Graphs and Graph Transformations for Object-Oriented and Service-Oriented Systems

Liang Zhao

Supervisor
Zhiming Liu

Supervisor
Roberto Bruni

May 25, 2012
Abstract

Theories of graphs and graph transformations form an important part of the mathematical foundations of computing, and have been applied in a wide range of areas from the design and analysis of algorithms to the formalization of various computer systems and programs. In this thesis, we study how graphs and graph transformations can be used to model the static structure and dynamic behavior of object-orientated and service-oriented systems.

Our work is mainly motivated by the difficulty in understanding and reasoning about object-orientated and service-oriented programs, which have more sophisticated features compared with traditional procedural programs. We show that the use of graphs and graph transformations provides both an intuitive visualization and a formal representation of object-orientated and service-oriented programs with these features, improving people's understanding of the execution states and behaviors of these programs.

We provide a graph-based type system, operational semantics and refinement calculus for an object-oriented language. In this framework, we define class structures and execution states of oo programs as directed and labeled graphs, called class graphs and state graphs, respectively. The type system checks whether a program is well-typed based on its class graph, while the operational semantics defines each step of program execution as a simple graph transformations between state graphs. We show the operational semantics is type-safe in that the execution of a well-typed program does not “go wrong”. Based on the operational semantics, we study the notion of structure refinement of oo programs as graph transformations between their class graphs. We provide a few groups of refinement rules for various purposes such as class expansion and polymorphism elimination and prove their soundness and relative completeness.

We also propose a graph-based representation of service-oriented systems specified in a service-oriented process calculus. In this framework, we define states of service-oriented systems as hierarchical graphs that naturally capture the hierarchical nature of service structures. For this, we exploit a suitable graph algebra and set up a hierarchical graph model, in which graph transformations are studied following the well-known Double-Pushout approach. Based on this model, we provide a graph transformation system with a few sets of graph transformation rules for various purposes such as process copy and process reduction. We prove that the graph transformation system is sound and complete with respect to the reduction semantics of the calculus.
I would like to express my most grateful thanks to my supervisor, Zhiming Liu, for his introduction of the interesting research topics, for his cooperation and guidance in my research and for his help and suggestions in revising this thesis. In the past four years, I learned a lot from him, not only in academic knowledge like how to solve a research problem or write a research paper, but also in attitudes towards work and life such as how to be confident and positive when communicating with people or facing problems. Without his constant help and encouragement, this thesis would not have come into reality.

I am grateful to Roberto Bruni, who is my supervisor in University of Pisa. It was he who brought me into the area of service-orientation and introduced to me the interesting research topic on CaSPiS. During our cooperation on this topic, we managed to overcome many dedicated problems and arrive at satisfactory results. I am also grateful to Zongyan Qiu, who was my supervisor during my MSc study in Peking University, for his guidance and cooperation in the development of the refinement calculus.

I would like to thank my parents, Tiewu Zhao and Baohua Liu, for their eternal love, for their constant support in my daily life and for their encouragement when I felt down. It is hard to express in words how much I owe thanks to them.

My sincere thanks also go to my colleagues Cristiano Bertolini, Zhenbang Chen, Ruzhen Dong, Wei Ke, Dan Li, Xiaojian Liu, Quan Long, Charles Morisset, Martin Schaf, Volker Stolz, Shuling Wang, Xiangpeng Zhao, and Jiaqi Zhu for their cooperation, discussion and friendly advice that are of great help in my work.

Special thanks to UNU-IIST and University of Pisa for their funding support throughout my PhD study. I would also like to thank Antonio Cerone, Kitty Chan, Pui Chan, Coffee Das Dores, Pierpaolo Degano, Ilaria Fierro, Peter Haddawy, Michelle Ho, Wendy Hoi, Sandy Lee, Sao Chi Lei, and Alice Pun for their administrative support these years.
## Contents

1 Introduction .................................................. 13
   1.1 Motivation .................................................. 14
   1.2 Contribution ................................................. 16
   1.3 Related Work ................................................ 19
   1.4 Structure of the Thesis and Origin of the Work .......... 22

2 Graph-Based Operational Semantics of oo Programs ........ 25
   2.1 Background: the Object-Oriented Language of rCOS .... 26
   2.2 Class Graphs, Object Graphs and State Graphs .......... 28
      2.2.1 Class graphs .......................................... 28
      2.2.2 Object graphs .......................................... 30
      2.2.3 State graphs ........................................... 31
      2.2.4 Correctly typed object graphs and state graphs ... 32
      2.2.5 Graph operations ...................................... 33
   2.3 Type System .................................................. 35
      2.3.1 Type checking of expressions ........................ 36
      2.3.2 Type checking of commands ............................ 36
      2.3.3 Type checking of class graphs and programs ....... 37
   2.4 Operational Semantics ........................................ 38
      2.4.1 Evaluation of expressions .............................. 38
      2.4.2 Type safety of expressions ............................ 39
      2.4.3 Execution of commands .................................. 40
      2.4.4 Type safety of commands ............................... 42
      2.4.5 Semantics and type safety of programs ............... 46
   2.5 Properties of OO Programs ................................... 46
   2.6 Summary ....................................................... 48

3 Graph-Based Structure Refinement of oo Programs .......... 49
   3.1 Interface and Structure Refinement ......................... 50
      3.1.1 A sufficient condition of structure refinement ... 52
   3.2 Structure Refinement for Graph Expansion ................. 54
      3.2.1 Refinement rules for structure expansion .......... 55
      3.2.2 Soundness and completeness of basic refinement rules ... 56
   3.3 Structure Refinement for Graph Compression ............... 59
      3.3.1 Removing redundant structure elements .............. 59
      3.3.2 Rules for removing redundant elements .............. 62
      3.3.3 Rules for combining classes .......................... 63
   3.4 Completeness of Refinement Rules .......................... 65
      3.4.1 Structure refinement for changing methods ........... 66
      3.4.2 Eliminating polymorphism ............................. 70
   3.5 Discussion: Practical Importance of Refinement Rules .... 73
   3.6 Summary ....................................................... 75
# CHAPTER 0. CONTENTS

4 Graph Representation of Service-Oriented Systems

4.1 Background: the Calculus CaSPiS
   4.1.1 Basic processes
   4.1.2 Operational semantics in terms of reduction
   4.1.3 Extension with replications

4.2 Background: Syntax of Hierarchical Graphs

4.3 Model of Hierarchical Graphs
   4.3.1 Graph transformation rules

4.4 Graph Representation of CaSPiS
   4.4.1 Processes as designs
   4.4.2 Tagged graph and tagging rules
   4.4.3 Rules for congruence
   4.4.4 Copy rules
   4.4.5 Rules for reduction
   4.4.6 Garbage collection rules
   4.4.7 Data assignment rules
   4.4.8 Soundness and completeness
   4.4.9 An example

4.5 Summary

5 Conclusions

Bibliography

A Proof of Soundness and Completeness of DPO Rules

A.1 Auxiliary Processes
A.2 Proof of Theorems 4.4.2 and 4.4.3 of Soundness
A.3 Proof of Theorems 4.4.4 and 4.4.5 of Completeness
   A.3.1 Variants of congruence and reduction
   A.3.2 Proof of Proposition A.3.1
   A.3.3 Proof of Proposition A.3.2
   A.3.4 Proof of Proposition A.3.3
   A.3.5 Proof of Proposition A.3.4

Glossary of Notations

Basic symbols

- $a, x, y, z, \ldots$: variable or class attribute
- $b, \ldots$: boolean expression or boolean attribute
- $c, \ldots$: command
- $cn, \ldots$: constant literal
- $con, \ldots$: configuration
- $d, \ldots$: edge
- $e, ve, \ldots$: expression (including assignable expression)
- $f, g, \ldots$: constructor
- $h, \ldots$: built-in operation of primitive data type
- $i, o, p, t, v, w, \ldots$: node
- $j, k, \ldots$: index or integer
- $l, L, \ldots$: label
- $le, re, \ldots$: assignable expression
- $m, \ldots$: class method
- $n, \ldots$: variable, session or service
- $q, \ldots$: predicate
- $r, \ldots$: session
- $s, \ldots$: service
- $u, \ldots$: (general) element
- $B, \ldots$: primitive data type
- $C, D, J, K, O, \ldots$: class
- $E, \ldots$: set of edge
- $F, \ldots$: pattern
- $G, H, \ldots$: graph
- $I, \ldots$: interface of class graph
- $M, \ldots$: set of methods
- $N, \ldots$: set of nodes
- $P, Q, \ldots$: process
- $R, \ldots$: rule
- $S, T, \ldots$: type (class or primitive data type)
- $Set, \ldots$: set of elements
- $U, \ldots$: sum
- $V, \ldots$: value
- $X, \ldots$: process variable
- $Z, \ldots$: design
Basic symbols (continued)

\[ \alpha, \ldots \] path
\[ \delta, \ldots \] set of rules
\[ \pi, \ldots \] prefix
\[ \rho, \ldots \] relation (including mapping)
\[ \sigma, \ldots \] substitution
\[ \Gamma, \ldots \] class graph
\[ \Delta, \ldots \] type context
\[ \Lambda, \ldots \] process context

Name Spaces

\[ B \] name space of primitive data types
\[ C \] name space of classes
\[ D \] name space of design labels
\[ F \] name space of design labels of flat designs
\[ G \] name space of constructors
\[ K \] name space of literals
\[ L \] name space of edge labels
\[ M \] name space of methods
\[ N \] name space of nodes
\[ O \] name space of node types
\[ R \] name space of sessions
\[ S \] name space of services
\[ T \] name space of types
\[ V \] name space of variables and class attributes

Mathematical symbols

\[ \emptyset \] empty set
\[ \epsilon \] empty sequence
\[ [u'/u] \] substitution of \( u \) by \( u' \)
\[ \vec{u} \] sequence of elements \( u_1, u_2, \ldots \)
\[ \vec{u}[j] \] \( j \)-th element of a sequence \( \vec{u} \)
\[ |\vec{u}| \] length of a sequence \( \vec{u} \)
\[ \{\vec{u}\} \] set of elements of a sequence \( \vec{u} \)
\[ \vec{u} \cdot \vec{u}' \] concatenation of sequences \( \vec{u} \) and \( \vec{u}' \)
\[ \#Set \] size (number of elements) of a set \( Set \)
\[ \text{dom}(\rho) \] domain of a relation \( \rho \) (which can be a partial or total function)
Chapter 2

$C \triangleright_{\Gamma} D$  
$S \preceq_{\Gamma} T$  
$S \triangleright_{\Gamma} T$

attr$_{\Gamma}(C)$  
Attr$_{\Gamma}(C)$  
dtype$_{\Gamma}(C, a)$  
dtype$_{\Gamma}(e)$  
init$_{\Gamma}(C, a)$

swing($G, \alpha, v$)  
new($G, C, \alpha$)  
push($G, \vec{x}, \vec{v}$)  
pop($G$)

Trace$_{\Gamma}(e : T)$  
$\Delta \vdash_{\Gamma} c : c$ with $k$ the minimum size of type contexts during the checking

trace($G, e$)  
eval($G, e$)  
rtype($G, e$)  
$c \rightarrow_{\Gamma} \text{con}$  
$c \rightarrow^{*}_{\Gamma} \text{con}$

$e_1 \approx e_2$  
$G_1 * G_2$

Chapter 3

$S_I(\Gamma)$  
MS$_I(\Gamma)$  
$\Gamma_1 \trianglelefteq_{I} \Gamma_2$  
$\Gamma_1 \equiv_{I} \Gamma_2$  
nodes($u$)  
edges($u$)  
meths($u$)  
Nodes($u$)  
Edges($u$)  
Meths($u$)
Chapter 4

\textbf{bn}(F) \quad \text{set of bound names of a pattern } F

\textbf{fn}(V) \quad \text{set of free names of a value } V

\textbf{fn}(P) \quad \text{set of free names of a process } P

P \equiv_c Q \quad \text{a process } P \text{ is (structural) congruent with another process } Q

P \rightarrow Q \quad \text{a process } P \text{ reduces to another process } Q

\textbf{match}(F;V) \quad \text{substitution that associates a pattern } F \text{ with a value } V

T(v) \quad \text{type of a node } v

T(l) \quad \text{type of an edge label } l

T(L) \quad \text{type of a design label } L

\text{AR}(l) \quad \text{arity of an edge label } l

\text{AR}(L) \quad \text{arity of a design label } L

\text{fn}(G) \quad \text{set of free nodes of } G

\mathcal{H}(G) \quad \text{underlying hypergraph of } G

G \Rightarrow_R G' \quad G' \text{ is a direct derivation of } G \text{ by a rule } R

G \Rightarrow^*_\delta G' \quad G' \text{ is a derivation of } G \text{ by rules } \delta

\llbracket F \rrbracket \quad \text{graph representation of a pattern } F

\llbracket V \rrbracket \quad \text{graph representation of a value } V

\llbracket P \rrbracket \quad \text{graph representation of a process } P

\llbracket P \rrbracket^\dagger \quad \text{tagged graph representation of a process } P
Chapter 1

Introduction

Graphs are among the most general models of various structures and relations, and have been applied in a large variety of domains involving computer science, physics, biology and sociology. In a traditional graph theory, a graph is defined as a simple mathematical structure that consists of nodes and edges between nodes. Such a structure on the one hand provides a good means of abstraction so that nodes represent objects in an application domain and edges the relations among these objects. On the other hand, a graph model is intuitive and visual and thus helps in the understanding of an application domain.

A graph supports different levels of details of the application domain with different annotations. The common annotations include directions of the edges, labels with different symbols and decorations such as colors on the nodes and edges. Therefore, there are theories of directed graphs, labeled graphs and colored graphs. In addition, a graph theory is often studied as a category [39] so that the algebraic technique of morphisms and isomorphisms can be used for studying important orders and equivalence relations among graphs. These orders and equivalence relations then represent important relations, either static or behavioral, among the programs modeled by the graphs.

The most basic applications of these simple-structured graphs in program modeling and analysis are possibly control flow graphs of sequential imperative programs [3], entity-relationship diagrams of database [27], different kinds of automata [62] and state machines [49]. These graph models of computer systems and programs are used for understanding, validation, simulation and later for verification of systems and programs, such as model checking [34, 89]. A more sophisticated application of graphs is the development of Petri-Net theories, colored and non-colored, for synchronization of concurrent and distributed systems [85]. Graphs with simple nodes, edges and necessary annotations and decorations are effective in modeling the behavior of a system with isolated states and symbolized transitions.

Hierarchical graph and hypergraph. Nowadays, however, we are facing software systems of a great scale of dimensions of complexity, e.g. the so called software-intensive systems [98]. We see these systems in our daily life, such as in aircraft, cars, banks and supermarkets [29]. These systems provide their users with a large varieties of services and features. They are becoming increasingly distributed, dynamic and mobile. In addition to the complexity of functional structure and behavior, modern software systems have complex aspects concerning organizational structure (i.e. system topology), distribution, interactions, security and real-time. This requires new programming paradigms [98], such as object-oriented programming, component-based programming and service-oriented programming.

The notions of types, values and states of a program in such a new paradigm have complex structures that affect the behavior. They cannot be effectively characterized by simple sets, data and nodes of a graph as they were in traditional procedure programming. Instead, they are appropriate to be represented as graphs themselves, called type graphs for types and instance graphs, or typed graphs, for their “values” and “states”. The behavior of a program is then defined as graphs that contain graphs, i.e. hierarchical graphs. As the name indicates, hierarchical graphs
provide a natural model for states of systems with different levels of objects and behaviors of systems with structured states.

On the other hand, the states of a system in a new paradigm are likely to have groups of objects or components that are closely related, e.g. mutually dependent. Such a relation is better represented as a hyperedge, i.e. an edge associated with a number of nodes through its tentacles, instead of a normal edge associated with just two. In this way, a system state is modeled as a hypergraph that consists of nodes and hyperedges. Besides relations, hyperedges can also be used to represent components of a system [24]. In such an approach, a group of components are related if their hyperedges are associated with a common node. The notions of hypergraphs and hierarchical graphs are not contradictory, i.e. there are hierarchical hypergraphs.

**Graph transformation.** In a theory of graph-based modeling, techniques of graph transformation or graph rewriting are developed for deriving new graphs from given ones to represent relations and transformations between models. There are different approaches to the formalization of graph transformations. The basic approach is simply adding or removing some nodes and edges. This approach is the easiest one to understand as it reflects the natural idea of transforming a graph. It is suitable for graph transformations that only change a small set of nodes and edges.

However, the basic approach is quite preliminary and needs to enumerate every single change. As a result, it would be verbose and even error-prone to characterize transformations of a large number of nodes and edges. In addition, the basic approach is not appropriate to define transformations of hierarchical graphs involving changes of nodes and edges which themselves are graphs too. Instead, these complex graph transformations are suitable to be formalized in algebraic approaches that are based on notions of category theory such as morphism and pushout. Among algebraic approaches, the most well known and widely used ones are the Double-Pushout (DPO) approach [39] and the Single-Pushout (SPO) approach [43].

There is some recent work on proposing a graph algebra where graphs are defined by terms that are obtained from a set of atom terms and a set of term operators, e.g. [24, 19]. Such an algebra is useful for symbolic manipulations of graphs and desirable for the study and formal representation of algebraic properties in an equational logic, that is, equivalence relations between graphs. For graphs defined in an algebra, graph transformations can also be formalized using term rewriting, e.g. in [24].

### 1.1 Motivation

Object-oriented (oo) programming is a programming paradigm evolved from the traditional procedural programming that uses classes of objects and their interactions to design applications and computer programs. Compared with a procedural program, the execution states of an oo program have complex structures of related objects, and the behavior of an oo program is difficult to understand and reason about. Such difficulty, for example, may come from the feature of dynamic binding of methods. For a variable $x$ and a method $m$, the behavior of a method invocation

$$x.m(...)$$

may not refer to the method $m$ of the class where $x$ is declared. By contrast, for a procedure invocation in a procedure program, the procedure to be called is always statically determined. Another example concerning the difficulty in understanding oo programs is the feature of aliasing [61]. For a variable $x$, an attribute $a$ and an expression $e$, the command

$$x.a := e$$

not only assigns the value of $e$ to the expression $x.a$, but also to every expression $y.a$ where $y$ is an alias of $x$. Poor understanding of these features may cause poor programming, which may further lead to bugs and even breakdowns of a system.
1.1. MOTIVATION

Because of the complexity and the challenge in understanding oo programs, there are a big number of traditional semantic theories of oo programs, e.g. [83, 1, 26, 66], operational or denotational, that use the basic theory of sets, functions and relations in defining the states of a program. However, as pointed out in [64], such an approach often needs to include in the syntax definitions of runtime concepts, such as locations, to indicate a value that may change over time. This and the lack of clarity about the structural properties of the states of oo programs are the main source of the complexity of these traditional theories. The complexity hinders the way of our thinking about the execution of a program and makes it difficult to formulate clear assertions about executions, which is important for analysis of the correctness of a program [59]. Such situation motivates us to provide a graph-based type system and operational semantics for oo programs. Our main concern is to provide a clarification of oo concepts and improve people’s understanding of the states and behaviors of oo programs. For this, a simple graph model is desirable which consists of simple-structured graphs and simple graph transformations.

In oo programming, the design of the class structure and the functional behavior are equally important and closely related. However, the former is likely to be ignored in most semantic theories of oo programs. For example, the theories in [83, 1, 26, 66] are mainly focused on functionalities of objects or methods of classes, without providing much help to the design of the classes and their relations. Such insufficiency motivates us to provide a calculus of structure refinement for oo programs. The notion of structure refinement is first studied in the Refinement calculus of Component and Object Systems (rCOS) [54]. It investigates what changes in the class structure maintain the capability of providing functionalities or services. Specifically, a class structure $\Gamma_1$ is a structure refinement of another class structure $\Gamma_2$, if $\Gamma_1$ provides as many services and as good (in term of functional refinement) services as $\Gamma_2$. Compared with the work [54] where structure refinement is defined through relations between mathematical sets and tuples, our refinement calculus will make use of our operational semantics so that the notion of structure refinement is formalized as graph transformations between graphs of class structures.

Service-oriented programming is an emerging programming paradigm evolved from object-oriented and component-based programming where applications and systems are constructed based on (possibly distributed) services with standard interfaces. Sometimes, people use the terminology service-oriented computing (SOC) instead of service-oriented programming to indicate that the design of a service system is more of software engineering at a higher level than of programming in the normal sense. To study and analyze the behaviors and properties of service-oriented systems, different computation models are proposed to formalize the key aspects of SOC, such as service autonomy, client-service interaction and orchestration. As process calculi [57, 76, 79] are quite mature in modeling concurrent systems and mobile systems, some people attempt to use process calculi, such as $\pi$-calculus [79], as a model of service systems [74, 40]. However, the modeling seems quite inefficient as the communication primitives of $\pi$-calculus are low level compared with aspects of SOC.

To improve this situation, a few service-oriented calculi are proposed. The Service Centered Calculus (SCC) [12] introduces service definition, service invocation and session handling as first class modeling elements, so as to model service systems at a better level of abstraction. However, SCC has a rudimentary mechanism for handling session closure, and it has no mechanism for orchestrating values arising from different activities. These aspects have been improved in the Calculus of Session and Pipelines (CaSPiS) [13]. CaSPiS supports most of the key features of SCC, but the notions of session and pipelining play a more central role. A session has two sides (or participating processes) and it is equipped with protocols followed by each side during an interaction between the two sides. A pipeline permits orchestrating the flow of data produced by different sessions. A structured operational semantics of CaSPiS is given in [13] based on labeled transitions. It does yet have a simpler and compact reduction semantics [16] that handles silent actions of processes in the labeled transition system.

The notions of service, session and pipeline introduce a strong hierarchical nature into service-oriented systems. For example, Figure 1.1 shows the sketch of a system, which consists of the definition (annotated by $D$) of a service $s$ with a protocol $P_1$, an invocation (annotated by $I$) of the service $s$ with a protocol $P_2$, and two sides of a session $r$ (annotated by $S$) with protocols $P_3$.
and $P_4$, respectively. The system is hierarchical with the service invocation nested in one side of the session. However, it is specified as a “linear” process expression

$$s.P_1 | r \triangleright (\pi.P_2|P_3)|r \triangleright P_4$$

in CaSPiS which fails to provide an intuitive visualization of the hierarchical nature. Such insufficiency motivates us to provide a hierarchical graph representation of service-oriented systems specified in CaSPiS. Besides providing an intuitive visualization that improves the understanding, such a representation enables us to formalize the behaviors of service-oriented systems as graph transformations in the DPO approach. This leads to a novel concurrent semantics [7] of CaSPiS which is helpful in recording causal dependencies between interactions and exploiting such information for detecting the possible source of faults and mis-behaviors. In order to make the theory work, we have to prove that the concurrent semantics is consistent with the original reduction semantics of CaSPiS.

1.2 Contribution

In this thesis, we provide a graph-based type system and operational semantics for a general object-oriented language from the rCOS [54] method with a large variety of oo features such as encapsulation, inheritance, type casting and dynamic method binding. The basic idea is to define the class structure of an oo program as a directed and labeled graph, called a class graph. In a class graph, a node is either a class node that represents a class or a leaf node that represents a primitive data type. An outgoing edge of a class node is labeled by an attribute of the class or a special symbol representing the inheritance relation, and it is targeted at the node representing the type of the attribute or the direct superclass of the class, respectively. Class graphs are crucial for the construction of the type system. This is because the type checking of an expression or a command needs to refer to a specific class graph to check whether an attribute is defined in a class, whether a type cast is valid, whether an expression is assigned with a type-consistent value, and so on. In addition, the type checking also depends on the type information of local variables and formal parameters of methods. We formalize such information as a type context, which is a directed and labeled graph, too.

If we regard the class graph of a program as a type, the execution states of the program are its instances. We define an execution state of an oo program also as a directed and labeled graph, called a state graph. A node in a state graph represents either an object or a simple datum. However, in the former case, the node is not labeled by an explicit reference value, but by the name of its runtime type that is a class name of the program. An edge represents an attribute of
1.2. CONTRIBUTION

the source object referring to the target object and thus it is labeled by the name of the attribute. To be a valid instance of a class graph, a state graph should satisfy two conditions.

1. The type of each object in the state graph is a class defined in the class graph.

2. For each attribute of an source object that refers to a target object, the attribute is defined in the class of the source object, and its type is consistent with the class of the target object.

It is well known that an object or a family of related objects can be represented as a graph, in which nodes are objects and edges are their attributes, e.g. [46, 55]. Intuitively, a state at a time of the execution of an oo program consists of the existing objects and their relationships at that time. Each step of the execution is to change the graph, and the changes of a graph can be defined by simple graph operations, such as swinging an edge, adding a new subgraph denoting a newly created object and removing an existing subgraph representing an existing object. However, the definitions of the execution states and the operational semantics are more subtle than this. On the one hand, an invocation of a method accesses to and operates on not only the fields of the self object (some oo languages use the keyword this instead of self) but also the formal parameters and local variables of the method. These constitute a scope of the execution and the scope changes when another method is called inside the invocation. For this, we introduce scope nodes into a state graph that represent scopes and arrange these nodes with their self objects, formal parameters and local variables on the top of the graph, just as a stack structure. A change of the scope of execution is done by pushing in or popping out its corresponding scope node in the state graph. On the other hand, the rules of a small-step semantics of a method invocation are challenging, in particular when, unlike in existing semantic definitions such as [66], we do not use address variables. Nevertheless, with the careful combination of the notions of scope stacks and objects in the concept of execution states, the model is indeed simple and defined as a classical Structural Operational Semantics (SOS) [86], using only the basic notions of graphs and graph operations.

We prove that the operational semantics is consistent with the type system. That is, the evaluation of a well-typed expression, the execution of a well-typed command and further the execution of a well-typed program will not get blocked, unless certain exception cases happen, for example the execution meets a null reference. In addition, it is worth pointing out that the operational semantic is location independent or address independent. Such a feature is important as oo programs only refer to variables and navigation paths, but not addresses or references. In this sense, our operational semantics is more abstract than most current oo semantics that explicitly refer to object references.

Based on the operational semantics, we define the notion of structure refinement in terms of graph transformations between class graphs (representing class structures) that preserve the functional behavior. Specifically, a structure refinement is a transformation \( \rho \) from one class graph \( \Gamma_1 \) to another class graph \( \Gamma_2 \), with a derived transformation \( \rho_s \) from the instances of \( \Gamma_1 \) to instances of \( \Gamma_2 \), such that

- for any instance \( G_1 \) of \( \Gamma_1 \) and \( G_2 \) of \( \Gamma_2 \) where \( G_1 \) can be transformed into \( G_2 \) by \( \rho_s \), if \( G_1 \) is transformed into another instance \( G'_1 \) of \( \Gamma_1 \) through the execution of a certain method invocation \( z.m() \), \( G_2 \) can be transformed into another instance \( G'_2 \) of \( \Gamma_2 \) through the execution of \( z.m() \) and \( G'_2 \) can be obtained from \( G'_1 \) by \( \rho_s \).

This is shown by the commute diagram in Figure 1.2.

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

1. The first group 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 are graph compression rules for combining classes, removing redundant classes, attributes, and collapsing an inheritance relation but do not involve method overriding.
The rules in the first group support object-oriented top down and incremental design, while the other rules are useful for program refactoring, abstraction, analysis and reverse engineering. It is straightforward to establish the soundness of these rules with respect to the definition of the structure refinement. For their completeness, we prove that the first group of rules are complete for a restricted set of class graph transformations, called well-typed structure transformations. In addition, the four groups of rules as a whole are complete in the sense that they are able to transform any class graph into a certain normal form, which has the same functionality as the original class graph while with a “minimal” number of classes.

We also provide a graph-based representation of service-oriented systems specified in the process calculus CaSPiS [13], a general service-oriented language focused on key features of SOC such as service autonomy, client-service interaction and orchestration. Since systems of services have a strong hierarchical nature, in both of their static structures and dynamic behaviors, we represent them as hierarchical graphs instead of simple-structured graphs as we used to represent oo systems.

As the basis of the representation, we set up a model of hierarchical graphs by exploiting a graph algebra [19, 20]. The algebra consists of a syntax of hierarchical graph terms with primitives for composition, name restriction and design hierarchy, and a semantics that interprets each hierarchical graph term as a multi-level structure of hypergraphs with mappings between different levels. We adopt the syntax of the algebra, as the primitives of the syntax are quite suitable to specify hierarchical graphs. Nevertheless, we provide a different semantics because in the original one it is very difficult to formalize the change of a hierarchy which involves the rebuilding of various levels of hypergraphs and mappings between them. In our semantics, a hierarchical graph term is interpreted as a single hypergraph with different groups of nodes and edges connected by a special kind of edges, called abstract edges. Such an interpretation enables us to deal with the change of a hierarchy conveniently, i.e. through adjusting the layout of abstract edges. Based on the semantics, it is straightforward to define the algebraic notions of morphism and pushout for hierarchical graphs, and further graph transformations in the DPO approach.

We first define a direct graph representation of CaSPiS processes, which specify the states of service systems, as hierarchical graphs in the graph model. The graph of a process is constructed simply according to the structure of the process. For example, the graph of a process $P|Q$, i.e. the parallel composition of two sub-processes $P$ and $Q$, is defined as an edge labeled by $Par$, which means parallel composition, associated with both the graphs of $P$ and $Q$. For a faithful visualization of the hierarchy of a process, the graph of each service, session and pipeline is encapsulated before it is composed with other graphs, so that the graph representation is hierarchical due to possibly nested services, sessions and pipelines.

Such a representation is straightforward to define and easy to understand, but it fails to distinguish sub-processes occurring in different contexts, i.e. static contexts and dynamic contexts. By contrast, the reduction semantics of CaSPiS only allows sub-processes in static contexts to take part in an interaction. To overcome such inconsistency, we define a tagged graph presentation of
processes, which is derived from the direct representation, where graphs of sub-processes occurring in static contexts are explicitly attached with tags.

Based on the tagged graph representation, we provide a graph transformation system to characterize the behavior of service systems specified by CaSPiS processes. The system consists of five sets of graph transformation rules.

1. The first set of rules are auxiliary rules for operating the tags. They are useful to obtain the tagged graph of a process from its untagged version.

2. The second set of rules aim at modeling the congruence relation between CaSPiS processes, which is defined within the reduction semantics of CaSPiS. For two congruent processes, this set of rules allows us to transform the tagged graph of one into that of the other.

3. The third set of rules are concerned with the interactions of services, sessions and pipelines. They are designed to simulate the reductions of CaSPiS processes. However, they may produce garbages as well as auxiliary edges for further data assignment which do not belong to the graph of any process.

4. The fourth and fifth sets of rules are auxiliary rules for garbage collection and data manipulation, respectively. They will remove the garbages and consume the auxiliary edges produced by the third set of rules.

We prove the soundness and completeness of the graph transformation system. Specifically, the first two sets of rules are sound and complete with respect to the congruence relation of CaSPiS processes, while the five sets of rules as a whole are sound and complete with respect to the reduction semantics of CaSPiS processes.

1.3 Related Work

Our work presented in this thesis is closely related with a few topics. It is worth introducing some representative work on each of them.

On formal semantics of oo programs. Model-based formalisms have been used extensively in conjunction with oo techniques, via languages such as Object-Z [95], VDM++ [41], and methods such as Syntropy [36] which uses the Z notation and Fusion [35] 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 mechanisms of oo programming languages, such as dynamic binding and polymorphism.

A modeling notation using the format of classes with inheritance and abstract specifications of class methods is more directly related to oo 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 define an oo programming language ROOL [26, 83] with subtypes and polymorphism using predicate transformers. Sekerinski provides a rich oo language [94, 75] 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 the complexity in the flow of control, it is infeasible to calculate the weakest precondition of an oo program for a given post condition. As a result, semantic proofs of refinement rules in ROOL are quite complex even without references. America and de Boer introduce a logic for the parallel language POOL [4]. It applies to imperative programs with object sharing, but without subtyping and method overriding. Abadi and Leino propose an axiomatic semantics for an imperative oo language with object sharing [2], but it does not permit recursive object types. Poetzsch-Heffter and Müller define a Hoare-style logic for oo programs [88] that relaxes many of the previous restrictions. However, as [70] points out, the specification of a method in this logic is derived from the implementation of the method, and thus the logic does not support the notion of refinement.
In [70], Leino presents a logic 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 OO programs. To improve this situation, Liu, et al. provide a comprehensive method rCOS [54, 72] for component-based and object-oriented model-driven development. rCOS is based on a formal language which supports most general OO notions such as reference type, visibility, inheritance, type casting, dynamic binding and polymorphism. The language is equipped with a denotational semantics defined in Unifying Theories of Programming (UTP) [58] and a refinement calculus, which consists of a set of refinement rules proved based on the denotational semantics, for studying the substitutability of class structures. We adopt this language as the basis of our graph-based operational semantics and refinement calculus. Our graph-based refinement rules and their soundness and completeness results that are proved based on our operational semantics agree with those proved based on the denotational semantics [54]. This gives the justification of the correctness of both semantic models. On the other hand, the use of graph notations in our model significantly improves the understanding of the semantics and refinement of OO programs.

Besides rCOS, there is some work that studies the notion of refinement in OO development. The work in [75, 5] handles class and interface refinements. However, the focus is substitutability of individual classes in a class structure. By contrast, our work investigates the refinement of a class model as a whole and supports structure design at different stages of the system development. In [48], 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.

**On graph-based models for OO systems.** Graph transformations are used in [44] 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 refinement calculus, however, is transformations of class graphs and how they determine the transformations on commands that preserve the 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 [64], a theory of graph transformation is applied to a definition of an OO execution semantics of a mini language. The simulation relation is then studied for programs in that language. The work in [96] formulates structural properties using graph constraints in type graphs with inheritance, and shows how to translate constrained type graphs with inheritance into 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 calculus goes beyond the concerns of these models 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 notations of graphs for pointer structures in [28], and the idea of using paths of a graph comes from the trace model of objects with pointers [59]. The notion of object graphs can be seen as an extension of the notion of execution states in classical imperative procedural programs. Based on this understanding, we claim that the theory of structure refinement between OO programs is a non-trivial extension of the theory of data refinement [53] for the support of OO software design. The calculus of OO 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 derive the data mappings. This extension is important as it advances the the classical refinement calculi to a design method applicable in large scale system development with effective tool support for model transformations [29].
1.3. RELATED WORK

On formal computation models for SOC. There are several service-oriented process calculi developed along with CaSPiS [13]. Bartoletti, et al. introduce \( \lambda^{req} \) [9] for characterizing the selection and composition of services. As an extension of \( \lambda \)-calculus with primitives for service request and local security policy, \( \lambda^{req} \) supports static analysis to ensure the services selected for composition match both the user’s requirements and the security requirements. Compared with \( \lambda^{req} \) which is focused on modeling and analyzing service composition, CaSPiS is mainly concerned with formalizing basic concepts of SOC such as service definition, service invocation and interaction. Guidi, et al. propose a multi-layer calculus named Service Oriented Computing Kernel (SOCK) [52] where the design of services is decomposed into three fundamental parts: the behavior, the declaration and the composition. To avoid interactions taking place between wrong pairs of service instances, SOCK uses the mechanism of correlation sets to distinguish different service instances, which is inspired by Web Services Business Process Execution Language (WS-BPEL) [99]. By contrast, CaSPiS solves the problem through the mechanism of dynamic creation of sessions, which is at a higher level of abstraction. Vieira, et al. provide Conversion Calculus [97] as a model for expressing service systems with emphasize on conversation context and context-sensitive message passing. The mechanism of message passing is similar to that of session in CaSPiS. However, there is no mechanism for orchestrating the flow of messages arising from different activities. By contrast, CaSPiS has the primitive of pipeline for orchestration of sessions. The notion of pipeline is inspired by Orc [81], a basic and elegant model for orchestration of computation entities. Another service-oriented calculus with primitives of orchestration is Stream-based Service Centered Calculus (SSCC) [68] proposed by Lanese, et al. As its name indicates, SSCC characterizes orchestration based on the notion of stream, where data can be stored into and retrieve from. This is different from the pipelining mechanism of CaSPiS.

On graph-based models for distributed and concurrent systems. There are various models of graphs and graph transformations that aim at characterizing and visualizing distributed, concurrent and mobile systems, including service systems. Gadducci proposes a graphical implementation of \( \pi \)-calculus in [47] based on term graph rewriting [8, 87]. In this work, processes of \( \pi \)-calculus, including recursive ones, are encoded into term graphs, which are directed and acyclic hypergraphs over a chosen signature representing “terms with shared sub-terms” over the signature. The use of term graphs, which are not hierarchical, makes it straightforward to reuse the standard graph rewriting technique, such as the DPO approach, which leads to a non-deterministic concurrent semantics. Then, the soundness and completeness of the encoding is verified by proving the equivalence of the concurrent semantics and the original reduction semantics of \( \pi \)-calculus. Milner provides a behavior semantics for condition-event Petri nets [92] in [78] based on bigraphs and their reactive systems [77, 63]. Generally, a bigraph consists of two orthogonal structures: a place graph and a link graph representing locality and connectivity of agents, respectively. In this work, a condition-event petri net is modeled as a bigraph whose place graph is flat, and then the behavior of the net is modeled as a bigraphical reactive system equipped with a labeled transition system and an associated bisimilarity equivalence. This bisimilarity is shown to coincide with the original one of condition-event petri nets. Another graph-based framework is presented by Hirsch and Tuosto [56] for specifying systems with high-level Quality of Service (QoS) aspects, where constraint-semirings [10] are used to describe QoS requirements of various criteria. The framework is based on Synchronized Hyperedge Replacement (SHR) [25, 45], a hypergraph rewriting mechanism for modeling the reconfiguration of (possibly distributed) systems. In SHR, the behavior of a single edge is defined by the notion of production, which indicates how and under what condition an edge can be replaced by a generic graph. Then, global transitions are obtained by synchronizing applications of productions with compatible conditions. A summary and comparison of graph models for distributed, concurrent and mobile systems can be found in the survey [21].

On algebras of graphs. There is some work that proposes and makes use of a graph algebra. Corradini and Gadducci introduce a preliminary algebra for term graphs [37] by showing that every term graph can be constructed from a small set of atom term graphs, each of which is
regarded as an atom term, using two basic operations (composition and union). The algebra is then used to establish an isomorphism from terms graphs to arrows of graph substitution monoidal (gs-monoidal) categories. Bruni, et al. present Architectural Design Rewriting (ADR) [24], a graph-based approach to the design of reconfigurable software architectures. In ADR, architectures are encoded as terms of a simple syntax of hierarchical graphs with a set of ad-hoc operators and atom constructs. Based on this algebra, architectural reconfigurations are defined inductively using standard term rewriting techniques. Inspired by ADR, Bruni, et al. provide an algebra of hierarchical graphs [19, 20] with primitives for composition, node restriction and nesting. It is a high-level language for specifying graphs with node sharing and embedded structures, thus well suited for representation of software systems where nesting and linking are key aspects. In this thesis, we adopt the syntax of this algebra, but define a new semantic model in order to support graph transformations in the DPO approach. A similar graph syntax, namely Algebra of graphs with nesting (AGN), can be found in [17] which is built on graphs with nesting and restriction (NR-graphs). AGN is also equipped with primitives for composition, restriction and design hierarchy, but it considers two kinds of restricted nodes, local and global, and unifies the notions of edges and designs, compared with the algebra of hierarchical graphs. In addition, the correspondence between NR-graphs and AGN terms is established indirectly, through encoding them into term graphs and arrows of gs-monoidal categories, respectively, and using the isomorphism between terms graphs and arrows of gs-monoidal categories [37]. By contrast, the relation of the algebra of hierarchical graphs and models of term graphs has not been exploited yet. Another graph algebra is proposed by Grohmann and Miculan [51] which is a typed language for the category of binding bigraphs, a generalization of the original pure bigraphs. Similar to the algebra of hierarchical graphs and AGN, the language has general constructs of graphs such as parallel composition and restriction, but it also has a few bigraph-specific primitives such as localization and globalization. The language is shown expressive as its certain sub-languages can be used to characterize the categories of pure, local and prime bigraphs. It can be tailored to formalize graph models of SHR and ADR as well.

It is worth pointing out that the algebra of hierarchical graphs is also applied in [20] to encode a couple of process calculi that characterize systems with nested structures, including CaSPiS. But there, the focus is on the encoding of states of systems rather than their behaviors. A step forward is made in [18] where behaviors of CaSPiS processes are also exploited. In this work, standard forms of graph transformation rules are provided to model reductions of processes, while each rule is defined in a context-sensitive way, i.e. it only deals with the case that the reduction occurs in a specific context. Therefore, to handle reductions in all possible contexts, an infinite number of rules is needed. This problem is solved in our graph model as we consider graph transformation rules in the DPO approach which are context-insensitive, i.e. one rule is enough to deal with one kind of reductions that occur in any possible context.

1.4 Structure of the Thesis and Origin of the Work

The rest of this thesis is organized in the following way.

Chapter 2: Graph-based Operational Semantics for oo Programs. This chapter presents our graph-based type system and operational semantics for oo programs specified in the formal language of the rCOS method [54]. The graphs we consider in this chapter are simple-structured ones, particularly directed and labeled ones. We define class graph and state graph that represent the class structure and an execution state of an oo program, respectively. Based on class graphs, we provide a type system that checks whether an expression, a command of a program, or the program itself is well-typed. And based on state graphs, we provide a small-step operational semantics that defines how an expression is evaluated and how a command or program is executed step by step. We prove that the operational semantics is type-safe with respect to the type system, i.e. the evaluation of a well-typed expression, the execution of a well-type command or well-type program does not “go wrong”. Finally in this chapter, we show the expressiveness of our graph notation by illustrating that a lot of interesting properties of oo programs can be stated and analyzed.
1.4. STRUCTURE OF THE THESIS AND ORIGIN OF THE WORK

A simplified version of this chapter has been published in [65]. The paper is focused on introducing the operational semantics, while the type system and the type safety of the operational semantics are only informally stated.

Chapter 3: Graph-based Structure Refinement for oo Programs. As an application of the operational semantics introduced in Chapter 2, we present in this chapter our graph-based refinement calculus for oo programs. The basic idea is to define the notion of structure refinement in terms of a graph transformation between two class graphs, whose instance state graphs are related with a derived graph transformation which is preserved by the execution of certain methods (See Figure 1.2). We first provide a group of graph transformation rules to study structure refinement that expands the class structure. We show that they are sound refinement rules and complete with respect to a subset of structure refinement, between class graphs with syntactic correspondence. We also provide a few groups of refinement rules for contracting the class structure, changing methods and combining classes with polymorphism. We prove that they are sound refinement rules, too, and complete in that they are able to transforming each class graph into a certain normal form.

A preliminary version of the refinement calculus is published in [71], where only structure refinement for expanding the class structure is studied. The current version of the calculus is published in [101] with all the refinement rules. However, the basis operational semantics, which was still under development then, is only informally stated, as well as the soundness of refinement rules.

Chapter 4: Graph Representation of Service-Oriented Systems. This chapter presents our framework of graph representation of service systems specified in the service-oriented process calculus CaSPIs [13]. The graphs we consider in this chapter are hierarchical ones that naturally capture the hierarchical nature of service systems. In particular, we provide an algebra of graphs for symbolic manipulation of graphs, where each hierarchical graph is specified by a term. The algebra is inspired by the one in [19] with primitives for composition, node restriction and design hierarchy. But differently from [19], our algebra supports the formalization of graph transformations in the DPO approach. After establishing the graph model, we show how to represent CaSPIs processes, i.e. states of service systems, as hierarchical graphs in the algebra. Based on this representation, we provide a graph transformation system to specify the behaviors of service systems in term of a few sets of graph transformation rules. Finally in this chapter, we prove the soundness and completeness of the graph transformation system with respect to the original congruence relation and reduction semantics of CaSPIs processes.

A simplified version of this chapter has been published in [23], where a restricted set of graph transformation rules is studied and thus a restricted result of soundness and completeness is proved. The full result of the chapter has been submitted to Journal of Science of Computer Programming [22].

Chapter 5: Conclusions. At the end of this thesis, we summarize the main results presented in the previous chapters, with an exploration of possible research topics for future work.
Chapter 2

Graph-Based Operational Semantics of OO Programs

In this chapter, we propose a graph-based type system and operational semantics for an object-oriented formal language. Our main objective is to provide a conceptual clarification for better understanding of states and behaviors of OO programs.

The formal language we consider is the OO language of the rCOS method [54] which supports a large variety of OO concepts such as encapsulation, inheritance, type casting and polymorphism. An OO program $\textit{cdecls Main}$ consists of a section of class declarations $\textit{cdecls}$ and a $\textit{main method Main}$. The class declaration section $\textit{cdecls}$ defines the class structure of the program, where each class is declared with its own attributes and methods, and may also inherit another class. A method $m(\vec{S} \vec{x}; \vec{T} \vec{y})\{c\}$ is declared with its value parameters $\vec{x}$ (of type $\vec{S}$), result parameters $\vec{y}$ (of type $\vec{T}$) and a body command $c$. Compared with (class) methods that have parameters, the main method $\textit{Main}$ consists of a sequence of external variables $\vec{ext}$ and a main command $c$. The external variables $\vec{ext}$ are both the input and output of the program, while the main command $c$ realizes the application task of the program through creating objects of classes declared in $\textit{cdecls}$, invoking methods of these objects and updating the external variables. Commands $c$ and expressions $e$ are defined in the same way as in most OO programming languages, except the introduction of non-deterministic choice commands for the purpose of specification and refinement.

For a visualization and clarification of OO concepts, we base our approach on simple-structured graphs and their transformations instead of mathematical sets, tuples and their relations, which are used in most existing semantic theories of OO programs [83, 75, 66]. We first define class graphs and object graphs that naturally capture the essential OO concepts. A class graph is a directed and labeled graph that represents the class structure of an OO program. In a class graph, a node represents either a class or a primitive data type such as integer and string. An outgoing edge of a class node $C$ is labeled by an attribute $a$ of $C$ or a special symbol $\triangleright$ representing the inheritance relation, and it is targeted at the node representing the type of the attribute $a$ or the class that $C$ inherits, respectively. An object graph is also a directed and labeled graph that represents a set of objects and their relations. In an object graph, a node represents either an object or a simple datum. In the former case, however, the object node $o$ is labeled by its runtime type $C$, i.e. the class of the object $o$, instead of an explicit reference value. An outgoing edge of an object node $o$ represents an attribute $a$ of the object, thus it is labeled by $a$ and targets at the node of object or datum $a$ refers to.

However, an object graph is not enough to represent an execution state of an OO program. This is because an execution state also involves scopes of variables, besides objects and their relations. When a local variable $x$ is declared, for example, the execution enters a new scope with $x$; and when $x$ is undeclared, the execution goes back to the scope before $x$ is declared. Similarly, when a method $m$ is invoked, the execution enters a new scope with formal parameters of $m$; and when the invocation terminates, the execution returns to the scope before the invocation. For a faithful
characterization of the execution states of an oo program, we define state graphs which extend object graphs with nodes that represent scopes. An outgoing edge of a scope node represents a variable of the scope and it targets at the value of the variable. In a state graph, the scope nodes are arranged as a stack structure, with the top node representing the current scope. When the execution enters a new scope, a new scope node is created and pushed into the stack; and when the execution leaves the current scope, the top scope node is popped out of the stack.

These graph notations enable us to provide a type system and operational semantics of the oo language. The type system checks whether an oo program is well-typed without executing the program, and the checking is based on the class graph, i.e. the class structure, of the program. We construct the type system step by step, from the checking of expressions, commands to that of class graphs and further programs. The type checking of an expression mainly checks whether a variable is correctly used and whether an attribute can be accessed in the context. For example, a variable has to be used after it is declared and before it is undeclared. The type checking of a command mainly checks the type consistency in an assignment or method invocation. Taking method invocation for example, the type of each actual parameter should be consistent with that of its corresponding formal parameter. The type checking of a class graph checks whether each class method is well-typed, i.e. whether the body command is well-typed with respect to the formal parameters. Finally, the type checking of a program checks whether its class graph is well-typed and whether its main command is well-typed with respect to its class graph and external variables.

The operational semantics is provided in the classical SOS style, where the execution of commands is defined in terms of transitions between configurations. A configuration is either a non-terminated one \((c, G)\) representing a state (graph) \(G\) with a command \(c\) to be executed, or a terminated one which is simply a state (graph) \(G\) at which the execution terminates. For most command constructs such as assignments and object creations, we define their execution through simple operations on state graphs, for example swinging an edge or adding an object node. Notice that the execution of a command may involve evaluation of expressions, but this can be done straightforwardly, through navigation over the state graph. The main challenge in providing the semantics is to define the execution of a method invocation \(x.m(\ldots)\) which involves a few dedicate issues. For example, for dynamic binding of methods, we have to look-up the method \(m\) declared in the class of the object that \(x\) refers to, instead of the class that \(x\) is declared. Another challenge is to ensure the consistency between the semantics and the type system. For example, the execution of a well-type command should not get blocked, unless certain exception cases happen.

It is worth pointing out that the oo language of the rCOS method is equipped with a UTP-based denotational semantics and, derived from the semantics, a set of refinement rules [54]. We will show the correctness of our graph-based operational semantics with respect to the UTP-based denotational semantics in Chapter 3, in that its derived refinement calculus agrees with the refinement rules derived from the denotational semantics.

Section 2.1 is a brief introduction of the oo language of the rCOS method [54]. Section 2.2 defines class graphs, object graphs and state graphs, followed by a set of basic operations on these graphs. Based on these graph notations, we present our type system and operational semantics in Section 2.3 and Section 2.4 for static type checking and dynamic execution of oo programs, respectively. We also prove that the operational semantics is consistent with the type system. Finally, in Section 2.5, we show examples of properties of programs that can be stated and analyzed in our graph model.

2.1 Background: the Object-Oriented Language of rCOS

In this section, we introduce the formal language of the rCOS method [54] which is used as the basis of our operational semantics to be presented in this chapter. We slightly adjust some of the syntactic constructs for the convenience of our discussion.

The vocabulary of the language consists of four disjoint sets: \(\mathcal{C}\) of class names, \(\mathcal{B}\) of names of primitive data types such as \(\text{Int}\) and \(\text{Bool}\), \(\mathcal{V}\) of names of variables and attributes, and \(\mathcal{M}\) of names of methods. Let \(\mathcal{T}\) be the names of types, i.e. the union of \(\mathcal{C}\) and \(\mathcal{B}\). The syntax of
Figure 2.1: Syntax of rCOS language
a method is allowed to be overridden in a subclass, but its signature, i.e. types of parameters, should be preserved.

Unlike (normal) methods that have parameters, the main method declares a sequence of *external variables* \(\text{ext}\) of the program, together with a main command \(c\). Each external variable declaration consists of its type, name and initial value. These variables are both the input and output of the program as the main command accesses and updates them. Due to the need of encapsulation, the main command is not allowed to access the attributes of classes. Instead, it calls methods that access the attributes.

A command can be simply \(\text{skip}\) that does not do anything. Command \(C.\text{new}(le)\) creates an object of class \(C\) and attach it to \(le\). The attributes of the object are initialized with the initial values declared in \(C\). Command \(le := \vec{c}\) assigns a sequence of expressions \(\vec{e}\) with a sequence (of the same length) of values \(\vec{c}\), respectively. We will write it as \(le := e\) if we are only concerned with a single assignment. Command \(e.m(\vec{c}; \vec{e})\) calls the method \(m\) of the object \(e\) refers to, with actual value parameters \(\vec{c}\) and actual result parameters \(\vec{e}\). While \(\text{var} \ T \; x = e\) declares a local variable \(x\) of type \(T\) with initial value \(e\). The scope of \(x\) can be ended by \(\text{end} \; x\) afterwards. Commands \(c_1; c_2\), \(c_1 \;\text{if} \; b \;\text{then} \; c_2\), \(c_1 \;\text{else} \; c_2\) stands for sequential composition, conditional choice, non-deterministic choice and loop, respectively. Here, we are not concerned with the detailed structure of a boolean expression \(b\), simply assuming it is of type \(\text{Bool}\) and can be evaluated to either \(\text{true}\) of \(\text{false}\).

Expressions include assignable expressions \(le\), or we simply say \(l\)-expressions, the special variable \(self\) that represents the currently active object, expressions with type casting \((C)e\), literals \(cn\), and expressions \(h(\vec{e})\) constructed with built-in operations \(h\) of primitive types. Notice that \(C.\text{new}(le)\) is a command rather than an expression, thus expressions of the language indeed have no side effects.

### 2.2 Class Graphs, Object Graphs and State Graphs

We define class graphs, object graphs, and state graphs in this section and discuss their relations. We also define graph operations that we need in the upcoming type system and operational semantics.

#### 2.2.1 Class graphs

The class structure of an object-oriented program can be represented as a directed and labeled graph. A node represents a class of objects or a type of data, so it is labeled by a type name in \(\mathcal{T}\). An edge is labeled by the name of an attribute, or a designated symbol \(\rhd\) to represent the inheritance of one class from another.

**Definition 2.2.1 (Class graph).** A class graph is a directed and labeled graph \(\Gamma = \langle N, E, M \rangle\), where

- \(N \subseteq \mathcal{T}\), denoted by \(\Gamma.\text{node}\), is the set of nodes. Each node represents a class or a primitive data type,
- \(E \subseteq N \times (V \cup \{\rhd\}) \times N\) is the set of edges, denoted by \(\Gamma.\text{edge}\). An edge \((C, a, D) \in E\) means class \(C\) has an attribute \(a\) of type \(D\), and an edge \((C, \rhd, D) \in E\) says \(C\) is a direct subclass of \(D\),
- \(M\) is a function that maps each class node to a set of method definitions, denoted by \(\Gamma.\text{method}\), and \(m(\bar{S} \; \bar{x}; \bar{T}\; \bar{y})\{e\} \in M(C)\) means the method \(m\) is defined in class \(C\).

