A Sound and Complete Theory of Graph Transformations for Service Programming with Sessions and Pipelines

Roberto Bruni, Zhiming Liu and Liang Zhao

November 2010
UNU-IIST and UNU-IIST Reports

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

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

UNU-IIST contributes through its programmatic activities:

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

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

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

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

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST’s home page: \url{http://www.iist.unu.edu}, if you would like to know more about UNU-IIST and its report series.

Peter Haddawy, Director
A Sound and Complete Theory of Graph Transformations for Service Programming with Sessions and Pipelines

Roberto Bruni, Zhiming Liu and Liang Zhao

Abstract

Graph transformation techniques, and the Double-Pushout (DPO) approach in particular, have been successfully applied in the modeling of concurrent systems. In this area, a research thread has addressed the definition of concurrent semantics for process calculi. In this report, we provide a theory of graph transformations for service programming with sessions and pipelines. The theory shows how graph transformation can cope with advanced features of service-oriented computing, such as several logical notions of scoping (like sessions and pipelines) together with the interplay between linking and containment. This is illustrated by encoding CaSPiS, a recently proposed process calculus with such sophisticated features. We first exploit a graph algebra and set up a graph model that supports DPO graph transformations. Then, we show how to represent CaSPiS processes as hierarchical graphs in the graph model and their behaviors as transformation rules over these graphs. Finally, we provide the soundness and completeness results of these rules with respect to the reduction semantics of CaSPiS.

Keywords: process calculus, hierarchical graph, graph transformation
Roberto Bruni is an Assistant Professor at University of Pisa. His research interests include software architectures, theory of graph and graph transformation, and formal methods for service-oriented computing. Email: bruni@di.unipi.it

Zhiming Liu is a Senior Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasizing sound methods and tools for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for object-oriented development. Email: Z.Liu@iist.unu.edu

Liang Zhao is a PhD Student of the joint PhD program of UNU-IIST and University of Pisa. His research interests include semantics and type systems of programming languages, graph transformation, and formal methods for object-oriented development. Email: liang@iist.unu.edu

The authors acknowledge support from the project GAVES funded by Macau Science and Technology Development Fund, NSFC 60970031, and by the Italian MIUR project IPODS (PRIN 2008).
# Contents

1. **Introduction** .................................................. 7

2. **The calculus CaSPiS** ........................................... 9
   2.1 Basic processes ............................................. 9
   2.2 Operational semantics in terms of reduction ............... 10
   2.3 Extension with replications ................................ 13

3. **Algebra of hierarchical graphs** .......................... 13
   3.1 Graph grammar .............................................. 14
      3.1.1 Hierarchical graph terms ............................. 14
   3.2 Interpretation of graph terms by hypergraphs ............ 16
   3.3 Graph transformation rules ................................. 18

4. **Graph representation of CaSPiS** ......................... 22
   4.1 Processes as designs ...................................... 22
   4.2 Tagged graph and tagging rules .......................... 25
   4.3 Rules for congruence ...................................... 28
   4.4 Copy rules .................................................. 29
   4.5 Rules for reduction ....................................... 35
   4.6 Garbage collection rules .................................. 39
   4.7 Data assignment rules .................................... 40
   4.8 Summary of graph transformation rules .................. 40
   4.9 An example .................................................. 41

5. **Soundness and completeness of graph transformation rules** 43
   5.1 Auxiliary processes ....................................... 44
   5.2 Proof of soundness ........................................ 50
      5.2.1 Proof of Theorem 2 .................................... 63
      5.2.2 Proof of Theorem 3 .................................... 63
   5.3 Variants of congruence and reduction ....................... 64
   5.4 Proof of completeness ..................................... 65
   5.5 Proof of Proposition 1 ...................................... 66
   5.6 Proof of Proposition 2 ...................................... 69
      5.6.1 Completeness of copy rules .......................... 70
      5.6.2 Proof of Proposition 5 ................................. 73
   5.7 Proof of Proposition 3 ...................................... 76
   5.8 Proof of Proposition 4 ...................................... 77
      5.8.1 Completeness of garbage collection rules .......... 77
      5.8.2 Completeness of data assignment rules .............. 80
      5.8.3 Proof of Proposition 6 ................................. 82

6. **Conclusion** .................................................. 84
1 Introduction

Process calculi are a flexible mathematical formalism that provides a convenient abstraction for concurrent systems, in the same way as $\lambda$-calculus lays the foundation of sequential computation. The main ingredients of process calculi are: an algebra (i.e. a signature and a set of structural congruence axioms) of computational entities, called processes with primitives for communication, parallel composition, etc., and an operational semantics modelling the evolution of processes either in terms of a labelled transition system or as a reduction system, that poses the basis for studying several notions of behavioural equivalence over processes.

