer, Heidelberg, pp 82–101
[Nau94] Naumann D (1994) Predicate transformer semantics of an Oberon-like language. In: Olderog E-R (ed) Proceedings of PROCOMET'94. North-Holland, Amsterdam, pp 467–487
[PHM99] Poetzsch-Heffter A, Muller P (1999) A programming logic for sequential Java. In: Swierstra SD (ed) Proceedings of programming languages and systems (ESOP'99), Lecture Notes in Computer Science, vol 1576. Springer, Heidelberg, pp 162–176
[Sek96] Sekerinski E (1996) A type-theoretical basis for an object-oriented refinement calculus. In: Kent SJH (ed) Formal methods and object technology. Springer, Heidelberg
[Ser] Tata Consultancy Services. Mastercraft. http://www.tata-mastercraft.com/

[Smi00] Smith G (2000) The object-Z specification language. Kluwer, Dordrecht
[TR05] Taentzer G, Rensink A (2005) Ensuring structural constraints in graph-based models with type inheritance. In: Cerioli M (ed) Fundamental approaches to software engineering (FASE), Edinburgh, UK, Lecture Notes in Computer Science, vol 3442. Springer, Heidelberg, pp 64–79
[WF02] Wermelinger M, Fiadero JL (2002) A graph transformation approach to software architecture reconfiguration. Sci Comput Program 44(2):133–155
[ZZLQ06] Zhao L, Zhao X, Long Q, Qiu Z (2006) A type system for the relational calculus of object systems. In: Proceedings of international conference on engineering complex computer systems. IEEE Computer Soceity, pp 189–198

Received 14 January 2007
Accepted in revised form 20 December 2007 by B. K. Aichernig, E. A. Boiten, M. J. Butler, J. Derrick and L. Groves
Published online 8 January 2008