We use \(C \rhd_\Gamma D\) to denote that \(C\) is a direct subclass of \(D\) in \(\Gamma\), \(\preceq_\Gamma\) the subtype relation defined by \(\Gamma\), which is the extension of the reflexive and transitive closure of \(\rhd_\Gamma\) on \(\mathcal{T}\), and \(\succ_\Gamma\) the supertype relation defined by \(\Gamma\), which is the inverse relation of \(\preceq_\Gamma\). We always omit the subscript \(\Gamma\) when there is no confusion.
Not every class graph defined above represents the class declarations of a syntactical program. We thus define the \textit{well-formed} graphs. A class graph \( \Gamma = \langle N, E, M \rangle \) is well-formed if the following conditions hold.

- Data types can only be used to label leaves: \((C, a, T) \in E \Rightarrow C \in C\).
- Labels of outgoing edges from a node are different, and thus attributes of a class are distinct and there is no multiple inheritance.
- Conditions for the inheritance:
  - the inheritance relation is only defined among classes,
  - there is no cycle formed by \( \triangleleft \) edges,
  - no attribute overriding is allowed: \( C_1 \preceq C \land C_1 \neq C \land (C, a, S) \in E \Rightarrow (C_1, a, T) \notin E \).
- Conditions for methods:
  - names of methods defined in each class are distinct, i.e. we do not consider method overloading. So, \( M \) can be regarded as a set, and each of its elements is of the form \( C :: m \), representing a method \( m \) defined in class \( C \),
  - class types of parameters of a method and class types used in the method body must be nodes of the graph: for each class name \( C \) that occurs in \( m(S \vec{x}; T \vec{y})\{c\} \), \( C \in N \),
  - overriding of a method preserves the method signature: \( m(S \vec{x}; T \vec{y})\{c\} \in M(C) \land C' \preceq C \land m(S' \vec{x}'; T' \vec{y})\{c'\} \in M(C') \Rightarrow S' = S \land T' = T \).

It is worth pointing out that a method can be overridden by one or more methods. These methods are called a set of \textit{polymorphic} methods and they have the same signature. In the rest of the paper, a class graph always means a well-formed class graph unless it is stated otherwise. Besides, we assume a data type occurs in a class graph when needed.

An example of class graph is shown in Fig. 2.2(1). It can be alternatively represented as a UML class diagram, depicted in Fig. 2.2(2), but UML class diagrams do not have the properties of the mathematical structure of directed and labeled graphs needed for formal reasoning and analysis.

An important point to note is that a class graph has three disjoint sets of edges.

- \textit{Data attributes} (also called \textit{data edges}) are those edges \((C, x, B)\) with \( C \in C \) and \( B \in B \).
- \textit{Association attributes} (also called \textit{association edges} or simply \textit{associations}) are those edges \((C, a, D)\) with \( C, D \in C \).
- \textit{Inheritance relations} (also called \textit{inheritance edges}) are the edges \((C, \triangleleft, D)\) with \( C, D \in C \).
We use the term attribute edge (or simply attribute) to denote either a data attribute or an association, and the term relational edge to denote an edge associated with two class nodes, i.e. either an association or an inheritance edge.

To represent more static features of the program, we extend the class graph with a set of functions. For a class \( C \) in a class graph \( \Gamma = (N, E, M) \), \( \text{attr}_\Gamma(C) \) denotes the labels of the outgoing edges from \( C \), i.e. the attributes directly defined in \( C \), and \( \text{Attr}_\Gamma(C) \) the set of attributes of \( C \) as well as those of all its superclasses, i.e. the actual attributes of \( C \) including those through inheritance. They are defined by

\[
\text{attr}_\Gamma(C) = \{ a \in V \mid \exists T \in N \bullet (C, a, T) \in E \} \\
\text{Attr}_\Gamma(C) = \{ a \in V \mid \exists D \bullet (C \preceq D \land a \in \text{attr}_\Gamma(D)) \}.
\]

For an attribute \( a \in \text{Attr}_\Gamma(C) \), we use \( \text{dtype}_\Gamma(C, a) \) to denote its declared type \( T \), i.e. \( (D, a, T) \in E \) for some \( D \succ C \), \( \text{init}_\Gamma(C, a) \) its initial value and \( \text{visib}_\Gamma(C, a) \) its visibility, which is either private, protected or public. Besides, we introduce two partial functions \( \text{mtype}_\Gamma(C, m) \) and \( \text{mbody}_\Gamma(C, m) \) for looking up the parameter type (signature) and the body of a method \( m \) from a class \( C \), respectively.

\[
\text{mtype}_\Gamma(C, m) \equiv \begin{cases} \{ (\vec{s}; \vec{t}) \} & \text{if } m(\vec{s}; \vec{t}) \{c\} \in M(C) \\ \text{mtype}(D, m) & \text{otherwise, if } C \succ D \end{cases}
\]

\[
\text{mbody}_\Gamma(C, m) \equiv \begin{cases} (\vec{x}; \vec{y}; c) & \text{if } m(\vec{s}; \vec{t}) \{c\} \in M(C) \\ \text{mbody}(D, m) & \text{otherwise, if } C \succ D \end{cases}
\]

We omit the subscript \( \Gamma \) in these notations when there is no confusion.

For an \( \text{oo} \) program \( \text{prog} = \text{cdecls} \bullet \text{Main} \), its class declaration section \( \text{cdecls} \) of can always be represented as a class graph \( \Gamma \). In the rest of the paper, we will denote \( \text{prog} \) as \( \Gamma \bullet \text{Main} \) instead of \( \text{cdecls} \bullet \text{Main} \), since we are more interested in graph notations of classes.

### 2.2.2 Object graphs

An object graph describes a family of objects and their relations. A node represents either an object, called an object node and labeled by its class, or a constant value, called a value node and labeled by the constant. An edge represents an attribute of the source object, and its target is the node representing the object or value that the attribute refers to.

Let \( N \) be an infinite set of node names and \( K \) the set of literals including the null reference and values of primitive types.

**Definition 2.2.2 (Object graph).** An object graph is a directed and labeled graph \( G = (N, E, \rho_t, \rho_v) \), where

- \( N \subseteq \mathcal{N} \) is the set of nodes, denoted by \( G.\text{node} \),
- \( E \subseteq N \times V \times N \) is the set of edges, denoted by \( G.\text{edge} \),
- \( \rho_t : N \rightarrow C \) is a partial function from nodes to types, denoted by \( G.\text{type} \),
- \( \rho_v : N \rightarrow K \) is a partial function from nodes to values, denoted by \( G.\text{value} \).
2.2. CLASS GRAPHS, OBJECT GRAPHS AND STATE GRAPHS

31

such that

1. a node is either an object node or a value node: $\text{dom}(\rho_t) \cap \text{dom}(\rho_v) = \emptyset \land \text{dom}(\rho_t) \cup \text{dom}(\rho_v) = N$,

2. labels of the outgoing edges from a node are different, and

3. all value nodes are leaves, having no outgoing edges.

An example of object graph is shown in Fig. 2.3.

Edge and Path. Let $G$ be an object graph. We write $v_1 \xrightarrow{a} v_2$ for an edge $(v_1, a, v_2) \in G.$

For a set $N \subseteq G.$node of nodes (or a single node), in($N$) and out($N$) respectively denote the sets of incoming edges to and outgoing edges from them. For a non-empty path $\alpha$, i.e. a sequence of consecutive edges, we define source($\alpha$) and target($\alpha$) to be the source and target node of $\alpha$, first($\alpha$) and last($\alpha$) the first and last edge, respectively. Notice that these notations for edges and paths are applicable to all directed and labeled graphs, not only object graphs.

2.2.3 State graphs

A state at a moment of time in the execution of an oo program consists of the existing objects, the attribute links between them, the values of data attributes, which form an object graph at that time; together with the variables and their values.

Roughly speaking, each step of the execution of the program in a state is to change the state by creating a new object, forming a new link, changing a link, or modifying a data attribute. Obviously, all these changes of the state can be considered as simple operations on the initial object graph.

However, we are interested in a small-step semantics, and we need to define semantics of changes of local variables and nested method invocations. For this, we define the notion of state graphs that introduces stacks into object graphs.

Definition 2.2.3 (State graph). A state graph is a directed and labeled graph $G = \langle N, E, \rho_t, \rho_v, t \rangle,$ where

- $N$, $\rho_t$ and $\rho_v$ are defined as in Definition 2.2.2 of object graphs,
- $E \subseteq N \times (V \cup \{\text{self}, \mathcal{S}\}) \times N$ is the set of edges, denoted by $G.$edge,
- $t \in N$ is the root of the graph, i.e. without incoming edges, denoted by $G.$root,
- starting from $t$, the $\mathcal{S}$-edges, if there are any, form a path such that except $t$ each node on the path has only one incoming edge.

We call the $\mathcal{S}$-path of $G$ the stack of the state graph and call the nodes on this path the scope nodes. When entering a new scope, a new node representing this scope is pushed onto the top of the stack, and when exiting a scope, the top node is popped out (together with its outgoing edges). A state graph in shown in Fig. 2.4. When the execution enters var $y$; var $x$; . . . ; end $x$; end $y$, it
pushes a new scope node $t_1$ onto the top of node $t_0$ with variable $y$ being attached to it; then the execution proceeds to `var $x$;...;end $x$;var $y$`, a new scope is entered and thus a new scope node $t_2$ is pushed on the top of node $t_1$ with the newly declared variable $x$ being attached to it. When `end` $x$ is executed, $t_2$ together with $x$ is popped out, then in the same way node $t_1$ will be popped out together with $y$.

A state graph represents a proper execution state of a program only if it satisfies the conditions 2 & 3 of object graphs and the following two well-formedness conditions:

1. a node is either a scope node, an object node or a value node, and
2. the source of each edge labeled by `self` is a scope node and its target is an object node.

In the rest of the paper, we always assume a state graph well-formed. Besides, a state graph is called stable if it does not contain `$-$`-edges, i.e. the stack is empty.

**Trace and graph isomorphism.** A node $v$ is accessible in $G$, denoted by $\text{access}(G,v)$, if it is reachable via a path starting from the root node, and $G$ is connected if all nodes are accessible. Given a state graph $G$, we can always get a connected subgraph by removing all the inaccessible nodes together with their associated edges. Such a subgraph of $G$ is unique, called the connected part of $G$, denoted by $G^\ast$.

The sequence of edge labels $a_1,a_2,\ldots,a_k$ uniquely determines the target node of a path from the root node $G.\text{root} \xrightarrow{a_1} \ldots \xrightarrow{a_k} v_k$, and it therefore uniquely represents an object or a value, depending on the type of the target node. We call such a sequence of edge labels a trace and ignore the difference between a path starting from the root and its trace.

In an abstract model, we do not distinguish graphs with only different choice of names of their nodes from $\mathcal{N}$, and this can be formalized by the notion of graph isomorphism. Two connected state graphs $G$ and $G'$ are isomorphic if there is a bijective function $\rho$ from $G.\text{node}$ to $G'.\text{node}$, such that

1. $\rho(G.\text{root}) = G'.\text{root}$,
2. $v_1 \xrightarrow{a} v_2 \in G.\text{edge} \iff \rho(v_1) \xrightarrow{a} \rho(v_2) \in G'.\text{edge}$, and
3. $G.\text{type}(v) = G'.\text{type}(\rho(v)) \land G.\text{value}(v) = G'.\text{value}(\rho(v))$.

Two state graphs are isomorphic if their connected parts are isomorphic. Isomorphic state graphs have the same set of traces. For simplicity, we assume the mapping $G.\text{value}$ is injective and thus all leaves nodes represent different values. From now on, we do not distinguish a value node from its value. And we assume a value node is in the state when needed, as otherwise it can always be added.

### 2.2.4 Correctly typed object graphs and state graphs

We have two kinds of types, namely class types and primitive data types. However, they are not enough if we want to reason about the type of the literal `null`, which is likely to be a value in an object or state graph. For this purpose, we introduce a special type `Null`, and assume that it is a subtype of every class type. Notice that such an assumption does not lead to multiple inheritance, since `Null` does not inherit attributes or methods from any class. We use $\mathbf{T}(\text{cn})$ to denote the type of a literal `cn`. For example, $\mathbf{T}(5) = \text{Int}$ and $\mathbf{T}(\text{null}) = \text{Null}$.

The allowable objects and states of a program, both represented by graphs, are determined by the class declarations of the program, which is represented by a graph too. For a class graph $\Gamma$, an object graph or a state graph $G$ is correctly typed with respect to $\Gamma$, or $\Gamma$-typed, if the following conditions hold.

1. The type of each object is defined in the class graph: $o \in \text{dom}(G.\text{type}) \Rightarrow G.\text{type}(o) \in \Gamma.\text{node}$.
2. Each attribute is correctly typed according to the class graph: $o \in \text{dom}(G.\text{type}) \land o \xrightarrow{a} v \in G.\text{edge} \Rightarrow \exists C, T \in \Gamma.\text{node} \bullet (o \ll C \land C \xrightarrow{a} T \in \Gamma.\text{edge} \land v \ll T)$.

Here and in the following discussion, we abuse the notation $v \ll T$ to denote that the type of an object node or a value node $v$ is a subtype of $T$, specifically:

- if $v$ is an object node, $G.\text{type}(v) \ll T$;
- if $v$ is a value node, $T(G.\text{value}(v)) \ll T$.

As an example, the object graph in Fig. 2.3 is correctly typed with respect to the class graph in Fig. 2.2(1).

An object graph or state graph $G$ is called complete if each object node $o$ in $G$ has all the actual attributes of its class, i.e. $a \in \text{Attr}(G.\text{type}(o)) \Rightarrow \exists v \xrightarrow{a} v \in G.\text{edge}$.

A state graph $G$ is a valid state of a program $\text{prog} = \Gamma \bullet \text{Main}$ if the following conditions hold.

1. $G$ is complete and correctly typed with respect to $\Gamma$.
2. The last node $t_0$ of the stack (i.e. the target of the $\bullet$-path) of $G$ records the external variables of $\text{Main}$: for each edge $t_0 \xrightarrow{x} v$ of $G$, $x$ is an external variable declared in $\text{Main}$ with some type $T$ such that $v \ll T$.

In the rest of the paper, we are only interested in correctly typed object graphs and valid state graphs.

### 2.2.5 Graph operations

We define a few basic operations on state graphs, which we are to use in the semantic definitions.

**Edge swing.** The most frequent operation for changing a state $G$ is done by an assignment $le.\alpha := e$. It causes the swing of the $a$-edge to point to the object or value of $e$. For an edge $d = v_1 \xrightarrow{a} v_2$ and a node $v$ of $G$, $\text{swing}(G, d, v) \cong G'$ such that $G'$ is the same as $G$ except that

$$G'.\text{edge} = (G.\text{edge} \setminus \{d\}) \cup \{v_1 \xrightarrow{a} v\}.$$ 

Notice that when defining a new graph, we just list the part different from the old one.

For a path $\alpha$, we use $\text{swing}(G, \alpha, v)$ for $\text{swing}(G, \text{last}(\alpha), v)$, i.e. swinging a path means swinging its last edge. See Fig. 2.5. It is allowed to swing a sequence of (more than one) paths at a time. We abuse the notation $\text{swing}()$ and define $\text{swing}(G, \alpha_0 \cdot \bar{\alpha}, v_0 \cdot \bar{v})$ inductively as $\text{swing}(\text{swing}(G, \alpha_0, v_0), \bar{\alpha}, \bar{v})$. Here and after, we use $\bar{u} \cdot \bar{w}$ to denote the concatenation of two sequences $\bar{u}$ and $\bar{w}$, and do not distinguish between an element and a singleton sequence.
CHAPTER 2. GRAPH-BASED OPERATIONAL SEMANTICS OF OO PROGRAMS

Object creation. Adding an object node is slightly tricky and we need to consider the type of the node and its attributes. Creating a new object of class C and attaching it to trace α in G is defined by

\[ \text{new}(G, C, \alpha) = \text{swing}(G', \alpha, o) \]

such that \( o \not\in G.\text{node} \), and

\[
\begin{align*}
G'.\text{node} &= G.\text{node} \cup \{o\} \\
G'.\text{edge} &= G.\text{edge} \cup \{o \xrightarrow{a} \text{init}(C, a) \mid a \in \text{Attr}(C)\} \\
G'.\text{type} &= G.\text{type} \cup \{o \mapsto C\}.
\end{align*}
\]

An example of object creation is shown in Fig. 2.6. In this example, an object of class C is created, with attributes \( a_1, \ldots, a_k \) set to their corresponding initial values. Then the path α is swung to the new object.

Stack operations. For a sequence of variables \( \vec{x} = x_1, \ldots, x_k \) and a sequence of nodes \( \vec{v} = v_1, \ldots, v_k \), \( \text{push}(G, \vec{x}, \vec{v}) \) adds a new scope with outgoing edges labeled by \( \vec{x} \) and pointing to the nodes \( \vec{v} \), accordingly:

\[ \text{push}(G, \vec{x}, \vec{v}) = G' \]

such that \( t' \not\in G.\text{node} \), and

\[
\begin{align*}
G'.\text{node} &= G.\text{node} \cup \{t'\} \\
G'.\text{edge} &= G.\text{edge} \cup \{t' \xrightarrow{x_1} v_1, \ldots, t' \xrightarrow{x_k} v_k, t' \xrightarrow{\$} G.\text{root}\} \\
G'.\text{root} &= t'.
\end{align*}
\]

As shown in Fig. 2.7, ending a scope pops the root out of the stack by simply removing it, as well as all its outgoing edges, from the graph, but the next node on the stack becomes the root.

\[ \text{pop}(G) = G' \]

such that

\[
\begin{align*}
G'.\text{node} &= G.\text{node} \setminus \{G.\text{root}\} \\
G'.\text{edge} &= G.\text{edge} \setminus \{G.\text{root} \xrightarrow{a} v \mid a \in V \cup \{\text{self}, \$\}, v \in N\} \\
G'.\text{root} &= t_{\text{next}}.
\end{align*}
\]

---

Figure 2.6: Object creation

Figure 2.7: Stack push and pop
2.3 Type System

The type system checks whether a program is type-correct (or well-typed) before executing the program. It is constructed in a bottom-up way: from the checking of expressions and commands to that of methods, class graphs and, finally, programs.

Generally, the type-correctness of an expression or a command depends on a specific class graph. For example, the command `C.new(x);x.a := 1` is well-typed only under a class graph with a class `C` which has an integer attribute `a`. In fact, a class graph itself is still not enough to decide whether an expression (or a command) is type-correct. For a variable `x`, it is always well-typed inside a local scope `var T x;...;end x`, but is likely to be ill-typed (undefined) outside. Therefore, the type-correctness of an expression also depends on its type environment, i.e. which variables are defined and in which order they are defined. We visualize such an environment as a graph, called a type context.

**Definition 2.3.1 (Type context).** A type context is a rooted, directed and labeled graph $\Delta = (N, E, t)$, where

- $N \subseteq N \cup T$ is the set of nodes, denoted by $\Delta\text{-node}$, including scope nodes $N \cap N$ and type nodes $N \cap T$,
- $E \subseteq N \times (V \cup \{self, $}$) \times N$ is the set of edges, denoted by $G\text{-edge}$, and
- $t \in N \cap N$ is the root of the graph, i.e. without incoming edges, denoted by $G\text{-root}$,

such that

1. labels of the outgoing edges from a node are different,
2. an `$-$edge is associated with two scope nodes; while an edge labeled by a variable name (or `self) starts from a scope node and ends at a type node, and
3. starting from the root $t$, the `$-$edges, if there are any, form a path such that except $t$ each node on the path has only one incoming edge.

A type context represents a snapshot of the type environment at a time of type checking, recording the types of variables (including `self) declared in all the scopes at that time. Similar to state graphs, a type context has a stack structure. When the type checking enters a new scope, a node with outgoing edges recording the variables in the new scope is pushed onto the top of the stack; when the checking exits a scope, the top node of the stack, together with its outgoing edges, is popped out, so that the type context recovers to the one exactly before entering the scope.

The notion of trace and graph isomorphism is also suitable for type contexts. Therefore, we do not distinguish type contexts differing only in the choice of names of their scope nodes from $N$. Moreover, we assume a type node is in a type context when needed, as otherwise it can always be added.

Let $\Delta$ be a type context, $t$ be one of its scope nodes and $z$ be a variable name or `self`, we define a partial function $search(\Delta, t, z)$ that searches for the trace of $z$ in $\Delta$ from $t$ node-by-node down the stack.

$$search(\Delta, t, z) \equiv \begin{cases} z & \text{if } \exists v \cdot t \xrightarrow{z} v \in \Delta\text{-edge} \\ $\text{search}(\Delta, t_1, z) & \text{otherwise, if } \exists t_1 \cdot t \xrightarrow{z} t_1 \in \Delta\text{-edge} \end{cases}$$

Notice that the recursion always terminates as there is no loop formed by `$-$edges. Based on this function, we define $\Delta(z)$ as the type of $z$ in $\Delta$:

$$\Delta(z) \equiv target(search(\Delta, \Delta\text{-root}, z))$$
where \(\Delta\) in the current scope, i.e. \(\vartriangle\)

For a type context \(\Delta\), a class graph \(\Gamma\), and a command \(c\), we use \(\Delta \vdash c : B\) to denote that \(c\) is well-typed and of type \(B\) under \(\Delta\) and \(\Gamma\). The subscript \(\Gamma\) can be omitted when there is no confusion. The type checking rules for expressions are given in Fig. 2.8.

The type checking of an expression mainly checks whether a variable is correctly used, whether a type cast is valid, and whether an attribute can be accessed under the type context. Rule \((T-\text{ATTR})\) is for the third purpose. In this rule, \(\Delta \vdash (C,a)\) means we can access the attribute \(a\) of class \(C\) in the context \(\Delta\), that is

- \(\text{visib}(C,a) = \text{public}\), or
- \(\text{visib}(C,a) = \text{protected}\) and \(\Delta(\text{self}) \ll C\), or
- \(\text{visib}(C,a) = \text{private}\) and \(\Delta(\text{self}) = C\).

### 2.3.2 Type checking of commands

To be type-correct, a command should satisfy certain requirements. For example, in an assignment \(le := e\) the types of expressions \(le\) and \(e\) should be consistent, and in a method invocation the types of actual parameters should match the signature of the method. Such kind of requirements is captured by the notion of local well-typedness. For type contexts \(\Delta\), \(\Delta'\), a class graph \(\Gamma\) and a command \(c\), we use \(\Delta \rightarrow \Delta' \vdash c\) to denote that \(c\) is locally well-typed under \(\Delta\) and \(\Gamma\), and that the type context turns to \(\Delta'\) after the checking of \(c\). Furthermore, a locally well-typed command \(c\) (under \(\Delta\) and \(\Gamma\)) is well-typed, denoted as \(\Delta \vdash c\), if its variable declarations and endings are
always matched. In these notations, the subscript $c$ always matched. In order to exclude such “bad” cases as $c$ defined in some class $C$ not matched. The extended notation

**Definition 2.3.2** *(Well-typed method)*. A method $m(S_1 \ x_1, \ldots, S_k \ x_k; T_1 \ y_1, \ldots, T_k' \ y_k') \{c\}$ defined in class $C$ of class graph $\Gamma$ is well-typed, denoted by $\vdash^c \ C :: m$, if $\Delta \vdash \ c$, where $\Delta = \{t, C, S_1, \ldots, S_k, T_1, \ldots, T_k\}$. Then, a graph is well-typed if each method defined in the graph is well-typed.

**Definition 2.3.3** *(Well-typed class graph)*. A class graph $\Gamma$ is well-typed, if for each attribute $a$ defined in some class $C$, $T(\text{init}(C, a)) \not\equiv \text{dtype}(C, a)$, and for each method $m$ defined in some class $C$, $\vdash^c \ C :: m$. 

\[ \Delta \vdash e : C \quad \Delta \vdash C \not\preceq C' \quad \frac{}{\Delta \vdash C.\text{new}(le)} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} := e} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]

\[ \Delta \vdash \bar{e} : \bar{T} \quad \Delta \vdash e : T' \quad \bar{T} \not\preceq T' \quad \frac{}{\Delta \vdash \bar{e} : \bar{T'}} \]
In a well-typed class graph $\Gamma$, each expression $e$ that occurs in the body of a method has a fixed type. We call it the declared type of $e$, denoted as $\text{dtype}_\Gamma(e)$.

At last, an oo program $\text{prog} = \Gamma \bullet \text{Main}$ is well-typed if $\Gamma$ is well-typed and $\text{Main}$ is well-typed under $\Gamma$.

**Definition 2.3.4** (Well-typed program). A program $\text{prog} = \Gamma \bullet \text{Main}$ is well-typed, where $\text{Main} = (T_1, x_1 = \text{cn}_1, \ldots, T_k, x_k = \text{cn}_k; e)$, if

- $\Gamma$ is well-typed,
- for $1 \leq j \leq k$, $T(\text{cn}_j) \leq T_j$, and
- $\Delta_{\text{init}} \vdash_\Gamma c$, where $\Delta_{\text{init}} = \{ \{ t, T_1, \ldots, T_k \}, \{ t \xrightarrow{x_1} T_1, \ldots, t \xrightarrow{x_k} T_k \}, t \}$.

We will see in the semantics that well-typed programs have certain good properties in execution. On the other hand, the execution of a program that is not well-typed would be error-prone.

### 2.4 Operational Semantics

Using the state graphs, we now simply follow the classical routine to define the evaluation of an expression and then the state transition rules.

#### 2.4.1 Evaluation of expressions

Given a state graph $G$, the evaluation of an expression $e$ returns an object node or a constant value. We use $\text{eval}(G, e)$ to denote the value of $e$, and $\text{rtype}(G, e)$ to denote the runtime type or current type of $e$ in state $G$.

In order to evaluate an expression, we should first calculate its trace. Given a state graph $G$, we use $\text{trace}(G, e)$ to denote the trace of an expression $e$. It is calculated inductively, using the function $\text{search}()$ defined in Section 2.3.

\[
\text{trace}(G, z) \triangleq \text{search}(G, G.\text{root}, z)
\]

\[
\text{trace}(G, e.a) \triangleq \text{trace}(G, e.a)
\]

\[
\text{trace}(G, (C)e) \triangleq \text{trace}(G, e)
\]

where $z$ is a variable or self. For the example graph $G_0$ in Fig. 2.4, $\text{trace}(G_0, x) = x$ and $\text{trace}(G_0, y) = \$.y$. From now on, when there is no confusion, we omit the argument $G$ in the graph operations that we have defined.

The value and the runtime type of an expression $e$ in $G$ are determined inductively as follows.

1. If $e$ is a constant value $\text{cn}$, then $\text{eval}(e) = \text{cn}$ and $\text{rtype}(e) = T(\text{cn})$.

2. If $e$ is a variable $x$ or self, $e$ can be evaluated only if $v = G(e)$ exists. If $v$ is an object node, $\text{eval}(e) = v$ and $\text{rtype}(e) = G.\text{type}(v)$, otherwise $\text{eval}(e) = G.\text{value}(v)$ and $\text{rtype}(e) = T(\text{eval}(e))$.

3. If $e$ is of the form $e'.a$, $e$ can be evaluated in $G$ only when $\text{trace}(e)$ exists in $G$. Let $v = \text{target}(\text{trace}(e))$. If $v$ is an object node, $\text{eval}(e) = v$ and $\text{rtype}(e) = G.\text{type}(v)$, otherwise $\text{eval}(e) = G.\text{value}(v)$ and $\text{rtype}(e) = T(\text{eval}(e))$.

4. If $e$ is a type cast $(C)e'$, then $\text{eval}(e) = \text{eval}(e')$ and $\text{rtype}(e) = \text{rtype}(e')$, provided that $\text{rtype}(e') \leq C$.

5. If $e$ is of the form $h(e_1, \ldots, e_k)$, $\text{eval}(e) = h(\text{eval}(e_1), \ldots, \text{eval}(e_k))$ and $\text{rtype}(e) = T(\text{eval}(e))$.

Notice that for an expression $e = e'.a$ such that $\text{eval}(e)$ exists, $\text{eval}(e')$ must be an object node with an attribute $a$. We call $\text{eval}(e')$ the parent object of $e'.a$. 

2.4.2 Type safety of expressions

We will show that well-typed expressions have good properties in evaluation. For this purpose, we need to study the consistency relation between type contexts and state graphs.

A type context \( \Delta \) and a state graph \( G \) are consistent if they have a similar structure. This is defined inductively by the following conditions, where \( z \) is either a variable \( x \) or \( x' \).

- For \( z \in var(\Delta) \), \( z \in var(G) \) and \( G(z) \triangleleft \Delta(z) \).
- For \( z \in var(G) \), \( z \in var(\Delta) \) if \( z \) occurs in \( \Delta \).
- \( \text{pop}(\Delta) \) and \( \text{pop}(G) \) are consistent, unless \( \text{size}(\Delta) = \text{size}(G) = 0 \).

By induction on the size of \( \Delta \), it is easy to verify that \( \Delta \) and \( G \) are consistent implies \( \text{size}(\Delta) = \text{size}(G) \) and \( G(z) \triangleleft \Delta(z) \) for any \( z \) occurring in \( \Delta \).

We expect to prove that a well-typed expression can be evaluated to a value, but there are the following cases of exceptions.

- **Exception 1 (null reference):** the evaluation of an expression \( e.a \) or the execution of a command \( e.m(ve, re) \) fails, if \( e \) is evaluated to null.
- **Exception 2 (illegal downcast):** the evaluation of an expression \( (C)e \) fails, if the runtime type of \( e \) is not a subtype of \( C \).

These two cases of exceptions can not be checked and avoided statically. However, the type system can help us to preclude all the other failures, i.e. the evaluation of a well-typed expression will not "go wrong".

**Theorem 2.4.1** (Type safety of expressions). Let \( \Delta \) be a type context, \( e \) be an expression, \( T \) be a type and \( G \) be a state graph consistent with \( \Delta \). If \( \Delta \vdash e : T \), then \( \text{eval}(G, e) \) exists and \( \text{rtype}(G, e) \triangleleft T \), unless one of the exception cases happens.

**Proof.** By induction on the structure of \( e \). Suppose none of the exception cases happens.

- **Case \( x \) (And similarly Case self).**
  Since \( \Delta \vdash x : T \) can only result from (T-VAR), we have \( \Delta(x) = T \). So, \( \text{target}(\text{trace}(x)) = G(x) \triangleleft \Delta(x) = T \), notice that \( \Delta \) is consistent with \( G \). Let \( v = \text{target}(\text{trace}(x)) \). If \( v \) is an object node, \( \text{eval}(x) = v \) and \( \text{rtype}(x) = G.\text{type}(v) \triangleleft T \). If \( v \) is a value node, \( \text{eval}(x) = G.\text{value}(v) \) and we still have \( \text{rtype}(x) = T(\text{eval}(x)) \triangleleft T \).

- **Case \( e.a \).**
  Since \( \Delta \vdash e.a : T \) can only result from (T-ATTR), we have \( \Delta \vdash e' : C \) and \( \text{dtype}(C, a) = T \) for some class \( C \). From the induction hypothesis, \( \text{eval}(e') \) exists and \( \text{rtype}(e') \triangleleft C \). Since Exception 1 does not happen, \( \text{eval}(e') \neq \text{null} \). So, \( \text{eval}(e') = \text{target}(\text{trace}(e')) = v' \) and \( \text{rtype}(e') = G.\text{type}(v') = C' \triangleleft C \) for some node \( v' \) and class \( C' \). From \( \text{dtype}(C, a) = T \) and \( C' \triangleleft C \), we know \( \text{dtype}(C', a) = T \), thus there exists \( v' \xrightarrow{a} v \in G.\text{edge} \) for some node \( v \) such that \( v \triangleleft T \). So, \( \text{target}(\text{trace}(e.a)) = v \). If \( v \) is an object node, \( \text{eval}(e.a) = v \) and \( \text{rtype}(e.a) = G.\text{type}(v) \triangleleft T \). If \( v \) is a value node, \( \text{eval}(e.a) = G.\text{value}(v) \) and we still have \( \text{rtype}(e.a) = T(\text{eval}(x)) \triangleleft T \), provided that Exception 2 does not happen.

- **Case \( (C)e' \).**
  Since \( \Delta \vdash (C)e' : T \) can only result from (T-UCast) or (T-DCast), we have \( T = C \), \( \Delta \vdash e' : C' \) from some \( C' \) satisfying \( C' \triangleleft C \) or \( C \not\triangleleft C' \). From the induction hypothesis, \( \text{eval}(e') \) exists and \( \text{rtype}(e') \triangleleft C' \). In the case \( C' \triangleleft C \), \( \text{rtype}(e') \triangleleft C \); and in the case \( C \not\triangleleft C' \), \( \text{rtype}(e') \triangleleft C \) still holds since Exception 3 does not happen. So we always have \( \text{eval}((C)e') = \text{eval}(e') \) and \( \text{rtype}((C)e') = \text{rtype}(e') \triangleleft C = T \).
(SKIP) \(\langle \text{skip}, G \rangle \rightarrow G\)  
(ASSIGN) \(\langle \overline{c} := \overline{e}, G \rangle \rightarrow \text{swing}(G, \text{trace}(\overline{c})), \text{eval}(\overline{e})\)  
(NEW) \(\langle C, \text{new}(\overline{e}), G \rangle \rightarrow \text{new}(C, G, \text{trace}(\overline{e}))\)  
(DCL-I) \(\langle \text{var } T \ x = \overline{e}, G \rangle \rightarrow \text{push}(G, x, \text{eval}(\overline{e}))\)  
(DCL) \(\langle \text{var } T \ x, G \rangle \rightarrow \text{push}(G, x, \text{init}(T))\)  
(EXIT) \(\langle \text{enter}(C, \overline{S}, \overline{T}, \overline{y}, e, v\overline{e}, r\overline{e}), G \rangle \rightarrow \text{push}(G, \text{self} \cdot \overline{x} \cdot \overline{y} \cdot \overline{y}^2, \text{eval}(e) \cdot \text{eval}(\overline{v\overline{e}}) \cdot \text{init}(\overline{T}) \cdot \text{po}(G, r\overline{e}))\)  
(LEAVE) \(\langle \text{leave}(\overline{y}, r\overline{e}), G \rangle \rightarrow \text{pop}(\text{swing}(G, \text{spo}(G, \overline{y}^2, r\overline{e}), \text{eval}(\overline{y})))\)  
(INVK) \(\langle r\text{type}(e) = C, m\text{type}(C, m) = (\overline{S}; \overline{T}), m\text{body}(C, m) = (\overline{x}; \overline{y}; c)\rangle \rightarrow \langle \text{enter}(C, \overline{S}, \overline{T}, \overline{y}, e, v\overline{e}, r\overline{e}); e; \text{leave}(\overline{y}, r\overline{e}), G \rangle\)  
(Seq) \(\langle c_1, G \rangle \rightarrow \langle c'_1, G' \rangle\)  
(Seq-Pri) \(\langle c_1, G \rangle \rightarrow G'\)  
(If-F) \(\langle c_1 < b \triangleright c_2, G \rangle \rightarrow \langle c_1, G \rangle\)  
(If-T) \(\langle c_1 < b \triangleright c_2, G \rangle \rightarrow \langle c_1, G \rangle, \overline{eval}(b) = \text{true}\)  
(While-F) \(\langle b \triangleright c, G \rangle \rightarrow \langle c, G \rangle, \overline{eval}(b) = \text{false}\)  
(While-T) \(\langle b \triangleright c, G \rangle \rightarrow \langle c, G \rangle, \overline{eval}(b) = \text{true}\)  
(NDet-L) \(\langle c_1, G \rangle \rightarrow \text{con} \langle c_1 \triangleright c_2, G \rangle \rightarrow \text{con}\)  
(NDet-R) \(\langle c_2, G \rangle \rightarrow \text{con} \langle c_1 \triangleright c_2, G \rangle \rightarrow \text{con}\)  

Figure 2.10: Operational semantics for commands in rCOS

- **Case \(\Delta \vdash cn : T\).** Since \(\Delta \vdash cn : T\) can only result from (T-Lrr), we know that \(T\) is a primitive data type such that \(T(cn) = T\). As a result, \(eval(cn) = cn\) and \(r\text{type}(cn) = T(cn) = T \lessdot T\).

- **Case \(\Delta \vdash h(\overline{c}) : T\).** Since \(\Delta \vdash h(\overline{c}) : T\) can only result from (T-Or), we know that \(T\) is a primitive data type, \(h : \overline{B} \rightarrow T\) and \(\Delta \vdash \overline{c} : \overline{B}\) for some primitive data types \(\overline{B}\). From the induction hypothesis, \(eval(\overline{c})\) exist and \(r\text{type}(\overline{c}) = T(eval(\overline{c})) \lessdot \overline{B}\), which implies \(T(eval(\overline{c})) = \overline{B}\) since they are primitive data types. As a result, \(eval(h(\overline{c})) = h(eval(\overline{c}))\) and \(r\text{type}(h(\overline{c})) = T(h(eval(\overline{c}))) = T \lessdot T\).

\[\square\]

### 2.4.3 Execution of commands

We define a small-step semantics for our language by giving the transition relation between configurations. A configuration \(\langle c, G \rangle\) is either

- a non-terminated one which is a pair \(\langle c, G \rangle\), representing a state \(G\) with a command \(c\) to be executed, or

- a terminated one which is simply a state \(G\), representing the state that the execution of a command terminates at.
Fig. 2.10 gives the semantic rules for the execution of commands. The rules for sequential composition, conditional choice and iteration are defined in the standard way in which an operational semantics for an imperative language is defined. The rules for non-deterministic choice allow different transitions from a configuration.

It is worth pointing out that the operator $\cap$ of non-deterministic choice is both commutative and associative. For example, the commands $(c_2 \cap c_1) \cap c_3$, $(c_1 \cap c_2) \cap c_3$ and $c_1 \cap (c_2 \cap c_3)$ have exactly the same semantic meaning, i.e. lead to the same set of transitions for any state $G$. This allows us to write a general non-deterministic command as $\cap_{j \in CI} c_j$, where $CI$ is a countable index set. Its semantics is clear: $\langle \cap_{j \in CI} c_j, G \rangle$ behaves like either $\langle c_j, G \rangle$ ($j \in CI$). And there is no problem even if $CI$ is infinite.

An assignment $\vec{e} := \vec{c}$ swings the traces of $\vec{e}$ to the values of $\vec{c}$, respectively. $C, new(\vec{e})$ creates a new initial instance of $C$ and swings the trace of $\vec{e}$ to point to the instance. A local variable declaration $\text{var } T \ x \ [\ = \ \vec{e}]$ adds the variable $x$ to a new scope by pushing it onto the stack of the state; while end $x$ pops the root out of the state. We use $init(T)$ to denote the initial value (or "zero" value) of type $T$. For example, $init(\text{Int}) = 0$, $init(\text{Bool}) = false$ and $init(C) = null$ for any class type $C$. An uninitialized variable will be temporarily set to the initial value of its declared type.

**Method invocation.** However, the semantics of a method invocation deserves some explanation because of the dynamic binding and early binding of result parameters. Intuitively, a method invocation $e.m(\vec{v}; \vec{r})$ first records the values of the actual value parameters $\vec{v}$ in the formal value parameters of $m$, and then executes the method body. After the execution, it returns the values of the formal return parameters of $m$ to the actual return parameters $\vec{r}$. However, the precise definition is trickier because of the following issues.

First, dynamic binding of the method to the runtime type of $e$ requires the look-up for the signature $\text{mtype}(C, m) = (\vec{S}; \vec{T})$ and the definition $\text{mbody}(C, m) = (\vec{x}; \vec{y}; e, \vec{c})$ of $m$. This is handled in Rule (Invk).

Then, the execution is entering the method body. In this phase, the parent object of each actual result parameter in the initial state should be recorded before it is possibly changed by the body command of the method. This is "early result parameter binding". For this, we need a sequence of auxiliary variables $\vec{y}$, which corresponds to the formal return parameters $\vec{y}$ and does not occur in the program, to record the parent objects of the actual result parameters $\vec{r}$ in the initial state. Thus, we introduce an implementation command $\text{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r})$ that sets a new scope with variables $\text{self}$, $\vec{x}$, $\vec{y}$ and $\vec{y}$ initialized by the values of $e$, $\vec{v}$, the initial values of $\vec{T}$ and the parent objects of $\vec{r}$, respectively. This is illustrated by Rule (Enter). In this rule, the function $\text{po}(G, \vec{r})$ returns the sequence of parent objects of $\vec{r}$ in $G$. It is defined by induction as follows, where $\varepsilon$ denotes the empty sequence.

$$\text{po}(G, \varepsilon) \equiv \varepsilon$$

$$\text{po}(G, re_0 \cdot \vec{r}) \equiv \begin{cases} \text{eval}(G, e) \cdot \text{po}(G, \vec{r}) & \text{if } re_0 = e.a \\ \text{null} \cdot \text{po}(G, \vec{r}) & \text{otherwise} \end{cases}$$

When the execution is leaving the body of $m$, if an actual result parameter $re$ is of the form $e.a$, the attribute $a$ of the old parent object $y^*$ of $re$ must be swung to the value of the corresponding formal parameter $y$. For this, we introduce another implementation command $\text{leave}(\vec{y}, \vec{r})$ for the return of a method invocation, whose semantics is defined in Rule (Leave) by swing and pop operations. The function $\text{spo}(G, \vec{y}, \vec{r})$ calculates the sequence of traces to be swung. It is defined by induction as follows.

$$\text{spo}(G, \varepsilon, \varepsilon) \equiv \varepsilon$$

$$\text{spo}(G, y^*_0 \cdot \vec{y}, re_0 \cdot \vec{r}) \equiv \begin{cases} y^*_0 \cdot \text{spo}(G, \vec{y}, \vec{r}) & \text{if } re_0 = e.a \\ \text{.$trace(pop(G), x) \cdot spo(G, \vec{y}, \vec{r})} & \text{if } re_0 = x \end{cases}$$
Theorem 2.4.2

We have already introduced three cases of exceptions in Section 2.4.2 which can not be avoided by type checking. If none of these exception cases occurs, the execution of a locally well-typed command will not be blocked.

Theorem 2.4.2 (Type safety of commands). Let $\Delta$, $\Delta_{\text{end}}$ be two type contexts, $c$ be a command
and $G$ be a state graph consistent with $\Delta$. If $\Delta \rightarrow \Delta_{end} \vdash c$, then

- there is at least one transition from the configuration $\langle c, G \rangle$,
- for each terminated configuration $G'$ such that $(c, G) \rightarrow G'$, $G'$ is consistent with $\Delta_{end}$, and
- for each non-terminated configuration $\langle \vec{c}', G' \rangle$ such that $(c, G) \rightarrow (\vec{c}', G')$, there is a type context $\Delta'$ consistent with $G'$ such that $\Delta' \rightarrow \Delta_{end} \vdash \vec{c}'$.

unless one of the exception cases happens.

**Proof.** By induction on the structure of $c$. Suppose none of the exception cases happens.

- **Case skip.**
  Since $\Delta \rightarrow \Delta_{end} \vdash \text{skip}$ can only result from (T-Skip), we have $\Delta_{end} = \Delta$. Thus $G$ is consistent with $\Delta_{end}$. According to (Skip), $(\text{skip}, G) \rightarrow G$, and this is the only transition from $(\text{skip}, G)$.

- **Case C.new(le).**
  Since $\Delta \rightarrow \Delta_{end} \vdash C.\text{new}(le)$ can only result from (T-New), we have $\Delta_{end} = \Delta$, $\Delta \vdash le : C'$ and $C \models C'$ for some $C'$. According to Theorem 2.4.1, $\text{eval}(le)$ exists, thus $\text{trace}(le)$ and $G' = \text{new}(G, C, \text{trace}(le))$ also exist. Here, $C \models C'$ ensures that $G'$ is also a valid state graph. According to (New), $(C.\text{new}(le), G) \rightarrow G'$, and this is the only transition from $(C.\text{new}(le), G)$. Notice that the graph operation new() does not affect scope nodes, and at most changes one of their outgoing edges: when $le = x$, it swings such an $x$-labeled edge to an object node $o$ of class $C$. Even in this case, we still have $G'(x) = \Delta(x)$ since $G'.\text{type}(G'(x)) = G'.\text{type}(o) = C \models C' = \Delta(x)$. As a result, $G'$ is consistent with $\Delta_{end}(= \Delta)$.

- **Case $\vec{le} := \vec{e}$.**
  Since $\Delta \rightarrow \Delta_{end} \vdash \vec{le} := \vec{e}$ can only result from (T-Assign), we have $\Delta_{end} = \Delta$, $\Delta \vdash \vec{le} : \vec{T}$, $\Delta \vdash \vec{e} : \vec{T}$ and $\vec{T} \models \vec{T}'$ for some $\vec{T}$ and $\vec{T}'$. According to Theorem 2.4.1, $\text{eval}(\vec{le})$, $\text{eval}(\vec{e})$ exist and $\text{rtype}(\vec{e}) \models \vec{T}$. Thus $\text{trace}(\vec{le})$ and $G' = \text{swing}(G, \text{trace}(\vec{le}), \text{eval}(\vec{e}))$ also exist. Here, the validity of $G'$ is ensured by $\text{rtype}(\vec{e}) \models \vec{T} \models \vec{T}'$. According to (Assign), $(\vec{le} := \vec{e}, G) \rightarrow G'$, and this is the only transition from $(\vec{le} := \vec{e}, G)$. Then, we need to prove $G'$ is consistent with $\Delta_{end}(= \Delta)$. For this, it is enough to show $G'(x) \models \Delta(x)$ for any variable $x$ that occurs in $\Delta$. If $x$ is not an expression in $\vec{le}$, we have $G'(x) = G(x) \models \Delta(x)$. Otherwise, suppose $x$ is the $j$-th expression of $\vec{le}$. Correspondingly, let $e_j$ be the $j$-th expression of $\vec{e}$, $T_j^x$ be the $j$-th type of $\vec{T}^e$, and $v_j$ be the node representing the value of $e_j$. We have $G'(x) = v_j \models \text{rtype}(e_j) \models T_j^x = \Delta(x)$.

- **Case var $T$ $x[= e]$.**
  We only consider the case that $x$ is initialized by $e$. The proof is similar (and even simpler) for the uninitialized case. Since $\Delta \rightarrow \Delta_{end} \vdash \text{var} T x = e$ can only result from (T-Dcl), we have $\Delta \vdash e : T', T' \models T$ and $\Delta_{end} = \text{push}(\Delta, x, T')$ for some $T'$. According to Theorem 2.4.1, $\text{eval}(e)$ exists and $\text{rtype}(e) \models T' \models T$, which means $G' = \text{push}(G, x, \text{eval}(e))$ also exists. Then, according to (Dcl-I), $(\text{var} T x = e, G) \rightarrow G'$, and this is the only transition from $(\text{var} T x = e, G)$. Notice that $G'$ is consistent with $\Delta_{end}$, since $\text{var}(G') = \{x\} = \text{var}(\Delta_{end})$, $G'(x) \models \text{rtype}(e) \models T = \Delta_{end}(x)$.

- **Case end $x$.**
  Since $\Delta \rightarrow \Delta_{end} \vdash \text{end} x$ can only result from (T-End), we have $\Delta_{end} = \text{pop}(\Delta)$, thus $\text{pop}(G)$ is consistent with $\Delta_{end}$. According to (End), $(\text{end} x, G) \rightarrow \text{pop}(G)$, and this is the only transition from $(\text{end} x, G)$. 

• Case \texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r}).

Since $\Delta \vdash \Delta_{\text{end}} \vdash \texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r})$ can only result from (T-ENTER), we have $\Delta_{\text{end}} = \texttt{push}(\Delta, \texttt{self} \cdot \vec{x} \cdot \vec{y} \cdot C \cdot \vec{S} \cdot \vec{T})$, $\Delta \vdash \vec{v} : \vec{S} \vdash \vec{S} \vdash \vec{S}$ and $\Delta \vdash \vec{r} : \vec{T}$ for some $\vec{S}$, $\vec{T}$. We also have $\texttt{rtype}(e) = C$ (thus $\texttt{eval}(e)$ exists), notice that $\texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r})$ can only be introduced by (INVR). According to Theorem 2.4.1, $\texttt{eval}(\vec{v})$, $\texttt{eval}(\vec{r})$ exist and $\texttt{rtype}(\vec{v}) \not\ll \vec{S} \ll \vec{S}$, which implies $\texttt{po}(G, \vec{r})$ also exists. As a result, $G' = \texttt{push}(G, \texttt{self} \cdot \vec{x} \cdot \vec{y} \cdot \vec{y}' \cdot \vec{v} \cdot \texttt{eval}(\vec{v}) \cdot \texttt{init}(\vec{T}) \cdot \texttt{po}(G, \vec{r})$) always exists. According to (ENTER), we have $(\texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r}), G) \rightarrow G'$, and this is the only transition from the configuration $(\texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r}), G)$. From (1) $\texttt{var}(G') = \texttt{var}(\Delta_{\text{end}}) \cup \{\vec{y}'\}$, where $\{\vec{y}'\}$ denotes the set of elements of the sequence $\vec{y}'$, (2) $\vec{y}'$ does not occur in $\Delta_{\text{end}}$. (3) $G'(\vec{s}) \not\ll \texttt{rtype}(\vec{v}) \ll \vec{S} = \Delta_{\text{end}}(\vec{x})$, and (5) $G'(\vec{y}) \ll \Delta_{\text{end}}(\vec{y})$, we conclude that $G'$ is consistent with $\Delta_{\text{end}}$.

• Case \texttt{leave}(\vec{y}, \vec{r}).

Since $\Delta \rightarrow \Delta_{\text{end}} \vdash \texttt{leave}(\vec{y}, \vec{r})$ can only result from (T-LEAVE), we have $\Delta_{\text{end}} = \texttt{pop}(\Delta)$, $\Delta \vdash \vec{y} : \vec{T}$, $\texttt{pop}(\Delta) \vdash \vec{r} \vdash \vec{T}$ and $\vec{T} \ll \vec{T}$ for some $\vec{T}$, $\vec{T}$. According to Theorem 2.4.1, $\texttt{eval}(\texttt{pop}(G), \vec{r})$ and $\texttt{eval}(\vec{y})$ exist with $\texttt{rtype}(\vec{y}) \ll \vec{T} \ll \vec{T}$. Thus $\texttt{spo}(G, \vec{y}, \vec{r})$, $G'' = \texttt{swing}(G, \texttt{spo}(G, \vec{y}, \vec{r}), \texttt{eval}(\vec{y}))$ and $G' = \texttt{pop}(G'')$ also exist. Here, the validity of $G''$ is ensured by $\texttt{rtype}(\vec{y}) \ll \vec{T}$, and $G'$ is valid since $G''$ is valid. According to (LEAVE), $(\texttt{leave}(\vec{y}, \vec{r}), G) \rightarrow G'$, and this is the only transition from $(\texttt{leave}(\vec{y}, \vec{r}), G)$. Notice that the graph operation $\texttt{swing}()$ does not affect scope nodes, and at most changes one of the their outgoing edges: when an actual result parameter $re = x$, it swings such an x-labeled edge to the node v representing the value of the corresponding formal result parameter y.

Even in this case, we still have $e \ll \texttt{rtype}(y) \ll \vec{T}$. As a result, $G''$ is consistent with $\Delta$, which implies $G'$ is consistent with $\Delta_{\text{end}}$.

• Case \texttt{c.m}(\vec{v}, \vec{r}).

Since $\Delta \rightarrow \Delta_{\text{end}} \vdash \texttt{c.m}(\vec{v}, \vec{r})$ can only result from (T-INVR), we have $\Delta_{\text{end}} = \Delta$, $\Delta \vdash e : C'$, $\Delta \vdash \vec{v} : \vec{S}$, $\Delta \vdash \vec{r} : \vec{T}$, $\texttt{rtype}(C', m) = (\vec{S}, \vec{T})$, $\vec{S} \ll \vec{S}$ and $\vec{T} \ll \vec{T}$ for some $\vec{S}$, $\vec{S} = S_1, \ldots, S_k$, $\vec{T} = T_1, \ldots, T_k$, $\vec{S} \ll \vec{T}$ and $\vec{T}$. According to Theorem 2.4.1, $\texttt{eval}(e)$ exists and $\texttt{rtype}(e) \ll C'$. Since Exception 1 does not happen, we have $\texttt{eval}(e) \neq \texttt{null}$, which means $\texttt{rtype}(e) = C$ for some class $C \ll C'$. So, $\texttt{rtype}(C, m) = \texttt{rtype}(C', m) = (\vec{S}, \vec{T})$. Suppose $\vec{mbody}(C, m) = (\vec{x} : \vec{y}, c_0)$, where $\vec{x} = x_1, \ldots, x_k$ and $\vec{y} = y_1, \ldots, y_k$, and let $e' = \texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r}) ; c_0 \texttt{. leave}(\vec{y}, \vec{r})$. According to (INVR), $(\texttt{c.m}(\vec{v}, \vec{r}), G) \rightarrow (\vec{c}, G)$, and this is the only transition from $(\texttt{c.m}(\vec{v}, \vec{r}), G)$. Then, we need to show that $e'$ is locally well-typed. This is done in three steps. (1) From (T-ENTER), we have $\Delta \rightarrow \Delta' \vdash \texttt{enter}(C, \vec{S}, \vec{T}, \vec{x}, \vec{y}, e, \vec{v}, \vec{r})$, where $\Delta' = \texttt{push}(\Delta, \texttt{self} \cdot \vec{x} \cdot \vec{y} \cdot C \cdot \vec{S} \cdot \vec{T})$. (2) Here, $m$ must be a well-typed method defined in some superclass $C''$ of $C$, i.e. $\Delta_1 \vdash c_0$ for $\Delta_1 = \{\{t_1, C'' ; S_1, \ldots, S_k, T_1, \ldots, T_k\} ; \{t_1 \xrightarrow{self} C'' ; t_1 \xrightarrow{\vec{x}} S_1, \ldots, t_1 \xrightarrow{\vec{y}} S_k, t_1 \xrightarrow{\vec{y}} T_1, \ldots, t_1 \xrightarrow{\vec{y}} T_k\}, t_1\}$). Notice that the change of the type of self to a subclass in the type context does not affect the type-correctness of a command, we have $\Delta_2 \vdash c_0$ for $\Delta_2 = \{\{t_2, C'' ; S_1, \ldots, S_k, T_1, \ldots, T_k\} ; \{t_2 \xrightarrow{self} C'' ; t_2 \xrightarrow{\vec{x}} S_1, \ldots, t_2 \xrightarrow{\vec{y}} S_k, t_2 \xrightarrow{\vec{y}} T_1, \ldots, t_2 \xrightarrow{\vec{y}} T_k\}, t_2\}$. Since $\Delta'$ is simply an extension of $\Delta_2$, $\Delta' \vdash c_0$, which implies $\Delta' = \Delta' \vdash c_0$. (3) From (T-LEAVE), $\Delta' \rightarrow \Delta' \vdash \texttt{leave}(\vec{y}, \vec{r})$, since $\Delta' \vdash \vec{y} : \vec{T}$ and $\texttt{pop}(\Delta') = \Delta$. According to (1), (2) and (3), we conclude that $\Delta \rightarrow \Delta' \vdash e'$ by using (T-Seq).

• Case \texttt{c.1; c_2}.

Since $\Delta \rightarrow \Delta_{\text{end}} \vdash \texttt{c.1; c_2}$ can only result from (T-Seq), we have $\Delta \rightarrow \Delta_{\text{mid}} \vdash \texttt{c.1}$ and $\Delta_{\text{mid}} \rightarrow \Delta_{\text{end}} \vdash \texttt{c.2}$ for some $\Delta_{\text{mid}}$. According to the induction hypothesis, there is at least one transition from $\langle \texttt{c.1}, G \rangle$. Then, according to (Seq-Pri) and (Seq), there is at least one transition from $\langle \texttt{c.1; c_2}, G \rangle$. There are two cases for each transition $\langle \texttt{c.1; c_2}, G \rangle \rightarrow \langle \vec{c}', G' \rangle$. 
consistent with \( \Delta \). Let us prove the following corollary by applying Theorem 2.4.2 repeatedly.

2.4. OPERATIONAL SEMANTICS

We say the execution of \( \Delta \) is consistent with \( \Delta_{\text{end}} \). In this case, we choose \( \Delta' = \Delta_{\text{mid}} \), so that \( \Delta' \to \Delta_{\text{end}} \mid \Delta' \). (2) If it is resulted from (Seq), \( \Delta' = \Delta' \) and \( \langle c_1, G \rangle \to \langle c_1', G' \rangle \) for some \( c_1' \).

According to the induction hypothesis, there is a type context \( \Delta' \) consistent with \( G' \) such that \( \Delta' \to \Delta_{\text{mid}} \mid \Delta' \). Then, according to (T-Seq), \( \Delta' \to \Delta_{\text{end}} \mid \Delta' \).

- Case \( c_1 \mid b \triangleright c_2 \).

Since \( \Delta \to \Delta_{\text{end}} \mid \Delta') \) can only result from (T-If), we have \( \Delta_{\text{end}} = \Delta, \mid \Delta' ) \) is block-free, and \( b \triangleright c_1 \) and \( \Delta' \). So, eval(b) equals true or false, \( \Delta \to \Delta_{\text{end}} \mid \Delta' \). (1) If eval(b) = true, according to (If-T), \( \langle c_1 \mid b \triangleright c_2, G \rangle \to \langle c_1, G \rangle \), and it is the only transition from \( \langle c_1 \mid b \triangleright c_2, G \rangle \). (2) Otherwise, according to (If-F), \( \langle c_1 \mid b \triangleright c_2, G \rangle \to \langle c_2, G \rangle \), and it is also the only transition from \( \langle c_1 \mid b \triangleright c_2, G \rangle \).

- Case \( b \mid c_1 \).

Since \( \Delta \to \Delta_{\text{end}} \mid b \mid c_1 \) can only result from (T-While), we have \( \Delta_{\text{end}} = \Delta, \mid \Delta' \) and \( \Delta \to c_1 \). As a result, eval(b) equals true or false and \( \Delta \to \Delta \mid c_1 \). (1) If eval(b) = false, according to (While-F), \( \langle b \mid c_1, G \rangle \to G \), and it is the only transition from \( \langle b \mid c_1, G \rangle \). (2) Otherwise, according to (While-T), \( \langle b \mid c_1, G \rangle \to \langle c_1, G \rangle \), and it is also the only transition from \( \langle b \mid c_1, G \rangle \). Then, according to (T-Seq), \( \Delta \to \Delta_{\text{end}} \mid b \mid c_1 \).

- Case \( c_1 \mid \bigcap \).

Since \( \Delta \to \Delta_{\text{end}} \mid c_1 \mid \bigcap \) can only result from (T-NDet), we have \( \Delta_{\text{end}} = \Delta, \mid \Delta' \) and \( \Delta \to \bigcap \). According to the induction hypothesis, there is a transition from \( \langle c_1 \mid \bigcap \rangle \), thus there is a transition from \( \langle c_1 \mid \bigcap \rangle \). Let \( con \) be a configuration such that \( \langle c_1 \mid \bigcap \rangle \to \langle con \rangle \). Notice that such a transition can only result from (NDet-L) and (NDet-R), either \( \langle c_1, G \rangle \to con \) or \( \langle c_2, G \rangle \to con \) holds. Without loss of generality, suppose \( \langle c_1, G \rangle \to con \). According to the induction hypothesis, if \( con \) is a non-terminated one \( \langle c', G' \rangle \), \( G' \) is consistent with \( \Delta_{\text{end}} \) if \( con \) is a non-terminated one \( \langle c', G' \rangle \), there is a type context \( \Delta' \) consistent with \( G' \) such that \( \Delta' \to \Delta_{\text{end}} \mid \Delta' \).

\( \square \)

Theorem 2.4.2 can be naturally extended to deal with sequences of transitions from a configuration \( \langle c, G \rangle \). Let \( \to^* \) be the reflexive and transitive closure of the transition relation \( \to \). We can prove the following corollary by applying Theorem 2.4.2 repeatedly.

Corollary 2.4.1. Let \( \Delta, \Delta_{\text{end}} \) be two type contexts, \( c \) be a command and \( G \) be a state graph consistent with \( \Delta \). If \( \Delta \to \Delta_{\text{end}} \mid \Delta \), then

- there is at least one transition from the configuration \( \langle c, G \rangle \),
- for each terminated configuration \( G' \) such that \( \langle c, G \rangle \to^* G' \), \( G' \) is consistent with \( \Delta_{\text{end}} \), and
- for each non-terminated configuration \( \langle c', G' \rangle \) such that \( \langle c, G \rangle \to^* \langle c', G' \rangle \), there is a type context \( \Delta' \) consistent with \( G' \) such that \( \Delta' \to \Delta_{\text{end}} \mid \Delta' \).

unless one of the exception cases happens.

Let \( \langle c, G \rangle \) be a non-terminated configuration. We say \( \langle c, G \rangle \) is block-free if

- \( c \) is not a non-deterministic choice and there is a transition from \( \langle c, G \rangle \), or
- \( c \) is a non-deterministic choice \( \bigcap_{j \in CI} j \), where each \( c_j \) (\( j \in CI \)) is not a non-deterministic choice, and there is a transition from \( \langle c_j, G \rangle \) for each \( j \in CI \).

We say the execution of \( \langle c, G \rangle \) always terminates if

- it never get blocked: for any non-terminated configuration \( \langle c', G' \rangle \) such that \( \langle c, G \rangle \to^* \langle c', G' \rangle \), \( \langle c', G' \rangle \) is block-free, and
it never diverge: there is a natural number \( k \) such that each sequence of \( k \) transitions from \( \langle c, G \rangle \) reaches a terminated configuration.

However, the well-typedness of a command \( c \) is not sufficient to make sure that the execution of \( \langle c, G \rangle \) always terminates.

### 2.4.5 Semantics and type safety of programs

The semantics of a program is the execution of the main command under the initial state, which is a stable state whose root records the external variable referring to their initial values. Specifically, the initial configuration of \( \text{prog} = \Gamma \bullet (T_1 x_1 = cn_1, \ldots, T_k x_k = cn_k; c) \) is \( \langle c, G_{\text{init}} \rangle \), where \( G_{\text{init}} = \langle \{t, v_1, \ldots, v_k\}, \{t \xrightarrow{z} v_j \mid 1 \leq j \leq k\}, \emptyset, \{v_j \mapsto cn_j \mid 1 \leq j \leq k\}, t \rangle \).

We now establish the consistency of the operational semantics with the type system. That is, for each terminated configuration \( \langle c, G \rangle \) such that \( \langle c, G_{\text{init}} \rangle \rightarrow \ast \langle c, G_{\text{end}} \rangle \), thus \( G_{\text{end}} \) is consistent with \( \Delta_{\text{init}} \) (see Definition 2.3.4), thus \( G_{\text{end}} \) is stable, and

- for each non-terminated configuration \( \langle c', G' \rangle \) such that \( \langle c, G_{\text{init}} \rangle \rightarrow \ast \langle c', G' \rangle \), there is a type context \( \Delta' \) consistent with \( G' \) such that \( \Delta' \vdash \Delta_{\text{init}} \vdash c' \),

unless one of the exception cases happens.

**Theorem 2.4.3 (Type safety of programs).** For a well-typed program \( \text{prog} = \Gamma \bullet (T_1 x_1 = cn_1, \ldots, T_k x_k = cn_k; c) \),

- there is at least one transition from the initial configuration \( \langle c, G_{\text{init}} \rangle \),
- for each terminated configuration \( G_{\text{end}} \) such that \( \langle c, G_{\text{init}} \rangle \rightarrow \ast G_{\text{end}}, G_{\text{end}} \) is consistent with \( \Delta_{\text{init}} \) (see Definition 2.3.4), thus \( G_{\text{end}} \) is stable, and
- for each non-terminated configuration \( \langle c', G' \rangle \) such that \( \langle c, G_{\text{init}} \rangle \rightarrow \ast \langle c', G' \rangle \), there is a type context \( \Delta' \) consistent with \( G' \) such that \( \Delta' \vdash \Delta_{\text{init}} \vdash c' \).

**Proof.** Since \( \text{prog} \) is well-typed, \( \Delta_{\text{init}} \rightarrow \Delta_{\text{init}} \vdash c \). Notice that \( \Delta_{\text{init}} \) and \( G_{\text{init}} \) are consistent, we can apply Corollary 2.4.1 and get the results.

### 2.5 Properties of OO Programs

A main motivation of the semantics is that we wish to help in reasoning about oo programs. For this it is crucial that properties can be clearly, easily and precisely thought about, described and understood. The advantage of our model in this aspect comes from intuitiveness and theoretical maturity of graphs. As shown in [59], many important properties of oo programs can easily be interpreted as assertions of state graphs. Simple but useful assertions include acyclic nodes, acyclic graphs, sink (or leaf) nodes, and reachability (credibility) of one node from another. In this section, we show how to describe properties of programs within our model without explicitly referring to locations.

**Object aliasing and confinement.** In an oo program, an accessible object is referred to by a navigation expression (or path) that is evaluated as a trace in our model. In a state, a navigation path \( e \) can represent an object that can further extend to \( e.a \) for any attribute \( a \) of the object or a sink node (or called leaf). It is a leaf, denoted by \( \text{leaf}(e) \), if in the state, \( e \) is either an object whose attributes are not defined (i.e. the evaluation of \( e.a \) fails for all attributes \( a \) of the object), an object whose class has no attribute, a null object, or a constant value of a data type.

Two navigation paths \( e_1 \) and \( e_2 \) are aliasing, denoted by \( e_1 \approx e_2 \), if their traces target at the same node. This is obviously an equivalence relation, and thus aliasing expressions share many properties. For example, they can reach the same objects, and they reach any of these objects through the same paths. Formally, for a path \( a \), \( e_1(a)e_2 \) means that the object referred to by \( e_2 \) can be reached from the object referred to by \( e_1 \) via \( a \). We have \( e \approx e_1 \land e_1(a)e_2 \Rightarrow e(a)e_2 \). We can use \( e_1 \xrightarrow{z} e_2 \) to denote that \( e_2 \) is reachable from \( e_1 \) through a non-empty path, and \( e_1 \xrightarrow{1} e_2 \) is
2.5. PROPERTIES OF OO PROGRAMS

...defined as \((e_1 \approx e_2) \lor (e_1 \mathrel{\rightarrow} e_2)\). Notice that aliasing is also a cause of cycles in a state. Formally, \(e\) is cyclic, denoted by \(\text{cyc}(e)\), if it can reach itself via a non-empty path, i.e. \(e \mathrel{\rightarrow} e\). We use \(\text{acyc}(e)\) to denote that \(e\) is acyclic.

There are more subtle and interesting graph properties, such as dominance of one node by another. Node \(v_1\) dominates node \(v_2\), denoted by \(v_1 \text{ dominates } v_2\), if every trace to \(v_2\) passes through \(v_1\). It holds for \(G\) if

\[ v_2 \notin (G \setminus \{v_1\})^*\text{node}. \]

where \(G \setminus N\) removes from \(G\) the nodes \(N\) and all their associated edges.

We can use these properties to define language mechanisms for managing aliasing and encapsulation of heap-allocated objects. Ownership \([33, 32]\) is one of them, and it provides a notion of object-level encapsulation. Each object has an owner, and it can only be accessed through its owner, i.e. it is dominated by its owner. With predicates of navigation paths, this relation can be represented as \(e_1 \text{ owns } e_2\), asserting that the object that \(e_1\) refers to owns the object that \(e_2\) refers to, if the node of \(e_1\) dominates the node of \(e_2\).

Similarly, an edge \(d\) is the bridge for node \(v\), denoted by \(d \text{ bridges } v\), if every trace to \(v\) goes through \(d\). It holds for \(G\) if

\[ v \notin (G \setminus \{d\})^*\text{node}. \]

where \(G \setminus E\) removes from \(G\) the edges \(E\). For two navigation paths \(e_1\) and \(e_2\), we can define the relation that \(e_1\) bridges \(e_2\): the last edge of \(e_1\) is the bridge for the object node \(e_2\) refers to. Then, the property of unique or alias-free references \([80, 14]\) can be specified. For a navigation path \(e\), we use \(\text{uniq}\) \(e\) to denote that \(e\) is the unique trace to its target object.

\[ \text{uniq } e \equiv \forall e' \leq e \bullet e' \text{ bridges } e \]

where \(e' \leq e\) denotes that \(e'\) is a (non-empty) prefix of \(e\).

Separation of graphs. For a connected state graph \(G\), let \(G.\text{store}\) be the subgraph, called the store of \(G\), that contains the nodes on the \(\mathcal{S}\)-path of \(G\) and their outgoing edges. The subgraph obtained from \(G\) by removing the edges of the store (and the nodes that become isolated because of the removal of these edges) is called the heap of \(G\), denoted by \(G.\text{heap}\). Notice that \(G = G.\text{store} \cup G.\text{heap}\), and \(G.\text{store}.\text{edge} \cap G.\text{heap}.\text{edge} = \emptyset\).

The separation logic \([93, 84]\) can be interpreted in our model. A state \(G\) is a separating composition of two graphs \(G_1\) and \(G_2\), denoted by \(G = G_1 \star G_2\), if \(G = G_1 \cup G_2\), \(G_1.\text{store} = G_2.\text{store}\) and \(G_1.\text{heap}.\text{edge} \cap G_2.\text{heap}.\text{edge} = \emptyset\). The separating conjunction \(q_1 \star q_2\), asserting that the heap graph can be split into two object graphs for which \(q_1\) and \(q_2\) hold respectively, is defined as

\[ [q_1 \star q_2] G \equiv \exists G_1, G_2 \bullet G = G_1 \star G_2 \land [q_1] G_1 \land [q_2] G_2. \]

For example, assume that \(q\) is an invariant of a class \(C\). To be sure that a method (that possibly overrides a method of \(C\)) of an object of a subclass \(D\) of \(C\) preserves this invariant, the assertion

\[ \{q \star \text{true}\} \text{mbody}(D, m) \{q \star \text{true}\} \]

is checked. Notice that \(q\) only mentions fields of \(C\), and the separation is to divide the state of the object into the attributes inherited from \(C\) and those newly declared in \(D\).

Chen and Sanders \([28]\) propose a pointer logic based on a mixed model of graphs and functions, that extends separation logic with more flexible relational compositions. Our graphs are simpler, but can also define those compositional relations such as the relation \(G_1\) access \(G_2\), which asserts that at least one node of \(G_1\) can access some node of \(G_2\).

Hoare and O’Hea\nren \([60]\) propose a unification of the ideas of separation in CSP and Concurrent Separation Logic \([15]\). We can also write properties by the idea of trace separation, as traces and nodes are unified in our model.
2.6 Summary

This chapter presents a graph-based type system and operational semantics for an oo formal language. The type system checks whether an oo program is well-typed according to its class structure, while the operational semantics defines the execution of the program through steps of state transitions. The operational semantics is proved consistent with the type system. That is, the execution of a well-typed program will not get blocked, unless certain exceptional cases occur. The type system and operational semantics is only based on simple graph notations and basic graph operations. In this way, it provides a clarification of a variety of oo concepts, improving people’s understanding of the states and behaviors of oo programs.

Another advantage of the operational semantics is that it is location independent. In this sense, it is more abstract compared with most existing oo semantic theories that refer to addresses or locations. In addition, the operational semantics is proved useful in theorem proving. It has been implemented in the theorem prover Isabelle [73, 50] to verify properties on executions of oo programs. In this next chapter, we will apply the operational semantics to the development of a graph-based refinement calculus for oo design. The refinement calculus agrees with all the refinement rules that are based on the original denotational semantics of the rCOS language [54], and this indicates the correctness of both the operational and denotational semantics.
Chapter 3

Graph-Based Structure Refinement of OO Programs

In this chapter, we study the notion of structure refinement of OO programs, based on the graph model and operational semantics provided in Chapter 2. Our objective is to develop a sound and complete refinement calculus for OO development.

As its name indicates, structure refinement is the refinement of the class structure so that the resulting class structure is able to substitute the original one in any context [54]. For this, the resulting class structure should provide at least as many and as good services or functionalities as the original one. Unlike other notions of refinement such as program refinement or functional refinement, structure refinement aims at investigating the relation between the change of the structure, i.e. the classes and their attributes, and the change of the functionality, i.e. the methods. Such a relation is important in the development of an OO program, but unfortunately it is often ignored in existing OO semantic theories, e.g. [83, 26, 66].

With the graph notations presented in Chapter 2, we formalize a structure refinement as a transformation $\rho$ from one class graph $\Gamma_1$ to another class graph $\Gamma_2$, so that the class structure $\Gamma_2$ defines provides better services or functionalities than the class structure $\Gamma_1$ defines. By “better”, we mean that each service, in terms of a method $m()$, provided by the class structure that $\Gamma_1$ defines is also provided by the class structure that $\Gamma_2$ defines, and that the service $m()$ provided by the latter class structure is less non-deterministic than it is provided by the former class structure. Specifically, there is a derived transformation $\rho_s$ from possible states of $\Gamma_1$ to those of $\Gamma_2$, such that

- for any state $G_1$ of $\Gamma_1$ and $G_2$ of $\Gamma_2$ where $G_1$ can be transformed into $G_2$ by $\rho_s$, if $G_1$ is transformed into another state $G'_1$ of $\Gamma_1$ through the execution of a method invocation $z.m()$ (under $\Gamma_1$), $G_2$ can be transformed into another state $G'_2$ of $\Gamma_2$ through the execution of $z.m()$ (under $\Gamma_2$) and $G'_2$ can be obtained from $G'_1$ by $\rho_s$.

This has been shown by the commute diagram in Figure 1.2 (in Chapter 1).

It is clear that a class structure provides a set of services in terms of its methods. However, a class structure does not need to provide all its methods as its services or functionalities. For the consideration of encapsulation, a class structure, or even a single class, is likely to have a part of its methods as the functional methods that contribute to its functionalities. By contrast, the other methods are defined for supporting the functional methods. The case is similar for classes. A class structure usually exposes a part of its classes as functional classes that can be accessed from outside, while the other classes are declared for supporting purposes. To formalize such information, we introduce the notion of interface. That is, an interface $I = \langle IC, IM \rangle$ of a class graph $\Gamma$ annotates the set of functional classes $IC$ and functional methods $IM$ of $\Gamma$. Besides, in a structure refinement, we assume that the refined class graph $\Gamma_2$ and the original class graph $\Gamma_1$ have a common interface $I$. This ensures that the class structure defined by $\Gamma_2$ always provides as many services as the class structure defined by $\Gamma_1$. 
To support stepwise development of oo programs, we develop a calculus of structure refinement that consist of a few groups of refinement rules.

- The first group of rules study structure refinements that expand the class structure. For example, we can add or rename a class, move an attribute to a superclass or decompose a data attribute. This kind of refinements is the most common ones we meet and they are useful for decomposition and incremental design.

- The second group of rules study structure refinements that compress the class structure. These rules allow us to combine different classes into one, or remove classes and attributes that are redundant, i.e. not actually accessed by the interface. They are useful for composition and abstraction.

- The third group of rules are auxiliary rules for transforming the methods while not changing the structure of classes. These rules make it convenient to add or rename methods, or copy methods to superclasses.

- The last group of rules are special rules for eliminating polymorphism. These rules enable us to collapse inheritance relations with method overriding.

Notice that each of the rules is just a simple transformation on a class graph, for example to add a new class or to move an attribute. However, they can realize various kinds of refinements when arranged properly and applied step by step.

For the soundness of the refinement calculus, we need to prove that each rule transforms a class graph $\Gamma_1$ to a refined one $\Gamma_2$. Our approach is to seek for a certain simulation relation between the execution of commands of $\Gamma_1$ and $\Gamma_2$. With such a simulation, it is straightforward to verify the structure refinement. For the completeness of the refinement calculus, however, we are not to prove that every structure refinement can be realized by the application of refinement rules. Because structure refinement is a semantic property while each rule is only a syntactic transformation, it is infeasible to cover all cases of structure refinement with a finite number of rules. Instead, we are going to investigate notions of relative completeness, with respect to structure transformation and normal form. A structure transformation is a transformation between two class graphs with good syntactic correspondences, while a normal form captures the essence structure of an oo program that contains no redundant classes, attributes or methods. We prove that every structure transformation can be realized by the first group of refinement rules and that every class graph can be transformed into a normal form through the application of refinement rules. In addition, the refinement rules and their soundness and completeness results, which are proved based on the operational semantics presented in Chapter 2, agree with those proved based on the original denotational semantics of rCOS [54]. This also gives the justification of the correctness of both semantic models.

Section 3.1 formalizes the basic notions of interface, structure refinement and simulation, where simulation is shown a sufficient condition of structure refinement. After that, we present the group of refinement rules for class graph expansion in Section 3.2. We also define the notion of structure transformation and study the completeness of these rules with respect to structure transformation. In Section 3.3, we provide the group of refinement rules for class graph compression. And in Section 3.4, we present the auxiliary rules for changing the methods and the special rules for eliminating polymorphism. We also define the notion of normal form and study the completeness of the refinement calculus with respect to normal form. Finally, we discuss the practical importance of the refinement calculus in Section 3.5.

### 3.1 Interface and Structure Refinement

As we have illustrated in Chapter 2, a class graph $\Gamma$ defines a class structure which contains a set of classes and a set of methods in these classes. However, the class structure defined by $\Gamma$ does not need to provide all of them to outside, e.g. to main methods, for direct access. Generally, the
services the class structure provides are a part of its classes and a part of methods defined in these classes, annotated through an interface. They are also called functional classes and functional methods of the class graph \( \Gamma \), respectively. By contrast, other classes and methods are supporting classes and supporting methods and they are not visible from outside. Notice that the notion of interface we are studying in this chapter is the interface especially for structure refinement. It is different from the notion of interface in oo programming languages such as Java.

**Definition 3.1.1 (Interface).** An interface \( I \) is a pair \( \langle IC, IM \rangle \), where

- \( IC \) is a set of class names, and
- \( IM \) is a set of methods, each of which has the form \( C :: m \) with \( C \in IC \).

For two interfaces \( I = \langle IC, IM \rangle \) and \( I' = \langle IC', IM' \rangle \), the predicate \( I \subseteq I' \) means \( IC \subseteq IC' \) \& \( IM \subseteq IM' \).

For a class graph \( \Gamma = \langle N, E, M \rangle \) and an interface \( I = \langle IC, IM \rangle \), we say \( I \) is an interface of \( \Gamma \) if \( IC \subseteq N \), \( IM \subseteq M \) and the parameter types of each method \( C :: m \in IM \) of \( \Gamma \) only consists of primitive data types and class types in \( IC \). Notice that a class graph may expose different (sub-)sets of classes and methods as its interface according to different needs. Since we do not consider method overloading (but only method overriding), it is fair to assume that methods in an interface of a class graph \( \Gamma \) have distinct names, except those with overriding relations in \( \Gamma \).

An interface \( I = \langle IC, IM \rangle \) of a class graph \( \Gamma \) restricts the set of classes and methods that can be directly accessed from the main method. Specifically, with such an interface, the main command or a segment of the main command \( c \) should be locally well-typed and should NOT contain

- any occurrence of a supporting class name \( D \not\in IC \), for example \( D.new(e) \),
- any call of a supporting method \( D :: m \not\in IM \), or
- any expression of the form \( e.a \). Recall that the main method is not allowed to access the attributes of classes directly.

A command satisfying the above conditions is called a valid command with respect to \( \Gamma \) and \( I \).

The execution of a program with interface \( I \) starts from a state graph that

- does not contain any object, and
- contains only one scope with a set of variables, including those of primitive data types whose values are constants of these data types and those of functional class types whose values are null.

Such a state graph is called an initial state with respect to \( I \). From such an initial state \( G_0 \), a valid command \( c \) with respect to \( \Gamma \) and \( I \) will be executed. So, the set of states achieved during such execution are all the states that the execution of a program with class graph \( \Gamma \) and interface \( I \) may arrive at. It is thus called the state space of \( \Gamma \) with respect to \( I \), denoted as \( S_I(\Gamma) \). An important subset of the state space involves those states the execution of a valid command may terminate at. It is called the main state space of \( \Gamma \) with respect to \( I \), denoted as \( MS_I(\Gamma) \). They are formally defined as follows.

\[
S_I(\Gamma) \equiv \{ G \text{ is a } \Gamma\text{-typed state graph } | \ (c,G_0) \rightarrow^* G \text{ or } (c,G_0) \rightarrow^* (c',G) \text{ for some initial state } G_0, \text{ valid command } c \text{ with respect to } \Gamma \text{ and } I, \\
\text{and command } c' \}
\]

\[
MS_I(\Gamma) \equiv \{ G \text{ is a } \Gamma\text{-typed state graph } | \ (c,G_0) \rightarrow^* G \text{ for some initial state } G_0, \text{ valid command } c \text{ with respect to } \Gamma \text{ and } I \}
\]

Notice that each state \( G \) in the state space \( S_I(\Gamma) \) is complete, since an object is always created with all its attributes. In addition, if the execution of a valid command \( c \) from a state \( G \in MS_I(\Gamma) \) terminates at a state \( G' \), \( G' \) is also a state in \( MS_I(\Gamma) \).
For two class graphs \( \Gamma_1 = \langle N_1, E_1, M_1 \rangle \), \( \Gamma_2 = \langle N_2, E_2, M_2 \rangle \) and an interface \( I = \langle IC, IM \rangle \), we say \( I \) is a common interface of \( \Gamma_1 \) and \( \Gamma_2 \) if \( I \) is an interface of both \( \Gamma_1 \) and \( \Gamma_2 \) and each method \( C :: m \in IM \) has the same signature in \( \Gamma_1 \) and \( \Gamma_2 \). Obviously, if \( I \) is a common interface of \( \Gamma_1 \) and \( \Gamma_2 \) and also a common interface of \( \Gamma_2 \) and \( \Gamma_3 \), \( I \) is a common interface of \( \Gamma_1 \) and \( \Gamma_3 \). So, the notion of common interface is also applicable for more than two class graphs.

With the notion of interface, we can formally define structure refinement between class graphs, through a relation between their main state spaces. We call a relation \( \rho \) from \( \text{Set}_1 \) to \( \text{Set}_2 \) total if it covers every element of \( \text{Set}_1 \).

**Definition 3.1.2 (Structure refinement).** Let \( \Gamma_1 = \langle N_1, E_1, M_1 \rangle \) and \( \Gamma_2 = \langle N_2, E_2, M_2 \rangle \) be two class graphs with a common interface \( I = \langle IC, IM \rangle \). \( \Gamma_2 \) is an \( I \)-interfaced structure refinement of \( \Gamma_1 \), denoted as \( \Gamma_1 \subseteq_I \Gamma_2 \), if there is a total relation \( \rho \) from \( \mathcal{MS}_I(\Gamma_1) \) to \( \mathcal{MS}_I(\Gamma_2) \) through which \( \Gamma_2 \) corresponds to \( \Gamma_1 \) in the invocation of functional methods.

Here, we say \( \Gamma_2 \) corresponds to \( \Gamma_1 \) through \( \rho \) in the invocation of functional methods, if for any well-typed invocation \( c_0 = z.m(\vec{x}; \vec{y}) \) of a method \( C :: m \in IM \), any states \( G_1 \) and \( G_2 \) such that \( \rho_G(G_1, G_2) \) and the execution of \( \langle c_0, G_1 \rangle \) always terminates,

1. the execution of \( \langle c_0, G_2 \rangle \) always terminates, and

2. for any state \( G_2' \) such that \( \langle c_0, G_2 \rangle \xrightarrow{*} G_2' \), there is a state \( G_1' \) such that \( \rho_G(G_1', G_2') \) and \( \langle c_0, G_1' \rangle \xrightarrow{*} G_1' \).

The above definition says that the class structure defined by the refined class graph \( \Gamma_2 \)

- provides at least as many services to the environment as the class structure defined by the original class graph \( \Gamma_1 \), that is, for any functional method \( C :: m \) defined in \( \Gamma_1 \), there is a corresponding functional method \( C :: m \) defined in \( \Gamma_2 \) with the same signature, and

- provides as good services to the environment as the class structure defined by the original class graph \( \Gamma_1 \), that is, the execution of any functional method \( C :: m \) defined in \( \Gamma_2 \) satisfies all properties of the execution of its original method \( C :: m \) defined in \( \Gamma_1 \). The two conditions in the definition actually mean that the behavior of the former method is less non-deterministic than the behavior of the latter one.

It is straightforward to verify that structure refinement is

- reflexive: \( \Gamma \subseteq_I \Gamma \) for any class graph \( \Gamma \) and any of its interfaces \( I \), and

- transitive: \( \Gamma_1 \subseteq_I \Gamma_2 \) and \( \Gamma_2 \subseteq_I \Gamma_3 \) implies \( \Gamma_1 \subseteq_I \Gamma_3 \) for any class graphs \( \Gamma_1, \Gamma_2, \Gamma_3 \) and interface \( I \).

In addition, if a class graph \( \Gamma_1 \) is refined by another class graph \( \Gamma_2 \) with respect to an interface \( I \), \( \Gamma_2 \) is also a refinement of \( \Gamma_1 \) with respect to any interface \( I' \) of \( \Gamma_1 \) such that \( I' \subseteq I \). We use \( \equiv \) to denote the equivalence relation between class graphs in terms of refinement, i.e. \( \Gamma_1 \equiv_I \Gamma_2 \) means \( \Gamma_1 \subseteq_I \Gamma_2 \) and \( \Gamma_2 \subseteq_I \Gamma_1 \).

### 3.1.1 A sufficient condition of structure refinement

In certain cases of structure refinement, the refined class graph “simulates” the original class graph well in the execution of commands.

**Definition 3.1.3 (Simulation).** Let \( \Gamma_1 \) and \( \Gamma_2 \) be two class graphs with a common interface \( I = \langle IC, IM \rangle \), \( \rho_\Gamma \) be a relation from \( \mathcal{S}_I(\Gamma_1) \) to \( \mathcal{S}_I(\Gamma_2) \) whose restriction to \( \mathcal{MS}_I(\Gamma_1) \times \mathcal{MS}_I(\Gamma_2) \) is a total relation, and \( \rho_\Gamma \) be a partial function from commands (including implementation commands) definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_\Gamma(c_0) = c_0 \) for any well-typed method call \( c_0 = z.m(\vec{x}; \vec{y}) \) of a method \( C :: m \in IM \). \( \Gamma_2 \) is a simulation of \( \Gamma_1 \) with respect to \( I \), \( \rho_\Gamma \) and \( \rho_\Gamma \), if the following conditions hold for any command \( c \) where \( \rho_\Gamma(c) \) exists, states \( G_1 \) and \( G_2 \) with \( \rho_\Gamma(G_1, G_2) \).
1. \( \langle c, G_1 \rangle \) is block-free implies \( (\rho_c(c), G_2) \) is block-free.

2. For any transition \( (\rho_c(c), G_2) \rightarrow G'_2 \) where \( G'_2 \in S_I(\Gamma_2) \), there is a state \( G'_1 \) such that \( \rho_c(G'_1, G'_2) \) and \( \langle c, G_1 \rangle \rightarrow G'_1 \).

3. For any transition \( (\rho_c(c), G_2) \rightarrow \langle c', G'_2 \rangle \) where \( G'_2 \in S_I(\Gamma_2) \), there is a configuration \( \langle c', G'_1 \rangle \) such that \( \rho_c(c') = c' \), \( \rho_c(G'_1, G'_2) \) and \( \langle c, G_1 \rangle \rightarrow \langle c', G'_1 \rangle \).

In order to verify that a class graph \( \Gamma_2 \) is a structure refinement of another class graph \( \Gamma_1 \), it is sufficient, but not necessary, to show that \( \Gamma_2 \) is a simulation of \( \Gamma_1 \).

**Theorem 3.1.1.** If \( \Gamma_2 \) is a simulation of \( \Gamma_1 \) with respect to \( I, \rho_s \) and \( \rho_c, I = \langle IC, IM \rangle \) is a common interface of \( \Gamma_1 \) and \( \Gamma_2 \). Let \( \rho^c_\Gamma \) be the restriction of \( \rho^c \) on \( MS_I(\Gamma_1) \). We show that \( \rho^c_\Gamma \) satisfies the two conditions in Definition 3.1.2. Let \( c_0 = z.m(\vec{x}, \vec{y}) \) be a well-typed method call of a method \( C :: m \in IM \). \( G_1 \) and \( G_2 \) be two states such that \( \rho^c_\Gamma(G_1, G_2) \), which implies \( \rho^c_\Gamma(G_1, G_2) \), and the execution of \( \langle c_0, G_1 \rangle \) always terminates, i.e. never get blocked or diverge. Notice that \( \rho^c_\Gamma(c_0) = c_0 \).

1. For any execution \( \langle c_0, G_2 \rangle \rightarrow \langle c'_1, G'_2 \rangle \rightarrow \ldots \rightarrow \langle c'_j, G'_2 \rangle \) where \( G'_2 \in S_I(\Gamma_2) \). According to Condition 3 in Definition 3.1.3, there is an execution \( \langle c_0, G_1 \rangle \rightarrow \langle c_1, G'_1 \rangle \rightarrow \ldots \rightarrow \langle c'_j, G'_1 \rangle \) such that \( \rho^c(c') = c'' \) and \( \rho^c(G'_1, G'_2) \). Since the execution of \( \langle c_0, G_1 \rangle \) never gets blocked, \( \langle c'_j, G'_1 \rangle \) is block-free. Thus the execution of \( \langle c_0, G_2 \rangle \) never gets blocked. On the other hand, since the execution of \( \langle c_0, G_1 \rangle \) never diverges, there is a natural number \( k \) such that any sequence of \( k \) transitions from \( \langle c_0, G_1 \rangle \) reaches a terminated configuration. According to Condition 3 in Definition 3.1.3, any sequence of \( k \) transitions from \( \langle c_0, G_2 \rangle \) also reaches a terminated configuration. Thus the execution of \( \langle c_0, G_2 \rangle \) never diverges.

2. Suppose \( G'_2 \) is a state such that \( \langle c_0, G_2 \rangle \rightarrow con_1 \rightarrow \ldots \rightarrow con_j = \langle c''_j, G''_2 \rangle \rightarrow G'_2 \). According to the above analysis, there is a sequence \( \langle c_0, G_1 \rangle \rightarrow con'_1 \rightarrow \ldots \rightarrow con'_j = \langle c''_j, G''_1 \rangle \) such that \( \rho^c(c') = c'' \) and \( \rho^c(G''_1, G''_2) \). Notice that \( G_2 \in MS_I(\Gamma_2) \) and \( c_0 \) is valid with respect to \( \Gamma_2 \) and \( I \). We have \( G'_2 \in MS_I(\Gamma_2) \subseteq S_I(\Gamma_2) \). According to Condition 2 in Definition 3.1.3, there is a state \( G'_1 \) such that \( \rho^c(G'_1, G'_2) \) and \( \langle c'_j, G''_1 \rangle \rightarrow G'_1 \), thus \( \langle c_0, G_1 \rangle \) on \( G'_1 \). Notice that \( G_1 \in MS_I(\Gamma_1) \) and \( c_0 \) is valid with respect to \( \Gamma_1 \) and \( I \). We have \( G'_1 \in MS_I(\Gamma_1) \). As a result, \( \rho^c_\Gamma(G'_1, G'_2) \).

A good case of simulation is that two class graphs can simulate each other in the execution of commands.

**Definition 3.1.4** (Bisimulation). Let \( \Gamma_1 \) and \( \Gamma_2 \) be two class graphs with a common interface \( I = \langle IC, IM \rangle \), \( \rho^c \) be a relation from \( S_I(\Gamma_1) \) to \( S_I(\Gamma_2) \) whose restriction to \( MS_I(\Gamma_1) \times MS_I(\Gamma_2) \) is a total and surjective relation, and \( \rho^c \) be an injective partial function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho^c(c_0) = c_0 \) for any well-typed method call \( c_0 = z.m(\vec{x}, \vec{y}) \) of a method \( C :: m \in IM \). \( \Gamma_2 \) is a bisimulation of \( \Gamma_1 \) with respect to \( I, \rho^c \) and \( \rho^c \), if the following conditions hold for any command \( c \) where \( \rho^c(c) \) exists, states \( G_1 \) and \( G_2 \) with \( \rho^c(G_1, G_2) \).

1. \( \langle c, G_1 \rangle \) is block-free if and only if \( \langle \rho^c(c), G_2 \rangle \) is block-free.

2. For any transition \( \langle c, G_1 \rangle \rightarrow G'_1 \) where \( G'_1 \in S_I(\Gamma_1) \), there is a state \( G'_2 \) such that \( \rho^c(G'_1, G'_2) \) and \( \langle \rho^c(c), G_2 \rangle \rightarrow G'_2 \).

3. For any transition \( \langle c, G_1 \rangle \rightarrow \langle c', G'_1 \rangle \) where \( G'_1 \in S_I(\Gamma_1) \), there is a configuration \( \langle c'', G'_2 \rangle \) such that \( c'' = \rho^c(c'), \rho^c(G'_1, G'_2) \) and \( \langle \rho^c(c), G_2 \rangle \rightarrow \langle \rho^c(c'), G'_2 \rangle \).

4. For any transition \( \langle \rho^c(c), G_2 \rangle \rightarrow G'_2 \) where \( G'_2 \in S_I(\Gamma_2) \), there is a state \( G'_1 \) such that \( \rho^c(G'_1, G'_2) \) and \( \langle c, G_1 \rangle \rightarrow G'_1 \).
5. For any transition $\langle \rho(c), G_2 \rangle \rightarrow \langle c', G'_2 \rangle$ where $G'_2 \in S_I(\Gamma_2)$, there is a configuration $\langle c', G'_1 \rangle$ such that $\rho(c') = c'$, $\rho_d(G'_1, G'_2)$, and $\langle c, \Gamma_1 \rangle \rightarrow \langle c', \Gamma'_1 \rangle$.

As its name indicates, a bisimulation between two class graphs implies they simulate each other. As a result, bisimulation is a sufficient (but not necessary) condition of equivalence in term of structure refinement.

**Theorem 3.1.2.** If $\Gamma_2$ is a bisimulation of $\Gamma_1$ with respect to $I$, $\rho_e$ and $\rho_c$, $\Gamma_1 \equiv_I \Gamma_2$.

**Proof.** $\Gamma_2$ is a bisimulation of $\Gamma_1$ with respect to $I$, $\rho_e$ and $\rho_c$ implies

- $\Gamma_2$ is a simulation of $\Gamma_1$ with respect to $I$, $\rho_e$ and $\rho_c$, and
- $\Gamma_1$ is a simulation of $\Gamma_2$ with respect to $I$, $\rho_e^{-1}$ and $\rho_c^{-1}$.

According to Theorem 3.1.1, we have $\Gamma_1 \preceq_I \Gamma_2$ and $\Gamma_2 \preceq_I \Gamma_1$, i.e. $\Gamma_1 \equiv_I \Gamma_2$.  

### 3.2 Structure Refinement for Graph Expansion

In this section, we study the class graph transformations that are used for class decomposition. Such a transformation shows how we can refine an oo program by expanding its class structure, without weakening its capability of providing functional services. We first identify a kind of class graph transformation where the original class graph and result class graph have good correspondences in their structures, i.e. nodes (classes) and edges (attributes and inheritance). We call such a transformation a **structure transformation**.

**Definition 3.2.1 (Structure transformation).** Let $\Gamma_1 = \langle N_1, E_1, \emptyset \rangle$ and $\Gamma_2 = \langle N_2, E_2, \emptyset \rangle$ be two class graphs, and $IC$ be a subset of their common classes. A mapping $\rho$ from $\Gamma_1$ to $\Gamma_2$ is a structure transformation with respect to $IC$, denoted by $\rho_{\mid IC}$, if the following conditions hold.

1. The restriction $\hat{\rho}$ of $\rho$ to the nodes is an injective mapping from $N_1$ to $N_2$, satisfying $\hat{\rho}(C) = C$ for each $C \in IC$.

2. The restriction $\hat{\rho}$ 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 $\Gamma_2$, and maps each data attribute $(C, a, B)$ in $E_1$ to a nonempty set of paths in $\Gamma_2$, each of which starts from $\hat{\rho}(C)$ and ends at a data type.

With Condition 2 in the above definition, we can decompose the mapping $\rho$ into its two restrictions

- the restriction of $\rho$ to the relational edges, denoted by $\rho_r$, and
- the restriction of $\rho$ to the data attributes, denoted by $\rho_d$.

So, a structure transformation $\rho$ is a tuple $\langle IC, \hat{\rho}, \rho_r, \rho_d \rangle$, characterizing the correspondence relations of nodes and edges between two class structures. We would expect that it “becomes” a structure refinement when we add methods into these class structures. For this, the structure transformation should satisfy certain typing conditions [42] or algebraic conditions [38].

**Definition 3.2.2.** A structure transformation $\rho_{\mid IC}$ from $\Gamma_1$ to $\Gamma_2$ is well-typed if it satisfies the following conditions.

1. For each inheritance edge $(C, \triangleright, D)$ in $\Gamma_1$, $\rho_r(C, \triangleright, D)$ is a path containing only inheritance edges.

2. For each association edge $(C, a, D)$ in $\Gamma_1$, the last edge of the path $\rho_r(C, a, D)$ is also an association edge.

3. For two different relational edges $(C_1, a_1, D_1)$ and $(C_2, a_2, D_2)$ in $\Gamma_1$, the path $\rho_r(C_1, a_1, D_1)$ is not a suffix of the path $\rho_r(C_2, a_2, D_2)$. 

3.2. STRUCTURE REFINEMENT FOR GRAPH EXPANSION

4. For two data attributes \((C_1, a_1, B_1)\) and \((C_2, a_2, B_2)\) in \(\Gamma_1\), and two paths \(\alpha_1 \in \rho_a(C_1, a_1, B_1)\) and \(\alpha_2 \in \rho_a(C_2, a_2, B_2)\), \(\alpha_1\) is not a suffix of \(\alpha_2\) unless \(\alpha_1 = \alpha_2, C_1 = C_2\) and \(a_1 = a_2\) (thus \(B_1 = B_2\) too).

5. For each data attribute \((C, a, B)\) in \(\Gamma_1\), there is a surjective operation \(h : B_1 \times \ldots \times B_k \rightarrow B\) such that the initial value of the attribute can be calculated by \(h\) from the initial values of its image attributes \(\rho_a(C, a, B)\): \(\text{init}(C, a) = h(\text{init}(D_1, a_1), \ldots, \text{init}(D_k, a_k))\), where \(\rho_a(C, a, B) = \{\hat{\rho}(C) \rightarrow \ldots \rightarrow D_j \xrightarrow{a_j} B_j \mid 1 \leq j \leq k\}\).

A structure transformation from \(\Gamma_1\) to \(\Gamma_2\) in fact defines an implementation of the classes, their attributes and associations in \(\Gamma_1\) by those of \(\Gamma_2\). A single inheritance relation is implemented by a number of steps of inheritance, a single association attribute or edge in \(\Gamma_1\) can be realized by a path, and a data attribute can be a set of paths in \(\Gamma_2\). These are captured by Conditions 1 to 5 of the well-typed structure transformations. Condition 1 requires an inheritance not be replaced by associations, while Condition 2 implies that association should not be implemented by inheritance either. The falsification of Condition 3 (similarly for Condition 4) implies that association should not be implemented by inheritance 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 \(\rho_a(C_1, a_1, D_1)\) is a suffix of \(\rho_a(C_2, a_2, D_2)\), 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 \(\hat{\rho}(D_2)\). 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 \(\Gamma_1\) can be “computed” by an expression of the data attributes of the classes in \(\Gamma_2\) that are the decomposition of the original class by the transformation. Conditions 4 and 5 allow the decomposition of a single attribute into a tree of classes and attributes.

Then, we would like to show that a well-typed structure transformation indeed “leads to” a structure refinement.

**Proposition 3.2.1.** Let \(\rho_{[IC]}\) be a well-typed structure transformation from \(\Gamma_1 = (N_1, E_1, \emptyset)\) to \(\Gamma_2 = (N_2, E_2, \emptyset)\). For any set of methods \(M_1\) that can be defined in \(\Gamma_1\) and any interface \(I = (IC, IM)\) of \(\Gamma_1 = (N_1, E_1, M_1)\), we can define a set of methods \(M_2\) in \(\Gamma_2\) such that \(\Gamma_1' \sqsubseteq I\Gamma_2' = (N_2, E_2, M_2)\).

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.

1. Soundness: provide a small set of rules that are structure refinements.
2. Completeness: prove that any structure transformation can be obtained by a sequential application of these refinement rules.

**Example 3.2.1.** Fig. 3.1 illustrates a structure transformation \(\rho_{[IC]}\), in which the class \(D\) corresponds to a class \(H\), the association \((C, a, D)\) is realized by two relational edges, and the data attribute \((D, x, \text{Int})\) is decomposed into two paths from \(H\) to data types in the resulting class graph. Specifically,

- \(IC = \{C\}\), \(\hat{\rho}(C) = C\) and \(\hat{\rho}(D) = O\),
- \(\rho_a(C, a, D) = C \xrightarrow{\hat{\rho}} J \xrightarrow{a} O\),
- \(\rho_a(D, x, \text{Int}) = \{O \xrightarrow{x} \text{Int}, O \xrightarrow{a} K \xrightarrow{x} \text{Int}\}\), and the addition operation on integers preserves the initial values of attributes: \(\text{init}(D, x) = \text{init}(O, x_1) + \text{init}(K, x_2)\).

3.2.1 Refinement rules for structure expansion

We give a set of basic refinement rules \(R1\) to \(R7\) in Fig. 3.3 which transform a class graph \(\Gamma_1 = (N_1, E_1, M_1)\) to another one \(\Gamma_2 = (N_2, E_2, M_2)\). The first and second columns of the table
are the names and descriptions of the rules, respectively. The precondition for each rule in the third column ensures that the class graph after the transformation is a well-formed one. Notice that each rule has an interface, shown in the last column, representing the classes whose names are not changed and methods in these classes whose signatures are not changed by the transformation.

In **R1**, we abuse the notation $M_1 \setminus \{C\}$ to denote the set of methods in $M_1$ that are neither defined in class $C$ nor containing $C$ as a parameter type. In **R2**, we use the notation $[C.y/C.x]$ to denote a substitution of each expression of the form $e.x$ by $e.y$ provided $dtype(e) \approx C$. The substitutions $[C.a.x/C.x]$ in **R5.1** and $[f(C.x_1, \ldots, C.x_k)/C.x]$ in **R6** are defined similarly. Notice that the latter substitution may lead to a non-deterministic assignment such as $h(e.x_1, \ldots, e.x_k) := cn$ which actually means $h(cn_1, \ldots, cn_k) = cn(e.x_1, \ldots, e.x_k) := (cn_1, \ldots, cn_k)$. This might be of unbounded nondeterminism, but it does not cause any problem in our framework.

In fact, **R5.1** and **R5.2** are two special cases of the general rule **R5** for forwarding attributes. It is depicted in Fig. 3.2 where each edge $(C_j, a_j, D)$ ($1 \leq j \leq k$) can be either an association or an inheritance. However, the two special cases are adequate to handle most cases of attribute forwarding.

In these refinement rules, the methods are transformed corresponding to the transformation of the class structure. This is important when we develop an object-oriented software in an incremental and iterative development process such as RUP [67]. The decomposition of low cohesive classes into simpler and more reusable classes [69] and the decomposition of functionality by delegation should be consistently combined [54].

### 3.2.2 Soundness and completeness of basic refinement rules

After providing the refinement rules, it is necessary to prove that they are sound, i.e. each rule indeed transforms a class graph to a refined one.

**Theorem 3.2.1** (Soundness of basic refinement rules). *If a class graph $\Gamma_1$ is transformed into another class graph $\Gamma_2$ through the application of a basic refinement rule $R_{[I]}$ (the application of $R$ with interface $I$), $\Gamma_1 \sqsubseteq_I \Gamma_2$.***

**Proof.** The application of $R_{[I]}$ transforms $\Gamma_1$ to $\Gamma_2$ implies $I = \langle IC, IM \rangle$ is a common interface of $\Gamma_1$ and $\Gamma_2$. According to Theorem 3.1.1, it is sufficient to construct a relation $\rho_\epsilon$ and a function $\rho_c$...
### 3.2. STRUCTURE REFINEMENT FOR GRAPH EXPANSION

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Precondition</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1</strong> Rename a class</td>
<td>a class $C$ is renamed to $D$; each occurrence of $C$ in any method is replaced by $D$</td>
<td>$D \notin N_1$</td>
<td>$\langle (N_1 \cap C \setminus {C}, \emptyset \rangle$</td>
</tr>
<tr>
<td><strong>R2</strong> Rename an attribute</td>
<td>an attribute $(C,x)$ is renamed to $(C,y)$; the body $c$ of each method is replaced by $c_{C,y/C,x}$</td>
<td>$y \neq x$; $y \notin \text{Attr}(C’)$ for any $C’ \not&lt; C$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R3</strong> Add a new class</td>
<td>add a new class $C$</td>
<td>$C \notin N_1$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R4.1</strong> Add an attribute</td>
<td>add an attribute $(C,x,T)$</td>
<td>$C \in C$; $x \notin \text{Attr}(C’)$ for any $C’ \not&lt; C$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R4.2</strong> Add an inheritance</td>
<td>add an inheritance $(C \triangleright D)$</td>
<td>$C, D \in C$; $(C \triangleright D’) \notin E_1$ for any $D’$; $D’ \not&lt; C$; $\text{attr}(C’) \cap \text{Attr}(D) = \emptyset$ for any $C’ \not&lt; C$; if $C’ : m, D’ : m \in M_1$ for some $C’ \not&lt; C$ and $D \not&lt; D’$, $C’ : m$ and $D’ : m$ have the same signature</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R5.1</strong> Forward an attribute through an association</td>
<td>an attribute $(C,x)$ is forwarded to $(D,x)$ through an association $(C,a,D)$; the body $c$ of each method is replaced by $c_{C,a,x/C,x}$</td>
<td>$x \notin \text{Attr}(D’)$ for any $D’ \not&lt; D$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R5.2</strong> Forward an attribute through an inheritance</td>
<td>an attribute $(C,x)$ is forwarded to $(D,x)$ through an inheritance $(C \triangleright D)$</td>
<td>$x \notin \text{Attr}(D’)$ for any $D’ \not&lt; D$, unless $D’ \not&lt; C$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R6</strong> Decompose a data attribute</td>
<td>a data attribute $(C,x,B)$ is decomposed into data attributes $(C,x_1,B_1), \ldots, (C,x_k,B_k)$; the body $c$ of each method turns to $c_{h(C,x_1, \cdots, C,x_k)/C,x}$ according to a primitive operation $h : B_1 \times \cdots \times B_k \rightarrow B$</td>
<td>$h$ is a surjective operation that preserves the initial values of attributes; $x_j \notin \text{Attr}(C’)$ for any $C’ \not&lt; C$ and $1 \leq j \leq k$</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
<tr>
<td><strong>R7</strong> Decompose an inheritance</td>
<td>an inheritance $(C \triangleright D)$ is decomposed into two inheritances $(C \triangleright J)$ and $(J \triangleright D)$ for some class $J$</td>
<td>$J \in C’$; $D \notin J$; $(J \triangleright C’) \notin E_1$ for any $C’$; $\text{attr}(J’) \cap \text{Attr}(D) = \emptyset$ for any $J’ \not&lt; J$; if $C’ : m, J : m \in M_1$ for some $C’ \not&lt; C$, $C’ : m$ and $J : m$ have the same signature; if $J’ : m, D’ : m \in M_1$ for some $J’ \not&lt; J$ and $D’ \not&lt; D$, $J’ : m$ and $D’ : m$ have the same signature</td>
<td>$\langle {1} \cap C, M_1 \rangle$</td>
</tr>
</tbody>
</table>

Figure 3.3: Basic refinement rules

for each basic refinement rule such that $\Gamma_2$ is a simulation of $\Gamma_1$ with respect to $I$, $\rho_a$, and $\rho_c$.

- **Case R1.** We define $\rho_a(c) = c_{[D/C]}$, and $\rho_a(G,G’)$ if $G’ = G[D/C]$. For a state graph $G$, the notation $G[D/C]$ denotes the state graph $G’$ that is the same as $G$ except that $G’.\text{type}(a) = G.\text{type}(a)[D/C]$ for each object node $o$.

- **Case R2.** We define $\rho_c(c) = c_{C,y/C,x}$, and $\rho_c(G,G’)$ if $G’$ is obtained from $G$ by replacing each edge $(o,x,v)$ with $(o,y,v)$ for each object node $o \not< C$. 
• Case R3, R4.2, R5.2 or R7. We define ρc(c) = c, and ρd(G, G') if G' = G.
• Case R4.1. We define ρc(c) = c, and ρd(G, G') if G' is obtain from G by adding edges
  (o, x, cn) where o \not\in C in G and cn = \text{init}(C, x).
• Case R5.1. We define ρc(c) = c[C.a.x/C.x], and ρd(G, G') if G' is obtained from G by
  replacing each edge (o, x, v), where o \not\in C and thus there is an edge (o, a, o'), with (o', x, v).
• Case R6. We define ρc(c) = c[h(C, x_1, \ldots, C, x_k)/C.x], and ρd(G, G') if G' is obtained from G
  by replacing each edge (o, x, cn), where o \not\in C, with a set of edges \{\(a, x_1, cn_1\), \ldots, \(a, x_k, cn_k\}\}
  such that h(cn_1, \ldots, cn_k) = cn.

Notice that in each case, \(I_2\) is actually a bisimulation of \(I_1\) with respect to I, \(\rho_a\) and \(\rho_c\). According

to Theorem 3.1.2, we also have \(\Gamma_1 \equiv_{I} I_2\).

As a direct consequence of Theorem 3.2.1, a sequential application of basic refinement rules
transforms a class graph to a refined one.

**Corollary 3.2.1.** If a class graph \(\Gamma_1\) is transformed into another class graph \(\Gamma_2\) by applying basic refinement rules \(R_1, \ldots, R_7\) sequentially, \(\Gamma_1 \equiv_{I} I_2\) for any interface I of \(\Gamma_1\) such that I \subseteq I_j (1 \leq j \leq k).

Then, we show that the basic refinement rules R1 - R7 are complete with respect to well-typed structure transformations.

**Theorem 3.2.2** (Completeness result). Let \(p_1[c] \not\in I\) be a well-typed structure transformation from
\(\Gamma_1 = \langle N_1, E_1, \emptyset\rangle\) to \(\Gamma_2 = \langle N_2, E_2, \emptyset\rangle\). For any set of methods \(M_1\) that can be defined in \(\Gamma_1\) and any interface \(I = \langle IC, IM\rangle\) of \(\Gamma_1' = \langle N_1, E_1, M_1\rangle\), there is a sequential application of basic refinement rules \(R_1, \ldots, R_7\), where I \subseteq I_j (1 \leq j \leq k), that transforms \(\Gamma_1'\) to \(\Gamma_2' = \langle N_2, E_2, M_2\rangle\) for some \(M_2\).

**Proof.** We identify a sequential application of refinement rules in four steps that transforms \(\Gamma_1'\) to \(\Gamma_2'\).

1. Rename each class C to \(\hat{\rho}(C)\) by R1.
2. Using R6, decompose each data attribute \((C, x, B)\) into a set of data attributes \{\((C, x_1, B_1)\),
   \ldots, \((C, x_k, B_k)\)\}, provided \(\rho_d(C, x, B) = \{\hat{\rho}(C) \rightarrow \ldots \rightarrow D_j \xrightarrow{\rho_d} B_j \mid 1 \leq j \leq k\}\).
3. Change each relational edge \((C, a, D)\) to a path \(\rho_t(C, a, D)\), using R2, R3, R4, R5 and
   R7; change each data attribute \((C, x_j, B_j)\) (1 \leq j \leq k) in the previous step to the path \(\hat{\rho}(C) \rightarrow \ldots \rightarrow D_j \xrightarrow{\rho_d} B_j\), using R3, R4 and R5.
4. Add nodes and edges that are not in the image of \(\hat{\rho}\), \(\rho_d\) or \(\rho_t\), through applications of R3 and R4.

Notice that the above application sequence transforms every class node C of \(\Gamma_1'\) into a node \(\hat{\rho}(C)\) of \(\Gamma_2'\), every relational edge \((C, a, D)\) of \(\Gamma_1'\) into a path \(\rho_t(C, a, D)\) of \(\Gamma_2'\), and every data attribute \((C, x, B)\) of \(\Gamma_1'\) finally into a set of paths \(\rho_d(C, x, B)\) of \(\Gamma_2'\). The application sequence is sufficient to covers all the cases.

**Example 3.2.2.** For the structure transformation illustrated in Example 3.2.1, Fig. 3.4 shows the applications of the rules that transform the source class structure \(\Gamma_1\) to the target structure \(\Gamma_2\).

We have proved that basic refinement rules are sound with respect to structure refinement and complete with respect to structure transformation. As a result, a (well-typed) structure transformation always leads to a structure refinement, i.e. Proposition 3.2.1 holds.
3.3. Structure Refinement for Graph Compression

A structure refinement can be achieved not only by expanding the class structure, but also by compressing the class structure. In this section, we study structure refinement by removing certain classes, attributes and methods or combining certain classes while preserving the functional services.

3.3.1 Removing redundant structure elements

Intuitively, we can remove classes, attributes and methods that are not referred to, either directly or indirectly, in the interface. In other words, structure elements of a class graph that are of no contribution to the provision of services can be removed.

In order to distinguish these structure elements from the others, we formalize the notion of accessed nodes, edges and methods in a class graph. For this, we introduce two notations. Given
a class graph \( \Gamma = (N, E, M) \) and two of its nodes \( C_1, C_2, \text{cb}(C_1, C_2) \) and \( \text{eb}(C_1, C_2) \) denote the set of classes and inheritance edges between \( C_1 \) and \( C_2 \), respectively.

\[
\text{cb}(C_1, C_2) \equiv \{ C \in N \cap C \mid C_1 \preceq C \preceq C_2 \} \\
\text{eb}(C_1, C_2) \equiv \{ (C, D) \in E \mid C_1 \preceq C, D \preceq C_2 \}
\]

Notice that both \( \text{cb}(C_1, C_2) \) and \( \text{eb}(C_1, C_2) \) are empty if \( C_1 \not\preceq C_2 \).

We start from defining nodes and edges that an expression directly refers to.

**Definition 3.3.1** (Directly accessed elements of expression). Given a class graph \( \Gamma = (N, E, M) \) and a well-typed expression \( e \), nodes\((e)\) and edges\((e)\) denote respectively the set of class nodes and edges that \( e \) accesses directly. They are defined inductively.

1. If \( e \) is a variable \( x \), nodes\((e)\) = \( \{ \text{dtype}(x) \} \) \( \setminus \mathcal{B} \) and edges\((e)\) = \( \emptyset \).
2. If \( e \) is a literal \( c_n \), nodes\((e)\) = edges\((e)\) = \( \emptyset \).
3. If \( e \) is self and the current class is \( C \), nodes\((e)\) = \( \{ C \} \) and edges\((e)\) = \( \emptyset \).
4. If \( e \) is \( e_0.a \), nodes\((e)\) = nodes\((e_0)\) \( \cup \text{cb}(C', C'') \cup \{ \text{dtype}(C'', a) \} \) \( \setminus \mathcal{B} \) and edges\((e)\) = edges\((e_0)\) \( \cup \text{eb}(C', C'') \cup \{ (C'', a, \text{dtype}(C'', a)) \} \), where \( C' = \text{dtype}(e_0) \) and \( C'' \) is the superclass of \( C' \) with \( a \in \text{attr}(C'') \).
5. If \( e \) is \( (C_0) e_0 \), nodes\((e)\) = nodes\((e_0)\) \( \cup \text{cb}(C_0, \text{dtype}(e_0)) \) \( \cup \text{eb}(\text{dtype}(e_0), C_0) \) and edges\((e)\) = edges\((e_0)\) \( \cup \text{cb}(C_0, \text{dtype}(e_0)) \) \( \cup \text{eb}(\text{dtype}(e_0), C_0) \).
6. If \( e \) is \( b(e_1, \ldots, e_k) \), nodes\((e)\) = nodes\((e_1) \cup \ldots \cup \text{nodes}\((e_k)\) and edges\((e)\) = edges\((e_1) \cup \ldots \cup \text{edges}\((e_k)\).

A simple variable or self refers only to its declared class type, while a literal does not refer to any classes or edges. An expression \( (C_0)e_0 \) or \( e_0.a \) accesses all the nodes and edges \( e_0 \) accesses. In addition, \( (C_0)e_0 \) refers to all the classes and inheritance edges between the declared type of \( e_0 \) and \( C_0 \), while \( e_0.a \) refers to the its declared type as well as the classes and inheritance edges between the declared type of \( e_0 \) and the class where \( a \) is declared. An expression \( b(e_1, \ldots, e_k) \) refers to all the classes and edges its sub-expressions access.

We allow the notations nodes\((\vec{e})\) and edges\((\vec{e})\) for a sequence of expressions \( \vec{e} \). They represent the set of class nodes and edges that an expression in \( \vec{e} \) accesses, respectively. In the same way, we allow the notations cb\((\vec{C}, \vec{C'})\) and eb\((\vec{C}, \vec{C'})\) for sequences of classes \( \vec{C} = C_1, \ldots, C_k \) and \( \vec{C'} = C'_1, \ldots, C'_k \). They are the union of cb\((C_j, C'_j)\) and eb\((C_j, C'_j)\) for \( 1 \leq j \leq k \), respectively. These notations enable us to define the sets of nodes, edges and methods that a command accesses directly.

**Definition 3.3.2** (Directly accessed elements of command). Given a class graph \( \Gamma = (N, E, M) \) and a well-typed command \( c \), nodes\((c)\), edges\((c)\) and meths\((c)\) denote respectively the set of class nodes, edges and methods \( c \) accesses directly. They are defined inductively.

1. If \( c \) is skip, nodes\((c)\) = edges\((c)\) = meths\((c)\) = \( \emptyset \).
2. If \( c \) is an object creation \( C.\text{new}(\ell_e) \), nodes\((c)\) = nodes\((\ell_e) \cup \text{cb}(C, \text{dtype}(\ell_e)) \), edges\((c)\) = edges\((\ell_e) \cup \text{cb}(C, \text{dtype}(\ell_e)) \) and meths\((c)\) = \( \emptyset \).
3. If \( c \) is \( \ell_e : = \vec{e} \), nodes\((c)\) = nodes\((\ell_e) \cup \text{nodes}\((\vec{e})\) \( \cup \text{cb}(\text{dtype}(\vec{e}), \text{dtype}(\ell_e)) \), edges\((c)\) = edges\((\ell_e) \cup \text{nodes}\((\vec{e})\) \( \cup \text{cb}(\text{dtype}(\vec{e}), \text{dtype}(\ell_e)) \) and meths\((c)\) = \( \emptyset \).
4. If \( c \) is var \( T \ x = e \), nodes\((c)\) = nodes\((e) \cup \text{cb}(\text{dtype}(e), T) \), edges\((c)\) = edges\((e) \cup \text{cb}(\text{dtype}(e), T) \) and meths\((c)\) = \( \emptyset \). If \( c \) is var \( T \ x \), nodes\((c)\) = \( \{ T \} \) \( \setminus \mathcal{B} \) and edges\((c)\) = meths\((c)\) = \( \emptyset \).
5. If \( c \) is end \( x \), nodes\((c)\) = \( \{ \text{dtype}(x) \} \) \( \setminus \mathcal{B} \) and edges\((c)\) = meths\((c)\) = \( \emptyset \).
6. If \( c \) is a method invocation \( e.m(\vec{e};\vec{t}) \), \( \text{nodes}(c) = \text{nodes}(e) \cup \text{nodes}(\vec{e}) \cup \text{nodes}(\vec{t}) \cup \text{cb}(\vec{S}, \vec{e}) \cup \text{cb}(\vec{T}, \vec{t}) \), \( \text{edges}(c) = \text{edges}(e) \cup \text{edges}(\vec{e}) \cup \text{edges}(\vec{t}) \cup \text{cb}(\vec{S}, \vec{S}) \cup \text{cb}(\vec{T}, \vec{T}) \) and \( \text{meths}(c) = \{ C'' : m \} \). where \( C' = \text{dtype}(e) \), \( C'' \) is the least superclass of \( C' \) where \( m \) is declared, \( \text{mtype}(C'', m) = (\vec{S}; \vec{T}) \), \( \vec{S} = \text{dtype}(\vec{e}) \) and \( \vec{T} = \text{dtype}(\vec{t}) \).

7. If \( c = c_1 \odot c_2 \) or \( c_1 \cap c_2 \), \( \text{nodes}(c) = \text{nodes}(c_1) \cup \text{nodes}(c_2), \text{edges}(c) = \text{edges}(c_1) \cup \text{edges}(c_2) \) and \( \text{meths}(c) = \text{meths}(c_1) \cup \text{meths}(c_2) \).

8. If \( c = c_1 \odot b \bowtie c_2 \), \( \text{nodes}(c) = \text{nodes}(c_1) \cup \text{nodes}(c_2) \cup \text{nodes}(b), \text{edges}(c) = \text{edges}(c_1) \cup \text{edges}(c_2) \cup \text{edges}(b) \) and \( \text{meths}(c) = \text{meths}(c_1) \cup \text{meths}(c_2) \).

9. If \( c = b \ast c_1 \), \( \text{nodes}(c) = \text{nodes}(c_1) \cup \text{nodes}(b), \text{edges}(c) = \text{edges}(c_1) \cup \text{edges}(b) \) and \( \text{meths}(c) = \text{meths}(c_1) \).

The \( \text{skip} \) command does not refer to any nodes, edges or methods. An object creation \( \text{cmd} \).new\( (\text{le}) \) refers directly to those structure elements \( \text{le} \) refers to, together with classes and inheritance edges between \( C \) and the declared type of \( \text{le} \). Similarly, an assignment \( \vec{e} \text{le} := \vec{e}' \) accesses what the expressions \( \vec{e} \text{le} \) and \( \vec{e}' \) access, together with classes and inheritance edges between their declared types. A variable declaration \( \text{var} \ T \ x = e \) accesses those \( e \) accesses, as well as classes and inheritance edges between \( T \) and the declared type of \( e \). By contrast, a variable ending command \( \text{end} \ x \) accesses only the class type \( x \) is declared as. A method invocation \( e.m(\vec{e};\vec{t}) \) directly accesses what \( e, \vec{e} \) and \( \vec{t} \) access, plus classes and inheritance edges between the types of formal and actual parameters. The directly accessed elements of structural commands, such as sequential composition and conditional choice, are defined inductively.

Based on the directly accessed elements of commands, it is natural to define the directly accessed elements of a method, as those its body command accesses directly.

**Definition 3.3.3 (Directly accessed elements of method).** Given a class graph \( \Gamma = (N, E, M) \) and one of its methods \( C_0 :: m_0 \), the set of class nodes, edges and methods that \( C_0 :: m_0 \) accesses directly are defined as

- \( \text{nodes}(C_0 :: m_0) = \text{nodes}(c_0), \)
- \( \text{edges}(C_0 :: m_0) = \text{edges}(c_0), \)
- \( \text{meths}(C_0 :: m_0) = \text{meths}(c_0), \)

where \( c_0 \) is the body of \( C_0 :: m_0 \).

Notice that a method may also access structure elements indirectly, through calling other methods. We formally define the elements accessed by a method here, including those directly or indirectly.

**Definition 3.3.4 (Accessed elements of method).** Given a class graph \( \Gamma = (N, E, M) \) and one of its methods \( C_0 :: m_0 \), we use \( \text{Nodes}(C_0 :: m_0) \), \( \text{Edges}(C_0 :: m_0) \) and \( \text{Meths}(C_0 :: m_0) \) to denote the sets of class nodes, edges and methods \( C_0 :: m_0 \) accesses, respectively. They are defined as follows.

1. For a class node \( C \in N \), \( C \in \text{Nodes}(C_0 :: m_0) \) if there is a sequence of methods \( C_1 :: m_1, \ldots, C_k :: m_k \) (\( k \geq 0 \)) such that \( C_j :: m_j \in \text{meths}(C_{j-1} :: m_{j-1}) \) for \( 1 \leq j \leq k \) and \( C \in \text{nodes}(C_k :: m_k) \).
2. For an edge \( d \in E \), \( d \in \text{Edges}(C_0 :: m_0) \) if there is a sequence of methods \( C_1 :: m_1, \ldots, C_k :: m_k \) (\( k \geq 0 \)) such that \( C_j :: m_j \in \text{meths}(C_{j-1} :: m_{j-1}) \) for \( 1 \leq j \leq k \) and \( d \in \text{edges}(C_k :: m_k) \).
3. For a method \( C :: m \in M \), \( C :: m \in \text{Meths}(C_0 :: m_0) \) if there is a sequence of methods \( C_1 :: m_1, \ldots, C_k :: m_k \) (\( k \geq 0 \)) such that \( C_j :: m_j \in \text{meths}(C_{j-1} :: m_{j-1}) \) for \( 1 \leq j \leq k \) and \( C :: m \in \text{meths}(C_k :: m_k) \).
Based on these notations, we can define the set of class nodes, edges and methods that are accessed by an interface. Let \( \text{Meths}(M) \) be a shorthand of \( \bigcup \{ \text{Meths}(C :: m) \mid C :: m \in M \} \) for a set of methods \( M \).

**Definition 3.3.5** (Accessed elements of interface). Given a class graph \( \Gamma = \langle N, E, M \rangle \) and one of its interfaces \( I = \langle IC, IM \rangle \), we use \( \text{Nodes}(I) \), \( \text{Edges}(I) \) and \( \text{Meths}(I) \) to denote the sets of class nodes, edges and methods \( I \) accesses, respectively. They are defined as

- \( \text{Nodes}(I) = \bigcup \{ cb(C, D) \mid C, D \in IC \} \cup \bigcup \{ \text{Nodes}(C :: m) \mid C :: m \in IM \} \),
- \( \text{Edges}(I) = \bigcup \{ cb(C, D) \mid C, D \in IC \} \cup \bigcup \{ \text{Edges}(C :: m) \mid C :: m \in IM \} \),
- \( \text{Meths}(I) = \{ C :: m \in M \mid C \lesssim D, cb(C, D) \subseteq \text{Edges}(I), D :: m \in (IM \cup \text{Meths}(IM)) \) for some class \( D \in N \} \).

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

**Theorem 3.3.1.** Let \( \Gamma = \langle N, E, M \rangle \) be a class graph and \( I = \langle IC, IM \rangle \) be an interface of \( \Gamma \).

1. An accessed edge of the interface is associated with accessed nodes: for each edge \( (C, a, D) \in \text{Edges}(I) \), \( C \in \text{Nodes}(I) \) and \( D \in \text{Nodes}(I) \) unless \( D \) is a primitive type.
2. An accessed method of the interface is defined in an accessed class: for each method \( C :: m \in \text{Meths}(I) \), \( C \in \text{Nodes}(I) \).
3. An method that overrides an accessed method through accessed inheritances is also an accessed one: for methods \( C :: m, D :: m \in M \) such that \( C :: m \in \text{Meths}(I) \), \( D \lesssim C \) and \( cb(D, C) \subseteq \text{Edges}(I) \), \( D :: m \in \text{Meths}(I) \).

**Proof.** Straightforward from the above definitions.

- \( (C, a, D) \in \text{Edges}(I) \) means either \( (C, a, D) \in cb(J, K) \) for some \( J, K \in IC \) or \( (C, a, D) \in \text{Edges}(J :: m) \) for some \( J :: m \in IM \). Suppose \( D \) is not a primitive type. In the former case, \( C, D \in cb(J, K) \) and thus \( C, D \in \text{Nodes}(I) \). In the latter case, \( C, D \in \text{Nodes}(J :: m) \) and we also have \( C, D \in \text{Nodes}(I) \).

- \( C :: m \in \text{Meths}(I) \) means there is a superclass \( C' \) of \( C \) such that \( cb(C, C') \subseteq \text{Edges}(I) \) and \( C' :: m \in (IM \cup \text{Meths}(IM)) \). If \( C' \neq C \), \( cb(C, C') \subseteq \text{Edges}(I) \) and thus \( C \in \text{Nodes}(I) \). If \( C' = C \), \( C :: m \in IM \) or \( C :: m \in \text{Meths}(C_0 :: m_0) \) for some method \( C_0 :: m_0 \in IM \). So, \( C \in IC \) or \( C \in \text{Nodes}(C_0 :: m_0) \). In either case, we have \( C \in \text{Nodes}(I) \).

- For methods \( C :: m, D :: m \in M \) such that \( C :: m \in \text{Meths}(I) \), \( D \lesssim C \) and \( cb(D, C) \subseteq \text{Edges}(I) \), there is a superclass \( C' \) of \( C \) such that \( cb(C, C') \subseteq \text{Edges}(I) \) and \( C' :: m \in (IM \cup \text{Meths}(IM)) \). So, \( C' \) is also a superclass of \( D \) and \( cb(D, C') \subseteq \text{Edges}(I) \). As a result, \( D :: m \in \text{Meths}(I) \).

\[ \square \]

### 3.3.2 Rules for removing redundant elements

An interface \( I \) is used to represent the functional classes and functional methods of a class graph \( \Gamma \). So, the set of nodes \( \text{Nodes}(I) \), edges \( \text{Edges}(I) \) and methods \( \text{Meths}(I) \) that are accessed by the interface are actually those contributing to the provision of services of \( \Gamma \). By contrast, other class nodes, edges and methods are redundant in that they do not take part in the services \( \Gamma \) provides. They can be simply removed without affecting the functionality of \( \Gamma \).
### 3.3. Structure Refinement for Graph Compression

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Precondition</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8.1</td>
<td>Remove a class</td>
<td>( M(C) = \emptyset; ) C is not associated with any edge; ( C \notin \text{nodes}(C' :: m) ) for each ( C' :: m \in M )</td>
<td>( I )</td>
</tr>
<tr>
<td>R8.2</td>
<td>Remove an edge</td>
<td>( d \notin \text{edges}(C' :: m) ) for each method ( C' :: m \in M )</td>
<td>( I )</td>
</tr>
<tr>
<td>R8.3</td>
<td>Remove methods</td>
<td>( M' = M \setminus \text{Meths}(I) )</td>
<td>( I )</td>
</tr>
</tbody>
</table>

Figure 3.5: Rules for removing redundant elements

**Theorem 3.3.2.** Let \( \Gamma = \langle N, E, M \rangle \) be a class graph and \( I = \langle IC, IM \rangle \) be one of its interfaces. Let \( \Gamma' = \langle N', E', M' \rangle \) be a (well-typed) class graph obtained from \( \Gamma \) by removing some of the redundant class nodes, edges and methods with respect to \( I \), i.e. \( \text{Nodes}(I) \subseteq N' \subseteq N \), \( \text{Edges}(I) \subseteq E' \subseteq E \) and \( \text{Meths}(I) \subseteq M' \subseteq M \). We have \( \Gamma' \equiv_I \Gamma \).

**Proof.** Let \( \rho_k \) be a relation from \( S_I(\Gamma) \) to \( S_I(\Gamma') \), such that \( \rho_k(G, G') \) if and only if \( G' \) is obtained from \( G \) by removing object attributes that correspond to attribute edges in \( \Gamma \) but not \( \Gamma' \). Let \( \rho_c \) be a partial function from commands definable in \( \Gamma \) to commands definable in \( \Gamma' \), such that \( \rho_c(c) = c \) for any command \( c \) definable in \( \Gamma' \). It is straightforward to verify that \( \rho_k \) and \( \rho_c \) satisfy all the conditions in Definition 3.1.4, thus \( \Gamma' \) is a bisimulation of \( \Gamma \) with respect to \( I \), \( \rho_k \) and \( \rho_c \). According to Theorem 3.1.2, \( \Gamma' \equiv_I \Gamma \).

For a well-typed class graph \( \Gamma \), the class graph \( \Gamma' \) obtained by removing all the redundant class nodes, edges and methods from \( \Gamma \) is also well-typed. This is because those redundant elements are not referred to by the rest part of the graph. According to Theorem 3.3.2, \( \Gamma' \) is equivalent to \( \Gamma \) in terms of structure refinement.

We provide a set of rules in Fig. 3.5 to remove redundant class nodes, edges and methods from a class graph \( \Gamma = \langle N, E, M \rangle \), with respect to an interface \( I \). Notice that the precondition of R8.1 implies \( C \notin \text{Nodes}(I) \) which means \( C \) is a redundant class node. Similarly, the precondition of R8.2 implies \( d \notin \text{Edges}(I) \) which means \( d \) is a redundant edge. The soundness of these rules is ensured by Theorem 3.3.2. The theorem also ensures each of them is an equivalence rule.

### 3.3.3 Rules for Combining Classes

Removing redundant structure elements is not the only way of structure refinement that makes the graph “smaller”. We can also achieve structure refinement by combining certain classes in a class structure.

Intuitively, in a class graph, a class \( D \) can be merged into another class \( C \) in the following way (See Fig. 3.6):

1. each outgoing edge from \( D \) is turned into an outgoing edge from \( C \); each incoming edge to \( D \) is turned into an incoming edge to \( C \),
2. each method of class \( D \) is moved into class \( C \), and
3. each parameter type \( D \) of a method is changed into \( C \).

The name of the combined class can be either \( C, D \) or any fresh class name. For simplicity, we just choose \( C \) in the following discussions.

We provide a set of rules in Fig. 3.7 for combining classes in a class graph \( \Gamma = \langle N, E, M \rangle \). Notice that there are two cases of class combination. R9.1 and R9.2 are for the case that one of the two classes to be combined is a direct subclass of the other, while R9.3 is for the case that there is no subclass relation between the two classes. The preconditions of two cases are different in their forms, but they are for the same purpose:
merge a class \( D \) into its direct superclass \( C \); the body command \( c \) of each method is replaced by \( c[C/D] \).

**Precondition**

\[
(D, \triangleright, C) \in E; \quad \forall C' \subseteq C \text{ but } C' \neq D; \quad \forall D \ni m \notin M \\
\text{if } C'' :: m \ni M \text{ for some } C'' \triangleright C \text{ or } C'' :: m \ni M \text{ for some } D'' \triangleright D \text{ or } D'' \ni D \text{ but } D'' \neq C
\]

**Interface**

\[
\{(N \triangleright \mathcal{C}) \setminus \{D\}, M \setminus \{D\}\} 
\]

---

**Figure 3.6:** Idea of class combination

**Figure 3.7:** Rules for combining classes

1. to ensure there is no name conflict in the attributes and methods of the two classes, thus the combination of classes with polymorphic methods is not allowed, and

2. to ensure the combination does not lead to new polymorphic methods.

By applying these rules sequentially, we are able to combine more than two classes into a single class. The soundness of these rules are proved in the following theorem. The theorem also shows that they are equivalent rules in terms of structure refinement.

**Theorem 3.3.3** (Soundness of R9). If a class graph \( \Gamma_1 \) is transformed into another class graph \( \Gamma_2 \) through an application of R9 with interface \( I \), \( \Gamma_1 \sqsubseteq_I \Gamma_2 \).

**Proof.** The application of R9 with interface \( I \) transforms \( \Gamma_1 \) to \( \Gamma_2 \) implies \( I \) is a common interface of \( \Gamma_1 \) and \( \Gamma_2 \).

- **Case R9.1.**

  Let \( \rho_b \) be a relation from \( S_I(\Gamma_1) \) to \( S_I(\Gamma_2) \), such that \( \rho_b(G_1, G_2) \) if and only if \( G_2 \) is obtained from \( G_1[C/D] \) by adding a set of edges \( (o, a, cn) \) where \( o \) is an object node of class \( C' \neq C \) in \( G_1[C/D] \), \( a \in \text{attr}_r(D) \) and \( cn = \text{init}_r(D, a) \). Let \( \rho_c \) be a function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_c(c) = c[C/D] \). It is straightforward to verify that \( \rho_b \) and \( \rho_c \) satisfy all the conditions in Definition 3.1.3, thus \( \Gamma_2 \) is a simulation of \( \Gamma_1 \) with respect to \( I, \rho_b \) and \( \rho_c \). According to Theorem 3.1.1, \( \Gamma_1 \sqsubseteq_I \Gamma_2 \).
3.4. Completeness of Refinement Rules

Let \( \rho'_k \) be a relation from \( \mathcal{S}_1(\Gamma_1) \) to \( \mathcal{S}_1(\Gamma_1) \) such that \( \rho'_k(G_2, G_1) \) if and only if \( G_1 = G_2[D/C] \). Let \( \rho'_k \) be a function from commands definable in \( \Gamma_2 \) and in the range of \( \rho_k \) to commands definable in \( \Gamma_1 \), such that \( \rho'_k(c) \) is obtained from \( c \) through replacing each object creation \( C\.new(le) \) by \( D\.new(le) \); each expression \( e.a \), where \( dtyp\Gamma_1(e) = C \) and \( a \in attr\Gamma_1(D) \), by \( (D)e;a \); and each method invocation \( e.m(c;le) \), where \( dtyp\Gamma_1(e) = C \) and \( D :: m \) is in \( \Gamma_1 \), by \( (D)e.m(c;le) \). It is straightforward to verify that \( \rho'_k \) and \( \rho'_k \) satisfy all the conditions in Definition 3.1.3, thus \( \Gamma_2 \) is a simulation of \( \Gamma_2 \) with respect to \( I \), \( \rho'_k \) and \( \rho'_k \). According to Theorem 3.1.1, \( \Gamma_2 \subseteq I \Gamma_1 \).

- **Case R9.2.**

Let \( \rho_k \) be a relation from \( \mathcal{S}_1(\Gamma_1) \) to \( \mathcal{S}_1(\Gamma_2) \), such that \( \rho_k(G_1, G_2) \) if and only if \( G_2 \) is obtained from \( G_1[C/D] \) by adding a set of edges \((o, a, cn)\) where \( o \) is an object node of class \( C' \approx_{\Gamma_2} C \) in \( G_1[C/D] \), \( a \in \text{attr}_{\Gamma_2}(C) \) and \( cn = \text{init}_{\Gamma_2}(C, a) \). Let \( \rho_k \) be a function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_k(c) = c[C/D] \). It is straightforward to verify that \( \rho_k \) and \( \rho_k \) satisfy all the conditions in Definition 3.1.3, thus \( \Gamma_2 \) is a simulation of \( \Gamma_2 \) with respect to \( I \), \( \rho_k \) and \( \rho_k \). According to Theorem 3.1.1, \( \Gamma_2 \subseteq I \Gamma_1 \).

- **Case R9.3.**

Let \( \rho_k \) be a relation from \( \mathcal{S}_1(\Gamma_1) \) to \( \mathcal{S}_1(\Gamma_2) \), such that \( \rho_k(G_1, G_2) \) if and only if \( G_2 \) is obtained from \( G_1[C/D] \) by adding a set of edges \((o, a, cn)\) where \( o \) is an object node of class \( C' \approx_{\Gamma_2} C \) in \( G_1[C/D] \), \( a \in \text{Attr}_{\Gamma_2}(C) \) and \( cn = \text{init}_{\Gamma_2}(C, a) \). Let \( \rho_k \) be a function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_k(c) = c[C/D] \). It is straightforward to verify that \( \rho_k \) and \( \rho_k \) satisfy all the conditions in Definition 3.1.3, thus \( \Gamma_2 \) is a simulation of \( \Gamma_2 \) with respect to \( I \), \( \rho_k \) and \( \rho_k \). According to Theorem 3.1.1, \( \Gamma_2 \subseteq I \Gamma_2 \).

For each state \( G_2 \in \mathcal{MS}_1(\Gamma_2) \), there is an initial state \( G \) with respect to \( I \) and a valid command \( c \) with respect to \( \Gamma_2 \) and \( I \) such that \( \langle c, G \rangle \rightarrow^* G_2 \) under \( \Gamma_2 \). Notice that \( c \) is also valid with respect to \( \Gamma_1 \) and \( I \). The same execution of \( \langle c, G \rangle \) under \( \Gamma_1 \) must terminate at a state \( G_1 \in \mathcal{MS}_1(\Gamma_1) \). We define a total relation \( \rho'_k \) from \( \mathcal{MS}_1(\Gamma_2) \) to \( \mathcal{MS}_1(\Gamma_1) \), such that \( \rho'_k(G_2, G_1) \) if and only if \( G_1 \) is obtained from \( G_2 \) in the above way. It is straightforward to verify that \( \rho'_k \) satisfies the two conditions in Definition 3.1.2, thus \( \Gamma_2 \subseteq I \Gamma_1 \).

\[ \square \]

**3.4 Completeness of Refinement Rules**

In this section, we study the completeness of our structure refinement rules. Notice that structure refinement is a semantic property while a refinement rule can only be a syntactic transformation. It is infeasible to establish a finite set of rules that is able to transform a class graph into any refined class graph. However, to demonstrate the power of the refinement rules we have given, we show that a class graph can always be transformed into a certain good form. Specifically, we study two normal forms and then show that the rules are powerful enough to transform any class graph into one of them.

**Definition 3.4.1 (Normal form).** Let \( \Gamma = \langle N, E, M \rangle \) be a class graph with an interface \( I = \langle IC, IM \rangle \). We say \( \Gamma \) is of Normal Form I if the following conditions hold.
1. \( \Gamma \) contains at least one supporting class, i.e. a class not in \( IC \).

2. All classes of \( \Gamma \) belong to one inheritance tree: there is a class \( C \in N \), which is the root of the inheritance tree, such that \( D \preceq C \) for any class \( D \in N \).

3. Each inheritance edge of \( \Gamma \) is polymorphic, unless it is associated with two functional classes, i.e. classes in \( IC \). We call an inheritance edges \((D, D', C)\) polymorphic if at least one method defined in \( C \), or a superclass of \( C \), is overridden in \( D \).

4. \( \Gamma \) contains no redundant class nodes, edges or methods: \( N \setminus T = \text{Nodes}(I) \), \( E = \text{Edges}(I) \) and \( M = \text{Meths}(I) \).

We say that \( \Gamma \) is of Normal Form II, if \( \Gamma \) satisfies both the following conditions.

1. All classes of \( \Gamma \) are functional classes: \( N \setminus T = IC \). This implies \( \Gamma \) contains no redundant class nodes.

2. \( \Gamma \) contains no redundant edges or methods: \( E = \text{Edges}(I) \) and \( M = \text{Meths}(I) \).

### 3.4.1 Structure refinement for changing methods

The refinement rules for graph expansion and graph compression so far are not enough to transform any class graph into a normal form, due to possible inconsistency, for example name conflict, between class methods. In order to adjust the methods in a class graph or make them consistent with one another, we study structure refinement that changes methods while preserving the class structure. In principle, this kind of refinement corresponds to the simple procedural refinement without changing the data space [82, 6].

We provide a set of rules in Fig. 3.8 to transform a class graph \( \Gamma = \langle N, E, M \rangle \) into another class graph \( \Gamma' = \langle N, E, M' \rangle \) without changing the set of nodes or edges. In **R10.1**, we use the notation \([C_0,m'/C_0,m]\) to denote the substitution of each method invocation \( e_0.m(e_0; \bar{e}) \) where \( \text{dtype}(e_0) \preceq C_0 \) by \( e_0.m'(e_0; \bar{e}) \). Notice that a special case of **R10.1** is to rename a method which is not polymorphic. We simply rename it to a fresh name \( C :: m' \) while substituting each invocation to \( C :: m \) by an invocation to \( C :: m' \) in the body of each method.

The precondition of **R10.1** and **R10.3** is to ensure the group of methods to be renamed and the method added are neither overriding nor overridden by other methods, respectively. By contrast, the preconditions of **R10.2** is to avoid copying the method to a class where the method has been overridden, otherwise the copy will be incorrect. Also notice that **R10.4** and **R10.5** are useful for the interchange between a method invocation and an equivalent command, which is obtained from the method body through certain substitution. The soundness of these rules is proved in the following theorem. The theorem also shows that they are equivalence rules in terms of structure refinement.

**Theorem 3.4.1 (Soundness of R10).** If a class graph \( \Gamma_1 \) is transformed into another class graph \( \Gamma_2 \) through an application of **R10** with interface \( I \), \( \Gamma_1 \equiv_I \Gamma_2 \).

**Proof.** The application of **R10** with interface \( I \) transforms \( \Gamma_1 \) to \( \Gamma_2 \) implies \( I \) is a common interface of \( \Gamma_1 \) and \( \Gamma_2 \).

- **Case R10.1.** Let \( \rho_c \) be a relation from \( S_I(\Gamma_1) \) to \( S_I(\Gamma_2) \) such that \( \rho_c(G_1, G_2) \) if and only if \( G_2 = G_1 \). Let \( \rho_e \) be a function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_e(c) = c[C_0.m'/C_0,m] \). It is straightforward to verify that \( \rho_c \) and \( \rho_e \) satisfy all the conditions in Definition 3.1.4, thus \( \Gamma_2 \) is a bisimulation of \( \Gamma_1 \) with respect to \( I \), \( \rho_c \) and \( \rho_e \). According to Theorem 3.1.2, \( \Gamma_1 \equiv_I \Gamma_2 \).

- **Case R10.2 or R10.3.** Let \( \rho_c \) be a relation from \( S_I(\Gamma_1) \) to \( S_I(\Gamma_2) \) such that \( \rho_c(G_1, G_2) \) if and only if \( G_2 = G_1 \). Let \( \rho_c \) be a function from commands definable in \( \Gamma_1 \) to commands definable in \( \Gamma_2 \) such that \( \rho_e(c) = c \). It is straightforward to verify that \( \rho_c \) and \( \rho_e \) satisfy all the conditions in Definition 3.1.4, thus \( \Gamma_2 \) is a bisimulation of \( \Gamma_1 \) with respect to \( I \), \( \rho_c \) and \( \rho_e \). According to Theorem 3.1.2, \( \Gamma_1 \equiv_I \Gamma_2 \).
3.4. COMPLETENESS OF REFINEMENT RULES

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Precondition</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R10.1 Rename methods</strong></td>
<td>Rename a group of polymorphic methods ( {C_0 :: m, C_1 :: m, \ldots, C_k :: m} ) ((k \geq 0)) to ({C_0 :: m', C_1 :: m', \ldots, C_k :: m'}) simultaneously; the body command (c) of each method is replaced by (e[C_0.m' / C_0.m])</td>
<td>(m') is a fresh method name; (C_j \not\equiv C_0) for each (1 \leq j \leq k; C :: m \not\in M) for each class (C) such that (C \not\subseteq C_0) or (C_0 \not\equiv C), unless (C \in {C_0, C_1, \ldots, C_k})</td>
<td>(\langle N \cap \mathcal{C}, M \setminus {C_0 :: m, C_1 :: m, \ldots, C_k :: m}\rangle)</td>
</tr>
<tr>
<td><strong>R10.2 Copy a method</strong></td>
<td>Copy a method (m) from a class (C) to one of its subclasses (D)</td>
<td>(D \not\subseteq C; D \not\equiv C; D' :: m \not\in M) for each class (D' \not\subseteq C) such that (D \not\subseteq D')</td>
<td>(\langle N \cap \mathcal{C}, M\rangle)</td>
</tr>
<tr>
<td><strong>R10.3 Add a method</strong></td>
<td>Add a well-typed method (m) into a class (C)</td>
<td>(C' :: m \not\in M) for each class (C') such that (C \not\subseteq C') or (C' \not\subseteq C)</td>
<td>(\langle N \cap \mathcal{C}, M\rangle)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Precondition</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R10.4 Invoke a method</strong></td>
<td>Replace a command (e'[e/\text{self}]) by the method invocation (e.m()) in the body of a method (m()) of a class (C), where (e.m()) invokes a method (m()) of a class (D)</td>
<td>(\text{dtype}(e) \not\subseteq D; \text{the method } D :: m') is not overridden in subclasses of (D)</td>
<td>(\langle N \cap \mathcal{C}, M\rangle)</td>
</tr>
<tr>
<td><strong>R10.5 Expand a method invocation</strong></td>
<td>Replace a method invocation (e.m()) by the command (e'[e/\text{self}]) in the body of a method (m()) of a class (C), where (e.m()) invokes a method (m'()) of a class (D)</td>
<td>(\text{dtype}(e) \not\subseteq D; \text{the method } D :: m') is not overridden in subclasses of (D)</td>
<td>(\langle N \cap \mathcal{C}, M\rangle)</td>
</tr>
</tbody>
</table>

Figure 3.8: Rules for changing methods

- Case **R10.4.** Let \(\rho_k\) be a total relation from \(\mathcal{M}S_I(\Gamma_1)\) to \(\mathcal{M}S_I(\Gamma_2)\) such that \(\rho_k(G_1, G_2)\) if and only if \(G_2 = G_1\). It is straightforward to verify that \(\rho_k\) satisfies the two conditions in Definition 3.1.2, thus \(\Gamma_1 \equiv_I \Gamma_2\). On the other hand, \(\rho_k^{-1}\) is a total relation from \(\mathcal{M}S_I(\Gamma_1)\) to \(\mathcal{M}S_I(\Gamma_2)\). And it is straightforward to verify that \(\rho_k^{-1}\) satisfies the two conditions in Definition 3.1.2, thus \(\Gamma_2 \equiv_I \Gamma_1\).

- Case **R10.5.** Since **R10.5** is exactly the inverse of **R10.4**, \(\Gamma_1 \equiv_I \Gamma_2\).

With **R10**, it is safe to say that the refinement rules we provide so far are complete in the sense that they transform any class graph into an equivalent graph of a normal form.

**Theorem 3.4.2** (Completeness result II). For each class graph \(\Gamma = \langle N, E, M\rangle\) with interface \(I = \langle IC, IM\rangle\), there exists a class graph \(\Gamma'\) also with interface \(I\) such that

1. \(\Gamma'\) is of either Normal Form I or Normal Form II,
2. \(\Gamma' \equiv_I \Gamma\), and
3. \(\Gamma'\) can be transformed from \(\Gamma\) through applications of refinement rules **R1** to **R10**.

**Proof.** For an arbitrary class graph \(\Gamma\), we have the following transformations.

1. First use **R8** to eliminate redundant classes nodes, edges and methods.
2. For each pair of classes, if the precondition of \( R9 \) holds, apply \( R9 \) to combine them into one. Notice that we may need to 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 renaming of attributes and methods. We get a class graph \( \Gamma' \).

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

1. Obviously, there is at least one supporting class in \( \Gamma' \). Otherwise, \( \Gamma' \) is of Normal Form II.

2. Assume \( \Gamma' \) contains at least two inheritance trees. Let \( D \) be a supporting class in \( \Gamma' \). Then, there exists at least one inheritance tree that does not contain \( D \). Let class \( C \) be the root of such an inheritance tree. We can use \( R2, R10.1 \) to rename attributes and methods of class \( D \) if needed and then use \( R9.3 \) 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 \( \Gamma' \) can be further merged.

3. Assume \( (C, \triangleright, D) \) is a non-polymorphic inheritance edge in \( \Gamma' \) where either \( C \) or \( D \) is a supporting class. If \( C \) is supporting, we can merge it into its direct superclass using \( R9.1 \). Similarly, if \( D \) is supporting, we can merge it into its direct subclass \( C \) using \( R9.2 \). This also contradicts with the fact that no pair of classes in \( \Gamma' \) can be further merged.

4. \( \Gamma' \) contains no redundant class nodes, edges or methods holds, since \( \Gamma' \) is obtained after the application of \( R8 \).

This completeness result shows that the essential difference between object-oriented programming and the traditional 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 3.4.1.** Let \( \Gamma \) be the class graph depicted in Fig. 3.9 with interface \( I = \{(C_1,C_2,C_3),\{C_1 :: get,C_2 :: set\}\} \). The methods of \( \Gamma \) are

\[
\begin{align*}
C_1 &:: \text{return}(\text{Int} \; j)\{j := \text{self}.x_1\} & D_1 &:: \text{set}(\text{Int} \; j)\{\text{self}.x_2 := j\} \\
J &:: \text{return}(\text{Int} \; j)\{\text{var} \; D_1 \; x; \; D_1.\text{new}(x); \; x.\text{set}(j); \; j := \text{self}.x_1 + x.\text{x_2}; \; \text{end} \; x\} \\
C_2 &:: \text{set}(\text{Int} \; j)\{(\text{var} \; \text{Int} \; k; \; \text{self}.a.\text{get}(k); \; \text{self}.x_2 := k; \; \text{end} \; k) \land \text{self}.b_2 \land \text{self}.x_2 := j\} \\
K &:: \text{return}(\text{Int} \; j)\{j := \text{self}.z\} & D_2 &:: \text{get}(\text{Int} \; j)\{j := \text{self}.y\} \\
C_3 &:: \text{get}(\text{Int} \; j)\{j := \text{self}.z \land \text{self}.b_3 \land (\text{var} \; C_1 \; x; J.\text{new}(x); x.\text{get}(j); \; \text{end} \; x)\}.
\end{align*}
\]
3.4. COMPLETENESS OF REFINEMENT RULES

We show that $\Gamma$ can be transformed into a class graph $\Gamma'$ of Normal Form I in the following steps, depicted in Fig. 3.10.

1. Remove redundant nodes and edges: It is easy to verify that $(O,b_1,\text{Bool}) \notin \text{Edges}(I)$ and $O \notin \text{Nodes}(I)$. We thus remove them using $\textbf{R8.2}$ and $\textbf{R8.1}$, respectively. The other nodes, edges and methods are accessed from the interface thus could not be further removed.

2. Merge classes: We first merge the supporting class $D_1$ into class $C_1$ using $\textbf{R9.3}$. The set of methods becomes

$$
\begin{align*}
C_1 &:: \text{get}(\text{Int } j)\{j := \text{self}.x_1\} & \quad C_1 &:: \text{set}(\text{Int } j)\{\text{self}.x_2 := j\} \\
J &:: \text{get}(\text{Int } j)\{\text{var } C_1 \ x; C_1.\text{new}(x); \text{set}(1); j := \text{self}.x_1 + x.x_2; \text{end } x\} \\
C_2 &:: \text{set}(\text{Int } j)\{(\text{var } \text{Int } k; \text{self}.a.\text{get}(k); \text{self}.x_2 := k; \text{end } k) \land \text{self}.b \triangleright \text{self}.x_2 := j\} \\
K &:: \text{get}(\text{Int } j)\{j := \text{self}.z\} & \quad D_2 &:: \text{get}(\text{Int } j)\{j := \text{self}.y\} \\
C_3 &:: \text{get}(\text{Int } j)\{j := \text{self}.z \land \text{self}.b \triangleright (\text{var } C_1 \ x; J.\text{new}(x); \text{get}(j); \text{end } x)\}.
\end{align*}
$$

Then, we use $\textbf{R10.1}$ to rename methods $K :: \text{get}$, $D_2 :: \text{get}$ and $C_3 :: \text{get}$ to $K :: \text{get}'$, $D_2 :: \text{get}'$ and $C_3 :: \text{get}'$. And after that, we use $\textbf{R9.3}$ to merge the supporting class $D_2$ into
<table>
<thead>
<tr>
<th>Rule</th>
<th>R11.1 Eliminate polymorphism - merge to superclass</th>
<th>R11.2 Eliminate polymorphism - merge to subclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>a class $D$ is merged into its direct superclass $C$, where $C$ and $D$ have polymorphic methods $m_0, \ldots, m_k$ ($k \geq 0$); an attribute $(C, b, Bool)$ is added whose initial value is false; each pair of polymorphic methods $m_j(S_j, x_j; T_j, y_j)(c_j) \in M(D)$ and $m_j(S_j, x_j; T_j, y_j')(c_j') \in M(C)$ is combined into one $m_j(S_j, x_j; T_j, y_j') { c_j \leftarrow \text{true} } (0 \leq j \leq k)$; then, the body command $c$ of each method is replaced by $c[D.new(le); le := true/D.new(le)]</td>
<td>C/D]$</td>
</tr>
<tr>
<td>Precondition</td>
<td>$(D, b, C) \in E$; $b$ is a fresh name; ${ m_0, \ldots, m_k } = { m \mid C \vdash m, D \vdash m \in M }$; for $D \vdash m \in M$ but $m \notin { m_0, \ldots, m_k }$, $C' \vdash m \notin M$ for each $C' \supseteq C$ or $C' \not\subseteq C$ but $C' \not\supseteq D$, $\text{attr}(C') \cap \text{attr}(D) = \emptyset$ for each $C' \not\subseteq C$ but $C' \not\supseteq D$; $D' \vdash m_j \in M$ for each $D' \supseteq D$ and $0 \leq j \leq k$</td>
<td>$(C, b, D) \in E$; $b$ is a fresh name; ${ m_0, \ldots, m_k } = { m \mid C \vdash m, D \vdash m \in M }$; for $C \vdash m \in M$ but $m \notin { m_0, \ldots, m_k }$, $D' \vdash m \notin M$ for each $D' \supseteq D$ or $D' \not\subseteq D$ but $D' \not\supseteq C$; $\text{attr}(D') \cap \text{attr}(C) = \emptyset$ for each $D' \not\subseteq D$ but $D' \not\supseteq C$; $D' \vdash m_j \in M$ for each $D' \supseteq D$ and $0 \leq j \leq k$</td>
</tr>
<tr>
<td>Interface</td>
<td>$\langle (N \cap C) \setminus { D }, M \setminus { D } \rangle$</td>
<td>$\langle (N \cap C) \setminus { D }, M \setminus { D } \rangle$</td>
</tr>
</tbody>
</table>

Figure 3.11: Rules for eliminating polymorphism

class $C_1$ and get the target graph $\Gamma'$. Notice that $\Gamma'$ is of Normal Form I and its methods are

$$
K \vdash \text{get('Int j'; j := self.x)} \quad C_1 \vdash \text{get('Int j'; j := self.x1)} \\
C_1 \vdash \text{set('Int j'; self.x2 := j)} \quad C_1 \vdash \text{get('Int j'; j := self.y)} \\
J \vdash \text{get('Int j'; var C_1; C_1.new(x); x.set(1); j := self.x1 + x.x2; end x)} \\
C_2 \vdash \text{set('Int j'; var Int k; self.a.get('k'); self.x2 := k; end k) \leftarrow self.b2 \supseteq self.x2 := j)} \\
C_3 \vdash \text{get('Int j'; j := self.z \supseteq self.b3 \supseteq \text{var C_1; x; C_1.new(x); x.get('j); end x})}.
$$

### 3.4.2 Eliminating polymorphism

The rules of structure refinement provided so far are not enough to eliminate every supporting class in a class graph. In fact, a supporting class can be associated with other classes through polymorphic inheritance edges, which can not be eliminated by any rule. In this sense, a normal form consists of either only functional classes or an inheritance tree with at least one supporting class.

We now consider the elimination of polymorphic inheritance edges so that all supporting classes can be combined into functional classes. For this purpose, we give a pair of rules in Fig. 3.11 to combine two classes $C, D$ with overriding methods $\{m_0, \ldots, m_k\}$ in a class graph $\Gamma = (N, E, M)$. In these rules, the notation $[D.new(le); le := true/D.new(le)]$ denotes a substitution that replaces each command of the form $D.new(le)$ by $D.new(le); le := true$. The preconditions of these rules are to avoid name conflicts of attributes and methods after the combination. In addition, the preconditions also requires an overriding method of $C$ and $D$ to be (re-)defined in each direct subclass of $D$, so that the combination of $C$ and $D$ does not affect the methods of other classes. In fact, such a requirement can always be satisfied through applications of R10.2.

The soundness of these rules are proved in the following theorem. The theorem also shows that they are equivalence rules in terms of structure refinement.
3.4. Completeness of Refinement Rules

Theorem 3.4.3 (Soundness of R11). If a class graph $\Gamma_1$ is transformed into another class graph $\Gamma_2$ through an application of R11 with interface $I$, $\Gamma_1 \equiv_I \Gamma_2$.

Proof. The fact that the application of R11 with interface $I$ transforms $\Gamma_1$ to $\Gamma_2$ implies that $I$ is a common interface of $\Gamma_1$ and $\Gamma_2$. In this proof, we slightly adjust the way that commands of $\Gamma_2$ are executed in the following aspects.

1. Notice that a command of the form $le.b := true$ in $\Gamma_2$ always occurs after $C.new(le)$. We regard each command of the form $C.new(le); le.b := true$ (but not $le.b := true$ itself) as an atom command which is executed in one step.

2. We redefine the execution of method invocations of polymorphic methods $\{m_0, \ldots, m_k\}$.

Let $con = \langle e, m_j(\ldots), G \rangle$ be a configuration where $0 \leq j \leq k$, $rtype(G,e) = C' \equiv_{\Gamma_2} C$ and $mbody_{\Gamma_2}(C', m_j) = (x_j; y_j); c_j \leftarrow self.b \triangleright c_j'. \triangleright$. If $eval(G,e,b) = true$, we define $con \rightarrow \langle e, m_j(\ldots), G \rangle$ where $c'$ is $c_j$ instead of $c_j \leftarrow self.b \triangleright c_j'$; otherwise, we define $con \rightarrow \langle e, m_j(\ldots), G \rangle$ where $c'$ is $c_j'$ instead of $c_j \leftarrow self.b \triangleright c_j'$.

The above adjustment does not affect the result of the execution of any valid command with respect to $\Gamma_2$ and $I$. Thus, it has no influence on the refinement relation between $\Gamma_2$ and other class graphs.

- Case R11.1.

Let $\rho_k$ be a relation from $S_I(\Gamma_1)$ to $S_I(\Gamma_2)$, such that $\rho_k(G_1, G_2)$ if and only if $G_2$ is obtained from $G_1[C/D]$ by adding edges $(a, a, c_n)$, $(a, b, false)$ for some object nodes $o \equiv_{\Gamma_2} C$ in $G_1[C/D]$, $a \in attr_{\Gamma_1}(D)$, and $cn = init_{\Gamma_1}(D, a)$, as well as edges $(a, b, true)$ for other object nodes $o \equiv_{\Gamma_2} C$ in $G_1[C/D]$. Notice that $\rho_k$ is in fact a bijection. Let $\rho_k$ be a function from commands definable in $\Gamma_1$ to commands definable in $\Gamma_2$ such that $\rho_k(c) = c[D.new(le); le.b := true/D.new(le)][C/D]$. It is straightforward to verify that $\rho_k$ and $\rho_k'$ satisfy all the conditions in Definition 3.1.3, thus $\Gamma_2$ is a simulation of $\Gamma_1$ with respect to $I$, $\rho_k$ and $\rho_k'$. According to Theorem 3.1.1, $\Gamma_1 \preceq_I \Gamma_2$.

- Case R11.2.

Let $\rho_k$ be a relation from $S_I(\Gamma_1)$ to $S_I(\Gamma_2)$, such that $\rho_k(G_1, G_2)$ if and only if $G_2$ is obtained from $G_1[C/D]$ by adding edges $(a, a, c_n)$, $(a, b, true)$ for some object nodes $o \equiv_{\Gamma_2} C$ in $G_1[C/D]$, $a \in attr_{\Gamma_1}(C)$, and $cn = init_{\Gamma_1}(C, a)$, as well as edges $(a, b, false)$ for other object nodes $o \equiv_{\Gamma_2} C$ in $G_1[C/D]$. Notice that $\rho_k$ is in fact a bijection. Let $\rho_k$ be a function from commands definable in $\Gamma_1$ to commands definable in $\Gamma_2$ such that $\rho_k(c) = c[D.new(le); le.b := true/D.new(le)][C/D]$. It is straightforward to verify that $\rho_k^{-1}$ and $\rho_k'$ satisfy all the conditions in Definition 3.1.3, thus $\Gamma_1$ is a simulation of $\Gamma_2$ with respect to $I$, $\rho_k^{-1}$ and $\rho_k'$. According to Theorem 3.1.1, $\Gamma_2 \preceq_I \Gamma_1$.
Example 3.4.2. Let Γ be the class graph depicted in Fig. 3.9 with interface I = \{\{C_1, C_2, C_3\}, \{C_1 :: get, C_2 :: set\}\}. In Example 3.4.1, we transformed Γ into a class graph Γ′, depicted in Fig. 3.12, of Normal Form I through applications of R8, R9 and R10. We can further transform Γ′ into a class graph Γ″ of Normal Form II in the following steps, depicted in Fig. 3.13.

1. Notice that J is a supporting class that is associated with class C_1 through a polymorphic inheritance edge. We merge class J into its direct superclass C_1 using R11.1. The set of methods becomes

\[
\begin{align*}
K &:: \text{get'}(\text{Int } j)\{j := \text{self}.z\} \\
C_1 &:: \text{get}(\text{Int } j)\{(\text{var } C_1 x; C_1\text{.new}(x); x\text{.set}(1)); j := \text{self}.x_1 + x.x_2; \text{end } x) \triangleleft \text{self}.b \triangleright j := \text{self}.x_1\} \\
C_1 &:: \text{set}(\text{Int } j); \{\text{self}.x_2 := j\} \quad C_1 &:: \text{get'}(\text{Int } j)\{j := \text{self}.y\} \\
C_2 &:: \text{set}(\text{Int } j); \{\text{var } C_1 k; \text{self}.a\text{.get'}(\text{Int } k); j := \text{self}.x_2 := k; \text{end } k) \triangleleft \text{self}.b \triangleright \text{self}.x_2 := j\} \\
C_3 &:: \text{get'}(\text{Int } j)\{j := \text{self}.z \triangleleft \text{self}.b \triangleright (\text{var } C_1 x; C_1\text{.new}(x); x.b := \text{true}; x\text{.get}(j); \text{end } x)\}.
\end{align*}
\]
3.5 Discussion: Practical Importance of Refinement Rules

We have provided a sound and complete refinement calculus that consists of a few sets of graph-based refinement rules. The completeness says that the refinement rules are expressive enough to achieve comprehensive refinements, e.g. to transform a class structure into a normal form, when applied in combination, though each of them is just a simple graph transformation. In this section, we show that the refinement calculus is also very important in practice and it has been applied in the development of many real software systems.

First, the refinement calculus is powerful in deriving useful design patterns. We show, for instance, in Fig. 3.14 two derived refinement rules DR1 and DR2 that characterize Expert Pattern
and Low Coupling design patterns [69], respectively. Expert Pattern is used to delegate some functionality of a class to its associated class(es), while Low Coupling is used to decompose a class into a few loosely coupled classes. Notice that Expert Pattern and Low Coupling are among the essential and most widely used design patterns for OO design. This indicates the expressiveness and usefulness of our refinement in the development of real OO systems.

In Fig. 3.14, each rule transforms a class graph \( \Gamma = \langle N, E, M \rangle \) into another class graph \( \Gamma' = \langle N', E', M' \rangle \). And in the description of these rules, we use the notation \( c(c_1) \) to denote a command \( c \) which has a sub-command \( c_1 \). This sub-command can be replaced by another command \( c_2 \) and we denote the resulting command as \( c(c_2) \). The following theorem shows the soundness of the derived rules, i.e. they are indeed derived from the refinement calculus.

**Theorem 3.5.1** (Soundness of derived refinement rules). If a class graph \( \Gamma \) is transformed into another class graph \( \Gamma' \) through an application of DR1 or DR2 with interface \( I \), \( \Gamma \) can be transformed into \( \Gamma' \) through a sequential application of refinement rules R1 to R11 with interface \( I \), thus \( \Gamma \equiv_I \Gamma' \).

**Proof.** We identify a sequential application of refinement rules that transforms \( \Gamma \) into \( \Gamma' \).

- Case \( \Gamma \) is transformed into \( \Gamma' \) by DR1.
  
  1. Add a method \( m'(\{c'\}) \) into class \( D \), where \( c' = c_1[\text{self}/\text{self}.a] \), using R10.3. (2) Replace

<table>
<thead>
<tr>
<th>Derived Rule (DR)</th>
<th>DR1 Expert pattern</th>
<th>DR2 Low Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td><img src="image1" alt="Diagram DR1 Expert pattern" /></td>
<td><img src="image2" alt="Diagram DR2 Low Coupling" /></td>
</tr>
<tr>
<td>add a method ( m'({c'}) ) into a class ( D ), where ( c' = c_1[\text{self}/\text{self}.a] ), provided ( (C, a, D) \in E, m(){c(c_1)} \in M(C) ) and the sub-command ( c_1 ) has direct access of an attribute ( x ) of class ( D ) (through ( \text{self}.a.x )); then, replace ( c_1 ) with ( \text{self}.a.m'()); as a result, ( C :: m ) access the attribute ( x \in \text{attr}(D) ) through calling ( D :: m' ) instead of accessing it directly via ( c_1 )</td>
<td>decompose a class ( C ) with attributes ( x_1, x_2 ) and methods ( m_1(){c_1} ) and ( m_2(){c_2[\text{self}.m_1()]} ) into classes ( C, D_1, D_2 ) and associations ( (C, a_1, D_1) ) and ( (D_2, a_1, D_1) ), provided ( c_1 ) and ( c_2 ) have direct accesses of ( x_1 ) and ( x_2 ), respectively, ( c_1 ) does not access ( x_2 ) and ( c_2 ) accesses ( x_1 ) only via ( c_1 ) (i.e. through calling ( \text{self}.m_1() )); ( x_1 ) and ( x_2 ) turn to attributes of ( D_1 ) and ( D_2 ), respectively</td>
<td></td>
</tr>
<tr>
<td>Original methods</td>
<td>( C :: m(){c(c_1)} )</td>
<td>( C :: m_1(){c_1} )</td>
</tr>
<tr>
<td>Resulting methods</td>
<td>( C :: m(){c(\text{self}.a.m'()}) ) ( D :: m'({c'}) ) with ( c' = c_1[\text{self}/\text{self}.a] )</td>
<td>( C :: m_1(){\text{self}.a.m_1()} ) ( C :: m_2(){\text{self}.a.m_2()} )</td>
</tr>
<tr>
<td>Precondition</td>
<td>( c_1 ) does not access any attribute or method of class ( C ); ( D' :: m' \not\in M ) for any class ( D' \not\subseteq D )</td>
<td>( D_1, D_2 \not\subseteq N ); ( a_2 \not\in attr(C') ) for any class ( C' \not\subseteq C ) or ( C' \not\supset C )</td>
</tr>
<tr>
<td>Interface</td>
<td>( (\langle N \cap \mathcal{C} \rangle, M) )</td>
<td>( (\langle N \cap \mathcal{C} \rangle, M) )</td>
</tr>
</tbody>
</table>

Figure 3.14: Derived refinement rules
the command $c_1$ by the method invocation $self.a.m'(\cdot)$ using R10.4. The resulting class graph is $\Gamma'$. 

- Case $\Gamma$ is transformed into $\Gamma'$ by DR2.

1. Add a class $D_2$ using R3 and then an association $(C, a_2, D_2)$ using R4.1. (2) Move the attributes $x_1$ and $x_2$ into class $D_2$ using R5.1, so that the body commands of $C :: m_1$ and $C :: m_2$ become $c_1[\text{self}.a_2.x_1/\text{self}.x_1]$ and $c_2[\text{self}.m_1(\cdot)][\text{self}.a_2.x_2/\text{self}.x_2]$, respectively. (3) Add a method $m_1(\cdot)|c_1|$ into class $D_2$ using R10.3 and then replace the body command of $C :: m_1$ by $\text{self}.a_2.m_1(\cdot)$ using R10.4. (4) Expand the method invocation $\text{self}.m_1(\cdot)$ in $C :: m_2$ using R10.5, so that the body command of $C :: m_2$ becomes $c_2[\text{self}.a_2.m_1(\cdot)][\text{self}.a_2.x_2/\text{self}.x_2]$.

(5) Add a method $m_2(\cdot)|c_2[\text{self}.m_1(\cdot)]|$ into class $D_2$ using R10.3 and then replace the body command of $C :: m_2$ by $\text{self}.a_2.m_2(\cdot)$ using R10.4. (6) Add a class $D_1$ using R3 and then an association $(D_2, a_1, D_1)$ using R4.1. (7) Move the attribute $x_1$ into class $D_1$ using R5.1, so that the body command of $D_2 :: m_1$ becomes $c_1[\text{self}.a_1.x_1/\text{self}.x_1]$. (8) Add a method $m_1(\cdot)|c_1|$ into class $D_1$ using R10.3 and then replace the body command of $D_2 :: m_1$ by $\text{self}.a_1.m_1(\cdot)$ using R10.4. (9) Expand the method invocation $\text{self}.m_1(\cdot)$ in $D_2 :: m_2$ using R10.5, so that the body command of $D_2 :: m_2$ becomes $c_2[\text{self}.a_1.m_1(\cdot)]$. The resulting class graph is $\Gamma'$.

\[ \] 

In addition to the characterization of OO design patterns, the refinement calculus has been successfully applied in the modeling of real software systems. One of them is the Common Component Modeling Example (CoCoME) [29, 30], an international benchmarking project for comparing and evaluating the practical appliance of existing component-based models and their corresponding specification techniques [90]. In particular, the refinement calculus has been used in the modeling of an extended version the point-of-sale terminal (POST), which was originally used as a running example in Larman’s book [69] to demonstrate the concepts, modeling and design of object-oriented systems. Another successful application of the refinement calculus is the development of a trustable medical system in telemedicine practice [100], which aims at improving the efficiency and quality of health care.

Also notice that the graph-based refinement rules and their soundness and completeness results, which are proved based on our operational semantics presented in Chapter 2, agree with those proved based on the original denotational semantics of rCOS [54]. This gives the justification of the correctness of both our graph-based operational semantics and the UTP-based denotational semantics of rCOS. On the other hand, the advantage of our operational semantics and refinement calculus lies in the visualization of OO concepts and the intuitiveness of graph notations. The graph-based operational semantics and refinement calculus has been used as the theoretical basis of the rCOS Tool [31, 91] for OO and component-based model-driven development.

### 3.6 Summary

In this chapter, we study the notion of structure refinement and investigate the relation between the transformation of the structure and that of the functionality of OO programs. We base our discussion on the graph model and operational semantics provided in Chapter 2, and define structure refinement as a graph transformation between class graphs that maintains the capability of providing services. As the main contribution of this chapter, we provide a calculus of structure refinement with a few groups of graph transformation rules. These rules characterize various cases of structure refinement, including refinement for class structure expansion and decomposition, for class structure compression and abstraction, as well as refinement for method transformation and polymorphism elimination.

We proved the soundness of the refinement calculus. That is, each graph transformation rule transforms a class graph to a refined one. We also proved the completeness of the refinement calculus with respect to the notions of structure transformation and normal form. That is, the
combination of refinement rules enables us to achieve every structure transformation, and transform every class graph into a normal form.

The refinement calculus, which is based on the operational semantics, agrees with all the refinement rules that are defined and proved based on the denotational semantics of rCOS [54]. This gives the justification of the correctness of both the operational semantics and the denotational semantics. In addition, the refinement calculus is shown expressive in characterizing fundamental design patterns for the design of OO programs. It has been further applied in meaningful case studies, e.g., enterprise systems [29, 30] and trustable medical systems [100], and used as the theoretical basis of the rCOS Tool [31, 91] for model-driven software development.
Chapter 4

Graph Representation of Service-Oriented Systems

In this chapter, we propose a hierarchical graph representation of structured service systems specified in a service-oriented formal language. Our main objective is to develop a sound and complete graph transformation system for service-oriented computing (SOC).

The formal language we consider is Calculus of Session and Pipelines (CaSPiS) [13], a general service-oriented computation model that supports the key features of SOC such as service autonomy, client-service interaction and orchestration. In CaSPiS, a service system is specified as a process expression which is constructed through an atom process 0 and a set of process operators. These process operators include basic operators such as prefixing, parallel composition and restriction, like in the π-calculus [79], and also service-specific operators such as service definition, service invocation, session and pipeline. A session $r$ has two sides $r \triangleleft P$ and $r \triangleright Q$ through which it associates two processes $P$ and $Q$ so that they are ready to interact with each other, while a pipeline $P > Q$ orchestrates two processes $P$ and $Q$ by “plugging” $Q$ after $P$, so that $P$ can produce values one by one for $Q$ to consume. Services, sessions and pipelines may occur nested, representing the hierarchy of service systems. CaSPiS is equipped with a relation of structural congruence between processes. For example, processes of parallel compositions $P|Q$ and $Q|P$ are congruent.

In CaSPiS, the behaviors of service systems are characterized by the notion of reduction. Specifically, a reduction is a step of evolvement of a process to another, up to congruence, due to an interaction of two of its sub-processes. According to the reduction semantics of CaSPiS, a reduction can be caused by the interaction between a service definition and service invocation, between two sides of a session, or between two processes orchestrated by a pipeline. In either case, each (sub-)process taking part in the interaction should occur in a certain context, specifically a static context. For example, prefixing is a non-static context and thus a prefixed process $\ldots \pi P \ldots$ is not allowed to interact with any other process until the prefix $\pi$ is eliminated due to some other interaction.

For graph representation of service systems and their hierarchy, we use a graph algebra to specify hierarchical graphs and study their algebraic properties. Generally, a graph algebra consists of a syntax that defines a set of terms for specification and a semantics that interprets each term as a graph. As for the syntax, we adopt the one of the graph algebra provided in [19]. The syntax supports general constructs of hierarchical graphs such as composition, nesting and node restriction, and it is suitable to specify graphs for structured service systems. On the other hand, the semantics of the algebra interprets a hierarchical graph term as a multi-level structure of hypergraphs with mappings between hypergraphs of different levels [19]. Based on this semantics, however, the formalization of graph transformations that change the hierarchy of a graph is complicated which involves the rebuilding of levels of hypergraphs as well as their mappings. To improve such situation, we define a novel semantics for the syntax which interprets a hierarchical graph term as a single hypergraph with different levels of nodes and edges connected by a special kind of edges, namely
abstract edges. As a result, it is much easier to handle the change of the hierarchy of a graph, i.e. through adjusting the layout of abstract edges. Furthermore, the graph model makes it straightforward to study the algebraic notions of morphism and pushout among hierarchical graphs, enabling us to define transformations of hierarchical graphs in the well-studied Double-Pushout (DPO) approach.

Based on the graph model, we propose a graph representation of service systems specified in CaSPiS, involving the representation of the states (processes) and that of the behavior (reduction). To represent the states, we first provide a direct representation of processes which is defined according to the structures of processes. For example, the graph of a parallel composition $P|Q$ is constructed from the graphs of $P$ and $Q$ that are associated with an edge labeled by $Par$ for parallel composition. To represent the hierarchy of a service system, we explicitly encapsulate the graph of every service, session and pipeline before composing it with the rest of the system. Such a representation of processes is natural to understand, but it is not sufficient to be the basis for further reductions. As the reduction semantics of CaSPiS requires, an interaction can only happen between two sub-processes occurring in static contexts. While in the direct representation, we cannot distinguish between sub-processes occurring in different contexts. To overcome such insufficiency, we provide a tagged graph presentation of processes. It is derived from the direct representation, while we attach tags to graphs of sub-processes occurring in static contexts.

As the representation of the behaviors of service systems, we define a graph transformation system to characterize the reductions of CaSPiS processes, which consists of the following sets of graph transformation rules in the DPO approach.

1. Tagging rules. These rules operate the tags so that we can obtain the tagged graph of a process from its untagged version.

2. Rules for congruence. These rules aim at modeling the congruence relation between CaSPiS processes. Each rule transforms the tagged graph of a process into that of a congruent process.

3. Rules for reduction. These rules aim at characterizing the reductions of CaSPiS processes and they deal with the interactions of services, sessions and pipelines. However, they may produce garbages as well as auxiliary edges for further data assignment which do not belong to the graph of any process.

4. Rules for garbage collection and data manipulation. These rules are auxiliary ones used to remove the garbages and consume the auxiliary edges produced by the third set of rules.

As for the soundness and completeness of the graph transformation system, we will show that the first two sets of rules are sound and complete with respect to the congruence relation of CaSPiS processes, while all the rules as a whole are sound and complete with respect to the the reduction semantics of CaSPiS processes.

We summarize the service-oriented calculus CaSPiS [13] in Section 4.1 and introduce the syntax of the algebra of hierarchical graphs [19] in Section 4.2. In Section 4.3, we present our semantic model of the graph algebra which supports graph transformations in the DPO approach. Based on this graph model, we show in Section 4.4 how to represent CaSPiS processes as hierarchical graphs. We also provide our graph transformation system on these graphs for representation of the behaviors of processes and study the soundness and completeness of the system.

4.1 Background: the Calculus CaSPiS

In this section, we introduce the key notions of the service-oriented calculus CaSPiS [13]. Assume two infinite sets $S$ and $R$ of service names and session names, respectively. Assume also an infinite set $G$ of constructors $f$, each with a fixed arity $\text{ar}(f)$. Constants are allowed in CaSPiS, and a constant $c$ is regarded as a constructor of arity 0.

For a sequence of elements $\vec{u}$, let $\vec{u}[j]$ denote the $j$-th element, $|\vec{u}|$ the length and $\{|\vec{u}\}$ the set of elements of $\vec{u}$, respectively.
4.1.1 Basic processes

We first introduce the fragment of CaSPiS without considering the replication of processes.

The simplest process is the nil process \(0\) that does not do anything. A process \(P\) can be prefixed by a concretion \(\langle V\rangle\) that generates a value \(V\); a return \(\langle V\rangle^\uparrow\) that returns a value \(V\) to the outside environment; or an abstraction \(\langle F\rangle\) that is ready to receive a value which matches the pattern \(F\). Such a process is called a prefixed process. In a prefixed process, a value is simply a value variable \(x\), or a constructed value \(f(\bar{V})\) composed of a sequence of values through a constructor. Similarly, a pattern can be a pattern variable of the form \(?x\) or a constructed pattern \(f(\bar{F})\).

The standard parallel composition \(P|Q\) is allowed. However, the choice operator “+”, called summation, is limited to the nil process and prefixed processes.

A service is declared by a service definition \(s.P\) and used by the environment through a service invocation \(s.Q\). A participant process of a session \(r\) is represented by \(r \triangleright P\), where \(P\) is the protocol process this participant follows. In CaSPiS, a session \(r\) can have only two participants, and they are also called the two sides of the session.

A process \(P\) can be pipelined with another process \(Q\), denoted by \(P > Q\), so that \(P\) can keep producing values for \(Q\) to consume. Service names, session names and variables can be restricted, in a way like the \(\pi\)-calculus [79] by \((\nu n)P\). In this process, \(P\) is the scope of the restriction, i.e. \((\nu n)\) restricts all the occurrences of the name \(n\) within \(P\).

**Definition 4.1.1 (Basic process).** A basic CaSPiS process is a term generated by the syntax:

\[
\begin{align*}
\text{Process} & \quad P, Q ::= U \mid P|Q \mid s.P \mid \pi.P \mid r \triangleright P \mid P > Q \mid (\nu n)P \\
\text{Sum} & \quad U ::= 0 \mid (F)P \mid \langle V \rangle P \mid \langle V \rangle^\uparrow P \mid U + U \\
\text{Pattern} & \quad F ::= ?x \mid f(\bar{F}) \\
\text{Value} & \quad V ::= x \mid f(\bar{V})
\end{align*}
\]

where \(s \in S\), \(r \in R\), \(x \in V\), \(f \in G\) and \(n \in S \cup R \cup V\). Recall that \(V\) is the name space of variables.

It is worth pointing out that the session construct \(r \triangleright P\) is a runtime syntax: it should not be used to model the initial state of a system, but can be dynamically generated upon service invocation.

A nil process \(0\) can be omitted when it is prefixed. For example, \(\langle ?x \rangle(x)0\) is a shorthand for \(\langle ?x \rangle(x)\). For a pattern \(F\), \(bn(F)\) denotes the set of its bound names, i.e. names \(x\) such that \(?x\) occurs in \(F\). A name \(n\) occurring in a process \(P\) can be bound by either a restriction \((\nu n)\) or an abstraction \(\langle F\rangle\) with \(n \in bn(F)\). Otherwise, it is a free name, and \(fn(P)\) denotes the set of free names of \(P\). For a value \(V\), the same notation \(fn(V)\) is used to denote the set of variables occurring in \(V\). Notice that a variable always occurs free, i.e. can not be bound, in a value.

**Congruence of processes.** As in the \(\pi\)-calculus [79], processes are equalized up to alpha-conversion. For example, \(\langle ?x \rangle \langle x \rangle \langle z \rangle\) and \(\langle ?y \rangle \langle y \rangle \langle z \rangle\) specify the same process. In addition, CaSPiS is equipped with a set of structural congruence rules among processes. They are classified as basic rules of commutativity and associativity (shown in Fig. 4.1) and special rules for moving a restriction “forward”, out of a parallel composition, session or pipeline (shown in Fig. 4.2). It can be inferred that congruent processes have the same set of free names.

4.1.2 Operational semantics in terms of reduction

The basic behavior of a process \(P\) is the communication and synchronization (called interactions) between its sub-processes. After an interaction, \(P\) evolves to another process \(Q\). Such a step of evolvement is called a reduction, denoted as \(P \rightarrow Q\).

The behaviors of prefixed processes, sum processes, parallel compositions and restrictions are similar to those in a traditional process calculus. A service definition process \(s.P\) and service invocation process \(\pi.Q\) synchronize on the service \(s\) and its corresponding invocation \(\pi\). After
offering the service \( s, s.P \) evolves to a session process \( r \triangleright P \) with a fresh session name \( r \). Symmetrically, after the service invocation \( \bar{s}, \bar{s}.Q \) becomes the other session side \( r \triangleright Q \) of \( r \). For example, \( s.P|\bar{s}.Q \rightarrow r \triangleright P|r \triangleright Q \). When a session \( r \) starts, the protocols \( P \) and \( Q \) of the session sides \( r \triangleright P \) and \( r \triangleright Q \) become active and produce and receive values from each other. For example, \( r \triangleright (\bar{y})P|r \triangleright (\bar{y})Q \rightarrow r \triangleright P[y/x]|r \triangleright Q \).

A pipelined process \( P > Q \) behaves as \( P \) but keeps the new state of \( P \) pipelined with \( Q \), until \( P \) produces a value. When \( P \) produces a value, a new instance of \( Q \) is created, that consumes the value produced by \( P \) and then runs in parallel with the original \( P \) and instances of \( Q \) created earlier. This is shown by the example \( (\bar{y})P > (\bar{x})Q \rightarrow (P > (\bar{x})Q)[Q[y/x]].\)

**Context.** The formal definition of reduction needs the notion of process context, i.e. a process expression with “holes”. Specifically, a context with \( k \) holes is a process term \( \Lambda[X_1, \ldots, X_k] \) defined in Definition 4.1.1, but containing processes variables \( X_1, \ldots, X_k \). When replacing these process variables respectively by processes \( P_1, \ldots, P_k \), we get a process \( \Lambda[P_1, \ldots, P_k] \). But the context itself can be simply denoted as \( \Lambda[\cdot, \ldots, \cdot] \), with the process variables omitted. In most cases in this chapter, we only need to consider contexts with one or two holes, i.e. \( \Lambda[\cdot] \) or \( \Lambda[\cdot, \cdot] \).

A context is called static if none of its holes occurs in the scope of a dynamic process operator, which is either a service definition \( s[\cdot] \), a service invocation \( \bar{s}[\cdot] \), a sum \([\cdot] + U \) or \( U + [\cdot] \), a prefix \( \pi[\cdot] \) or the right-hand side of a pipeline \( P > [\cdot] \). A context is called session-immune and restriction-immune if its hole(s) does not occur in the scope of a session and restriction, respectively. Moreover, a 2-hole context is called restriction-balanced if the holes occur in the same restriction environment. For example, \((\bar{vm})[\cdot][r \triangleright [\cdot]] > Q \) is not restriction-balanced, as only its first hole is bound by the restriction \( (\bar{vm}) \). Nevertheless, it is a static context.

**Reduction rules.** Following the discussion about the informal behavior of processes, we summarize the reduction rules for service definition, service invocation, session and pipelined processes in Fig. 4.3, where each rule shows a pair of processes \( P \) and \( Q \) such that \( P \rightarrow Q \). In these rules, it is required \( \Lambda_0[\cdot] \) is static; \( \Lambda[\cdot, \cdot] \) is static and restriction-balanced; \( \Lambda_1[\cdot] \) and \( \Lambda_2[\cdot] \) are static, session-immune and restriction-immune. So, there is no rule that allows a reduction to take place in a non-static context. In addition, the last four rules require that the substitution \( \sigma = \text{match}(F; V) \) exists, which is calculated from the pattern \( F \) and the value \( V \). For example, \( \text{match}(f(\bar{x}, ?y); f(z, g(1))) = [z, g(1)/x, y] \), while \( \text{match}(f(\bar{x}, ?y); g(2)) \) does not exist as pattern \( f(\bar{x}, ?y) \) and value \( g(2) \) do not match. Formally, \( \sigma = \text{match}(F; V) \) is the substitution such that \( \text{dom}(\sigma) = \text{bn}(F) \) and \( \hat{F}\sigma = V \), where \( \hat{F} \) denotes the value obtained from \( F \) by replacing each \(?x\) with \( x \).
4.1. BACKGROUND: THE CALCULUS CASPIS

Well-formed process. To represent a meaningful service system, a process term must satisfy
structural of processes that are not likely to occur in a service system. From now on, a process by
time, through the reduction rule (Sync). The other conditions are used to rule out certain con-
tions for sessions reflect the consideration that a session can only be generated at run-
Example. Let us consider the process $Q | (Cl > (?y)P)$, where $Q = req.(\nu\ell) (((\ell) + \langle\text{null}\rangle))$ is a
service to allocate new resources (if available), $Cl = \overline{req}.(?x)\langle x \rangle$ is a client of $Q$ and $P$ is a generic
process. Then the above process can evolve as illustrated below.

$$
Q | (Cl > (?y)P) \rightarrow (\nu\ell)(r \triangleright (\nu\ell)((\ell) + \langle\text{null}\rangle) \mid (r \triangleright (?x)\langle x \rangle > (?y)P)) \quad \text{(Sync)}
$$

$$
\equiv_c (\nu\ell)(r \triangleright (\nu\ell)((\ell) + \langle\text{null}\rangle) \mid (r \triangleright (?x)\langle x \rangle > (?y)P))
\rightarrow (\nu\ell)(r \triangleright (\nu\ell)((\ell) + \langle\text{null}\rangle) \mid (r \triangleright (?x)\langle x \rangle > (?y)P)) \quad \text{(S-Sync)}
\rightarrow (\nu\ell)(r \triangleright 0 \mid (r \triangleright 0 > (?y)P) \mid P[\text{null}/y]) \quad \text{(P-Sync-Ret)}
$$

Figure 4.3: Reduction rules

Notice that $r \triangleright 0$ is inert and therefore $r \triangleright 0 > (?y)P$ is also inert, then the reached process
amounts essentially to $P[\text{null}/y]$. An analogous computation could have led (up to the presence
of inert processes) to the process $(\nu\ell)P[\ell/y]$.

Well-formed process. To represent a meaningful service system, a process term must satisfy
the following well-formedness conditions.

1. Conditions for sessions:
   - each session occurs in a static context,
   - each session name occurs at most twice (module alpha-conversion), i.e. a session has at
     most two sides, and
   - sessions are nested in an acyclic way. For example, terms like $r \triangleright r \triangleright P$ or $r \triangleright r_1 \triangleright P|r_1 \triangleright r \triangleright Q$ are illegal.

2. Conditions for patterns, sums and pipelines:
   - a pattern variable occurs at most once in each pattern, and
   - a sum has at most one kind of prefixes, for example $(?x)P + (?y)Q$ or $\langle x \rangle P + \langle y \rangle Q + 0$,
     and
   - the right-hand side of a pipeline is an abstraction or a sum of abstractions.

The conditions for sessions reflect the consideration that a session can only be generated at run-
time, through the reduction rule (Sync). The other conditions are used to rule out certain con-
structs of processes that are not likely to occur in a service system. From now on, a process by
default means a well-formed one unless it is stated otherwise.
4.1.3 Extension with replications

A service system may contain a service definition that can be invoked repeatedly, or an abstraction that is always ready to receive a value and take corresponding actions. In order to specify such systems, a new construct of processes, replication, is introduced into CaSPiS.

Definition 4.1.2 (Process). The syntax of CaSPiS processes is an extension of the basic one in Definition 4.1.1 given by:

\[
\text{Process} \quad P, Q ::= \ldots \quad \text{(Basic constructs)} \quad | \quad !P \quad \text{(Replication)}
\]

A replication !P is well-formed if its body P is either a service definition, an abstraction or a sum of abstractions. In the following discussion, a replication always means a well-formed one unless it is stated otherwise.

The newly introduced construct ![·] is a dynamic operator. So, no reduction is allowed to occur inside the body of a replication. Instead, the behavior of a replication is defined by a new special congruence rule.

\[
!P \equiv_c P || P
\]

The extended set of special congruence rules is shown in Fig. 4.4.

A replication can take part in a reduction (only) indirectly, i.e. after it is “unfolded” by the new congruence rule. For example, given \( P|Q \rightarrow R, P||Q \equiv_c P||Q \rightarrow R||Q \).

4.2 Background: Syntax of Hierarchical Graphs

In this section, we summarize the syntax of the algebra of hierarchical graphs introduced in [19], which we adopt as the basis of our graph model.

For simplicity, we first present the syntax of graph terms without hierarchy. Let \( \mathcal{N} \) be a set of nodes and \( \mathcal{L} \) be a set of edge labels.

Definition 4.2.1 (Graph term). A graph term is generated by the syntax

\[
\text{Graph} \quad G ::= 0 \mid v \mid l(\vec{v}) \mid G|G \mid (\nu v)G
\]

where \( v \in \mathcal{N} \) and \( l \in \mathcal{L} \).

Term 0 specifies the empty graph, \( v \) specifies the graph of only one node named by \( v \), \( l(\vec{v}) \) is used to specify the graph of an \( l \)-labeled edge attached to nodes \( \vec{v} \) through its tentacles, \( G_1|G_2 \) is for the composition of two graphs and \( (\nu v)G \) is a restriction that binds the node \( v \) in \( G \) so that it is invisible outside. A node \( v \) occurring in a graph term is free if it is not bound by a restriction \( (\nu v) \). The intuition of two example graph terms is shown in Fig. 4.5. We will strictly define the interpretation of each graph term, i.e. the graph that it specifies, in the next section.
Hierarchical graph terms. The syntax presented above is suitable to specify closed graphs that represent closed systems. However, it is not sufficient to deal with open systems and their compositions. This motivates the extension of the syntax with hierarchical graph terms, through the notion of design. Assume a set $D$ of design labels.

**Definition 4.2.2** (Hierarchical graph term). A hierarchical graph term, graph term for short, is either a graph or a design generated by the syntax

\[
\text{Graph} \quad G \ ::= \ 0 \ | \ v \ | \ l(v) \mid G(G) \mid (\nu v)G \mid Z(v)
\]

\[
\text{Design} \quad Z \ ::= \ L_w[G]
\]

where $v \in \mathcal{N}$, $l \in \mathcal{L}$ and $L \in \mathcal{D}$.

A design $Z = L_w[G]$ makes an encapsulation of a graph $G$ and exposes a sequence of free nodes $\vec{w}$ of $G$ for composition with the environment. Thus $G$ and $\vec{w}$ are called the body graph and the (sequence of) exposed nodes of $Z$, respectively. Given a design $Z$, a design edge $Z(\vec{w})$ is obtained from $Z$ by attaching its exposed nodes to the nodes $\vec{v}$. Notice that the body graph of a design may further contain design edges of different designs. With nested designs, a graph term as well as the graph it specifies is indeed hierarchical. A node $v$ occurring in a hierarchical graph term is free if it is neither bound by a restriction ($\nu v$) nor exposed by a design. The intuition of an example hierarchical graph term is shown in Fig. 4.6.

A graph has different types of nodes for modeling different entities. Assume a set $\mathcal{O}$ of node types so that each node $v$ has a fixed type $T(v) \in \mathcal{O}$. Besides, each edge label $l$ (or design label $L$) has a fixed arity $AR(l)$ and a fixed type $T(l)$ which is a sequence of node types with $|T(l)| = AR(l)$. This means each $l$-labeled edge should be of arity $AR(l)$ and be associated with a sequence of nodes of types $T(l)$, respectively. Similarly, each design label $L$ has a fixed arity $AR(L)$ and a fixed type $T(L)$ with $|T(L)| = AR(L)$. This means each $L$-labeled design should be of arity $AR(L)$, and its exposed nodes should be of types $T(L)$ so that they can only be attached to nodes of types $T(L)$.

Specifically, a graph term $G_0$ is well-typed if

- for each occurrence of $L_{\vec{w}}[G]$ in $G_0$, $\vec{w}$ only consists of free nodes of $G$ and $T(\vec{w}) = T(L)$,
- for each occurrence of $Z(\vec{v})$ in $G_0$, $T(\vec{v}) = T(L)$, where $L$ is the label of $Z$, and
- for each occurrence of $l(\vec{v})$ in $G_0$, $T(\vec{v}) = T(l)$.

From now on, a graph term always means a well-typed one unless it is stated otherwise.
CHAPTER 4. GRAPH REPRESENTATION OF SERVICE-ORIENTED SYSTEMS

Flat design edges. It is worth pointing out that a design edge \( L_{\vec{w}}[\vec{v}] \) plays two roles in a graph term. First, it represents the interface of a (sub-)graph \( G \) through which \( G \) can be associated with the surrounding nodes and edges. In other words, the design edge is introduced for graph composition. The second role of a design edge is to represent an encapsulation of a sub-graph which contributes to the hierarchy of the whole graph, i.e. the design edge is for the clarification of the hierarchy. When \( L_{\vec{w}}[\vec{v}] \) is introduced merely for the first role, it can be made flat by collapsing the exposed nodes \( \vec{w} \) of the design with the nodes \( \vec{v} \) they are attached to. For example, Fig. 4.7(b) shows the flat version of the design edge of Fig. 4.7(a).

To syntactically indicate in a graph term whether a design edge is flat, we assume a designated subset of flat design labels \( F \subseteq D \), i.e. a design edge is flat if and only if its label \( L \in F \). In the next section, we will strictly define the interpretation of graph terms, where designs edges with normal and flat design labels are interpreted in different ways.

4.3 Model of Hierarchical Graphs

In this section, we provide our graph model that interprets a hierarchical graph term defined in the previous section as a graph. Notice that such a graph is actually a hypergraph as it allows an edge to be associated with more than two nodes. To characterize the hierarchy of graph terms introduced by designs, we introduce a special kind of binary edges, namely abstract edges, into their interpretation hypergraphs. An abstract edge attaches an exposed node of a design to a node outside, so that the design is represented by a set of abstract edges through which the body graph of the design is associated with the environment.

Specifically, a hypergraph is a tuple with (normal) edges, abstract edges, free nodes and interface nodes.

Definition 4.3.1 (Hypergraph). A hypergraph is a 5-tuple \( \langle N, E, AE, fn, in \rangle \), where

- \( N \subseteq \mathcal{N} \) is the set of nodes,
- \( E \) is the set of edges, each of which is of the form \( l(\vec{v}) \) where \( l \in \mathcal{L} \) and \( \vec{v} \) is a sequence of nodes in \( N \),
- \( AE \) is the set of abstract edges, each of which is of the form \( L^k_j(v_1, v_2) \) where \( v_1, v_2 \in N \) and the label \( L^k_j \) is a design label \( L \in D \) with an integer subscript \( j \) \((1 \leq j \leq AR(L))\) and an optional integer superscript \( k \),
- \( fn \subseteq N \) is the set of free nodes, and
- \( in \) is the sequence of interface nodes, which only consists of nodes in \( N \setminus fn \), i.e. an interface node is not free.

Notice that we define the label of an abstract edge \( L^k_j \) as a design label with a subscript and a superscript. The design label \( L \) and the subscript \( j \) are used to indicate that the abstract edge is to associate the \( j \)-th exposed node of an \( L \)-labeled design to the environment. The superscript \( k \) is used to discriminate different occurrences of \( L \)-labeled designs, i.e. different superscripts are used to label abstract edges of different \( L \)-labeled designs. This superscript can be omitted when
there is only one $L$-labeled design of our concern. Now, we formally define the interpretation of graph terms.

Definition 4.3.2 (Interpretation of graph terms). A graph term $G$ is interpreted as a hypergraph $\mathcal{H}(G) = \langle N(G), E(G), AE(G), \text{fn}(G), \text{in}(G) \rangle$ defined as follows, where $N(G)$ is the set of nodes names, $E(G)$ the set of edges, $AE(G)$ the set of abstract edges, $\text{fn}(G)$ the set of free nodes, and $\text{in}(G)$ the sequence of interface nodes. We call $\mathcal{H}(G)$ the hypergraph of $G$.

\[
\begin{align*}
\mathcal{H}(0) &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
\mathcal{H}(v) &= \langle \{v\}, \emptyset, \{v\}, \emptyset \rangle \\
\mathcal{H}(l(v)) &= \langle \{l\}, \{l\}, \emptyset, \{l\}, \emptyset \rangle \\
\mathcal{H}((\nu v)G_1) &= \langle N(G_1), E(G_1), AE(G_1), \text{fn}(G_1) \setminus \{v\}, \emptyset \rangle \\
\mathcal{H}(G_1 G_2) &= \langle N(G_1) \cup N(G_2), E(G_1) \cup E(G_2), AE(G_1) \cup AE(G_2), \text{fn}(G_1) \cup \text{fn}(G_2), \emptyset \rangle \\
\mathcal{H}(L_w[G_1]) &= \langle N(G_1) \cup \{\vec{w}\}, E(G_1), AE(G_1) \cup \{\vec{w}\} \cup \{\vec{w}\}, \text{fn}(G_1) \setminus \{\vec{w}\}, \text{fn}(G_1) \setminus \{\vec{w}\}, \emptyset \rangle \\
\mathcal{H}(L_w[G_1]) &= \langle N(G_1) \setminus \{v\}, E(G_1) \setminus \{v\}, AE(G_1) \setminus \{v\}, \text{fn}(G_1) \setminus \{\vec{v}\}, \text{fn}(G_1) \setminus \{\vec{v}\}, \emptyset \rangle \\
\mathcal{H}(L_w[G_1]) &= \langle N(Z) \setminus \{v\}, E(Z) \setminus \{v\}, \text{fn}(Z) \setminus \{v\}, \text{fn}(Z) \setminus \{v\}, \emptyset \rangle \\
\mathcal{H}(L_w[G_1]) &= \langle L \not\in F, Z = L_w[G_1] \rangle
\end{align*}
\]

According to the intuition given in the previous subsection, the interpretation of a node, edge, restriction composition of a graph term is straightforward and easy to understand. An $L$-labeled design $Z$ is generally characterized by a set of abstract edges labeled $L^1_k, \ldots, L^k_{AR(L)}$ that link the exposed nodes to a set of fresh nodes as the interface nodes for composition with the environment. As we have illustrated, the superscript $k$ is different for different $L$-labeled designs. A design edge $Z(\vec{v})$ is interpreted in the same way as its design $Z$, while the sequence of interface nodes are replaced by $\vec{v}$. For design edges with flat design labels, we further collapse its corresponding abstract edges, which actually leads to the unification of the exposed nodes with $\vec{v}$. Notice that terms $(\nu v)(\nu v)G$ and $(\nu v)(\nu v)G$ are interpreted as the same hypergraph. We thus extend the restriction operator to a set of nodes and write, for example, $(\nu \{v, w\})G$ for $(\nu v)(\nu w)G$ and $(\nu v)(\nu w)G$.

We show two examples of the interpretation in Fig. 4.8, where the hypergraphs of the following terms are depicted, with $L \not\in F$.

\[
G_1 = L_{(w_1, w_2)}[l(w_1, v) \mid w_2](v_1, v_2) \mid L_{(w_1, w_2)}[w_1 \mid l(w_2, v)](v_1, v_2) \\
G_2 = L_{(w_1, w_2)}[l(w_1, v) \mid l(w_2, v)](v_1, v_2) \mid L_{(w_1, w_2)}[w_1 \mid l(w_2, v)](v_1, v_2)
\]

A free node is labeled with its name, while a bound one is not since its naming is not significant. An edge is depicted as a box with tentacles, the number of which is exactly its arity. We don’t always need to order these tentacles explicitly. Instead, we use a convention in most cases: for an edge with more than one tentacle, we usually order its tentacles clockwise, with the first one drawn as an incoming arrow and others as outgoing arrows. For an edge with only one tentacle, however, it is not significant whether the tentacle is shown as an incoming or outgoing edge. An abstract edge is represented as a dotted arrow with its label $L^k_j$. In Fig. 4.8, for example, the abstract edges in the upper and lower parts of $G_1$ are labeled with different superscripts as $(L^1_1, L^1_2)$ and $(L^2_1, L^2_2)$, respectively. Since there are two $L$-labeled designs, the superscripts are necessary. In fact, without these superscripts, we cannot distinguish between the hypergraphs of $G_1$ and $G_2$. 

![Figure 4.8: Hypergraphs of terms](image)
A hypergraph full of abstract edge labels looks complicated. We can simplify its graphic representation, putting the body of each \( L \)-labeled design into a dotted box labeled by \( L \) and removing all the abstract edge labels. We regard the dotted box as a special “edge” and use the same convention for edges to order these tentacles. For example, \( G_1 \) and \( G_2 \) in Fig. 4.8 are re-depicted in Fig. 4.9. Notice that a free node is shared by different designs, such as \( v \) in \( G_1 \). Besides encapsulation, designs also provide a mechanism of abstraction, enabling us to hide elements that are not significant in the current view. In Fig. 4.9, for example, the design \( Z \) (labeled by \( L \)) is simply depicted as a “double box” (with tentacles) as we are not concerned with the details of its body.

**Morphism.** For a formal definition of graph transformations, we need to study the relations between hypergraphs, which is captured by the notion of morphism.

**Definition 4.3.3** (Morphism). A morphism \( \rho : G_1 \rightarrow G_2 \) is a mapping from one hypergraph \( G_1 \) to another hypergraph \( G_2 \) such that

1. \( \rho( u ) \) has the same type as \( u \), where \( u \) is either a node, an edge or an abstract edge,
2. If \( \rho \) maps an edge or abstract edge \( l( \overrightarrow{u} ) \) to \( l( \overrightarrow{v} ) \), \( \rho \) maps \( \overrightarrow{v} \) to \( \overrightarrow{w} \), and
3. \( \rho \) maps the sequence of interface nodes of \( G_1 \) to those of \( G_2 \).

A morphism \( \rho : G_1 \rightarrow G_2 \) is called fn-preserving if it maps each free node of \( G_1 \) to a free node of \( G_2 \) with the same node name. A fn-preserving morphism is called strongly fn-preserving if it further maps each bound node of \( G_1 \) to a bound node of \( G_2 \).

Two hypergraphs \( G_1 \) and \( G_2 \) are isomorphic, denoted as \( G_1 \equiv \_G_2 \), if there is a morphism between them that is bijective and strongly fn-preserving. As a result, isomorphic hypergraphs have the same set of free node names.

When there is no confusion, we allow the interchange between a graph term and its hypergraph. Therefore, we can define the relation that two terms \( G_1 \) and \( G_2 \) are isomorphic, i.e. their hypergraphs are isomorphic. It is straightforward to verify the isomorphism relations between hierarchical graphs in Fig. 4.10.
### 4.3.1 Graph transformation rules

A graph-based theory of programming often requires the formalization of rules of graph transformations for defining the behavior of a program or the derivation of one program from another. In our graph model, we study graph transformation rules in the Double-Pushout (DPO) approach [39] which is based on the algebraic notion of pushout. Intuitively, a pushout combines a pair of graphs by injecting them into a larger graph with certain common parts.

**Definition 4.3.4** (Pushout). Let $G_0$, $G_1$ and $G_2$ be three graphs with two morphisms $\rho_1 : G_0 \rightarrow G_1$ and $\rho_2 : G_0 \rightarrow G_2$. A pushout of the pair $\langle \rho_1, \rho_2 \rangle$ is a graph $G_3$ together with two morphisms $\rho_{13} : G_1 \rightarrow G_3$ and $\rho_{23} : G_2 \rightarrow G_3$ (see Fig. 4.11), such that:

1. $\rho_{13} \circ \rho_1 = \rho_{23} \circ \rho_2$, and
2. if there exist a graph $G'_3$ and two morphisms $\rho'_{13}$ and $\rho'_{23}$ such that $\rho'_{13} \circ \rho_1 = \rho'_{23} \circ \rho_2$, there is a unique morphism $\rho_3 : G_3 \rightarrow G'_3$ such that $\rho'_{13} = \rho_3 \circ \rho_{13}$ and $\rho'_{23} = \rho_3 \circ \rho_{23}$.

We call $\langle \rho_2, \rho_{23} \rangle$ a pushout complement of $\langle \rho_1, \rho_{13} \rangle$, and vice versa.

This definition shows that the pushout graph $G_3$ is the union of $G_1$ and $G_2$ with the images of $G$ by $\rho_1$ and $\rho_2$ being equalized.

Following the DPO approach, we formalize a graph transformation rule in terms of a pair of morphisms. We simply call such a graph transformation rule a DPO rule. As we will show later, the application of a DPO rule makes use of a pair of pushouts based on its morphisms.

**Definition 4.3.5** (DPO rule). A DPO rule $R : G_L \xleftarrow{\rho_L} G_1 \xrightarrow{\rho_R} G_R$ consists of a pair of morphisms $\rho_L : G_1 \rightarrow G_L$ and $\rho_R : G_1 \rightarrow G_R$, where $\rho_L$ is injective. Graphs $G_L$, $G_1$ and $G_R$ are called the left-hand side, the interface and the right-hand side of the rule, respectively.

In most DPO rules, $\rho_L$ and $\rho_R$ are the identity mappings or they change only a small part of nodes. We thus simply represent a DPO rule by listing the three graphs as $G_L \| G_1 \| G_R$ with additional annotations for nodes that are not mapped identically. Here, "\|" is just used to separate the graphs, it does not represent a graph composition. We show two examples of DPO rules $R_1$ and $R_2$ in Fig. 4.12. In Rule $R_1$, both morphisms are the identity mapping and thus no annotation is needed. For Rule $R_2$, however, we use $v / v' \rightarrow v$ to annotate that $\rho_R$ maps different nodes $v$ and $v'$ in the interface to the same one $v$ in the right-hand side.

Now we show how a DPO rule can be applied to derive one graph from another.

**Definition 4.3.6** (Direct derivation). Let $R : G_L \xleftarrow{\rho_L} G_1 \xrightarrow{\rho_R} G_R$ be a DPO rule. Given a graph $G$ and a morphism $\rho_1 : G_L \rightarrow G$, $G'$ is a direct derivation of $G$ by $R$ (based on $\rho_1$), denoted as $G \Rightarrow_R G'$, if there exist the morphisms in Fig. 4.13 such that

1. both squares are pushouts,
2. \( \rho_L \) is strongly fn-preserving, and
3. \( \rho'_R \) is fn-preserving whose image includes all the free names of \( G' \). This actually implies \( \text{fn}(G') = \text{fn}(G'') \).

In this definition, \( \rho_1 \) is called the match of the derivation as it actually matches graph \( G_L \) with the subgraph \( \rho_1(G_L) \) of \( G \). When \( \rho_1 \) is found, a graph \( G'' \) can be constructed with morphisms \( \rho_2, \rho'_L \) so that the square on the left of Fig. 4.13 is a pushout. The intuition is that \( G'' \) is obtained from \( G \) by removing the elements, i.e. nodes, edges and abstract edges, in \( \rho_1(G_L \setminus \rho_L(G_I)) \) and preserving those in \( \rho_1(G_I) \). Then, a graph \( G' \) can be constructed with morphisms \( \rho_3, \rho'_R \) so that the square on the right of Fig. 4.13 is a pushout. The intuition is that \( G' \) is obtained from \( G'' \) by adding the elements corresponding to \( G_R \setminus \rho_R(G_I) \). The second and third conditions of this definition ensure that \( G' \) and \( G'' \) are unique. An example of direct derivation by Rule \( R_1 \) from Fig. 4.12 is shown in Fig. 4.14.

It is not the case that for any match \( \rho_1 : G_L \rightarrow G \), the pair of pushouts exist as in Fig. 4.13. For them to exist, \( \rho_1 \) must satisfy the following conditions.

- **Identification condition.** The (same) image of two different elements of \( G_L \) by \( \rho_1 \) should be preserved: for elements \( u_1 \neq u_2 \) in \( G_L \), \( \rho_1(u_1) = \rho_1(u_2) \) implies \( u_1, u_2 \in \rho_L(G_I) \).

- **Dangling condition.** If a node of \( G \) is removed, all of its associated edges and abstract edges should also be removed so that no tentacles are left dangling: no edge or abstract edge in \( G \setminus \rho_1(G_L) \) is attached to a node in \( \rho_1(G_L \setminus \rho_L(G_I)) \).
4.4. GRAPH REPRESENTATION OF CASPiS

In this section, we apply our graph model provided in the previous section for the representation of CASPiS processes and their behaviors. We first define a direct representation of each CASPiS process \( P \) as a hierarchical graph \( [P] \). This representation is easy to understand, but it is hard to define the reductions of processes. To overcome this problem, we define a tagged version \( [P]^t \) of \( [P] \), and show that \( [P]^t \) can be derived from the untagged version \( [P] \). Based on the tagged graph representation, we provide a graph transformation system with a few sets of graph transformation rules for characterizing the congruence and reductions of processes.

To represent a process as a hierarchical graph, we consider the following sets as our vocabularies:

\[
\begin{align*}
\text{(Node types)} & \quad \mathcal{O} = \{ \bullet, \triangleright, \circ \} \\
\text{(Edge labels)} & \quad \mathcal{L} = \{ \mathbf{p}, \mathbf{v}, \mathbf{v}, \text{Nil}, \text{Abs}, \text{Con}, \text{Ret}, \text{Sum}, \text{Par}, \text{Def}, \text{Inv}, \text{Ses}, \text{Pip}, \text{Res}, \text{Rep} \} \\
& \quad \cup \{ A, \mathbf{r}, C, \text{PC}, \text{VC}, \text{RC}, \text{AS} \} \cup \mathcal{G} \\
\text{(Design labels)} & \quad \mathcal{D} = \{ \mathbb{P}, \mathbb{F}, \mathbb{V}, \mathbb{D}, \mathbb{I}, \mathbb{S}, \mathbb{R} \} \text{ with } \mathcal{F} = \{ \mathbb{P}, \mathbb{F}, \mathbb{V} \}
\end{align*}
\]

We define three node types \( \bullet, \triangleright \) and \( \circ \), representing the control flow, data and channels of a process, respectively. We introduce a set of edge labels. Some of them represents the operators on processes, such as \( \text{Abs} \) for abstraction, \( \text{Par} \) for parallel composition and \( \text{Ses} \) for session. The others are for auxiliary purposes such as tagging \( A \), copy \( C \) and data assignment \( AS \), which will become clear later on. The design labels represent processes \( \mathbb{P} \), patterns \( \mathbb{F} \), values \( \mathbb{V} \), service definitions \( \mathbb{D} \), invocations \( \mathbb{I} \), sessions \( \mathbb{S} \) and right-hand sides of pipelines \( \mathbb{R} \), respectively. Design edges labeled with \( \mathbb{P} \), \( \mathbb{F} \) and \( \mathbb{V} \) are flat, so the hierarchy of a graph is introduced only by services, sessions and pipelines.

4.4.1 Processes as designs

In order to define the graph representation of processes, we first need to specify how patterns and values are represented.

**Definition 4.4.1** (Graph representation of patterns). The graph representation of a pattern \( F \), denoted as \( [F]_\mathbb{F} \), is a design of label \( \mathbb{F} \) and type \( \triangleright \), defined as follows (depicted in Fig. 4.15).

\[
\begin{align*}
[?x]_\mathbb{F} & \overset{\text{def}}{=} \mathbb{F}_v[\mathbf{v}(v, x)] \\
[f(F_1, \ldots, F_k)]_\mathbb{F} & \overset{\text{def}}{=} \mathbb{F}_v[\mathbf{v}(v_1, \ldots, v_k)](f(v_1, \ldots, v_k)[[F_1]_\mathbb{F}(v_1)] \ldots [[F_k]_\mathbb{F}(v_k)])
\end{align*}
\]

A pattern variable is represented as an edge \( \mathbf{p}v \) of type \( \langle \mathbf{v}, \triangleright \rangle \) attached to the node of this variable; a constructor \( f \) is represented by an edge labelled by \( f \) which is of arity \( \text{AR}(f) = \text{ar}(f) + 1 \) and
CHAPTER 4. GRAPH REPRESENTATION OF SERVICE-ORIENTED SYSTEMS

Figure 4.15: Graph representation of patterns

Figure 4.16: Graph of an example pattern

Figure 4.17: Graph representation of values

type $\triangleright$ for each rank. It is straightforward to verify that $\text{fn}(\llbracket F \rrbracket_F) = \text{bn}(F)$ for each pattern $F$. The graph of an example pattern $f(\triangleright x, g(\triangleright y, \triangleright z))$ is shown in Fig. 4.16.

**Definition 4.4.2 (Graph representation of values).** The graph representation of a value $V$, denoted as $\llbracket V \rrbracket_V$, is a design of label $V$ and type $\triangleright$, defined as follows (depicted in Fig. 4.17).

$$
\llbracket x \rrbracket_V \overset{\text{def}}{=} V_v[v(v, x)]
$$

$$
[\llbracket f(V_1, \ldots, V_k) \rrbracket_V] \overset{\text{def}}{=} V_v[\nu(v_1, \ldots, v_k)](f(v, v_1, \ldots, v_k)][\llbracket V_1 \rrbracket_V(v_1)] \ldots [\llbracket V_k \rrbracket_V(v_k)]
$$

A value variable is represented as an edge $vv$ of type $(\triangleright, \triangleright)$ attached to the node of this variable. A constructed value is represented in the same way as a constructed pattern. We also have $\text{fn}(\llbracket V \rrbracket_V) = \text{fn}(V)$ for each value $V$.

A process is represented as a $\mathcal{P}$-labeled design of type $(\bullet, \circ, \circ, \circ)$. In this design, a $\bullet$ node $p$ is exposed as the start of the control flow and three $\circ$ nodes $i$, $o$ and $t$ are exposed as the input, output and return channels, respectively.

**Definition 4.4.3 (Graph representation of processes).** The graph representation of a process $P$, denoted as $\llbracket P \rrbracket$, is defined by induction on the structure of $P$. The representative cases are depicted
4.4. Graph representation of CASPIS

\[ \text{Nil}(p) \]
\[ (F)P \]
\[ P \mid Q \]
\[ r \triangleright P \]
\[ [P > Q] \]
\[ (\nu m)P \]
\[ [P] \]

Figure 4.18: Graph representation of processes

in Fig. 4.18.

\[ \text{[0]} \equiv P_{(p,i,o,t)}[|o|t]|\text{Nil}(p) \]
\[ \text{[(F)P]} \equiv P_{(p,i,o,t)}[|(\nu(p_1,v)) \triangleright \text{bn}(F))(\text{Abs}(p,v,p_1,i)[(F)[v]][P](p_1,i,o,t)) \]
\[ \text{[(V)P]} \equiv P_{(p,i,o,t)}[(\nu(p_1,v))(\text{Con}(p,v,p_1,o)[V][v])[P](p_1,i,o,t)) \]
\[ \text{[(V)^*P]} \equiv P_{(p,i,o,t)}[(\nu(p_1,v))(\text{Ret}(p,v,p_1,t)[V][v])[P](p_1,i,o,t)) \]
\[ \text{[U + U']} \equiv P_{(p,i,o,t)}[(\nu(p_1,p_2))(\text{Sum}(p,p_1,p_2))[U](p_1,i,o,t))[U'(p_2,i,o,t))] \]
\[ \text{[P|Q]} \equiv P_{(p,i,o,t)}[(\nu(p_1,p_2))(\text{Par}(p,p_1,p_2))[P](p_1,i,o,t)[Q](p_2,i,o,t)) \]
\[ \text{[s,P]} \equiv P_{(p,i,o,t)}[|t|\text{D}(p,t)[|(\nu(p_1,i_1,o_1))(\text{Dec}(p,s,p_1,i_1,o_1))[P](p_1,i_1,o_1,t))]|P,o) \]
\[ \text{[\nu,P]} \equiv P_{(p,i,o,t)}[|t|\text{D}(p,t)[|(\nu(p_1,i_1,o_1))(\text{Inv}(p,s,p_1,i_1,o_1))[P](p_1,i_1,o_1,t))]|P,o) \]
\[ \text{[r \triangleright P]} \equiv P_{(p,i,o,t)}[|t|\text{S}(p,t)[|(\nu(p_1,i_1,o_1))(\text{Sel}(p,r,p_1,i_1,o_1))[P](p_1,i_1,o_1,t))]|P,o) \]
\[ \text{[(\nu m)P]} \equiv P_{(p,i,o,t)}[|(\nu(p_1,n))(\text{Res}(p,n,p_1))[P](p_1,i,o,t)) \]
\[ \text{[P > Q]} \equiv P_{(p,i,o,t)}[|(\nu(p_1,p_2,0_1))(\text{Imp}(p,p_1,p_2,o_1,i,o,t))][P](p_1,i,o,t)[Q](p_2,i,o,t)) \]
\[ != P_{(p,i,o,t)}[|(\nu(p_1,i_1,o_1,t_1))(\text{Rep}(p,p_1,i,o,t))[P](p_1,i_1,o_1,t_1)) \]

The nil process 0 is represented as an edge Nil of type •. An abstraction process (F)P is represented as a graph with an edge Abs of type (•,•,•,•) connected with the graphs of F and P and attached to the input channel of the whole process. Similar to an abstraction, a concretion and a return process is represented, but with a Con and a Ret edge of type (•,•,•,•) associated with the output channel and the return channel, respectively.

In the graph of a parallel composition P|Q or a sum P+Q, the graphs of P and Q are connected by a Par or Sum edge of type (•,•,•,•), and the channels of P and Q are combined. The graph
of a session process $r \triangleright P$ is defined by attaching the graph of $P$ with a session edge $\text{Ses}$ of type $(\bullet, \triangleright, \bullet, \circ, \circ)$. The $\text{Ses}$ edge is also connected with the input and output channels of $P$. This subgraph is then encapsulated by an $S$-labeled design of type $(\bullet, \circ)$ with the return channel of $P$ exposed to the output channel of the session. The graphs of a service definition and a service invocation are defined similarly.

A pipeline $P > Q$ is represented as an edge $\text{Pip}$ of type $(\bullet, \bullet, \bullet, \circ, \circ, \circ, \circ)$ connected with the graphs of $P$, $Q$, the channels of $P$ and the output channel of the whole pipeline, where the graph of the right-hand side $Q$ is encapsulated by a $\Lambda$-labeled design of type $\bullet$. A restriction $(\nu n)P$ and a replication $!P$ are respectively represented as an edge $\text{Res}$ of type $(\bullet, \triangleright, \bullet)$ and $\text{Rep}$ of type $(\bullet, \bullet, \circ, \circ, \circ)$, attached to the graph of $P$. In the latter case, the channels of $P$ are invisible from outside.

It is worth pointing out that, for any process $P$, names of free nodes in the graph representation of $P$ are exactly the free names of $P$, i.e. $\text{fn}([P]) = \text{fn}(P)$.

### 4.4.2 Tagged graph and tagging rules

In the graph term $[P]$ of a process $P$, each control flow node $\bullet$ is actually the start of a sub-processes $Q$ of $P$. In this sense, the $\bullet$ node corresponds to a context $\Lambda[\cdot]$ with $\Lambda[Q] = P$. Recall that in a process reduction only sub-processes occurring in static contexts are allowed to interact with each other. To define reductions on graphs, we need to distinguish active control flow nodes that correspond to static contexts, from inactive ones that correspond to non-static contexts. For this, we tag the former with $A$-labeled unary edges ($A$ means “active”) of type $\bullet$, called tag edges.

**Definition 4.4.4** (Tagged graph of processes). The tagged graph representation of $P$, denoted as $[P]^{\dagger}$, is defined by induction on the structure of $P$. The representative cases are depicted in Fig. 4.19.

\[
\begin{align*}
[0]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[i | o | t | \text{A}(p) | \text{Nil}(p)] \\
[(F)P]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}([\nu \{p_1, v\} \cup \text{bn}(F)](\text{A}(p)|\text{Abs}(p,v,p_1,t))[F](v)\langle [P]^{\dagger}(p_1,i,o,t) \rangle) \\
[(V)P]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}([\nu \{p_1, v\}]\langle \text{A}(p) | \text{Con}(p,v,p_1,o) \rangle [V](v)\langle [P]^{\dagger}(p_1,i,o,t) \rangle) \\
[(V')P]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}([\nu \{p_1, v\}]\langle \text{A}(p) | \text{Ret}(p,v,p_1,t) \rangle [V](v)\langle [P]^{\dagger}(p_1,i,o,t) \rangle) \\
[U + U']^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}([\nu \{p_1, p_2\}]\langle \text{A}(p) | \text{Sum}(p_1,p_2) \rangle [U](p_1,i,o,t)\langle [U']^{\dagger}(p_2,i,o,t) \rangle) \\
[P \triangleright Q]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}([\nu \{p_1, p_2\}]\langle \text{Pip}(p,p_1,p_2) | [P]^{\dagger}(p_1,i,o,t) \rangle [Q]^{\dagger}(p_2,i,o,t) \rangle) \\
[S]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle t | \text{A}(p) | \text{Nil}(p) \rangle (\nu \{p_1, p_2\}) \langle \text{Def}(p,s,p_1,i_1,o_1) | [P]^{\dagger}(p_1,i_1,o_1) \rangle (p,o)] \\
[S']^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle t | \text{A}(p) | \text{Nil}(p) \rangle (\nu \{p_1, p_2\}) \langle \text{Inv}(p,s,p_1,i_1,o_1) | [P]^{\dagger}(p_1,i_1,o_1) \rangle (p,o)] \\
[r \triangleright P]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle \nu \{p_1, i_1, o_1\} \rangle(\text{Ses}(p,r,p_1,i_1,o_1)) \langle [P]^{\dagger}(p_1,i_1,o_1) \rangle (p,o)] \\
[(\nu n)P]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle \nu \{p_1, i_1, o_1\} \rangle(\text{Nil}(p)) \langle [P]^{\dagger}(p_1,i_1,o_1) \rangle (p,o)] \\
[P > Q]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle \nu \{p_1, p_2, o_1\} \rangle(\text{Pip}(p,p_1,p_2,o_1,i,o,t)) \langle [P]^{\dagger}(p_1,i_1,o_1) \rangle (p,o)] \\
[I]^{\dagger} & \stackrel{\text{def}}{=} \mathbb{P}_{(p,i,o,t)}[\langle \nu \{p_1, i_1, o_1, l_1\} \rangle(\text{P}(p_1,i_1,o_1) \langle [P]^{\dagger}(p_1,i_1,o_1,l_1) \rangle (p,o))] \\
\end{align*}
\]

In a tagged graph $[P]^{\dagger}$, each occurrence of abstraction, concretion, return, service definition or invocation in a static context is tagged by an $A$-edge. Notice that in the case of a restriction, $[(\nu n)P]^{\dagger}$ is quite different from its untagged version. In $[(\nu n)P]^{\dagger}$, a new value is generated and it is denoted by an $n$-labeled edge ($n$ means “restricted value”) of type $\triangleright$, thus the original $\text{Res}$-labeled edge of the untagged version is not needed. For each case of a process $P$, names of free nodes in the tagged graph of $P$ are exactly the free names of $P$, i.e. $\text{fn}([P]^{\dagger}) = \text{fn}(P)$.

To obtained a tagged graph $[P]^{\dagger}$ from its untagged version $[P]$, we first add a tag edge to the start of the control flow of $[P]$, which leads to the graph $\mathbb{P}_{(p,i,o,t)}[\text{A}(p)|[P]^{\dagger}(p,i,o,t)]$, and then apply a sequence of graph transformation rules. These rules are called tagging rules, denoted as $\delta_T$, and they are shown in Fig. 4.20.
The tagging starts with a tag $\mathcal{A}$ at the start of the control. The tag moves step by step along the flow of control, until it arrives at a nil process or a dynamic operator. In each step, the tag may
go through a session, through a pipeline into its left-hand side or through a parallel composition into both of its branches. If the tag meets a restriction, the restriction edge \( \text{Res} \) is removed, the associated control flow nodes are combined, and an \( rv \) edge is added to the associated value node.

We show that these rules are sufficient to transform the untagged graph of every process into its tagged version.

**Theorem 4.4.1** (Completeness of tagging rules). \( \mathbb{P}_{(p,i,o,t)}[A(p)][P] \langle p, i, o, t \rangle \Rightarrow_{\delta_r} [P]^{\dagger} \) for any process \( P \).

**Proof.** By induction on the structure of \( P \). In this proof, we use "IH" as a shorthand for "induction hypothesis".

- **Case** \( P = Q \langle Q' \rangle \).
  \[\Rightarrow (\text{Par}-\text{Tag}) \quad \mathbb{P}_{(p,i,o,t)}[A(p)][Q][Q'][\langle p, i, o, t \rangle] \]
  \[\Rightarrow (\text{IH}) \Rightarrow_{\delta_r} [Q][Q'][\langle p, i, o, t \rangle] \equiv_d [Q]^{\dagger} \]

- **Case** \( P = r \triangleright Q \).
  \[\Rightarrow (\text{Rcs}-\text{Tag}) \quad \mathbb{P}_{(p,i,o,t)}[A(p)][r \triangleright Q][\langle p, i, o, t \rangle] \]
  \[\Rightarrow (\text{IH}) \Rightarrow_{\delta_r} [r \triangleright Q][\langle p, i, o, t \rangle] \equiv_d [r \triangleright Q]^{\dagger} \]

- **Case** \( P = (vn)Q \).
  \[\Rightarrow (\text{Rcs}-\text{Tag}) \quad \mathbb{P}_{(p,i,o,t)}[A(p)][(vn)Q][\langle p, i, o, t \rangle] \]
  \[\Rightarrow (\text{IH}) \Rightarrow_{\delta_r} [(vn)Q][\langle p, i, o, t \rangle] \equiv_d [(vn)Q]^{\dagger} \]

- **Case** \( P = Q > Q' \).
  \[\Rightarrow (\text{Par}-\text{Tag}) \quad \mathbb{P}_{(p,i,o,t)}[A(p)][Q > Q'][\langle p, i, o, t \rangle] \]
  \[\Rightarrow (\text{IH}) \Rightarrow_{\delta_r} [Q > Q'][\langle p, i, o, t \rangle] \equiv_d [Q > Q']^{\dagger} \]

- For other cases of process \( P \), \( \mathbb{P}_{(p,i,o,t)}[A(p)][P] \langle p, i, o, t \rangle \equiv_d [P]^{\dagger} \).

\[\Box\]

**4.4.3 Rules for congruence**

We provide a set of graph transformation rules \( \delta_C \) to characterize the congruence relation between CaSPiS processes. The set includes rules for commutativity, associativity and restrictions, as well as rules for making a copy of a (sub-)process. We first introduce the former subset of rules, and leave the copy rules to the next subsection.

The basic rules are for commutativity and associativity of sums and parallel compositions, shown in Fig. 4.21. In the case of commutativity, we simply change the order of tentacles of the \textit{Sum} and \textit{Par} edges; while in the case of associativity, we rearrange the configuration of these edges.
4.4. Graph Representation of CASPIS

In order to represent the congruence relation $U + 0 \equiv_c U$, we provide a few unit rules for sums, shown in Fig. 4.22. The first one of these rules simply removes a nil from a sum process, while the other ones work in the opposite way, to add nils to sums, each corresponding to a specific case of sum process.

We also defined a set of rules for restrictions, shown in Fig. 4.23. These rules include the unit rules for both untagged and tagged forms of restrictions, as well as rules to move a restriction forward, out of another restriction, a parallel composition, a pipeline (from the left-hand side) or a session.

4.4.4 Copy rules

In order to make copies of processes (or sub-processes), we introduce a set of copy rules $\delta_P \subset \delta_C$. In these rules, we use edges of label $C$, which are of type $(\bullet, \bullet, \circ, \circ, \circ)$, to copy the control flow of
CHAPTER 4. GRAPH REPRESENTATION OF SERVICE-ORIENTED SYSTEMS

CHAPTER 4. GRAPH REPRESENTATION OF SERVICE-ORIENTED SYSTEMS

Figure 4.23: Rules for restrictions

Figure 4.24: Rule for replication

processes, and edges of label $PC$, $VC$ and $RC$, which are of type ($\odot$, $\odot$), to copy patterns, values and restrictions, respectively. They are called copy edges.

**First step.** Given a replication process to be copied, we first create a copy edge $C$ and put it in parallel with the original process. This is done by Rule (Rep-Step), depicted in Fig. 4.24.

We require that this rule can only be applied to graphs or tagged graphs of processes, e.g. without any copy edges. That is, we do not consider the interplay among different copy procedures. Alternatively, such a requirement can be specified as a set of negative application conditions (NAC) of the rule. Each NAC takes the form of a graph, e.g. a single copy edge, and a DPO rule with NACs can not be applied to a graph that contains either of them as a subgraph.

It is worth pointing out that a copy edge $C$ can also be generated by a reduction of a pipeline. We will show this by rules for reduction in the next subsection. The same requirement applies to those rules.
Copy of the control flow. We provide a group of rules in Fig. 4.25, 4.26 and 4.27 to copy the control flow and rebuilding the channels of a process step by step. Each of these rules corresponds to a specific process construct, such as nil, abstraction, service definition, pipeline and restriction. After the copy of an abstraction, a $PC$ edge is generated which will further copy the pattern of
In addition, after the copy of a restriction, an RC invocation, a the abstraction. Similarly, after the copy of a concretion, a return, a service definition or a service invocation, a VC edge is generated for subsequent copy of the corresponding value or service name. In addition, after the copy of a restriction, an RC edge is introduced in order to copy the restricted value.

Recall that no session is allowed to occur in the body of a replication or the right-hand side of a pipeline, which is a non-static context. As a result, we don’t need to consider the copy of a session.

Figure 4.26: Control-copy rules (Part II)
4.4. GRAPH REPRESENTATION OF CASPI

Copy of data. Besides the copy of the control flow, we also need to copy the data of a process. For this purpose, we provide a group of rules in Fig. 4.28 that aim at copying patterns and values, using the copy edges $PC$ and $VC$ generated during the copy of the control flow, respectively.

Elimination of copy edges. The copy edges are just auxiliary ones and do not occur in the graph representation of any process. So, we have to eliminate them at the end of a copy procedure, in order to achieve the graph of the target process. Rules for this purpose are provided in Fig. 4.29.
We should be careful in applying these rules. Specifically, there is a priority order among them (and the other copy rules).

- \((\text{VC-Elim-PC}) > (\text{PC-Elim}) \text{ or } (\text{VC-Elim})\)
- \((\text{VC-Elim-RC}) > (\text{RC-Elim}) \text{ or } (\text{VC-Elim})\)
- Any control-copy rule or data-copy rule > \((\text{PC-Elim}) \text{ or } (\text{RC-Elim}) \text{ or } (\text{VC-Elim})\)

Alternatively, such a priority order can be specified as NACs of rules (PC-Elim), (RC-Elim) and (VC-Elim).

In the case that more than one data copy rules are applicable to a graph during the copy procedure, the one with higher (highest) priority should be applied first, otherwise the copy may be incorrect. For example, during the copy of \((?x)(x)\), a PC-edge for copying the pattern \(?x\) and a VC-edge for copying the value \(x\) will both target at the data node representing \(x\) in the original graph. For a correct copy, we combine the source nodes of the two edges by (VC-Elim-PC) and eliminate the two edges by (PC-Elim) and (VC-Elim). However, if we apply either (PC-Elim) or (VC-Elim) before (VC-Elim-PC), the connection of the two edges would be destroyed and in consequence we could never arrive at the desired graph.

### 4.4.5 Rules for reduction

We provide a set of graph transformation rules \(\delta_R\) to characterize the reduction behavior of CaSPiS processes. Each rule is designed for a specific case of reduction.

The first rule is for the synchronization between a pair of service definition and service invocation, provided in Fig. 4.30. The synchronization causes the creation of a new session, whose name is restricted thus inaccessible from other parts of the graph. It is possible that the data node representing the service name become isolated after the synchronization, but it can be eliminated by garbage collection. We will introduce rules for garbage collection later.

We also have a pair of rules for the reduction of a session, shown in Fig. 4.31. Rule (Ses-Sync) is for the interaction between a concretion and an abstraction of a session \(r\). The shared channel node by the edges \(\text{Con}\) and \(\text{Ses}\) makes sure that the concretion belongs to one side of \(r\). Similarly, the abstraction belongs to the other session side. Both of the abstraction and concretion are removed after the communication, with the value of the concretion connected to the pattern of the abstraction through an AS-edge of type \((\circ, \circ)\). Such an edge is used for further data assignment. Notice that the concretion and abstraction originally occur in two sums, respectively.
Their communication makes the other branches of the sums isolated in the graph. These isolated parts will be removed by garbage collection. Rule (Ses-Sync-Ret) is for the interaction between a return and an abstraction in different sides of a session \( r \). It has a similar form to Rule (Ses-Sync),
while the return edge $\text{Ret}$ occurs in the body of another session, which is nested inside $r$. Due to the limit of space, we draw these rules vertically, i.e. from top to bottom.

We have two more rules for the reduction of a pipeline, shown in Fig 4.32. Rule (Pip-Sync) is for the interaction between a concretion and an abstraction of a pipeline. The shared channel node by the edges $\text{Con}$ and $\text{pip}$ makes sure that the concretion belongs to left-hand side of the pipeline, so that it can communicate with the abstraction $\text{Abs}$ on the right-hand side. The concretion is removed after the communication, and a copy edge $C$ is generated which is put in parallel with the whole pipeline and aims at copying the right-hand side. In addition, the value of the original concretion is connected to the pattern of the abstraction through an $\text{AS}$ edge and a $\text{PC}$ edge for further data assignment and pattern copy. Notice that the concretion originally occurs in a sum. After the reduction, the other branch of the sum becomes isolated in the graph and can be removed by garbage collection. Rule (Pip-Sync-Ret) is for the interaction between a return on the left-hand side of a pipeline and an abstraction on the right-hand side. It has a similar form to Rule (Ses-Sync), while the return edge $\text{Ret}$ occurs in the body of an additional session. Due to the limit of space, we draw these rules vertically, i.e. from top to bottom.

It is worth pointing out that each rule for the reduction of a session or a pipeline has vari-

Figure 4.32: Rules for reduction (Part III)
ants. Take Rule (Ses-Sync) for example, which characterizes the communication between a sum of concretions and a sum of abstractions on different sides of a session. It has the following variants:

1. on one session side is a concretion, on the other side is an abstraction,

2. on one session side is a concretion, on the other side is a sum of abstractions, and

3. on one session side is a sum of concretions, on the other side is an abstraction.

Fig. 4.33 shows the first one (on the left) and the second one (on the right). Nevertheless, each variant rule is equivalent to the original rule, since an abstraction or concretion can always be represented in the form of a sum, i.e. \((F)P \equiv_c (F)P + 0\), \((V)P \equiv_c (V)P + 0\).

We require that the each rule for reduction can only be applied to tagged graphs of processes, e.g. without isolated parts or auxiliary edges other than tag edges. This requirement reflects our consideration that after a reduction we would expect to finish all the relevant assignment, garbage collection, as well as necessary copy and tagging procedures, before starting the next reduction. Alternatively, such a requirement can be specified as NACs of these rules.

### 4.4.6 Garbage collection rules

After the application of a reduction rule, certain nodes and edges may become isolated, and they will make no contribution to the further transformations of the whole graph. We provide a set
of graph transformation rules $\delta_G$, called garbage collection rules, to remove these parts from the graph. These rules are shown in Fig. 4.34 and 4.35, covering all the cases of isolated process constructs, data and channels.

Figure 4.34: Garbage collection rules (Part I)
4.4. GRAPH REPRESENTATION OF CASPIS

4.4.7 Data assignment rules

After the application of a reduction rule, we also need to assign values to their corresponding patterns, according to those AS edges produced by the reduction. After the assignment, some of the values may not in their correct form so we have to normalize them. For these purposes, we provide a set of graph transformation rules \(\delta_D\) for data assignment, as well as the subsequent data normalization. They are called data assignment rules, shown in Fig 4.36.

Figure 4.35: Garbage collection rules (Part II)

Figure 4.36: Data assignment rules
In order to avoid unnecessary complexity of graphs, we require that each of these rules can only be applied to graphs without copy edges. That is, we do not consider the case to perform copy and data assignment at the same time. Alternatively, such a requirement can be specified as NACs of these rules.

4.4.8 Soundness and completeness

To sum up, we have provided a graph transformation system, denoted as $\delta_A$, that consists of

1. a set of tagging rules $\delta_T$ (Fig. 4.20),
2. a set of rules for congruence $\delta_C$ (Fig. 4.21, 4.22 and 4.23), including a subset $\delta_P$ of copy rules (Fig. 4.24, 4.25, 4.26, 4.27, 4.28 and 4.29),
3. a set of rules and reduction $\delta_R$ (Fig. 4.30, 4.31 and 4.32),
4. a set of garbage collection rules $\delta_G$ (Fig. 4.34 and 4.35), and
5. a set of data assignment rules $\delta_D$ (Fig. 4.36).

We show that the graph transformation system $\delta_A$ is sound and complete with respect to both congruence and reduction of CaSPiS processes.

The soundness with respect to congruence means that two processes $P$ and $Q$ are congruent if the tagged graph of $P$ can be transformed to that of $Q$, through applications of rules for congruence $\delta_C$ as well as auxiliary tagging rules $\delta_T$.

**Theorem 4.4.2** (Soundness w.r.t. congruence). For two processes $P$ and $Q$, $\llbracket P \rrbracket \Rightarrow^*_{\delta_C \cup \delta_T} \llbracket Q \rrbracket$ implies $P \equiv_c Q$.

The soundness with respect to reduction means that a process $P$ reduces to another process $Q$ if the tagged graph of the $P$ can be transformed to that of $Q$, through applications of rules of reduction $\delta_R$.

**Theorem 4.4.3** (Soundness w.r.t. reduction). For two processes $P$ and $Q$, if $\llbracket P \rrbracket \Rightarrow_{\delta_D} \llbracket Q \rrbracket$ with exactly one application of $\delta_R$, $P \rightarrow Q$.

The completeness with respect to congruence means that the tagged graphs of two congruent processes $P$ and $Q$ can be transformed to a common tagged graph of some process $Q'$, through applications of rules for congruence $\delta_C$ as well as auxiliary tagging rules $\delta_T$ when necessary.

**Theorem 4.4.4** (Completeness w.r.t. congruence). For two processes $P$ and $Q$, $P \equiv_c Q$ implies $\llbracket P \rrbracket \Rightarrow_{\delta_C \cup \delta_T} \llbracket Q' \rrbracket$ and $\llbracket Q \rrbracket \Rightarrow_{\delta_C \cup \delta_T} \llbracket Q' \rrbracket$ for some process $Q'$.

It is worth pointing out that the completeness with respect to congruence does not mean that the tagged graph of a process can always be transformed to that of a congruent process. In fact, such a conjecture is too strong to be valid. For congruent processes $|P|$ and $|P|P$, for example, we are able to “unfold” the graph $|P|$ into $|P|P$ by making a copy of $P$ using the copy rules. However, we can hardly transform from $|P|P$ back to $|P|$ by applications of any set of DPO rules, as the DPO approach does not have a mechanism to check whether two parts of a graph are equivalent, i.e. representing the same process.

For the same reason, the completeness with respect to reduction does not mean that the tagged graph of $P$ can be transformed to that of $Q$ for any reduction $P \rightarrow Q$. Instead, the tagged graph of $P$ can be transformed to that of some process $Q'$ congruent with the reduced process $Q$.

**Theorem 4.4.5** (Completeness w.r.t. reduction). For two processes $P$ and $Q$, $P \rightarrow Q$ implies $\llbracket P \rrbracket \Rightarrow_{\delta_D} \llbracket Q' \rrbracket$ for some process $Q' \equiv_c Q$.

The proof of these theorems is presented in Appendix A.
4.4.9 An example

We use an example to show the application of our graph transformation rules. Consider a service named `time` which is ready to output the current time `T`. This service can be used by a process that invokes the service, receives values it produces and returns them. The composition of the service and the process is specified in CaSPiS as `P_0 = time.(T)|time.(?x)(x)`\(^\dagger\). The synchronization between `time` and `time` creates a session with a fresh name `r`, and `P_0` evolves to `P_1 = (νr)(r\triangleright(T)|r\triangleright(?x)(x))\(^\dagger\)`. Then, the concretion `T` on one session side and the abstraction `(?x)` on the other side can communicate, assigning `x` on the latter side with `T`, and `P_1` evolves to `P_2 = (νr)(r\triangleright0|r\triangleright(T))\(^\dagger\)`.

The same behavior can be simulated by graph transformations shown in Fig. 4.37. The left graph in the first row is `P_0|\langle p, i, o, t \rangle`. It is transformed to `P_1|\langle p, i, o, t \rangle` (the right graph in the second row) through a sequential application of DPO rules (Ser-Sync), (D-GC) and (Ses-Tag). Such a graph can be further transformed to `P_2|\langle p, i, o, t \rangle` (the right graph in the last row) by applying the DPO rules (Ses-Sync), (PV-Assign) and (Ctr-Norm).
4.5 Summary

In this chapter, we propose a graph representation of structured service systems specified in the service-oriented process calculus CaSPiS. Instead of simple-structured graphs, we use hierarchical graphs for the representation which naturally capture the hierarchical nature of service systems. As the basis of the representation, we set up a graph model by exploiting a suitable graph algebra. In particular, we adopt the syntax of the algebra but define a novel semantic model in which the hierarchy of a graph is realized through a mechanism of abstract edges. This mechanism makes it convenient to study the algebraic notions of morphism and pushout for hierarchical graphs, so that the graph model supports the DPO approach well.

Following the DPO approach, we provide a graph-based concurrent semantics of CaSPiS in terms of a graph transformation system that characterizes the behaviors of processes. Specifically, the system consists of a few sets of graph transformation rules, including basic rules for congruence and reduction as well as rules for auxiliary purposes such as replication, data assignment and garbage collection. As the main result of this chapter, we proved that the graph transformation system is sound and complete with respect to the congruence and reductions of processes. Therefore, our graph-based concurrent semantics is indeed consistent with the original reduction semantics of CaSPiS.
Chapter 5

Conclusions

Nowadays, we are facing software systems with a large scale of complexity and most of them are developed under more sophisticated programming paradigms such as oo programming and service-oriented programming rather than the traditional procedure programming. It thus becomes more and more important to understand the behaviors of systems under these new paradigms clearly, so as to avoid poor programming which may lead to bugs or even system breakdowns. In this thesis, we propose a graph-based approach to the modeling of oo and service-oriented systems. We show that the use of graphs and graph transformations provides both an intuitive visualization and a formal characterization which improve people's understanding of the static states and dynamic behaviors of these systems.

In Chapter 2, we provide a graph-based type system and operational semantics for oo programs specified in a general oo language, namely the formal language of the rCOS method [54]. We define class graphs, type context graphs and state graphs that naturally capture the class structures, type contexts and execution states of oo programs, respectively. The type system checks whether a command, and further a program, is well-typed according to its type context graph and class graph. The operational semantics, which is in the classical SOS style, executes well-typed commands and programs step by step, where each step of the execution is simply a transformation from one state graph to another. We proved the type safety of the operational semantics, that is, the operational semantics is consistent with the type system.

A distinct feature of the operational semantics is that it is only based on directed and labeled graphs and a set of simple operations on these graphs, compared with most existing semantic theories of oo programs, e.g. [83, 75, 66], which are based on mathematical tuples and their relations. The graph notations provide a conceptual clarification of a variety of oo features, such as inheritance, type casting, aliasing and dynamic binding of methods. In addition, the operational semantics is location independent and thus more abstract than most oo semantics that explicitly refer to addresses or locations.

In Chapter 3, we provide a calculus of structure refinement which investigates the relation between transformations in class declarations and changes in method definitions of oo programs. Based on the graph notations and operational semantics defined in Chapter 2, we formalize a structure refinement as a transformation from one class graph to another so that the resulting class structure provides at least as many and as good services as the original class structure. The calculus consists of a few groups of refinement rules, characterizing various cases of structure refinement, including expanding a class structure through adding classes and associations, compressing a class structure through combining classes, transforming methods and eliminating polymorphism. We proved that they are sound refinement rules and complete with respect to the notions of structure transformation and normal form. That is, the combination of refinement rules enables us to achieve every structure transformation, and transform every class graph into a normal form. The result of completeness shows that the refinement rules are expressive and powerful in achieving meaningful refinement when applied in combination. On the other hand, each refinement rule is defined in terms of a simple graph transformation so that it is easy to understand.
Besides the theoretical importance, the graph-based operational semantics and refinement calculus of oo programs have successful applications in practice. First, due to its advantage of intuitiveness and location independency, the graph-based operational semantics has been adopted in theorem proving [73, 50] to verify properties of oo programs. Besides the application of its underlying semantics, the graph-based refinement calculus has been applied in the Common Component Modeling Example (CoCoME) [29, 30], an international benchmarking project for comparing and evaluating the practical appliance of existing component-based models and their corresponding specification techniques [90]. The refinement calculus has also been used in the development of a trustable medical systems in telemedicine practice [100], which improves the efficiency and quality of health care. So, there is no need for us to provide another case study of the application of the refinement calculus in this thesis. In addition, as we have shown in Section 3.5, the refinement calculus is expressive enough to characterize fundamental design patterns for oo development such as Expert Pattern and Low Coupling, and further rCOS refinement rules [54] provided and proved based on the denotational semantics of rCOS. This actually gives the justification of the correctness of both the graph-based operational semantics we provide in Chapter 2 and the UTP-based denotational semantics of rCOS. Due to these advantages, the refinement calculus, together with the underlying operational semantics, has been used as the theoretical basis of the rCOS Tool [31, 91] for oo and component-based model-driven development.

In Chapter 4, we provide a graph representation of structured service systems specified in the process calculus CaSPiS [13], a general service-oriented language focused on key features of SOC. For a faithful visualization of the nested structures of service systems, we set of a graph model of hierarchical graphs through exploiting a graph algebra [19, 20]. Specifically, we adopt the syntax of the algebra with primitives such as design and composition that are suitable to specify hierarchical graphs, while we provide a novel semantic model which enables the formalization of graph transformations in the well-studied DPO approach. In addition to representing states of service systems as hierarchical graphs in the graph model, we characterize their behaviors by providing a graph transformation system which consists of a few sets of DPO graph transformation rules. We proved that these graph transformation rules are indeed sound and complete with respect to the congruence and reduction rules of CaSPiS processes.

The advantage of the graph-based approach is gained from the intuitive understanding of graphs, together with the mathematical elegance and large body of theory available on graphs and graph transformations. The use of design in the graph algebra provides a natural mechanism of abstraction, allowing us to hide information that we are not concerned with in different views. The graph model is new compared with the one given in [20] in that hierarchy is realized by proper combinations of abstract edges between nodes of different designs. And this is a key nature that enables us to define graph transformations in the DPO approach. Furthermore, given the fact that the DPO approach comes with a natural concurrent semantics [7], we have provided a concurrent semantics of CaSPiS in terms of the graph transformation system which is proved consistent with the original reduction semantics of CaSPiS. The concurrent semantics is helpful in recording causal dependencies between interactions and detecting possible sources of faults and malfunctions of services.

5.1 Future Work

We have shown that the graph-based refinement calculus, which is established on the graph-based operational semantics, is consistent with the original rCOS refinement calculus, which is established on the UTP-based denotational semantics [54]. Our objective is to provide a unified and solid theoretical framework for the development of oo programs, and the rest of the work is to prove the equivalence of the two underlying semantics. For this, we are going to study and compare the execution states of the two semantics and the main challenge is to find a correspondence relation between their different notations.

As we have shown in Section 2.5, our graph model is expressive and useful in formulating clear assertions about executions of oo programs. Based on the graph model, we are now developing...
5.1. FUTURE WORK

A graph-based OO Hoare Logic with graph assertions as pre and post conditions of specifications. Such a logic is important for static analysis of OO programs. For sequential OO programs, we prefer first-order predication logic as the basis for defining assertions, because it is natural to characterize many interesting properties of these programs such as aliasing and confinement. On the other hand, temporal or modal logic should be a better choice for concurrent or multi-thread programs, and we would also like to explore how much separation logic is useful to formulate assertions for these programs. Besides the choice of logic, there are two challenges in the development of logic axioms. First, we have to deal with the problem of aliasing. Due to the existence of aliasing navigation expressions, the Hoare Logic law of backward substitution

\[ \{ q[e/le] \} le := e \{ q \} \]

for procedure programs is not valid any more for OO programs. For example,

\[ \{ x_2.a = 4 \} x_1.a := 3 \{ x_2.a = 4 \} \]

is false if \( x_2 \) is an alias of \( x_1 \). Another challenge lies in the definition of the law for object creation. Given a command \( C.\text{new}(x) \), which creates an object and assigns it to \( x \), and a postcondition \( q_2 \), it is different to calculate the precondition \( q_1 \) such that

\[ \{ q_1 \} C.\text{new}(x) \{ q_2 \}. \]

This is because after the execution of \( C.\text{new}(x) \), objects originally referred to by \( x \) may not be accessible any more. A possible solution is to define the logic “reversely”, calculating the post-condition from the precondition. Besides the development of the logic theory, it is also important to investigate its tool support for application in automated techniques of verification and analysis of OO programs, involving the automation of OO design patterns supported by the refinement calculus.

Future work also includes the application of our graph-based model of service systems to a more substantial case study, and further exploration of the power of the theories of graphs and graph transformations for analysis of service-oriented systems. In addition, we are going to implement our graph transformation system with existing graph-based tools. Due to the complexity of the underlying mathematical structures of graphs, we have to consider possible optimization in the implementation so as to reduce the computation scale as well as the consumption of computer resources. The information hiding provided by the mechanism of design would be the key to the optimization.

So far, the graph models for characterizing OO systems and service systems are different yet. For clarification and improvement in understanding of OO concepts, we adopt a model of simple-structured graphs with graph transformations in the basic approach, while for a faithful visualization of the nested structure of service systems and further a concurrent semantics, we use a model of hierarchical graphs with graph transformations in the DPO approach. On the one hand, this suggests a general approach to analogous problems. That is, we choose, according to the properties of the problem, a notation of graphs for the modeling of states and an approach of graph transformations for the modeling of behaviors. On the other hand, the development a unified while efficient graph-based method is useful, both theoretically and practically, for the modeling of various systems. The first step towards this can be the exploration of relations between graph models for OO and service systems, and a key observation is that standard techniques for dealing with simple-structured graphs can also be used to deal with hierarchical graphs.
Bibliography


[99] OASIS. Web Services Business Process Execution Language Version 2.0. [http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html](http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html).
Appendix A

Proof of Soundness and Completeness of DPO Rules

In this appendix, we prove Theorems 4.4.2, 4.4.3, 4.4.4 and 4.4.5 of soundness and completeness. For this, we need to reason about all possible “intermediate” graphs during the applications of DPO rules. Our approach is to introduce auxiliary processes into CaSPiS so that each intermediate graph corresponds to an auxiliary process.

A.1 Auxiliary Processes

Let $\mathcal{L}C$ and $\mathcal{L}S$ be two disjoint infinite set, representing labels for copy and labels for value-share, respectively. We extend the syntax of CaSPiS as follows.

| Process $P$ ::= & \ldots & | l : P & | (l : s).P & | \bar{l}: s.P & | (\nu l : n)P & | Copy(l) & | VC(l).P & | \overline{VC(l)}.P & | (\nu RC(l,n))P & | \vdash P & | \text{GB}(P; GI) & | \text{AS}(\vec{F})P |
| Pattern $F$ ::= & \ldots & | l : F & | pv(l : x) & | PC(l) & | pv(PC(l,x)) & | \nu l & | \nu\nu l & | \nu\nu\nu l & | \nu\nu\nu\nu l |
| Value $V$ ::= & \ldots & | l : V & | vv(l : x) & | VC(l) & | vv(VC(l)) & | \nu V & | L : x & | L : vv(V) & | Sh(L) |
| Garbage Item $GI$ ::= & s & | ch & | var(x) & | F & | V & | P & | Set ::= \{GI, \ldots, GI\} |

where $l \in \mathcal{L}C$, $L \in \mathcal{L}S$, $s \in S$, $n \in S \cup R \cup \mathcal{V}$, $ch$ is a channel name and $Set$ is a set of names, which may include names of variables, services, sessions and channels.

A process can be a labeled process or a copy process. In a labeled process, a label can apply to the whole process as in $l : P$, to a service name as in $(l : s).P$ or $\bar{l}: s.P$, or to a restriction as in $(\nu l : n)P$. Their corresponding copy processes are $\text{Copy}(l)$, $\text{VC}(l).P$, $\overline{\text{VC}(l)}.P$ and $(\nu \text{RC}(l,n))P$, respectively. Intuitively, $\text{Copy}(l)$ is a special process that aims at copying the process labeled by $l$ somewhere. Similarly, $\text{VC}(l)$ (or $(\nu \text{RC}(l,n))$) is a special value (or restriction of $n$) that aims at copying the value (or restriction) labeled by $l$. A labeled process can also contain labeled patterns or labeled values. A labeled pattern is of the form $l : F$, where the label $l$ applies to the whole pattern $F$, or $pv(l : x)$, where the label $l$ applies to the pattern variable $x$. Their corresponding copy patterns are $\text{PC}(l)$ and $pv(\text{PC}(l,x))$, respectively. Intuitively, $\text{PC}(l)$ (or $pv(\text{PC}(l,x))$) is a special pattern (or pattern variable of $x$) that aims at copying the pattern labeled by $l$ somewhere. Similarly, a labeled value is of the form $l : V$, where the label $l$ applies to the whole value $V$, or $vv(l : x)$, where the label $l$ applies to the value variable $x$. Their corresponding copy values are $\text{VC}(l)$ and $vv(\text{VC}(l))$, respectively. (*)

In addition to labeled values and copy values, a process is allowed to contain prefixed values, shared values and sharing values. A prefixed value is of the form $vv(V)$. It is similar to $V$ but, as we will show later, its graph contains an extra $vv$-labeled edge. A variable and a prefixed value
can be shared. So, a shared value is of the form \( L : x \) or \( L : vv(V) \). It corresponds to zero or more sharing values of the form \( Sh(L) \), i.e. it can be shared any times. Notice that we do not call \( L : x \) or \( L : vv(V) \) a labeled value, in order to avoid ambiguity.

A process can be of the pre-tagged form \( \dagger P \), representing the state that \( P \) is ready for tagging. A process can also be equipped with an assignment, as \( AS(\vec{V}; \vec{F})P \), where a name \( x \in \text{fn}(P) \) is bound by the patterns \( \vec{F} \) if \( x \in \text{bn}(\vec{F}) \) (thus it can be renamed through an alpha-conversion). It represents the state that we are ready to assign the values of \( \vec{V} \) to variables of \( P \) according to the patterns \( \vec{F} \). Besides, a process may contain garbage items \( GI \), as in \( GB(P; GI) \). A garbage item is either single or composite. A single garbage item can be a service name \( s \), a channel name \( ch \), a variable \( \text{var}(x) \), a pattern \( F \), a value \( V \) or a process \( P \), while a composite garbage item \( \text{Set} :: \{ GI_1, \ldots, GI_k \} \) consists of a set of garbage items \( GI_1, \ldots, GI_k \) and bound by a set of names \( \text{Set} \).

From now on, we use the terminology “process” to denote any process defined by the extended syntax above, and normal processes to denote a process defined by the original CaSPiS syntax (given in Section 4.1). Similarly, we have normal patterns and normal values. In addition, a process, a pattern or a value is called label-free, if it does not contain any label \( l \in \mathcal{LC} \) or \( L \in \mathcal{LS} \).

With the extension of processes, we extend the notions of process operators and thus contexts at the same time. For example, we have new process operators such as \( [\dagger : \cdot] \) and \( AS(V; F)[\cdot] \). We require that each of them is dynamic, so that the notion of static context remains the same. Similar to processes, a context defined in Section 4.1 is called normal, and a context without labels is called label-free. Moreover, a one-hole context is called garbage-free if its hole does not occur inside a garbage item.

**Well-formedness.** In order to represent a legal state of graphs, i.e. either an intermediate graph or a graph of process, a process \( P \) should satisfy the following well-formedness conditions.

1. Basic well-formedness conditions provided originally for normal processes (given in Section 4.1). Among these conditions, we need to slightly modify two of them, with the extension of processes. The first one is that "each session occurs in a static context". We make it more general as: a session is allowed to occur in the scope of \( \dagger [\cdot] \), but not other dynamic operators. The other one is that "the right-hand side of a pipeline is a sum of abstractions." We make it more general as: the right-hand side of a pipeline can be a sum of abstractions or its labeled version or a copy process with one of these labels.

2. Conditions for label and copy.
   - Only normal patterns, values and sub-process of \( P \) can be labeled.
   - The labeled patterns, values and sub-processes of \( P \) have distinct labels, so do copy patterns, values and sub-processes of \( P \).
   - A copy pattern (value or process) of \( P \) is matched by a labeled pattern (value or process) of \( P \), and vice versa, in the way stated in the paragraph (*).

3. Conditions for shared and sharing value.
   - For each shared value \( L : vv(V) \) of \( P \), \( V \) does not contain any sharing (sub-)value.
   - The shared values of \( P \) have distinct labels.
   - A sharing value \( Sh(L) \) of \( P \) is matched by a shared value \( L : x \) or \( L : vv(V) \) of \( P \). However, such a shared value can be matched by zero or more sharing values \( Sh(L) \).

   - Each pattern, value or sub-process of \( P \) that occurs in a garbage item is normal.
   - For each sub-process \( GB(Q; GI) \) of \( P \), \( GI \) is a composite garbage item. This is just a technical assumption, making each single garbage item occur in a composite one. In fact, we will show that each garbage item \( GI \) is equivalent to a composite one \( \emptyset :: \{ GI \} \).
A garbage item GI is called empty if \([GI]_g \equiv_a 0\), e.g. \(\{\} : \emptyset \equiv_a \{\} : \emptyset\).
APPENDIX A. PROOF OF SOUNDNESS AND COMPLETENESS OF DPO RULES

The graph representation of extended processes are defined as follows.

\[
\begin{align*}
[[t : P]] & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[P\{p', i, o, t\}] \\
\text{[Copy(t)]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[C(p, s', i, o, t)] \\
[[t : s).P]] & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[t]D_{(p, 0)}(\nu(p, i, o, t))[[P\{p, i, o, t\}]](\nu(p, i, o, t))] \\
\text{[VC(t).P]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[t]D_{(p, 0)}(\nu(p, i, o, t))[[P\{p, i, o, t\}]](\nu(p, i, o, t))] \\
[[l : s).P]] & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[t]D_{(p, 0)}(\nu(p, i, o, t))[[P\{p, i, o, t\}]](\nu(p, i, o, t))] \\
\text{[VC(l).P]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[t]D_{(p, 0)}(\nu(p, i, o, t))[[P\{p, i, o, t\}]](\nu(p, i, o, t))] \\
[[\nu l : n)P]] & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[\nu(p, n)](\nu(p, n')[[P\{p, i, o, t\}]](\nu(p, n'))] \\
\text{[\nu RC(l, n))P]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[\nu(p, n)](\nu(p, n'[[P\{p, i, o, t\}]](\nu(p, n'))] \\
[[t)] & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[\nu(p, n)](\nu(p, n')[[P\{p, i, o, t\}]](\nu(p, n'))] \\
\text{[GB(P; G1)]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[P\{p, i, o, t\}][G1]_g] \\
\text{[AS(V; \hat{F})P]} & \overset{\text{def}}{=} P_{\{p, i, o, t\}}[[\nu(v_1, ..., v_k, w_1, ..., w_k)]](\nu(v_1, w_1, ..., v_k, w_k))[[V_1 \cup ... \cup V_k]([v_1, ..., v_k, w_1, ..., w_k]](\nu(bn(F_1) \cup ... \cup bn(F_k))(\nu(bn(F_1) \uplus ... \uplus bn(F_k))(\nu(F_1) \uplus ... \uplus \nu(F_k)](\nu(bn(w_k)))]
\end{align*}
\]

where \( \bar{V} = V_1, ..., V_k \) and \( \hat{F} = F_1, ..., F_k \). The representative ones are depicted in Fig. A.2.
For each of these new process constructs $P_0$, we define its tagged graph as:

$$\llbracket P_0 \rrbracket^\ast \overset{\text{def}}{=} P_{(p, i, o, t)}[A(p)][P_0](p, i, o, t).$$

So, Theorem 4.4.1 actually means $\llbracket \dagger P \rrbracket^\ast \Rightarrow \overset{\text{def}}{=} \llbracket P \rrbracket^\ast$, and it is valid for every process $P$, not only normal ones. Besides, it is worth pointing out that the graph representation, both tagged and untagged versions, of $GB(P; GI)$ and $\dagger P$ are isomorphic, if $GI$ is empty. So are the graph representation of $\dagger P$ and $P$, if $P$ is constructed through a dynamic process operator.

**Normal form.** In order to study the relations among intermediate graphs, we need to introduce a notion of congruence between extended processes. For this purpose, we map all the extended processes into normal processes so that we can make use of the congruence relation between normal processes (given in Section 4.1).

To map an auxiliary process $P$ into a normal process, we first eliminate all its labels, according to the following rules.

$$P(l : Q, Copy(l)) \Rightarrow P(Q, Q)$$
$$P((l : s), Q, VC(l), Q') \Rightarrow P(s, Q, s, Q')$$
$$P(l : s, Q, VC(l), Q') \Rightarrow P(s, Q, s, Q')$$
$$P((\nu l : n)Q, (\nu RC(l, n))Q') \Rightarrow P((\nu n)Q, (\nu n)Q')$$
$$P(l : F, PC(l)) \Rightarrow P(F, F)$$
$$P(l : V, VC(l)) \Rightarrow P(V, V)$$
$$P((\nu v(l : x), pv(PC(l, x))) \Rightarrow P(?x, ?x)$$
$$P(l : V, VC(l)) \Rightarrow P(V, V)$$
$$P(L : x, Sh(L), \ldots, Sh(L)) \Rightarrow P(x, x, \ldots, x)$$
$$P(L : vv(V), Sh(L), \ldots, Sh(L)) \Rightarrow P(V, V, \ldots, V)$$ if $V$ contains no shared value

In the last two rules, it is required that $Sh(L), \ldots, Sh(L)$ are all the occurrences of $Sh(L)$ in $P$. Notice that in these rules, we use a new kind of notations. For example, $P(l : Q, Copy(l))$ represents a process $P$ which has two separate sub-processes $l : Q$ and $Copy(l)$. With such a notation, $P(Q_1, Q_2)$ represents the process $P$ by replacing $l : Q$ and $Copy(l)$ in $P$ with $Q_1$ and $Q_2$, respectively. Similarly, a notation like $P(l : V, VC(l))$ denotes a process $P$ which contains two separate values $l : V$ and $VC(l)$ which can be replaced. These notations are flexible in that any element (processes, patterns, values, names or garbage items) can occur in the bracket and the number of elements is not restricted. We will use this kind of notations throughout this section.

For a well-formed process, all its labels, copies, shared values and sharing values can be eliminated through applications of the above rules. However, the order of applying these rules is not significant. For a well-formed process $P$, the result is unique and it is a well-formed and label-free process, called the *label-free form* of $P$ and denoted as $nf(P)$. Especially, if $P$ is label-free itself, $nf(P) = P$.

For a well-formed and label-free process $P$, we can always map it to a well-formed normal process, denoted as $nf(P)$. It is defined inductively as follows.

- $nf(0) \overset{\text{def}}{=} 0$
- $nf(GB(P; GI)) \overset{\text{def}}{=} nf(P)$
- $nf((F)P) \overset{\text{def}}{=} (F)nf(P)$
- $nf((\nu l)P) \overset{\text{def}}{=} (\nu l)^\dagger nf(P)$
- $nf(P(Q) \overset{\text{def}}{=} nf(P) \mid nf(Q)$
- $nf(s.P) \overset{\text{def}}{=} s \cdot nf(P)$
- $nf((\nu n)P) \overset{\text{def}}{=} (\nu n)nf(P)$
- $nf(P(t)) \overset{\text{def}}{=} !nf(P)$
- $nf((A.S(V; F))P) \overset{\text{def}}{=} nf(P\sigma)$
- $nf((V)^\dagger P) \overset{\text{def}}{=} (V)^\dagger nf(P)$
- $nf(U \triangleright P) \overset{\text{def}}{=} U \triangleright nf(P)$
- $nf(U + U') \overset{\text{def}}{=} nf(U) + nf(U')$
- $nf(P \triangleright Q) \overset{\text{def}}{=} nf(P) \triangleright nf(Q)$
- $nf(P) \overset{\text{def}}{=} nf(P)$
where $\sigma = \text{match}(\vec{F}; \vec{V})$, and $\vec{V}$ is the value obtained from $V$ by replacing each $\nu v(V')$ by $V'$. We can infer from the above definition that $\text{nf}(\Lambda[P]) = \Lambda[\text{nf}(P)]$ for any normal context $\Lambda[\cdot]$ and label-free process $P$. In addition, if $P$ is a normal process itself, we have $\text{nf}(P) = P$.

For a well-formed process $P$, we can always eliminate its labels and then map it to a normal process, in the way stated above. Such a normal process, i.e., $\text{nf}(\text{lf}(P))$, is called the normal formal of $P$.

**Nf-congruence.** With the notion of normal form, we can define the congruence relation between processes. Specifically, two processes $P$ and $Q$ are called nf-congruent, if their normal forms are congruent, i.e., $\text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q))$.

It is worth pointing out that different processes $P$ and $Q$ may have the same tagged graph representation, i.e., $\llbracket P \rrbracket \equiv_d \llbracket Q \rrbracket$, in one of the following basic cases or their combinations.

1. $P$ and $Q$ are alpha-convertible, or different only in the naming of labels. For example, $P = !l : P'[\text{Copy}(l)]$ and $Q = !l' : P'[\text{Copy}(l')]$.
2. $P = \Lambda[\Lambda'[(\nu v)P']]$ and $Q = \Lambda[(\nu v)\Lambda'[P']]$, or vice versa, where both $\Lambda[\cdot]$ and $\Lambda'[\cdot]$ are static.
3. $P = \Lambda[\nu P']$ and $Q = \Lambda[P']$, or vice versa, where $P'$ is constructed through a dynamic process operator if $\Lambda[\cdot]$ is static.
4. $P = P(L : x, Sh(L), \ldots, Sh(L))$ and $Q = P(x, x, \ldots, x)$, where $Sh(L), \ldots, Sh(L)$ are all the occurrences of $Sh(L)$ in $P$, or vice versa.
5. $P = P(L : \nu v(V))$ and $Q = P(\nu v(V))$, or vice versa, where no $Sh(L)$ occurs in $P$.
6. $P = \Lambda[\text{GB}(P'; GI)]$ and $Q = \Lambda[\nu P']$, or vice versa, where $GI$ is empty.
7. $P$ and $Q$ are different only in the distribution of garbage items. For example, $P = GB(P_1; GI) | P_2$ and $Q = P_1 | GB(\text{lf}(P_2; GI))$.

In either case, $P$ and $Q$ are nf-congruent. So, for any graph $H$, the process $P$ such that $\llbracket P \rrbracket \equiv_d H$ is unique up to nf-congruence, if it exists.

### A.2 Proof of Theorems 4.4.2 and 4.4.3 of Soundness

We first show that nf-congruence is preserved by any context, so that it is indeed a congruence relation.

**Lemma A.2.1.** $\text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q))$ implies $\text{nf}(\text{lf}(\Lambda[P])) \equiv_c \text{nf}(\text{lf}(\Lambda[Q]))$ for any processes $P$, $Q$ and context $\Lambda[\cdot]$.

**Proof.** If $\Lambda[\cdot]$ is not label-free, there is a label-free context $\Lambda'[\cdot]$ such that $\text{lf}(\Lambda[P]) = \Lambda'[\text{lf}(P)]$ for any process $P$. In this case, we only need to prove the result for $\Lambda'[\cdot]$.

If $\Lambda[\cdot]$ is not garbage-free, the hole $[\cdot]$ occurs inside a garbage item $GI$. According to the fact that $\text{nf}(\text{GB}(P'; GI)) = \text{nf}(P')$, i.e., a garbage item can be simply removed, we always have $\text{nf}(\text{lf}(\Lambda[P])) = \text{nf}(\text{lf}(\Lambda[Q]))$.

In the rest of the proof, we assume $\Lambda[\cdot]$ is label-free and garbage-free, and make induction on its structure. We usually use “IH” as a shorthand for “induction hypothesis”.

1. $\Lambda[\cdot] = [\cdot]$. This case is trivial.
2. $\Lambda[\cdot] = (F)\Lambda'[\cdot]$. 
   - $\text{nf}(\text{lf}(\Lambda[P])) = (F) \text{nf}(\text{lf}(\Lambda'[P]))$ 
   - (IH) $\equiv_c (F) \text{nf}(\text{lf}(\Lambda'[Q])) = \text{nf}(\text{lf}(\Lambda[Q]))$
(3) $\Lambda[\cdot] = (V)\Lambda'[\cdot]$ (Case $\Lambda[\cdot] = (V)^\dagger\Lambda'[\cdot]$ is similar).
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = (V)\text{nf}(\text{lf}(\Lambda'[\cdot]))$
   (IH) $\equiv_c (V)\text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(4) $\Lambda[\cdot] = \Lambda'[\cdot] + U$ (Case $\Lambda[\cdot] = U + \Lambda'[\cdot]$ is similar).
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = \text{nf}(\text{lf}(\Lambda'[\cdot])) + \text{nf}(\text{lf}(U))$
   (IH) $\equiv_c \text{nf}(\text{lf}(\Lambda'[\cdot])) + \text{nf}(\text{lf}(U)) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(5) $\Lambda[\cdot] = \Lambda'[\cdot] | P_1$ (Case $\Lambda[\cdot] = P_1|\Lambda'[\cdot]$ is similar).
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = \text{nf}(\text{lf}(\Lambda'[\cdot])) | \text{nf}(\text{lf}(P_1))$
   (IH) $\equiv_c \text{nf}(\text{lf}(\Lambda'[\cdot])) | \text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(6) $\Lambda[\cdot] = \sigma.\Lambda'[\cdot]$ (Case $\sigma.\Lambda'[\cdot]$ is similar).
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = \sigma.\text{nf}(\text{lf}(\Lambda'[\cdot]))$
   (IH) $\equiv_c \sigma.\text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(7) $\Lambda[\cdot] = r \triangleright \Lambda'[\cdot]$. 
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = r \triangleright \text{nf}(\text{lf}(\Lambda'[\cdot]))$
   (IH) $\equiv_c r \triangleright \text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(8) $\Lambda[\cdot] = \Lambda'[\cdot] > P_1$ (Case $\Lambda[\cdot] = P_1 > \Lambda'[\cdot]$ is similar).
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = \text{nf}(\text{lf}(\Lambda'[\cdot])) > \text{nf}(\text{lf}(P_1))$
   (IH) $\equiv_c \text{nf}(\text{lf}(\Lambda'[\cdot])) > \text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(9) $\Lambda[\cdot] = (\nu n)\Lambda'[\cdot]$. 
   $\text{nf}(\text{lf}(\Lambda[\cdot])) = (\nu n)\text{nf}(\text{lf}(\Lambda'[\cdot]))$
   (IH) $\equiv_c (\nu n)\text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(10) $\Lambda[\cdot] = !\Lambda'[\cdot]$ (Case $\Lambda[\cdot] = !\Lambda'[\cdot]$ is similar).
    $\text{nf}(\text{lf}(\Lambda[\cdot])) = !\text{nf}(\text{lf}(\Lambda'[\cdot]))$
    (IH) $\equiv_c !\text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(11) $\Lambda[\cdot] = GB(\Lambda[\cdot]; GI)$. 
    $\text{nf}(\text{lf}(\Lambda[\cdot])) = \text{nf}(\text{lf}(\Lambda'[\cdot]))$
    (IH) $\equiv_c \text{nf}(\text{lf}(\Lambda'[\cdot])) = \text{nf}(\text{lf}(\Lambda[\cdot]))$

(12) $\Lambda[\cdot] = AS(\bar{V}; \bar{F})\Lambda'[\cdot]$. 
    $\text{nf}(\text{lf}(\Lambda[\cdot])) = \text{nf}(\text{lf}(\Lambda'[\cdot]))\sigma$
    (IH) $\equiv_c \text{nf}(\text{lf}(\Lambda'[\cdot]))\sigma = \text{nf}(\text{lf}(\Lambda[\cdot])) \quad \sigma = \text{match}(\bar{F}; \bar{V})$

With this lemma, we are able to prove the soundness graph transformation rules we have defined. For tagging rules $\delta_T$, copy rules $\delta_P$, rules for congruence $\delta_C$, garbage collection rules $\delta_G$, and data assignment rules $\delta_D$, they are sound in that they always transform the tagged graph of a process to that of an nf-congruent one.

**Theorem A.2.1** (Soundness of tagging rules). For a process $P$, a DPO rule $R \in \delta_T$ and a graph $H$ such that $[P]^\dagger \simeq_R H$, there exists a process $Q$ such that $[Q]^\dagger \equiv_d H$ and $\text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q))$.

**Proof.** We prove for each rule $R \in \delta_T$. 


(1) \( R = (\text{Ses-Tag}). \)

\( P \) must be of the form \( \Lambda[\tau \triangleright P_1] \) for some static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\tau \triangleright P_1] \). In addition, we have \( \text{nf}(\text{lf}(\tau \triangleright P_1)) = \text{nf}(\text{lf}(\tau \triangleright P_1)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(2) \( R = (\text{Pip-Tag}). \)

\( P \) must be of the form \( \Lambda[\{P_1 > P_2\}] \) for some static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\{P_1 > P_2\}] \). In addition, we have \( \text{nf}(\text{lf}(\{P_1 > P_2\})) = \text{nf}(\text{lf}(\{P_1 > P_2\})) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(3) \( R = (\text{Res-Tag}). \)

\( P \) must be of the form \( \Lambda[\{P_1|P_2\}] \) for some static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\{P_1|P_2\}] \). In addition, we have \( \text{nf}(\text{lf}(\{P_1|P_2\})) = \text{nf}(\text{lf}(\{P_1|P_2\})) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(4) \( R = (\text{Par-Tag}). \)

\( P \) must be of the form \( \Lambda[\{\nu\}P_1] \) for some static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\{\nu\}P_1] \). In addition, we have \( \text{nf}(\text{lf}(\{\nu\}P_1)) = \text{nf}(\text{lf}(\{\nu\}P_1)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

Theorem A.2.2 (Soundness of copy rules). For a process \( P \), a DPO rule \( R \in \delta_P \) and a graph \( H \) such that \([P]^\ddagger \Rightarrow_R H \), there exists a process \( Q \) such that \([Q]^\ddagger \equiv_d H \) and \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

Proof. We prove for each rule \( R \in \delta_P \).

(1) \( R = (\text{Rep-Step}). \)

\( P \) must be a normal process of the form \( \Lambda[P_1] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[l : P_1] \). Since \( \text{nf}(\text{lf}(P_1)) = !P_1 \equiv_c !P_1 = \text{nf}(\text{lf}(l : P_1)) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \), according to Lemma A.2.1.

(2) \( R = (\text{Nil-Copy}). \)

\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : 0] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[l : 0] \) and thus \( \text{lf}(Q) = \text{lf}(P) \).

(3) \( R = (\text{Abs-Copy}) \) (Case \( R = (\text{Con-Copy}) \) or \( (\text{Ret-Copy}) \) is similar).

\( P \) must be of the form \( \Lambda[(PC(l'))\text{Copy}(l'), l : F]l : P_1 \) in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[(PC(l'))\text{Copy}(l'), l' : F]l : P_1 \) and thus \( \text{lf}(Q) = \text{lf}(\Lambda[(PC(l')\text{Copy}(l'), l : F)]l : P_1) = \text{lf}(P) \).

(4) \( R = (\text{Par-Copy}) \) (Case \( R = (\text{Sum-Copy}) \) is similar).

\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : (P_1|P_2)] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\text{Copy}(l), l : (P_1|P_2)] \) and thus \( \text{lf}(Q) = \text{lf}(\Lambda[\text{Copy}(l), l : (P_1|P_2)]) = \text{lf}(P) \).

(5) \( R = (\text{Def-Copy}) \) (Case \( R = (\text{Inv-Copy}) \) is similar).

\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : (s.P_1)] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\text{Copy}(l), l : (s.P_1)] \) and thus \( \text{lf}(Q) = \text{lf}(\Lambda[s.P_1, s.P_1]) = \text{lf}(P) \).

(6) \( R = (\text{Pip-Copy}). \)

\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : (P_1 > P_2)] \), in order that \( R \) can be applied to \([P]^\ddagger\). In this case, \( H \equiv_d \llbracket Q \rrbracket^\ddagger \), where \( Q = \Lambda[\text{Copy}(l) > \text{Copy}(l'), l : P_1 > P_2] \) and thus \( \text{lf}(Q) = \text{lf}(\Lambda[P_1 > P_2, P_1 > P_2]) = \text{lf}(P) \).
(7) \( R = (\text{Res-Copy}) \).
\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : (\nu n)P_1] \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[\nu RC(l', n)\text{Copy}(l), (\nu l' : n)l : P_1] \) and thus \( \text{if}(Q) = \text{if}(\Lambda[(\nu n)P_1, (\nu n)P_1]) = \text{if}(P) \).

(8) \( R = (\text{Rep-Copy}) \).
\( P \) must be of the form \( \Lambda[\text{Copy}(l), l : \neg P_1] \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[\text{Copy}(l), \neg l : P_1] \) and thus \( \text{if}(Q) = \text{if}(\Lambda[\neg P_1, \neg P_1]) = \text{if}(P) \).

(9) \( R = (\text{PV-PCopy}) \).
\( P \) must be of the form \( P(PCA(l), l : ?x) \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = P(\nu(PCA(l), \nu(l : x))) \) and thus \( \text{if}(Q) = \text{if}(P(?x, ?x)) = \text{if}(P) \).

(10) \( R = (\text{VV-VCopy}) \).
\( P \) must be of the form \( P(VC(l), l : x) \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = P(\nu(VC(l), \nu(l : x))) \) and thus \( \text{if}(Q) = \text{if}(P(x, x)) = \text{if}(P) \).

(11) \( R = (\text{Ctr-PCopy}) \) (Case \( R = (\text{Ctr-VCopy}) \) is similar).
\( P \) must be of the form \( P(PCA(l), l : f(F_1, \ldots, F_k)) \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = P(f(PCA(l)), f(l : F_1, \ldots, l : F_k)) \) and thus \( \text{if}(Q) = \text{if}(P(\nu RC(l, x))) = \text{if}(P) \).

(12) \( R = (\text{VC-Elim}) \).
\( P \) must be of the form \( \Lambda[(F_1(\nu(PCA(l, x))), P_1)(\nu(VC(l'))), (F_2(\nu(l : x)))P_2(\nu(l' : x))] \), in order that \( R \) can be applied to \([P]^\dagger\). Without loss of generality, suppose \( x \) is never bound in \( P_1 \) or \( P_2 \). In fact, this can always be achieved by alpha-conversions. In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[(F_1(\nu RC(l, x)))P_1(x), (F_2(\nu(l : x)))P_2(x)] \) and thus \( \text{if}(Q) = \text{if}(P(\nu RC(l, x))) = \text{if}(P) \).

(13) \( R = (\text{PC-Elim}) \).
\( P \) must be of the form \( \Lambda[(F_1(\nu(PCA(l, x))), P_1)(\nu(VC(l'))), (F_2(\nu(l : x)))P_2(\nu(l' : x))] \), in order that \( R \) can be applied to \([P]^\dagger\). If \( P_2 \) does not contain any variable \( \nu(l' : x) \), or \( l' : V \) with \( x \in \text{fn}(V) \), or any sub-process \( l' : P' \) with \( x \in \text{fn}(P') \). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[(F_1(\nu l : x))P_1(x), (F_2(\nu l : x))P_2(x)] \) and thus \( \text{if}(Q) = \text{if}(P) \).

(14) \( R = (\text{VC-Elim-RC}) \).
\( P \) must be of the form \( \Lambda[(\nu RC(l, x), P_1)(\nu(VC(l'))), (\nu l : x)P_2(\nu(l' : x))] \) for some variable \( x \), or \( \Lambda[(\nu RC(l, s))P_1(VC(l'))), (\nu l : s)P_2(l' : s)] \) for some service \( s \), in order that \( R \) can be applied to \([P]^\dagger\). In the former case, suppose \( x \) is never bound in \( P_1 \) or \( P_2 \), which can always be achieved by alpha-conversions. In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[(\nu RC(l, x))P_1(x), (\nu l : x)P_2(x)] \) and thus \( \text{if}(Q) = \text{if}(P) \). In the latter case, suppose \( s \) is never bound in \( P_1 \) or \( P_2 \), which can always be achieved by alpha-conversions. In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[(\nu RC(l, s))P_1(s), (\nu l : s)P_2(s)] \) and we also have \( \text{if}(Q) = \text{if}(P) \).

(15) \( R = (\text{RC-Elim}) \).
\( P \) must be of the form \( \Lambda[(\nu RC(l, n))P_1, (\nu l : n)P_2] \), in order that \( R \) can be applied to \([P]^\dagger\), where \( P_2 \) does not contain any variable \( \nu(l' : n) \), or \( l' : P' \) with \( n \in \text{fn}(V) \), or any sub-process \( l' : P' \) with \( n \in \text{fn}(P') \). In this case, \( H \equiv_a [Q]^\dagger \), where \( Q = \Lambda[(\nu n)P_1, (\nu n)P_2] \) and thus \( \text{if}(Q) = \text{if}(P) \).
(16) \( R = (\text{VC-Elim}) \). 
\( P \) must be of the form \( \Lambda[P(lv(VC(l)), lv(l : x))] \) or \( \Lambda[P_1(VC(l)), l : s] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \), where \( x \) or \( s \) is never bound in \( P_1 \). In the former case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[P_1(x, x)] \) and thus \( \text{lf}(Q) = \text{lf}(P) \). In the latter case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[P_1(s, s)] \) and we also have \( \text{lf}(Q) = \text{lf}(P) \). \( \square \)

**Theorem A.2.3** (Soundness of rules for congruence). For a process \( P \), a DPO rule \( R \in \delta_C \) and a graph \( H \) such that \( \left[ P \right]^\dagger \Rightarrow_R H \), there exists a process \( Q \) such that \( \left[ Q \right]^\dagger \equiv_d H \) and \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

**Proof.** According to Theorem A.2.2, the set of copy rules \( \delta_P \) are sound. So, we only need to prove the soundness of each rule \( R \in \delta_C \setminus \delta_P \).

(1) \( R = (\text{Par-Comm}) \) (Case \( R = (\text{Sum-Comm}) \) is similar).
\( P \) must be of the form \( \Lambda[P_1[P_2]] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[P_2[P_1]] \). Since \( \text{nf}(\text{lf}(P_1[P_2])) \equiv_c \text{nf}(\text{lf}(P_2[P_1])) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \) according to Lemma A.2.1.

(2) \( R = (\text{Par-Assoc}) \) (Case \( R = (\text{Sum-Assoc}) \) is similar).
\( P \) must be of the form \( \Lambda[(P_1P_2)P_3] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[P_1[P_2P_3]] \). Since \( \text{nf}(\text{lf}(P_1[P_2P_3])) \equiv_c \text{nf}(\text{lf}(P_1P_2P_3)) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \) according to Lemma A.2.1.

(3) \( R = (\text{Sum-Unit}) \).
\( P \) must be of the form \( \Lambda[U + 0] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[U] \). In addition, \( \text{nf}(\text{lf}(U + 0)) = \text{nf}(\text{lf}(U)) + 0 \equiv_c \text{nf}(\text{lf}(U)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(4) \( R = (\text{Nil-toSum}) \). 
\( P \) must be of the form \( \Lambda[0] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[0 + 0] \). In addition, \( \text{nf}(\text{lf}(0 + 0)) = 0 + 0 \equiv_c 0 = \text{nf}(\text{lf}(0)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(5) \( R = (\text{Abs-toSum}) \) (Case \( R = (\text{Con-toSum}) \) or \( (\text{Ret-toSum}) \) is similar).
\( P \) must be of the form \( \Lambda[(F)P_1] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[(F)P_1 + 0] \). In addition, \( \text{nf}(\text{lf}((F)P_1)) \equiv_c \text{nf}(\text{lf}((F)P_1 + 0)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(6) \( R = (\text{Res-Unit}) \) (Case \( R = (\text{Res-Unit-A}) \) is similar).
\( P \) must be of the form \( \Lambda[(vn)0] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[0] \). In addition, \( \text{nf}(\text{lf}((vn)0)) = (vn)0 \equiv_c 0 = \text{nf}(\text{lf}(0)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(7) \( R = (\text{Nil-toRes}) \) (Case \( R = (\text{Nil-toRes-A}) \) is similar).
\( P \) must be of the form \( \Lambda[0] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[(vn)0] \). In addition, \( \text{nf}(\text{lf}(0)) = 0 \equiv_c (vn)0 = \text{nf}(\text{lf}((vn)0)) \). According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).

(8) \( R = (\text{Res-Comm}) \).
\( P \) must be of the form \( \Lambda[(vn)(vn')P_1] \) for some non-static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \( \left[ P \right]^\dagger \). In this case, \( H \equiv_d \left[ Q \right]^\dagger \), where \( Q = \Lambda[(vn')(vn)P_1] \). In addition, 
\[ \begin{align*}
    \text{nf}(\text{lf}((vn)(vn')P_1)) &= (vn)(vn') \text{nf}(\text{lf}(P_1)) \\
    &\equiv_c (vn')(vn) \text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}((vn')(vn)P_1)).
\end{align*} \]
According to Lemma A.2.1, \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).
(9) \( R = (\text{Par-Res-Comm}) \).
\( P \) must be of the form \( \Lambda[P_i[(vn)P_j] \) for some non-static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \( [P]^\dagger \). Without loss of generality, suppose \( n \notin \text{fn}(P_i) \), which can always be achieved by alpha-conversions. In this case, \( H \equiv_d \llbracket Q \rrbracket^\dagger \), where \( Q = \Lambda[(vn)(P_1|P_2)] \). Since \( \text{nf}(\llbracket (vn)(P_1|P_2) \rrbracket) = \text{nf}(\llbracket (vn)(P_1|P_2) \rrbracket) = \text{nf}(\llbracket (vn)(P_1|P_2) \rrbracket) \), \( \text{nf}(\llbracket (vn)(P_1|P_2) \rrbracket) = \text{nf}(\llbracket (vn)(P_1|P_2) \rrbracket) \) according to Lemma A.2.1.

(10) \( R = (\text{Pip-Res-Comm}) \).
\( P \) must be of the form \( \Lambda[(vn)P_i > P_j] \) for some non-static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \( [P]^\dagger \). Without loss of generality, suppose \( n \notin \text{fn}(P_j) \), which can always be achieved by alpha-conversions. In this case, \( H \equiv_d \llbracket Q \rrbracket^\dagger \), where \( Q = \Lambda[(vn)(P_i > P_j)] \). Since \( \text{nf}(\llbracket (vn)(P_i > P_j) \rrbracket) = \text{nf}(\llbracket (vn)(P_i > P_j) \rrbracket) \) according to Lemma A.2.1.

(11) \( R = (\text{Ses-Res-Comm}) \).
\( P \) must be of the form \( \Lambda[r > (vn)P_i] \) for some non-static context \( \Lambda[\cdot] \), in order that \( R \) can be applied to \( [P]^\dagger \). Without loss of generality, suppose \( n \neq r \), which can always be achieved by alpha-conversions. In this case, \( H \equiv_d \llbracket Q \rrbracket^\dagger \), where \( Q = \Lambda[(vn)(r > P_i)] \). Since \( \text{nf}(\llbracket (vn)(r > P_i) \rrbracket) = \text{nf}(\llbracket (vn)(r > P_i) \rrbracket) \) according to Lemma A.2.1.

**Theorem A.2.4** (Soundness of garbage collection rules). For a process \( P \), a DPO rule \( R \in \delta_G \) and a graph \( H \) such that \( [P]^\dagger \Rightarrow_R H \), there exists a process \( P \) such that \( [Q]^\dagger \equiv_d H \) and \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket Q \rrbracket) \).

**Proof.** We prove for each rule \( R \in \delta_G \).

(1) \( R = (\text{Nil-GC}) \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI)] \), where \( GI \) has a sub-item \( Set \equiv \langle 0, GI_1, \ldots, GI_k \rangle \), in order that \( R \) can be applied to \( [P]^\dagger \). Without loss of generality, suppose \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket P \rrbracket) \) according to Lemma A.2.1, \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket Q \rrbracket) \).

(2) \( R = (\text{Abs-GC}) \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI((F)P_2))] \), in order that \( R \) can be applied to \( [P]^\dagger \). In this case, \( H \equiv_d \llbracket Q \rrbracket^\dagger \), where \( Q = \Lambda[GB(P_1; GI((F)P_2))] \). In addition, \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket Q \rrbracket) \).

(3) \( R = (\text{Con-GC}) \) (Case \( R = (\text{Ret-GC}) \) is similar).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(\langle\{V\}P_2, GI_1, \ldots, GI_k\rangle))] \), in order that \( R \) can be applied to \( [P]^\dagger \). We choose \( Q = \Lambda[GB(P_1; GI(\langle\{V\}P_2, GI_1, \ldots, GI_k\rangle))] \), so that \( [Q]^\dagger \equiv_d H \). In addition, \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket Q \rrbracket) \).

(4) \( R = (\text{Par-GC}) \) (Case \( R = (\text{Sum-GC}) \) is similar).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(\langle\{P\}P_2, GI_1, \ldots, GI_k\rangle))] \), in order that \( R \) can be applied to \( [P]^\dagger \). We choose \( Q = \Lambda[GB(P_1; GI(\langle\{P\}P_2, GI_1, \ldots, GI_k\rangle))] \), so that \( [Q]^\dagger \equiv_d H \). In addition, \( \text{nf}(\llbracket P \rrbracket) = \text{nf}(\llbracket Q \rrbracket) \).
(5) \( R = \text{(Def-GC)} \) (Case \( R = \text{(Inv-GC)} \) is similar).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(Set :: \{ s, P_2, GI_1, \ldots, GI_k \})]] \), in order that \( R \) can be applied to \([P]^\dagger\). We choose \( Q = \Lambda[GB(P_1; GI(Set :: \{ s, i, o, t, \{ i, o, t \} :: \{ P_2 \}, GI_1, \ldots, GI_k \})] \), so that \([Q]^\dagger \equiv_d H \). In addition,
\[
\text{nf}(\text{if}(GB(P_1; GI(Set :: \{ s, P_2, GI_1, \ldots, GI_k \})))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(Set :: \{ s, i, o, t, \{ i, o, t \} :: \{ P_2 \}, GI_1, \ldots, GI_k \}))).
\]
According to Lemma A.2.1, \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \).

(6) \( R = \text{(Pip-GC)} \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(Set :: \{ \{ P_2 > P_3, GI_1, \ldots, GI_k \} \})] \), in order that \( R \) can be applied to \([P]^\dagger\). We choose \( Q = \Lambda[GB(P_1; GI(Set :: \{ \{ P_2 \}, \{ n \} :: \{ P_3 \}, GI_1, \ldots, GI_k \})] \), so that \([Q]^\dagger \equiv_d H \). In addition,
\[
\text{nf}(\text{if}(GB(P_1; GI(Set :: \{ \{ P_2 > P_3, GI_1, \ldots, GI_k \} \}))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(Set :: \{ \{ P_2 \}, \{ n \} :: \{ P_3 \}, GI_1, \ldots, GI_k \}))).
\]
According to Lemma A.2.1, \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \).

(7) \( R = \text{(Res-GC)} \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI((\nu n)P_2))] \), in order that \( R \) can be applied to \([P]^\dagger\). If \( n \) is a variable name, we choose \( Q = \Lambda[GB(P_1; GI(\{ n \} :: \{ \text{var}(n), P_2 \})]] \), so that \([Q]^\dagger \equiv_d H \) and
\[
\text{nf}(\text{if}(GB(P_1; GI((\nu n)P_2)))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(\{ n \} :: \{ \text{var}(n), P_2 \}))).
\]
Otherwise, \( n \) is a service name. We choose \( Q = \Lambda[GB(P_1; GI(\{ n \} :: \{ n, P_2 \})]] \), so that \([Q]^\dagger \equiv_d H \) and
\[
\text{nf}(\text{if}(GB(P_1; GI((\nu n)P_2)))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(\{ n \} :: \{ n, P_2 \}))).
\]
In either case, we have \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \), according to Lemma A.2.1.

(8) \( R = \text{(Rep-GC)} \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(Set :: \{ !P_2, GI_1, \ldots, GI_k \})]] \), in order that \( R \) can be applied to \([P]^\dagger\). We choose \( Q = \Lambda[GB(P_1; GI(Set :: \{ i, o, t, \{ i, o, t \} :: \{ P_2 \}, GI_1, \ldots, GI_k \})] \), so that \([Q]^\dagger \equiv_d H \). In addition,
\[
\text{nf}(\text{if}(GB(P_1; GI(Set :: \{ !P_2, GI_1, \ldots, GI_k \})))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(Set :: \{ i, o, t, \{ i, o, t \} :: \{ P_2 \}, GI_1, \ldots, GI_k \}))).
\]
According to Lemma A.2.1, \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \).

(9) \( R = \text{(Ctr-GC)} \).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(Set :: \{ \{ FV_1, \ldots, FV_j \}, GI_1, \ldots, GI_k \})]] \) in order that \( R \) can be applied to \([P]^\dagger\), where \( FV_1, \ldots, FV_j \) is either a sequence of patterns or a sequence of values. We choose \( Q = \Lambda[GB(P_1; GI(Set :: \{ \{ FV_1, \ldots, FV_j \}, GI_1, \ldots, GI_k \})] \), so that \([Q]^\dagger \equiv_d H \). In addition,
\[
\text{nf}(\text{if}(GB(P_1; GI(Set :: \{ \{ FV_1, \ldots, FV_j \}, GI_1, \ldots, GI_k \})))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(Set :: \{ \{ FV_1, \ldots, FV_j \}, GI_1, \ldots, GI_k \}))).
\]
According to Lemma A.2.1, \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \).

(10) \( R = \text{(PV-GC)} \) (Case \( R = \text{(VV-GC)} \) is similar).
\( P \) must be of the form \( \Lambda[GB(P_1; GI(\{ ?x \})]] \), in order that \( R \) can be applied to \([P]^\dagger\). We choose \( Q = \Lambda[GB(P_1; GI(\text{var}(x)))] \), so that \([Q]^\dagger \equiv_d H \). In addition,
\[
\text{nf}(\text{if}(GB(P_1; GI(\{ ?x \})))) = \text{nf}(\text{if}(P_1)) = \text{nf}(\text{if}(GB(P_1; GI(\text{var}(x))))).
\]
According to Lemma A.2.1, \( \text{nf}(\text{if}(P)) \equiv_c \text{nf}(\text{if}(Q)) \).

(11) \( R = \text{(D-GC)} \).
\( P \) must be of the form \( \Lambda[GB(P_2; GI(Set :: \{ \text{var}(x), GI_1, \ldots, GI_k \})]] \) or \( \Lambda[GB(P_1; GI(Set :: \{ s, GI_1, \ldots, GI_k \})]] \), in order that \( R \) can be applied to \([P]^\dagger\). In either case, we choose \( Q = \)}
A.2. PROOF OF SOUNDNESS

131

A\[ GB(P_1; GI(Set :: \{ GI_1, \ldots, GI_k \})) \], so that \([Q]^\dagger \equiv_d H\). In addition, 

\[ \text{nf\{lf(GB(P_1; GI(Set :: \{ \text{var}(x), GI_1, \ldots, GI_k \}))\}} \]

\[ = \text{nf\{lf(GB(P_1; GI(Set :: \{ s, GI_1, \ldots, GI_k \}))\}} \]

\[ = \text{nf\{lf(P_1)\}} = \text{nf\{lf(GB(P_1; GI(Set :: \{ GI_1, \ldots, GI_k \}))\}} \).

According to Lemma A.2.1, \(\text{nf\{lf(P)\}} \equiv_c \text{nf\{lf(Q)\}}\).

\[ (12) \quad R = (\text{Ch-GC}) \]

\(P\) must be of the form \([\Lambda\{GB(P_1; GI(Set :: \{ ch, GI_1, \ldots, GI_k \}))\} \], in order that \(R\) can be applied to \([P]^\dagger\). We choose \(Q = \Lambda\{GB(P_1; GI(Set :: \{ GI_1, \ldots, GI_k \}))\} \), so that \([Q]^\dagger \equiv_d H\). In addition, 

\[ \text{nf\{lf(GB(P_1; GI(Set :: \{ \text{ch}, GI_1, \ldots, GI_k \}))\}} \]

\[ = \text{nf\{lf(P_1)\}} = \text{nf\{lf(GB(P_1; GI(Set :: \{ GI_1, \ldots, GI_k \}))\}} \).

According to Lemma A.2.1, \(\text{nf\{lf(P)\}} \equiv_c \text{nf\{lf(Q)\}}\). □

**Theorem A.2.5** (Soundness of data assignment rules). For a process \(P\), a DPO rule \(R \in \delta_D\) and a graph \(H\) such that \([P]^\dagger \Rightarrow_R H\), there exists a process \(Q\) such that \([Q]^\dagger \equiv_d H\) and \(\text{nf\{lf(P)\}} \equiv_c \text{nf\{lf(Q)\}}\).

**Proof.** We prove for each rule \(R \in \delta_D\).

(1) \(R = (\text{PV-Assign})\).

\(P\) must be of the form \([\Lambda\{AS(V_1, \ldots, V_j, F_1, \ldots, F_j, ?x)P_1\} \] for some context \(\Lambda\{\cdot\}\) and \(j \geq 0\), in order that \(R\) can be applied to \([P]^\dagger\). Assume the bound names of \(F_1, \ldots, F_j\) do not occur in \(V\).

This can always be achieved by alpha-conversions.

If \(x \not\in \text{fn}(P_1)\), there are two cases.

1. If \(j = 0, H \equiv_d [Q]^\dagger\), where \(Q = \Lambda\{GB(P_1; \emptyset :: \{ V \})\} \). In this case, we have:

\[ \text{nf\{lf(AS(V; ?x)P_1)\}} = \text{nf\{lf(P_1 [V/x] P_1)\}} = \text{nf\{lf(GB(P_1; \emptyset :: \{ V \}))\}} \).

2. Otherwise, \(H \equiv_d [Q]^\dagger\), where \(Q = \Lambda\{GB(AS(V_1, \ldots, V_j; F_1, \ldots, F_j)P_1; \emptyset :: \{ V \})\} \). In this case, we have:

\[ \text{nf\{lf(P_1)\}} = \text{nf\{lf(GB(AS(V_1, \ldots, V_j; F_1, \ldots, F_j)P_1; \emptyset :: \{ V \}))\}} \]

\[ = \text{nf\{lf(P_1)\}} = \text{nf\{lf(GB(AS(V_1, \ldots, V_j; F_1, \ldots, F_j)P_1; \emptyset :: \{ V \}))\}} \]

\[ \text{nf\{lf(P_1)\}} \equiv_c \text{nf\{lf(Q)\}}\). In either case, \(\text{nf\{lf(P)\}} = \text{nf\{lf(Q)\}}\), according to Lemma A.2.1.

If \(x \in \text{fn}(P_1)\), we can write \(P_1\) in the form \(P_1(x, x, \ldots, x)\), where \(x, x, \ldots, x\) are all its free occurrences of \(x\). There are also two case.

1. If \(j = 0, H \equiv_d [Q]^\dagger\), where \(Q = \Lambda\{P_1(L : vv(V), Sh(L), \ldots, Sh(L))\} \). In this case, 

\[ \text{nf\{lf(AS(V; ?x)P_1)\}} = \text{nf\{lf(P_1 [V/x] P_1)\}} \]

\[ = \text{nf\{lf(P_1 [V/V, V])\}} = \text{nf\{lf(P_1 (L : vv(V), Sh(L), \ldots, Sh(L)))\}} \).

2. Otherwise, \(Q = \Lambda\{AS(V_1, \ldots, V_j; F_1, \ldots, F_j)P_1 (L : vv(V), Sh(L), \ldots, Sh(L))\} \), so that \([Q]^\dagger \equiv_d H\). In this case, 

\[ \text{nf\{lf(AS(V_1, \ldots, V_j; F_1, \ldots, F_j, ?x)P_1)\}} = \text{nf\{lf(P_1 [V/x] P_1)\}} \]

\[ = \text{nf\{lf(P_1 [V/V, V])\}} \]

\[ = \text{nf\{lf(AS(V_1, \ldots, V_j; F_1, \ldots, F_j)P_1 (L : vv(V), Sh(L), \ldots, Sh(L)))\}} \]

\[ \text{nf\{lf(P_1)\}} \equiv_c \text{nf\{lf(Q)\}}\). In either case, \(\text{nf\{lf(P)\}} = \text{nf\{lf(Q)\}}\), according to Lemma A.2.1.

(2) \(R = (\text{Ctr-Assign})\).

\(P\) must be of the form \([\Lambda\{AS(V_1, \ldots, V_j, f(V_1', \ldots, V_k') ; F_1, \ldots, F_j; f(F_1', \ldots, F_k')P_1\} \] where \(j \geq 0\) and \(k \geq 0\), in order that \(R\) can be applied to \([P]^\dagger\). There are two cases.

1. If \(j = k = 0, H \equiv_d [Q]^\dagger\), where \(Q = \Lambda\{P_1\} \). In this case, we have:

\[ \text{nf\{lf(AS(f; f)P_1)\}} = \text{nf\{lf(P_1)\}} \).


2. Otherwise, $H \equiv_d [Q]^\dagger$, where $Q = \Lambda[AS(V_1, \ldots, V_j, V'_1, \ldots, V'_k; F_1, \ldots, F_j, F'_1, \ldots, F'_k)P_1]$. In this case, we have:
\[
\begin{align*}
\text{nf}(\text{lf}(AS(V_1, \ldots, V_j, f(V'_1, \ldots, V'_k); F_1, \ldots, f(F'_1, \ldots, F'_k)P_1)) &= \text{nf}(\text{lf}(P_1\sigma)) \\
\text{where } \sigma = \text{match}(F_1, \ldots, F'_1, F'_k; V_1, \ldots, V_j, V'_1, \ldots, V'_k).
\end{align*}
\]
In either case, $\text{nf}(\text{lf}(P)) = \text{nf}(\text{lf}(Q))$, according to Lemma A.2.1.

(3) $R = (VV\text{-Norm}).$  
$P$ must contain a value $\nu r(V)$, i.e. $P$ is of the form $P(\nu r(V))$, in order that $P$ can be applied to $[P]^\dagger$. According to $R$, $H \equiv_d [Q]^\dagger$, where $Q = P(\nu r(V))$ and thus $\text{nf}(\text{lf}(P)) = \text{nf}(\text{lf}(Q))$.

There is only one exception that $V$ is a shared value $L : V_0$ and the value $\nu r(L : V_0)$ is also shared as $L' : \nu r(L : V_0)$, i.e. $P$ is of the form $P(L' : \nu r(L : V_0))$. In this case, we choose $Q = P(L : V_0)[Sh(L)/Sh(L')]$, and still have $[Q]^\dagger \equiv_d H$ and $\text{lf}(P) = \text{lf}(Q)$.

(4) $R = (Ctr\text{-Split}).$  
$P$ must be of the form $P(L : \nu r(f(V_1, \ldots, V_k)), Sh(L))$, in order that $P$ can be applied to $[P]^\dagger$. We choose $Q = P(L : \nu r(f(L_1 : \nu r(V_1), \ldots, L_k : \nu r(V_k)), f(Sh(L_1), \ldots, Sh(L_k)))$, where $L_1, \ldots, L_k$ are fresh, so that $[Q]^\dagger \equiv_d H$ and $\text{lf}(P) = \text{lf}(Q)$.

(5) $R = (Ctr\text{-Norm}).$  
$P$ must be of the form $P(\nu r(f(\bar{V})))$, in order that $P$ can be applied to $[P]^\dagger$. We choose $Q = P(f(\bar{V}))$, so that $[Q]^\dagger \equiv_d H$ and $\text{nf}(\text{lf}(P)) = \text{nf}(\text{lf}(Q))$.

Then, we show that the set of rules for reduction $\delta_R$ are sound, in that they transform the tagged graph of a normal process to the tagged graph of one it reduces to (up to nf-congruence).

**Theorem A.2.6** (Soundness of rules for reduction). For a normal process $P$, a DPO rule $R \in \delta_R$ and a graph $H$ such that $[P]^\dagger \Rightarrow_R H$, there exists a process $Q$, which may not be well-matched, such that $[Q]^\dagger \equiv_d H$. And if $Q$ is well-matched, it is also well-formed and $P \rightarrow \text{nf}(\text{lf}(Q))$.

**Proof.** We prove for each rule $R \in \delta_R$.

(1) $R = (Ser\text{-Sync}).$  
$P$ must be of the form $\Lambda[s, P_1, s, P_2]$ for some static and restriction-balanced context $\Lambda[\cdot, \cdot, \cdot]$ in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = (\nu r)s, \Lambda[\cdot, \cdot, \cdot]R \triangleright GB(P_1; \emptyset : \{s\})$, $\triangleright GB(P_2)$ with $r$ fresh. As a result, $P \rightarrow (\nu r)s, \Lambda[\cdot, \cdot, \cdot]R \triangleright P_1, r \triangleright P_2 = \text{nf}(\text{lf}(Q))$.

(2) $R = (Ses\text{-Sync}).$  
$P$ must be of the form $\Lambda[r \triangleright \Lambda'[\cdot](V)P_1 + U_1], r \triangleright \Lambda_2[(F)P_2 + U_2]$, in order that $R$ can be applied to $[P]^\dagger$, where $\Lambda[\cdot, \cdot, \cdot]$ is static and restriction-balanced, $\Lambda_2[\cdot]$ and $\Lambda'[\cdot]$ are static, session-immune and restriction-immune, and the hole of $\Lambda'[\cdot]$ does not occur in the scope of a pipeline. In other words, the hole of $\Lambda'[\cdot]$ can occur only in the scope of parallel compositions, i.e. there exists a normal process $P'$ such that $\Lambda'[Q]' \equiv_c P'Q'$ for any normal process $Q'$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = \Lambda[r \triangleright \Lambda'[\cdot]GB(P_1; \emptyset : \{U_1\})], r \triangleright \Lambda_2[\cdot]GB(AS(V; F)P_2; \emptyset : \{U_2\})]$. If $Q$ is well-matched, the match $\sigma = \text{match}(F; V)$ exists, so that $Q$ is well-formed and
\[
\begin{align*}
P & \equiv_c \Lambda[r \triangleright (P'|((V)P_1 + U_1)), r \triangleright \Lambda_2[(F)P_2 + U_2)] \\
& \rightarrow \Lambda[r \triangleright (P'|P_1), r \triangleright \Lambda_2[P_2\sigma]] \\
& \equiv_c \Lambda[r \triangleright \Lambda'[P_1], r \triangleright \Lambda_2[P_2\sigma]] \\
& = \text{nf}(\text{lf}(Q)).
\end{align*}
\]
(3) \( R = (\text{Ses-Sync-Ret}) \). 
\( P \) must be of the form \( \Lambda[r \triangleright \Lambda'[r' \triangleright \Lambda_1(V)\uparrow P_1 + U_1], \ r \triangleright \Lambda_2[(F)P_2 + U_2]] \), in order that \( R \) can be applied to \([P]\), where \( \Lambda[\cdot, \cdot] \) is static and restriction-balanced, \( \Lambda_1[\cdot] \), \( \Lambda_2[\cdot] \) and \( \Lambda'[\cdot] \) are static, session-immune and restriction-immune, and the hole of \( \Lambda'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( \Lambda'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( \Lambda'[Q'] \equiv_c P' \equiv Q' \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]\), where \( Q = \Lambda[r \triangleright \Lambda'[r' \triangleright \Lambda_1(GB(P_1; \emptyset \equiv \{U_1\})], \ r \triangleright \Lambda_2[GB(AS(V; F)P_2; \emptyset \equiv \{U_2\})]] \). If \( Q \) is well-matched, the match \( \sigma = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and
\[
P \equiv_c \Lambda[r \triangleright (P[r' \triangleright \Lambda_1(V)\uparrow P_1 + U_1], \ r \triangleright \Lambda_2[(F)P_2 + U_2])]
\equiv_c \Lambda[r \triangleright \Lambda'[r' \triangleright \Lambda_1[P_1]], \ r \triangleright \Lambda_2[P_2\sigma]]
\equiv_c \text{nf}(\text{lff}(Q)).
\]

(4) \( R = (\text{Pip-Sync}) \). 
\( P \) must be of the form \( \Lambda_0[\Lambda'[\{V\}P_1 + U_1] > ((F)P_2 + U_2)] \), in order that \( R \) can be applied to \([P]\), where \( \Lambda_0[\cdot] \) is static, \( \Lambda'[\cdot] \) is static, session-immune and restriction-immune, and the hole of \( \Lambda'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( \Lambda'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( \Lambda'[Q'] \equiv_c P' \equiv Q' \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]\), where \( Q = \Lambda_0[\Lambda'[\{GB(P_1; \emptyset \equiv \{U_1\})] > ([l' : F]P_2 + U_2)AS(V; PC(l') \text{Copy}(l))]). \) If \( Q \) is well-matched, the match \( \sigma = \text{match}(PC(l'); V) = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and
\[
P \equiv_c \Lambda_0[\Lambda'[\{V\}P_1 + U_1]] > ((F)P_2 + U_2])
\equiv_c \Lambda_0[P_2\sigma((P\uparrow P_1) > ((F)P_2 + U_2)])
\equiv_c \text{nf}(\text{lff}(Q)).
\]

(5) \( R = (\text{Pip-Sync-Ret}) \). 
\( P \) must be of the form \( \Lambda_0[\Lambda'[r \triangleright \Lambda_1(V)\uparrow P_1 + U_1] > ((F)P_2 + U_2)] \), in order that \( R \) can be applied to \([P]\), where \( \Lambda_0[\cdot] \) is static, \( \Lambda_1[\cdot] \) and \( \Lambda'[\cdot] \) are static, session-immune and restriction-immune, and the hole of \( \Lambda'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( \Lambda'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( \Lambda'[Q'] \equiv_c P' \equiv Q' \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]\), where \( Q = \Lambda_0[\Lambda'[r \triangleright \Lambda_1(GB(P_1; \emptyset \equiv \{U_1\})] > ([l' : F]t : P_2 + U_2)AS(V; PC(l') \text{Copy}(l))]). \) If \( Q \) is well-matched, the match \( \sigma = \text{match}(PC(l'); V) = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and
\[
P \equiv_c \Lambda_0[P'\uparrow r \triangleright \Lambda_1(V)\uparrow P_1 + U_1)] > ((F)P_2 + U_2)]
\equiv_c \Lambda_0[P_2\sigma((P'\uparrow r \triangleright \Lambda_1[P_1]) > ((F)P_2 + U_2)])
\equiv_c \text{nf}(\text{lff}(Q)).
\]

With the soundness of each individual rule set, we are able to prove the soundness of the whole graph transformation system, with respect to congruence (Theorem 4.4.2) and reduction (Theorem 4.4.3).

**Proof of Theorem 4.4.2**

Proof. \( [P] \uparrow \equiv_{(c \cup \delta_T)} [Q] \) means \( [P] \uparrow \Rightarrow R_1, \ H_1 \Rightarrow R_2 \ldots \Rightarrow R_k, \ H_k \equiv_d [Q] \), for some graphs \( H_1, \ldots, H_k \) and rules \( R_1, \ldots, R_k \in \delta_T \). According to Theorems A.2.1 and A.2.3, there exist a sequence of processes \( P_1, \ldots, P_k \) such that \( [P_j] \uparrow \equiv_d H_j \) for \( 1 \leq j \leq k \) and \( P \equiv_c \text{nf}(\text{lff}(P_1)) \equiv_c \ldots \equiv_c \text{nf}(\text{lff}(P_k)) \). Since \( [P_k] \uparrow \equiv_d H_k \equiv_d [Q] \), \( \text{nf}(\text{lff}(P_k)) \equiv_c Q \). As a result, \( P \equiv_c Q \). \( \square \)
Proof of Theorem 4.4.3

Proof. \([P] \vdash_{\delta_A} \llbracket Q \rrbracket \) means \([P] \equiv_d H_0 \Rightarrow_{R_1} H_1 \Rightarrow_{R_2} \ldots \Rightarrow_{R_k} H_k \equiv_d \llbracket Q \rrbracket\), for some graphs \(H_0, H_1, \ldots, H_k\) and rules \(R_1, \ldots, R_k \in \delta_A\). Suppose \(R_{j_0}(1 \leq j_0 \leq k)\) is the only one among these rules that belongs to \(\delta_R\), i.e. each of the others belongs to \(\delta_F \cup \delta_G \cup \delta_D\). According to Theorems A.2.1, A.2.3, A.2.4, A.2.5 and A.2.6, there exist a sequence of processes \(P_0 \equiv P_1, \ldots, P_{j_0}\), where \(P_{j_0}\) may not be well-formed, such that \([P_j]\) \equiv_d H_j, for \(0 \leq j \leq j_0\) and \(P \equiv_c \text{nf}(\text{lf}(P_1)) \equiv_c \ldots \equiv_c \text{nf}(\text{lf}(P_{j_0}-1))\). Recall that each rule in \(\delta_R\) can only be applied to graphs of normal processes, there must be a normal process \(P'\) such that \([P'] \equiv_d H_{j_0} \equiv_d [P_{j_0}]\), thus \(P' \equiv_c \text{nf}(\text{lf}(P_{j_0}-1))\). Notice that \([P_0]\) \equiv_d H_{j_0} \Rightarrow_{\delta_A} [Q]\), \(P_{j_0}\) must be well-matched, since no rule in \(\delta_A\) is able to transform the tagged graph of a process which is not well-matched to that of a well-matched process. According to Theorem A.2.6, \(P_{j_0}\) is well-formed and \(P' \rightarrow \text{nf}(\text{lf}(P_{j_0}))\). Then, according to Theorems A.2.1, A.2.3, A.2.4 and A.2.5, there exist a sequence of processes \(P_{j_0+1}, \ldots, P_k\) such that \([P'] \equiv_d H_{j_0} \rightarrow_{\delta_R} [Q]\). Since \([P_k]\) \equiv_d H_{j_0} \equiv_d [Q]\), \(P_{j_0}\) is well-formed, such that \(0 \equiv_c \text{nf}(\text{lf}(P_{j_0})) \equiv_c \ldots \equiv_c \text{nf}(\text{lf}(P_k))\). As a result, \(P \equiv_c P' \rightarrow \text{nf}(\text{lf}(P_{j_0})) \equiv_c Q\). \(\square\)

A.3 Proof of Theorems 4.4.4 and 4.4.5 of Completeness

In order to prove the completeness of graph transformation rules, we need to extend the notions of congruence and reduction and consider a few of their variants.

A.3.1 Variants of congruence and reduction

In this subsection, we only consider normal processes and normal contexts.

Strict congruence. For two processes \(P\) and \(Q\), we say \(P\) is one-step congruent with \(Q\), denoted as \(P \equiv_{s} Q\), if there is a congruence rule \(P' \equiv Q'\) (see Fig. 4.1 and Fig. 4.4 in Section 4.1) such that \(P = \Lambda[P']\) and \(Q = \Lambda[Q']\), or \(P = \Lambda[Q']\) and \(Q = \Lambda[P']\), for some context \(\Lambda[\cdot]\). As a result, the congruence relation \(\equiv_{s}\) is the reflexive and transitive closure of \(\equiv_{s}\).

For two processes \(P\) and \(Q\), we say \(P\) is one-step strictly congruent with \(Q\), denoted as \(P \equiv_{s} Q\), if there is a basic congruence rule \(P' \equiv Q'\) (see Fig. 4.1 in Section 4.1) such that \(P = \Lambda[P']\) and \(Q = \Lambda[Q']\), or \(Q = \Lambda[Q']\) and \(Q = \Lambda[P']\), for some context \(\Lambda[\cdot]\). Let \(\equiv_{s}\) be the reflexive and transitive closure of \(\equiv_{s}\). We say \(P\) is strictly congruent with \(Q\) if \(P \equiv_{s} Q\).

Expansion, generalization and reorganization. For two processes \(P\) and \(Q\), we say \(Q\) is a one-step expansion of \(P\), denoted as \(P \triangleright_{*} Q\), if there is a special congruence rule \(P' \equiv_{s} Q'\) (see Fig. 4.4 in Section 4.1) such that \(P = \Lambda[P']\) and \(Q = \Lambda[Q']\) for some context \(\Lambda[\cdot]\). So, \(P \equiv_{s} Q\) means either \(P \equiv_{s} Q\) or \(P \triangleright_{*} Q\) or \(Q \triangleright_{*} P\). Furthermore, if the congruence rule is one of the three for moving restrictions forward, we say \(Q\) is a one-step res-forwardness of \(P\), denoted as \(P \triangleright_{*} Q\). Otherwise, the congruence rule is the one for unfolding applications. In this case, we say \(Q\) is a one-step unfolding of \(P\), denoted as \(P \triangleright_{*} Q\).

For two processes \(P\) and \(Q\), we say \(Q\) is a flexible unfolding of \(P\), denoted as \(P \triangleright_{f} Q\), if \(P = \Lambda[\Pi_1, \ldots, \Pi_k]\) and \(Q = \Lambda[\Pi_1\Pi_1, \ldots, \Pi_k][\Pi_k]\) for some k-hole context \(\Lambda[\cdot, \ldots, \cdot]\) and processes \(P_1, \ldots, P_k\), \(k \geq 0\). Such a flexible unfolding can be achieved by applying one-step unfolding k times, to \(P_1, \ldots, \Pi_k\), respectively. Notice that the order of these \(k\) applications are not significant. This is why we call it “flexible”. In addition, it is worth pointing out that a one-step unfolding is a special case of flexible unfolding with \(k = 1\), i.e. \(P \triangleright_{f} Q\) implies \(P \triangleright_{f} Q\).

For two processes \(P\) and \(Q\), we say \(Q\) is a one-step generalization of \(P\), denoted as \(P \triangleright_{*} Q\), if either \(P \equiv_{s} Q\) or \(P \triangleright_{*} Q\). As a result, \(P \equiv_{s} Q\) if and only if \(P \equiv_{s} Q\) or \(Q \triangleright_{*} P\). Let \(\triangleright_{*}\) be the reflexive and transitive closure of \(\triangleright_{*}\). We say \(Q\) is a generalization of \(P\) if \(P \triangleright_{*} Q\).

For two processes \(P\) and \(Q\), we say \(Q\) is a one-step reorganization of \(P\), denoted as \(P \triangleright_{r} Q\), if either \(P \triangleright_{*} Q\) or \(P \equiv_{s} Q\). As a result, \(P \triangleright_{*} Q\) if and only if \(P \triangleright_{*} Q\) or \(P \triangleright_{*} Q\). Let \(\triangleright_{*}\) be the reflexive and transitive closure of \(\triangleright_{*}\). We say \(Q\) is a reorganization of \(P\) if \(P \triangleright_{*} Q\).
For two processes $P$ and $Q$, we first introduce a couple of lemmas. In this subsection, we only consider normal processes and normal contexts. To prove Proposition A.3.1, we first introduce a couple of lemmas.

**Lemma A.3.1.** If $P \succeq Q$ and $P \succeq Q'$, then $P' \succeq Q'$ and $Q \succeq Q'$ for some process $Q'$ (see Fig. A.3).
Figure A.4: Cases in the proof of Lemma A.3.1

Proof. Suppose from $P$ to $Q$, $!P_0$ is unfolded to $P_0 || P_0$; while from $P$ to $P'$, $!P_1, \ldots, !P_k$ are unfolded to $P_1 || P_1, \ldots, P_k || P_k$, respectively. Since $P'$ is a flexible unfolding of $P$, the replications $!P_1, \ldots, !P_k$ are pairwise irrelevant and their order is not important. As for the relation of $!P_0$ and $!P_1, \ldots, !P_k$ in $P$, there are three cases (see Fig. A.4). (1) $!P_0$ is irrelevant with $!P_1, \ldots, !P_k$. So, $P = \Lambda[!P_0, !P_1, \ldots, !P_k]$ for some context $\Lambda[\cdot, \ldots, \cdot]$. As a result, $Q = \Lambda[!P_0 || P_0, !P_1, \ldots, !P_k]$ and $P' = \Lambda[!P_0, !P_1, \ldots, P_k || P_k]$. In this case, we choose $Q' = \Lambda[!P_0 || P_0, !P_1 || P_1, \ldots, P_k || P_k]$ so that $P' \Rightarrow_c Q'$ and $Q \not\Rightarrow_f Q'$. (2) $!P_0$ is included in one of $!P_1, \ldots, !P_k$. Without loss of generality, suppose it is included in $!P_k$. So, $P = \Lambda[!P_1, \ldots, !P_{k-1}, !P_k]$, and $Q = \Lambda[!P_0 || P_0, !P_1, \ldots, !P_k]$ for some contexts $\Lambda[\cdot, \ldots, \cdot]$ and $\Lambda_1[\cdot, \cdot]$. As a result, $Q = \Lambda[!P_0 || P_0, !P_1, \ldots, !P_{k-1}, !P_k]$, and $P' = \Lambda[!P_0 || P_0, !P_1, \ldots, !P_{k-1}, !P_k]$, where $P_k' = \Lambda_1[!P_0 || P_0]$. In this case, we choose $Q' = \Lambda[!P_0 || P_0, !P_1 \ldots, P_k' || P_k]$ so that $P' \Rightarrow_c Q'$ and $Q \not\Rightarrow_f Q'$. (3) Part of $!P_1, \ldots, !P_k$ is included in $!P_0$. Without loss of generality, suppose $!P_0$ contains $!P_1, \ldots, !P_j$ for some $j \leq k$. So, $P = \Lambda[!P_0, !P_{j+1}, \ldots, !P_k]$, and $Q = \Lambda[!P_0 || P_0, !P_1, \ldots, !P_j]$, where $P_k' = \Lambda_1[!P_0 || P_0]$. As a result, $Q = \Lambda[!P_0 || P_0, !P_{j+1}, \ldots, !P_k]$ and $P' = \Lambda[!P_0 || P_0, !P_{j+1} || P_{j+1}, \ldots, P_k || P_k]$, where $P_0'$ is a shorthand for $\Lambda_1[!P_0 || P_0, \ldots, !P_j]$. Let $Q' = \Lambda[!P_0 || P_0, !P_{j+1} || P_{j+1}, \ldots, P_k || P_k]$. Then $P' \Rightarrow_c Q'$ and $Q \not\Rightarrow_f Q'$. □

Lemma A.3.2. If $P \Rightarrow_c Q$ and $P \not\Rightarrow_f P'$, then $P' \Rightarrow_c Q'$ and $Q \not\Rightarrow_f Q'$ for some $Q'$ (see Fig. A.5).

Proof. Suppose $P = \Lambda[!P_1, \ldots, !P_k]$ and $P' = \Lambda[!P_1 || P_1, \ldots, !P_k || P_k]$. Notice that $P \Rightarrow_c Q$. There are two cases (see Fig. A.5). (1) One of $!P_1, \ldots, !P_k$ is changed when $P$ transforms into $Q$. Without loss of generality, suppose $P_1$ is changed into $P_1'$, i.e. $Q = \Lambda[!P_1', !P_2, \ldots, !P_k]$, and $P_1 \Rightarrow_c P_1'$. In this case, we choose $Q' = \Lambda[!P_0 || P_1'|| P_2 || P_2, \ldots, !P_k || P_k]$ so that $P' \Rightarrow_c Q'$.
A.3. PROOF OF COMPLETENESS

Figure A.5: Idea of Lemma A.3.2

\[
\begin{array}{ccc}
P & \xrightarrow{f} & Q \\
\Rightarrow & & \Rightarrow \\
P' & \xrightarrow{c} & Q'
\end{array}
\]

Figure A.6: Cases in the proof of Lemma A.3.2

Figure A.7: Idea of Lemma A.3.3

Figure A.8: Idea of Lemma A.3.4

and \( Q \xrightarrow{f} Q' \). (2) None of \( P_1, \ldots, P_k \) is changed when \( P \) transforms into \( Q \). Notice that each replication is simply preserved by any congruence rule (except the one \( !P \equiv_c P' !P \)). There must be a \( k \)-hole context \( \Lambda_1[\cdot, \ldots, \cdot] \) such that \( Q = \Lambda_1[!P_1, \ldots, !P_k] \) and that for any processes \( X_1, \ldots, X_k, \Lambda[X_1, \ldots, X_k] \Rightarrow^* \Lambda_1[X_1, \ldots, X_k] \). Let \( Q' \) be \( \Lambda_1[!P_1, \ldots, !P_k] \), then we have \( P' \Rightarrow_c Q' \) and \( Q \xrightarrow{f} Q' \). \( \square \)

A direct deduction of Lemma A.3.1 and Lemma A.3.2 is as follows. Recall that each step of a generalization (\( \Rightarrow^*_c \)) is either an unfolding (\( \xrightarrow{c} \)) or a reorganization (\( \Rightarrow^*_c \)).

**Lemma A.3.3.** If \( P \Rightarrow_c Q \) and \( P \xrightarrow{f} P' \), then \( P' \Rightarrow_c Q' \) and \( Q \xrightarrow{f} Q' \) for some \( Q' \) (see Fig. A.7).

With Lemma A.3.3, we can prove the following lemma.

**Lemma A.3.4.** If \( P \Rightarrow_c Q \) and \( P \Rightarrow^*_c P' \), then \( P' \Rightarrow_c Q' \) and \( Q \Rightarrow_c Q' \) for some \( Q' \) (see Fig. A.8).

**Proof.** There are two case for \( P \Rightarrow^*_c P' \) (see Fig A.9). (1) \( P \Rightarrow^*_c P' \). It is a special case of \( P \xrightarrow{f} P' \). According to Lemma A.3.3, there exists a process \( Q' \) such that \( P' \Rightarrow_c Q' \) and \( Q \xrightarrow{f} Q' \). Notice
that \( G \Rightarrow Q \) implies \( Q \Rightarrow Q' \). (2) \( P \Rightarrow^* P' \), which implies \( rp(P) \equiv_s rp(P') \). In this case, we choose \( Q' = rp(Q) \), so that \( Q \Rightarrow_q Q' \). Also notice that \( P \Rightarrow^e Q \) implies \( rp(P) \Rightarrow^e rp(Q) \). We have \( P' \Rightarrow^e rp(P') \equiv_s rp(P) \Rightarrow^e rp(Q) = Q' \).

With Lemma A.3.4, Proposition A.3.1 can be trivially proved by induction on the number \( k \) of one-step congruences from \( P \) to \( Q \), i.e. \( P = P_0 \equiv^* P_1 \equiv^* \ldots \equiv^* P_k = Q \).

### A.3.3 Proof of Proposition A.3.2

In order to prove Proposition A.3.2, we only need to prove the following proposition, since a generalization is composed of a sequence of one-step generalizations.

**Proposition A.3.5.** \( P \Rightarrow^e Q \) implies \( [P]^\dagger \Rightarrow^{\delta^* \cup \delta_T} [Q]^\dagger \).

We first show that derivations of graphs are preserved by process contexts.

**Lemma A.3.5.** Let \( \delta \) be a set of DPO rules, \( P, Q \) be two processes such that \( [P] \Rightarrow^\ast [Q] \) and \( [P]^\dagger \Rightarrow^{\delta^*} [Q]^\dagger \). Then, for any context \( \Lambda[\cdot] \), \( [\Lambda[P]] \Rightarrow^{\delta^*} [\Lambda[Q]] \) and \( [\Lambda[P]^\dagger] \Rightarrow^{\delta^*} [\Lambda[Q]^\dagger] \).

**Proof.** For any context \( \Lambda[\cdot] \) and process \( X \), \( [\Lambda[X]] \) is constructed based on \( [X] \), i.e. \( [\Lambda[X]] \) is of the form \( G([X]) \). As a result, \( [\Lambda[P]] \equiv_d G([P]) \Rightarrow^{\delta^*} G([Q]) \equiv_d [\Lambda[Q]] \).

If \( \Lambda[\cdot] \) is a static context, \( [\Lambda[X]^\dagger] \) is constructed based on \( [X]^\dagger \) for any process \( X \), i.e. \( [\Lambda[X]^\dagger] \) is of the form \( G'([X]^\dagger) \). In this case, \( [\Lambda[P]^\dagger] \equiv_d G'([P]^\dagger) \Rightarrow^{\delta^*} G'([Q]^\dagger) \equiv_d [\Lambda[Q]^\dagger] \). If \( \Lambda[\cdot] \) is non-static, \( [\Lambda[X]^\dagger] \) is constructed based on the untagged graph \( [X] \) for any process \( X \), i.e. \( [\Lambda[X]^\dagger] \) is of the form \( G''([X]) \). In this case, \( [\Lambda[P]^\dagger] \equiv_d G''([P]^\dagger) \Rightarrow^{\delta^*} G''([Q]^\dagger) \equiv_d [\Lambda[Q]^\dagger] \).

To prove Proposition A.3.5, we also need to study the completeness of copy rules.

**Completeness of copy rules**

We would expect to prove the copy rules \( \delta_P \) are complete, in that they can “unfold” the graph of any replication \( !P \) to that of \( !P|P \). For this purpose, we first show that any pattern, value, and (sub-)process can be correctly copied through applications of copy rules.

**Lemma A.3.6.** Let \( F(\xi_1, \ldots, \xi_k) \) be a normal pattern, where \( x_1, \ldots, x_k \) are all its pattern variables. We have

\[
[P(l : F(\xi_1, \ldots, \xi_k)))\mid Q(PC(l))] \Rightarrow^\delta_P [P(F(pv(l_1 : x_1), \ldots, pv(l_k : x_k)))\mid Q(F(pv(PC(l_1, x_1)), \ldots, pv(PC(l_k, x_k)))]
\]

**Proof.** By induction on the structure of \( F = F(\xi_1, \ldots, \xi_k) \).

1. \( F = ?x. \)
   \[
   [P(l : x)\mid Q(PC(l))] \Rightarrow_{PCopy} [P(pv(l : x))\mid Q(pv(PC(l, x)))]
   \]
A.3. PROOF OF COMPLETENESS

(2) \( F = f(F_1, \ldots, F_j) \).

For \( 1 \leq jj \leq j \), let \( F_{jj} \) be of the form \( F_{jj}(?x_{ij}^{jj}, \ldots, ?x_{kj}^{jj}) \), where \( x_{ij}^{jj}, \ldots, x_{kj}^{jj} \) are all its pattern variables. So, \( x^{1}_{k_j}, x^{2}_{k_j}, \ldots, x^{k}_{k_j} \) are all the pattern variables of \( F \).

\[ \Rightarrow (\text{Abs} - \text{Copy}) \]
\[ \Rightarrow (\text{VC} - \text{Elim} - \text{PC}) \]

Lemma A.3.7. Let \( V(x_1, \ldots, x_k) \) be a normal value, where \( x_1, \ldots, x_k \) are all the occurrences of its value variables. We have \( \|P[l : V(x_1, \ldots, x_k)]Q(VC(l))\| \Rightarrow (\text{Abstr} - \text{Copy})[\|P[\text{copy}(l_0)]\|]Q(VC(l_0)) \).

Proof. By induction on the structure of the value \( V \), similar to Lemma A.3.6.

Lemma A.3.8. Let \( P(x_1, \ldots, x_k, s_1, \ldots, s_j) \) be a normal process without sessions, where \( x_1, \ldots, x_k \) and \( s_1, \ldots, s_j \) are all the occurrences of its free variables and free service names, respectively. For any contexts \( \Lambda_1[\cdot] \) and \( \Lambda_2[\cdot] \), \( \|\Lambda_1[l : P(x_1, \ldots, x_k, s_1, \ldots, s_j)]\| \Rightarrow (\text{Abstr} - \text{Copy})\|\Lambda_2[\text{copy}(l)]\| \).

Proof. By induction on the structure of \( P = P(x_1, \ldots, x_k, s_1, \ldots, s_j) \).

(1) \( P = 0 \).

\[ \|\Lambda_1[l : 0]\|\Lambda_2[\text{copy}(l)]\| \Rightarrow (\text{Abstr} - \text{Copy})\|\Lambda_1[0]\|\Lambda_2[0]\| \]

(2) \( P = (F)Q \).

Let \( F \) be of the form \( F(?y_1, \ldots, ?y_j) \), where \( y_1, \ldots, y_j \) are all its pattern variables. Let \( Q \) be of the form \( Q(x_1, \ldots, x_k, x_{k'}, s_1, \ldots, s_j) \), where \( s_1, \ldots, s_j \) are all the occurrences of its free service names and \( x_1, \ldots, x_k \) are all the occurrences of its free variables, of which \( x_1, \ldots, x_{k'} \) are bounded by \( F \). So, \( x_1, \ldots, x_k \) are all the occurrences of free variables of \( P \).

\[ \Rightarrow (\text{Abs} - \text{Copy}) \]
\[ \Rightarrow (\text{VC} - \text{Elim} - \text{PC}) \]

(3) \( P = \langle V \rangle Q \). (Case \( P = \langle V \rangle \) is similar.)

Let \( V \) be of the form \( V(x_1, \ldots, x_j) \), where \( x_1, \ldots, x_j \) are all the occurrences of its value variables. Let \( Q \) be of the form \( Q(y_1, \ldots, y_{k'}; s_1, \ldots, s_j) \), where \( y_1, \ldots, y_{k'} \) and \( s_1, \ldots, s_j \) are all the occurrences of its free variables and free service names, respectively. So, \( x_1, \ldots, x_j, y_1, \ldots, y_{k'} \) are all
the occurrences of free variables of $P$.

\[ \Rightarrow_{(\text{Con-Copy})} [A_1[l : (V)Q]A_2[Copy(l)]] \]

(\text{Lemma A.3.7, IH}) \Rightarrow_{\delta_P}^* \[
\begin{align*}
&[A_1[l : (V)Q]A_2[Copy(l)]] \\
&[A_1[l' : V]l : Q][A_2[(VC(l'))Copy(l)]]
\end{align*}
\]

\[ \Rightarrow_{(\text{Par-Copy})} [A_1[l : (Q')Q][A_2[Copy(l)]]Copy(l)] \]

\[ \Rightarrow_{(\text{Def-Copy})} [A_1[l : (Q')Q][A_2[(VC(l'))Copy(l)]]Copy(l)] \]

(4) $P = Q$. (Case $P = Q + Q'$ or $P = Q > Q'$ is similar.)

Let $Q$ be of the form $Q(x_1, \ldots, x_k, s_1, \ldots, s_{j_1})$, where $x_1, \ldots, x_k$ and $s_1, \ldots, s_{j_1}$ are all the occurrences of its free variables and free service names, respectively. Similarly, let $Q'$ be of the form $Q'(x'_1, \ldots, x'_{k_2}, s'_1, \ldots, s'_{j_2})$. So, $x_1, x_2, x'_1, \ldots, x'_{k_2}$ and $s_1, s_1', s_2, \ldots, s'_{j_2}$ are all the occurrences of free variables and free service names of $P$, respectively.

\[ \Rightarrow_{(\text{Def-Copy})} [A_1[l : (Q')Q][A_2[(VC(l'))Copy(l)]]Copy(l)] \]

(5) $P = s.Q$. (Case $P = \sigma.Q$ is similar.)

Let $Q$ be of the form $Q(x_1, \ldots, x_k, s_1, \ldots, s_{j_1})$, where $x_1, \ldots, x_k$ and $s_1, \ldots, s_{j_1}$ are all the occurrences of its free variables and free service names, respectively. So, $s_1, \ldots, s_{j_1}$ are all the occurrences of free service names of $P$.

\[ \Rightarrow_{(\text{Def-Copy})} [A_1[l : (Q')Q][A_2[(VC(l'))Copy(l)]]Copy(l)] \]

(6) $P = \alpha.Q$.

There are two cases: $n$ is a variable or a service name. If $n$ is a variable $y$, $P = \alpha.Q$. Let $Q$ be of the form $Q(x_1, \ldots, x_k, y, \ldots, y, s_1, \ldots, s_y)$ (may occur more than once in $Q$), where $x_1, \ldots, x_k$ and $y, s_1, \ldots, s_y$ are all the occurrences of its free variables and free service names, respectively. So, $x_1, \ldots, x_k$ are all the occurrences of free variables of $P$.

\[ \Rightarrow_{(\text{Def-Copy})} [A_1[l : (Q')Q][A_2[(VC(l'))Copy(l)]]Copy(l)] \]

Otherwise, $n$ is a service name $s$. Let $Q$ be of the form $Q(x_1, \ldots, x_k, s, \ldots, s, s_1, \ldots, s_y)$ (may occur more than once in $Q$), where $x_1, \ldots, x_k$ and $s, s_1, \ldots, s_y$ are all the occurrences of its free variables and free service names, respectively. So, $s_1, \ldots, s_y$ are all the occurrences of free service names of $P$. 


A.3. PROOF OF COMPLETENESS

\[\Rightarrow_{(Res-Copy)}\] 
\[\Rightarrow_{(IH)}\Rightarrow_{sp}\] 
\[\Rightarrow_{(VC-Elim-RC)}\] 
\[\Rightarrow_{(RC-Elim)}\] 
\[\equiv_d\] 
\[\Rightarrow_{(Rep-Copy)}\] 
\[\Rightarrow_{(Rep-Step)}\] 
\[\Rightarrow_{(Lemma A.3.8)}\] 
\[\Rightarrow_{(VC-Elim)}\] 
\[\equiv_d\] 

(7) \(P = \parallel Q\).

Let \(Q\) be of the form \(Q(x_1, \ldots, x_k, s_1, \ldots, s_j)\), where \(x_1, \ldots, x_k\) and \(s_1, \ldots, s_j\) are all the occurrences of its free variables and free service names, respectively.

\[\Rightarrow_{(Res-Copy)}\] 
\[\Rightarrow_{(IH)}\Rightarrow_{sp}\] 
\[\Rightarrow_{(VC-Elim-RC)}\] 
\[\Rightarrow_{(RC-Elim)}\] 
\[\equiv_d\] 

Then, we draw the conclusion that the set of copy rules are complete.

**Theorem A.3.1** (Completeness of copy rules). For any normal process \(\parallel P\), \([\parallel P]\Rightarrow_{sp}^{*} [\parallel P] \parallel P\).

**Proof.** Let \(P\) be of the form \(P(x_1, \ldots, x_k, s_1, \ldots, s_j)\), where \(x_1, \ldots, x_k\) and \(s_1, \ldots, s_j\) are all the occurrences of its free variables and free service names, respectively.

\[\Rightarrow_{(Rep-Step)}\] 
\[\Rightarrow_{(Lemma A.3.8)}\] 
\[\Rightarrow_{(VC-Elim)}\] 
\[\equiv_d\] 

**Proof of Proposition A.3.5**

With the completeness of copy rules, we are able to prove that each congruence rule can be simulated by graph transformation rules.

**Lemma A.3.9.** For each basic congruence rule \(LS \equiv_{sc} RS\), \([LS] \Rightarrow_{sc}^{*} [RS] \Rightarrow_{sc}^{*} [LS]^{t}\) and \([LS]^{t} \Rightarrow_{sc}^{*} [RS]^{t} \Rightarrow_{sc}^{*} [LS]^{t}\).

**Proof.** Straightforward for each rule.

1. \(LS = P \parallel P', RS = P' \parallel P\).
\([LS] \Rightarrow_{(Par-Comm)} [RS] \Rightarrow_{(Par-Comm)} [LS]. \ [LS]^{t} \Rightarrow_{(Par-Comm)} [RS]^{t} \Rightarrow_{(Par-Comm)} [LS]^{t}.

2. \(LS = (P \parallel P'), RS = P \parallel (P' \parallel P').\)
\([LS] \Rightarrow_{(Par-Assoc)} [RS] \Rightarrow_{(Par-Comm)} [(P \parallel P')] [P] \Rightarrow_{(Par-Comm)} [(P' \parallel P) [P]]\]
\([LS] \Rightarrow_{(Par-Assoc)} [P' \parallel (P' \parallel P)] \Rightarrow_{(Par-Comm)} [P' \parallel (P' \parallel P)]\]
\([LS] \Rightarrow_{(Par-Comm)} [LS].\)

In the same way, \([LS]^{t} \Rightarrow_{(Par-Assoc)} [RS]^{t} \Rightarrow_{(Par-Comm),(Par-Assoc)} [LS]^{t}.\)

3. \(LS = U + U', RS = U' + U\).
\([LS] \Rightarrow_{(Sum-Comm)} [RS] \Rightarrow_{(Sum-Comm)} [LS]. \ [LS]^{t} \Rightarrow_{(Sum-Comm)} [RS]^{t} \Rightarrow_{(Sum-Comm)} [LS]^{t}.\)
APPENDIX A. PROOF OF SOUNDNESS AND COMPLETENESS OF DPO RULES

(4) \( LS = (U + U') + U'', RS = U + (U' + U'') \).
\[
[LS] \Rightarrow (\text{Sum} - \text{Assoc}) [RS]
\]
\[
\Rightarrow (\text{Sum} - \text{Comm}) [(U + U'') + U'] \Rightarrow (\text{Sum} - \text{Comm}) [(U'' + U') + U]
\]
\[
\Rightarrow (\text{Sum} - \text{Assoc}) [U'' + (U' + U)] \Rightarrow (\text{Sum} - \text{Comm}) [U'' + (U + U')]
\]
\[
\Rightarrow (\text{Sum} - \text{Comm}) [LS].
\]
In the same way, \( [LS]^\dagger \Rightarrow (\text{Sum} - \text{Assoc}) [RS] \Rightarrow [\text{(Sum} - \text{Comm}, (\text{Sum} - \text{Assoc})]) [LS]^\dagger \).

(5) \( LS = U + 0, RS = U \).
\[
[LS] \Rightarrow (\text{Sum} - \text{Unit}) [RS]. \quad [LS]^\dagger \Rightarrow (\text{Sum} - \text{Unit}) [RS]^\dagger.
\]
In order to prove \( [RS] \Rightarrow^*_{\delta_C} [LS] \) and \( [RS]^\dagger \Rightarrow^*_{\delta_C} [LS]^\dagger \), we make induction on the structure of \( U \).

- Case \( U = 0 \).
\[
[RS] \Rightarrow (\text{Nil} - \text{toSum}) [LS]. \quad [RS]^\dagger \Rightarrow (\text{Nil} - \text{toSum}) [LS]^\dagger.
\]

- Case \( U = (F)P \) (Case \( U = (V)P \) or \( (V)^\dagger P \) is similar).
\[
[RS] \Rightarrow (\text{Abs} - \text{toSum}) [LS]. \quad [RS]^\dagger \Rightarrow (\text{Abs} - \text{toSum}) [LS]^\dagger.
\]

- Case \( U = U_1 + U_2 \).
\[
[RS] \Rightarrow (\text{Nil} - \text{toSum}) [LS]. \quad [RS]^\dagger \Rightarrow (\text{Nil} - \text{toSum}) [LS]^\dagger.
\]

In the same way, \( [RS]^\dagger \Rightarrow^*_{\delta_C} [LS]^\dagger \).

(6) \( LS = (\nu m)(\nu m')P, RS = (\nu m')(\nu m)P \).
\[
[LS] \Rightarrow (\text{Res} - \text{Comm}) [RS] \Rightarrow (\text{Res} - \text{Comm}) [LS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger \equiv_d [LS]^\dagger.
\]

(7) \( LS = (\nu m)0, RS = 0 \).
\[
[LS] \Rightarrow (\text{Res} - \text{Unit}) [RS] \Rightarrow (\text{Nil} - \text{toRes}) [LS]. \quad [LS]^\dagger \Rightarrow (\text{Res} - \text{Unit} - A) [RS]^\dagger \Rightarrow (\text{Nil} - \text{toRes} - A) [LS]^\dagger.
\]

Lemma A.3.10. For each special congruence rule \( LS \equiv_c RS \), \( [LS] \Rightarrow^*_{\delta_C} [RS] \) and \( [LS]^\dagger \Rightarrow^*_{\delta_C \cup \delta_T} [RS]^\dagger \).

Proof. Straightforward for each rule.

(1) \( LS = P|(\nu m)Q, RS = (\nu m)(P|Q) \).
\[
[LS] \Rightarrow (\text{Par} - \text{Res} - \text{Comm}) [RS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger.
\]

(2) \( LS = (\nu m)Q > P, RS = (\nu m)(Q > P) \).
\[
[LS] \Rightarrow (\text{Par} - \text{Res} - \text{Comm}) [RS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger.
\]

(3) \( LS = r \triangleright (\nu m)P, RS = (\nu m)(r \triangleright P) \).
\[
[LS] \Rightarrow (\text{Res} - \text{Res} - \text{Comm}) [RS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger.
\]

(4) \( LS = \neg P, RS = P|\neg P \).
According to Theorem A.3.1, \([P] \Rightarrow^*_{\delta_p} [\neg P|P] \). As a result, \([LS] \Rightarrow^*_{\delta_p} [P|P] \Rightarrow (\text{Par} - \text{Comm}) [RS] \).
And
\[
[LS]^\dagger \equiv_d P_{(p,i,o,t)}[A(p)][LS](p,i,o,t)
\]
\[
\Rightarrow^*_{\delta_C} P_{(p,i,o,t)}[A(p)][RS](p,i,o,t)
\]
(4.1.4) \Rightarrow^*_{\delta_T} [RS]^\dagger.

Now, we are ready to prove Proposition A.3.5.

Proof. \( P \equiv^*_c Q \) means \( P \equiv^*_Q \) or \( P \triangleright Q \).
A.3. PROOF OF COMPLETENESS

\[ P \xrightarrow{s} Q \]
\[ P' \xrightarrow{s} Q' \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]
\[ \Lambda[|P_0|] \xrightarrow{s} \Lambda_1[|P_0|] \]
\[ \Lambda[|P_0|] \xrightarrow{s} Q \]
\[ \Lambda[|P_0|] \xrightarrow{s} Q \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]

Figure A.10: Idea of Lemma A.3.11

\[ \Lambda[|P_0|] \xrightarrow{s} \Lambda_1[|P_0|] \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]
\[ \Rightarrow \cdot \]

Figure A.11: Cases in the proof of Lemma A.3.11

- If \( P \equiv_s^* Q \), there exist a basic congruence rule \( LS \equiv_c RS \) such that \( P = \Lambda[LS], Q = \Lambda[RS] \) or \( P = \Lambda[RS], Q = \Lambda[LS] \) for some context \( \Lambda[\cdot] \). According to Lemma A.3.9, we have \( [LS] \Rightarrow_s^* \Lambda_{\cdot} \) \( RS \Rightarrow_s^* [LS] \) and \( [LS] \Rightarrow_s^* [RS] \). Then, according to Lemma A.3.5, we have \( [P]^\downarrow \Rightarrow_s^* [Q]^\downarrow \) (and also \( [Q]^\downarrow \Rightarrow_s^* [P]^\downarrow \)).

- If \( P \triangleright^* Q \), there exist a special congruence rule \( LS \equiv_c RS \) such that \( P = \Lambda[LS] \) and \( Q = \Lambda[RS] \) for some context \( \Lambda[\cdot] \). According to Lemma A.3.10, we have \( [LS] \Rightarrow_s^* [RS] \) and \( [LS] \Rightarrow_s^* [RS] \). Then, according to Lemma A.3.5, we have \( [P]^\downarrow \Rightarrow_s^* [Q]^\downarrow \).

A.3.4 Proof of Proposition A.3.3

In this subsection, we only consider normal processes and normal contexts. In order to prove Proposition A.3.3, we need the following lemma.

Lemma A.3.11. If \( P \rightarrow_s Q \) and \( P \Rightarrow^*_c P' \), then \( P' \rightarrow_s Q' \) and \( Q \Rightarrow^*_c Q' \) for some \( Q' \) (see Fig. A.10).

Proof. There are two cases for \( P \Rightarrow^*_c P' \) (see Fig. A.11). (1) \( P \triangleright^* P' \). Suppose \( P = \Lambda[|P_0|] \) and \( P' = \Lambda[|P_0|] \) for some context \( \Lambda[\cdot] \) and process \( P_0 \). Notice that \( !P_0 \) can not take part in the strict reduction \( P \rightarrow_s Q \). It will be either preserved or simply deleted by the reduction. (1.1) If it is preserved, then there exists a context \( \Lambda_1[\cdot] \) such that \( Q = \Lambda_1[|P_0|] \), and \( \Lambda_1[X] \rightarrow_s \Lambda_1[X] \) for any process \( X \). In this case, we can choose \( Q' = \Lambda_1[|P_0|] \), so that \( P' \rightarrow_s Q' \) and \( Q \Rightarrow^*_c Q' \). (1.2) If \( !P_0 \) is deleted by the reduction, then for any process \( X \), \( \Lambda[X] \rightarrow_s Q \). In this case, we choose \( Q' = \Lambda_1[|P_0|] \), so that \( P' \rightarrow_s Q' \) and \( Q \Rightarrow^*_c Q' \). (2) \( P \Rightarrow^*_c P' \), which implies \( \text{rp}(P) \equiv_s \text{rp}(P') \). In this case, we choose \( Q' = \text{rp}(Q) \), so that \( Q \Rightarrow^*_c Q' \). Also notice that \( P \rightarrow_s Q \) implies \( \text{rp}(P) \rightarrow_s \text{rp}(Q) \). We have \( P' \rightarrow_r \text{rp}(P') \equiv_s \text{rp}(P) \rightarrow_r \text{rp}(Q) = Q' \).

A direct deduction of Lemma A.3.11 is as follows. Notice that a generalization (\( \Rightarrow^*_c \)) is a sequence of one-step generalizations (\( \Rightarrow^*_c \)).

Lemma A.3.12. If \( P \rightarrow_s Q \) and \( P \Rightarrow^*_c P' \), then \( P' \rightarrow_s Q' \) and \( Q \Rightarrow^*_c Q' \) for some \( Q' \) (see Fig. A.12).

Now, we are ready to Proposition A.3.3. The idea of the proof is shown in Fig. A.13.
Proof. $P \rightarrow Q$ means $P \equiv_c P_0 \rightarrow_s Q_0 \equiv_c Q$ for some $P_0$ and $Q_0$. According to Proposition A.3.1, there exists a process $P'$ such that $P \Rightarrow_c P'$ and $P_0 \Rightarrow_c P'$. Then, according to Lemma A.3.12, there exists a process $Q'$ such that $P' \rightarrow_s Q'$ and $Q_0 \Rightarrow_c Q'$. From $Q_0 \Rightarrow_c Q'$, we know that $Q' \equiv_c Q_0 \equiv_c Q$. \qed

A.3.5 Proof of Proposition A.3.4

In order to prove Proposition A.3.4, we only need to prove the following proposition.

**Proposition A.3.6.** $P \rightarrow_p Q$ implies $\llbracket P \rrbracket^\dagger \Rightarrow^{\delta_A \dagger} _{c} \llbracket Q \rrbracket^\dagger$.

For this, we study the completeness of garbage collection rules and data assignment rules.

**Completeness of garbage collection rules**

We would expect to prove that the set of rules $\delta_G$ are complete. That is, in the graph of any process $GB(P; GI)$, the garbage $GI$ can be removed through applications of $\delta_G$. For this, we propose a notion of size of garbage. We are going to show that the total size of garbage strictly decreases through the application of each garbage collection rule, so that the garbage can be removed in finite steps.
Formally, the size of a garbage item $GI$, denoted as $sz(GI)$, is defined inductively as follows.

$$sz(GI) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } GI = s, \text{ch or } \mathit{var}(x) \\ 2 & \text{if } GI = ?x \text{ or } x \\ sz(F_1) + \ldots + sz(F_k) + 1 & \text{if } GI = f(F_1, \ldots, F_k) \\ sz(V_1) + \ldots + sz(V_k) + 1 & \text{if } GI = f(V_1, \ldots, V_k) \\ 4 & \text{if } GI = 0 \\ sz(F) + sz(P) + 1 & \text{if } GI = (FP) \\ sz(V) + sz(P) + 1 & \text{if } GI = (VP) \text{ or } (VP) \uparrow P \\ sz(P_1) + sz(P_2) + 1 & \text{if } GI = P_1 | P_2 \text{ or } P_1 + P_2 \\ sz(P) + 5 & \text{if } GI = sP \text{ or } \overline{s}P \\ sz(P_1) + sz(P_2) + 2 & \text{if } GI = P_1 > P_2 \\ sz(P) + 2 & \text{if } GI = (vn)P \\ sz(P) + 4 & \text{if } GI = !P \\ 0 & \text{if } GI = \mathit{Set} :: \emptyset \\ sz(GI_1) + \ldots + sz(GI_k) & \text{if } GI = \mathit{Set} :: \{GI_1, \ldots, GI_k\} \end{cases}$$

With this definition, we show that a garbage item can always be removed.

**Lemma A.3.13.** For any process $P$ and composite garbage item $GI$, $[GB(P; GI)] \Rightarrow_{s_0}^* [P]$.

**Proof.** By induction on the size of $GI$. If $sz(GI) = 0$, $GI$ is an empty garbage item, so that $[GB(P; GI)] \equiv_d^* [P]$. For $sz(GI) > 0$, $GI$ contains at least one single garbage item $GI_0$. That is, $GI$ is of the form $GI(\mathit{Set} :: \{GI_0, GI_1, \ldots, GI_k\})$. We consider different cases of $GI_0$, and prove in either case $[GB(P; GI)] \Rightarrow_{s_0}^* [P]$, according to IH.

1. $GI_0 = s$ (Case $GI_0 = ch$ or $\mathit{var}(x)$ is similar).
   
   If $s$ is a common node of $[GI]_0$ and $[P]_{p, i, o, t}$, $[GB(P; GI)] \equiv_d^* [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{GI_1, \ldots, GI_k\})$. Otherwise, $[GB(P; GI)] \Rightarrow_{(\mathit{GC})} [GB(P; GI')]$. Since $sz(GI') < sz(GI)$, we have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.

2. $GI_0 = ?x$ (Case $GI_0 = ?x$ is similar).
   
   $[GB(P; GI)] \Rightarrow_{(PV-\mathit{GC})} [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{\mathit{var}(x), GI_1, \ldots, GI_k\})$. Since $sz(GI') < sz(GI)$, we have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.

3. $GI_0 = f(F_1, \ldots, F_j)$ (Case $GI_0 = f(V_1, \ldots, V_j)$ is similar).
   
   $[GB(P; GI)] \Rightarrow_{(\mathit{GC})} [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{F_1, \ldots, F_j, GI_1, \ldots, GI_k\})$. Since $sz(GI') < sz(GI)$, we have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.

4. $GI_0 = 0$.
   
   $[GB(P; GI)] \Rightarrow_{(\\mathit{GC})} [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{i, o, t, GI_1, \ldots, GI_k\})$. Since $sz(GI') < sz(GI)$, we have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.

5. $GI_0 = (F)P_1$.
   
   $[GB(P; GI)] \Rightarrow_{(\mathit{Abs-\mathit{GC})}} [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{bn(F) :: \{F, P_1\}, GI_1, \ldots, GI_k\})$. Notice that $sz(GI') < sz(GI)$. We have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.

6. $GI_0 = (V)P_1$ (Case $GI_0 = (V)P_1$ is similar).
   
   $[GB(P; GI)] \Rightarrow_{(\mathit{Con-\mathit{GC})}} [GB(P; GI')]$, where $GI' = GI(\mathit{Set} :: \{V, P_1, GI_1, \ldots, GI_k\})$. Since $sz(GI') < sz(GI)$, we have $[GB(P; GI')] \Rightarrow_{s_0}^* [P]$, according to IH.
(7) \( G\ell_0 = P_1 | P_2 \) (Case \( G\ell_0 = P_1 + P_2 \) is similar).
\[
[GB(P; G\ell)] \Rightarrow_{(P\sigma\neg-GC)} [GB(P; G\ell')], \text{ where } G\ell' = G\ell(Set \triangleright {P_1, P_2, G\ell_1, \ldots, G\ell_k}).
\]
Since \( sz(G\ell') < sz(G\ell) \), we have \( [GB(P; G\ell')] \Rightarrow_{\delta_G} [P] \), according to IH.

(8) \( G\ell_0 = s.P_1 \) (Case \( G\ell_0 = s.P_1 \) is similar).
\[
[GB(P; G\ell)] \Rightarrow_{(Def-GC)} [GB(P; G\ell')], \text{ where } G\ell' = G\ell(Set \triangleright {s, i, o, t, \{i, o, t\} \triangleright {P_1}, G\ell_1, \ldots, G\ell_k}).
\]
Since \( sz(G\ell') < sz(G\ell) \), we have \( [GB(P; G\ell')] \Rightarrow_{\delta_G} [P] \), according to IH.

(9) \( G\ell_0 = P_1 > P_2 \).
\[
[GB(P; G\ell)] \Rightarrow_{(P\sigma\neg-GC)} [GB(P; G\ell')], \text{ where } G\ell' = G\ell(Set \triangleright {\{s, P_1\}, G\ell_1, \ldots, G\ell_k}).
\]
In either case, \( sz(G\ell') < sz(G\ell) \). According to IH, we have \( [GB(P; G\ell')] \Rightarrow_{\delta_G} [P] \).

(10) \( G\ell_0 = (\nu n)P_1 \).
\[
[GB(P; G\ell)] \Rightarrow_{(Res-GC)} [GB(P; G\ell')], \text{ where } G\ell' = G\ell(Set \triangleright {\{x\} \triangleright \{\text{var}(x), P_1\}, G\ell_1, \ldots, G\ell_k}).
\]
Otherwise, \( n \) is a service name \( s \), and \( [GB(P; G\ell)] \Rightarrow_{(Res-GC)} [GB(P; G\ell')], \text{ where } G\ell' = G\ell(Set \triangleright {\{s\} \triangleright {s, P_1, G\ell_1, \ldots, G\ell_k}}). \)
In either case, \( sz(G\ell') < sz(G\ell) \). According to IH, we have \( [GB(P; G\ell')] \Rightarrow_{\delta_G} [P] \).

As a natural deduction of Lemma A.3.13, the set of garbage collection rules are complete.

**Theorem A.3.2** (Completeness of garbage collection rules). For any process \( P \) and composite garbage item \( G\ell \), \( [GB(P; G\ell)] \Rightarrow_{\delta_G} [P] \) and \( [GB(P; G\ell)] \Rightarrow^{\dagger}_{\delta_G} [P]^\dagger \).

**Proof.** According to Lemma A.3.13, \( [GB(P; G\ell)] \Rightarrow_{\delta_G} [P] \). As a result,
\[
[GB(P; G\ell)] \Rightarrow_{d} [P_{(p,i,o,t)}][A(P)] [GB(P; G\ell)][p,i,o,t] \Rightarrow_{\delta_G} [P_{(p,i,o,t)}][A(P)][p,i,o,t].
\]
According to Theorem 4.4.1, \( [P_{(p,i,o,t)}][A(P)][p,i,o,t] \Rightarrow_{\dagger_G} [P]^\dagger \).

**Completeness of data assignment rules**

We would expect to prove that the set of rules \( \delta_D \) are complete. That is, the graph of \( AS(V; F)P \) can be transformed into that of \( P\sigma \) through applications of these rules, where \( \sigma = \text{match}(F; V) \).

For this purpose, we need to consider the form in which a (normal) value can be shared. Formally, the sharing form of a normal value \( V \), denoted as \( sf(V) \), is defined as follows.
\[
sf(V) \overset{\text{def}}{=} \begin{cases} V & \text{if } V \text{ is a variable } x \\ vv(V) & \text{if } V \text{ is a constructed value } (V_1, \ldots, V_k) \end{cases}
\]
We claim that the graph of the sharing form of a value can be transformed into the graph of the value through applications of data assignment rules.

**Lemma A.3.14.** For any process \( P \) and normal value \( V \), \( [P(sf(V))] \Rightarrow_{\delta_D} [P(V)] \).

**Proof.** If \( V \) is a variable \( x \), we have \( [P(sf(x))] \equiv_d [P(x)] \). If \( V \) is a constructed value \((V_1, \ldots, V_k)\), we have \( [P(sf(f(V_1, \ldots, V_k)))] \equiv_d [P(vv(f(V_1, \ldots, V_k)))] \Rightarrow_{(Ctri-Norm)} [P(f(V_1, \ldots, V_k))] \).

Then, we show that a shared value can be copied through applications of data assignment rules.

**Lemma A.3.15.** For any process \( P \) and normal value \( V \), \( [P(L: vv(V), Sh(L))] \Rightarrow_{\delta_D} [P(L : sf(V), V)] \).

**Proof.** By induction on the structure of \( V \).
(1) \( V = x \).
\[ P(L : vv(x), Sh(L)) \implies_{(V \leftarrow \text{Norm})} P(L : x, Sh(L)) \equiv_d P(L : x, x) \]

(2) \( V = f(V_1, \ldots , V_k) \).
\[ \Rightarrow (\text{Ctr-Split}) \]
\[ \Rightarrow_{\delta_d} \]
\[ \equiv_d \]
\[ P(L : vv(f(V_1, \ldots , V_k), Sh(L))) \]
\[ P(L : vv(f(L_1 : vv(V_1), \ldots , L_k : vv(V_k)), Sh(L_1), \ldots , Sh(L_k))) \]
\[ P(L : vv(f(L_1 : sf(V_1), \ldots , L_k : sf(V_k)), f(V_1), \ldots , f(V_k))) \]
\[ P(L : vV(V_1, \ldots , V_k), f(V_1, \ldots , V_k)) \]

As a natural deduction of Lemma A.3.15, we have the following lemma.

**Lemma A.3.16.** For any process \( P \) and normal value \( V \), \[ P(L : vv(V), Sh(L), \ldots , Sh(L)) \implies_{\delta_d} P(V, V, \ldots , V) \], where \( Sh(L), \ldots , Sh(L) \) are all the occurrences of \( Sh(L) \) in \( P \).

**Proof.** If \( V \) is a variable \( x \),
\[ P(L : vv(x), Sh(L), \ldots , Sh(L)) \equiv_d P(x, x, \ldots , x) \]
If \( V \) is a constructed value \( f(V_1, \ldots , V_k) \),
\[ P(L : vv(V), V_1, \ldots , V_k) \]
\[ P(L : vV(V), V_1, \ldots , V_k) \]
\[ P(L : vV(V), V_1, \ldots , V_k) \]
Now, we are ready to prove the completeness of data assignment rules.

**Theorem A.3.3** (Completeness of data assignment rules). For any normal process \( P \), normal pattern \( F \) and normal value \( V \) such that \( \sigma = \text{match}(F; V) \) exists, \([AS(V; F)P] \Rightarrow_{\delta_d} [P\sigma] \) and \([AS(V; F)P] \Rightarrow_{\delta_d} [P\sigma] \).

**Proof.** Let \( F \) be of the form \( F(\bar{x}_1, \ldots , ?x_k) \), where \( \bar{x}_1, \ldots , x_k \) are all its pattern variables. In order that \( \sigma = \text{match}(F; V) \) exists, \( V \) must be of the form \( F(V_1, \ldots , V_k) \) for some values \( V_1, \ldots , V_k \). That is, \( \sigma = [V_1, \ldots , V_k \mid x_1, \ldots , x_k] \). Let \( P = P(x_1, x_1, \ldots , x_k, x_1, \ldots , x_k) \), where \( x_1, x_1, \ldots , x_1, \ldots , x_k, x_1, \ldots , x_k \) are all the occurrences of its free variables bound by \( F \).

\[ \Rightarrow_{\text{Ctr-Assign}} \]
\[ \Rightarrow_{\text{PV-Assign}} \]
\[ P(L : vv(V_1), Sh(L_1), \ldots , Sh(L_k), \ldots , Sh(L_k)) \]
\[ P(L : vV(V_1, \ldots , V_k), V_1, \ldots , V_k, V_1, \ldots , V_k) \]
\[ [P(V_1, V_1, \ldots , V_k, V_1, \ldots , V_k)] \]

As a result,
\[ [AS(V; F)P] \]
\[ [P\sigma] \]
\[ P_0[i, o, t] \]
\[ [P\sigma] \]
\[ \Rightarrow_{\delta_d} [P\sigma] \]
\[ \Rightarrow_{\delta_d} [P\sigma] \]
\[ \Rightarrow_{\delta_d} [P\sigma] \]

**Proof of Proposition A.3.6**

Finally, Theorems A.3.2 and A.3.3 enable us to prove Proposition A.3.6.

**Proof.** Straightforward for each case of \( P \rightarrow P \).

1. \( P = \Lambda[s.P_1, s.P_2], Q = (vr)\Lambda[r \triangleright P_1, r \triangleright P_2], \) where \( \Lambda[\cdot, \cdot] \) is static and restriction-balanced.

\[ \Rightarrow_{(\text{Ser-sync})} \]
\[ \Rightarrow_{\delta_d \cup \delta_T} \]
\[ \Rightarrow_{\delta_T} \]
\[ \Rightarrow_{\delta_T} \]
\[ [Q] \]
(2) \( P = \lambda[r \triangleright (P'[\{(V)P_1 + U_1\}]), \ r \triangleright \Lambda_2[(F)P_2 + U_2]], \ Q = \lambda[r \triangleright (P'[P_1]), \ r \triangleright \Lambda_2[P_2\sigma]], \) where \( \sigma = \text{match}(F; V), \ \Lambda_1[\cdot, \cdot] \) is static and restriction-balanced, \( \Lambda_2[\cdot] \) is static, session-immune and restriction-immune.

\[
\Rightarrow_{(\text{Ses-Sync})} [P]^\dagger
\]

(\text{Theorem A.3.2, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(\text{Theorem A.3.3, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(3) \( P = \lambda[r \triangleright (P'[\{(V)P_1 + U_1\}]), \ r \triangleright \Lambda_2[(F)P_2 + U_2]], \ Q = \lambda[r \triangleright (P'[P_1]), \ r \triangleright \Lambda_2[P_2\sigma]], \) where \( \sigma = \text{match}(F; V), \ \Lambda_1[\cdot, \cdot] \) is static and restriction-balanced, \( \Lambda_1[\cdot] \) and \( \Lambda_2[\cdot] \) are static, session-immune and restriction-immune.

\[
\Rightarrow_{(\text{Ses-Sync-Ce})} [P]^\dagger
\]

(\text{Theorem A.3.2, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(\text{Theorem A.3.3, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(4) \( P = \lambda_0[[P'[\{(V)P_1 + U_1\} > ((F)P_2 + U_2)]]; \ Q = \lambda_0[P_2\sigma][((P'[P_1] > (F)P_2 + U_2))], \) where \( \sigma = \text{match}(F; V), \ \lambda_0[\cdot] \) is static.

Let \( F \) be of the form \( F(\text{y}_1, \ldots, \text{y}_{1'}), \) where \( \text{y}_1, \ldots, \text{y}_{1'} \) are all its pattern variables. Let \( P_2 \) be of the form \( P_2(x_1, \ldots, x_k, x_{1'}, \ldots, x_{k'}, s_1, \ldots, s_j) \), where \( s_1, \ldots, s_j \) are all the occurrences of its free service names and \( x_1, \ldots, x_{k'} \) are all the occurrences of its free variables, of which \( x_1', \ldots, x_{k'}', \) are all its pattern variables. Let

\[
\Rightarrow_{(\text{Ses-Sync})} [P]^\dagger
\]

(\text{Theorem A.3.2, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(\text{Lemma A.3.6, Lemma A.3.8}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(\text{Theorem A.3.3, Lemma A.3.5}) \( \Rightarrow \sigma_{\lambda_1\lambda_2}^P \)

(5) \( P = \lambda_0[[P[r \triangleright \Lambda_1[\{(V)P_1 + U_1\} > ((F)P_2 + U_2)]]; \ Q = \lambda_0[P_2\sigma][((P[r \triangleright \Lambda_1[P_1]] > ((F)P_2 + U_2))], \) where \( \sigma = \text{match}(F; V), \ \lambda_0[\cdot] \) is static, \( \lambda_1[\cdot] \) is static, session-immune and restriction-immune.

Let \( F \) be of the form \( F(\text{y}_1, \ldots, \text{y}_{1'}), \) where \( \text{y}_1, \ldots, \text{y}_{1'} \) are all its pattern variables. Let \( P_2 \) be of the form \( P_2(x_1, \ldots, x_k, x_{1'}, \ldots, x_{k'}, s_1, \ldots, s_j) \), where \( s_1, \ldots, s_j \) are all the occurrences of its free service names and \( x_1, \ldots, x_{k'} \) are all the occurrences of its free variables, of which \( x_1', \ldots, x_{k'}' \) are bound by \( F \).
A.3. PROOF OF COMPLETENESS

\[ \Rightarrow_{(Pip\text{-}Sync\text{-}Rel)} [P] \]

\[ \Rightarrow_{(\text{Theorem A.3.2, Lemma A.3.5})} \Rightarrow_{\mathbb{A}_P \cup \mathbb{A}_T}^* [A_0[\{(P' \triangleright \Delta_1[P_1]) > ((l' : F)l : P_2 + U_2)\}] AS(V; PC(l'')Copy(l))] \]

\[ (\text{Lemma A.3.6, Lemma A.3.8}) \Rightarrow_{\mathbb{A}_P}^* [A_0[\{(P' \triangleright \Delta_1[P_1]) > ((F(pv(l''_1 : y_1), \ldots, pv(l''_j : y_j))) P_2(vv(l_1 : x_1), \ldots, vv(l_k : x_k), vv(l'_1 : x'_1), \ldots, vv(l'_j : x'_j) + U_2)\}] AS(V; F(pv(PC(l''_1 : y_1), \ldots, pv(PC(l''_j : y_j)))) P_2(vv(VC(l_1)), \ldots, vv(VC(l_k)), vv(VC(l'_1)), \ldots, vv(VC(l'_j))) \]

\[ \Rightarrow_{(VC\text{-}Elim\text{-}PC)} [A_0[\{(P' \triangleright \Delta_1[P_1]) > ((F(pv(l''_1 : y_1), \ldots, pv(l''_j : y_j))) P_2(vv(l_1 : x_1), \ldots, vv(l_k : x_k), x'_1, \ldots, x'_{k'()}, l''_1 : s_1, \ldots, l''_j : s_j + U_2)\}] AS(V; F(pv(PC(l''_1 : y_1), \ldots, pv(PC(l''_j : y_j)))) P_2(vv(VC(l_1)), \ldots, vv(VC(l_k)), x'_1, \ldots, x'_{k'()}, VC(l''_1), \ldots, VC(l''_j)) \]

\[ \Rightarrow_{(\text{Theorem A.3.3, Lemma A.3.5})} \Rightarrow_{\mathbb{A}_P \cup \mathbb{A}_T}^* [A_0[\{(P' \triangleright \Delta_1[P_1]) > ((F)P_2 + U_2)\}] AS(V; F)P_2] \]

\[ \Rightarrow_{(Par\text{-}Comm)} [Q] \]