Process calculi have become quite mature in the study of traditional concurrent and communicating systems [11, 14], and even advanced to specification and verification of mobile systems [15]. However, these traditional process calculi does not match certain advanced features of service-oriented computing (SOC) like the nested scoping of sessions or pipelining workflows, or the interplay between linking and containment. Though there exist attempts of using $\pi$-calculus [15] as a model of service systems [13, 8], the encoding is quite low level and different first-class aspects in SOC, such as client-service interaction and orchestration, are mixed up and obfuscated. The low level communication primitives of $\pi$-calculus make the analysis quite complicated. Particularly, since the same communication pattern is used to encode many different aspects, it is almost infeasible to re-use static analysis techniques to provide any guarantees about safe interactions.

In order to improve this situation, a few service-oriented calculi are proposed. The Service Centered Calculus (SCC) [1] 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) [2]. CaSPiS still supports the important features of SCC with respect to service autonomy, client-service interaction and orchestration. However, the notions of session and pipelining play a more central role. In CaSPiS, 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 in CaSPiS permits orchestrating the flow of data produced by different sessions. The concept of pipeline is inspired by Orc [16], a basic and elegant programming model for structured orchestration of services. A structured operational semantics of CaSPiS is given in [2] based on labeled transitions. It does yet have a simpler and compact reduction semantics [3], on which we focus, that handles silent actions of processes in the labeled transition system.

As illustrated by a large body of literature, graphs and graph transformations provide useful insights into distributed, concurrent and mobile systems [10, 12, 9]. Following this direction, we are going to define a graph-based concurrent semantics for CaSPiS. This can help, for example, to record causal dependencies between interactions and exploit such information for detecting the possible source of faults and misbehaviors. In order to succeed we need to address two issues. The first is that sessions and pipelines introduce a strong hierarchical nature to a service oriented system in both of its static structure and dynamic behavior. The hierarchical structure also changes during the evolution of the system, due to dynamic creation of sessions by invocations of services, and dynamic creation of processes in pipelines.
Therefore we must deal with hierarchical graphs. The second is that the graph transformation semantics must be “compatible” with the existing interleaving one.

In this report, we propose a hierarchical graph representation of service systems and show how to use graph transformation rules to study their behaviors. More precisely, our first contribution is to set up a model of hierarchical graphs by exploiting a suitable graph algebra. In this model, graph transformations are studied following the well-known Double-Pushout (DPO) approach [7]. Then, we map CaSPiS processes to hierarchical graphs in the graph algebra and define a graph transformation system with a few sets of graph transformation rules. As an important result of this report, we proved that the graph transformation system is not only sound, but also complete with respect to the congruence relation and reduction semantics of CaSPiS.

This report extends the previous workshop paper [6] in several aspects. In this report, we modeled more sophisticated features of CaSPiS such as replication processes and constructed values. Such an extension improves the expressiveness of our framework, enabling us, for example, to reason about persistent services that are always available for invocations. Another progress made in this report is the provision of a full graph transformation framework of reduction, including rules for process copy, data assignment and garbage collection. The workshop version, however, only considered a simple and restricted model that does not involve these behaviors. Besides, a main contribution of this report is the proof of soundness and completeness of the whole graph transformation system. By contrast, the workshop version made the proof only for basic graph transformation rules of congruence.

There is some work that also aims at providing a graph model for SOC. In [5], states of service systems are interpreted as terms of a graph algebra that supports names, name restrictions and design hierarchy. We adopt the grammar of the algebra, but provide a different semantic model where hierarchy is realized by a special kind of edges, called abstract edges. With such a model, we are able to define graph transformation rules in the DPO form that change the hierarchical structure of a graph, i.e. through adding or removing the corresponding abstract edges.

An extension of the work [5] is presented in [4] where the behavior of processes are also studied. The authors defined standard forms for prefixed processes, while the prefix can only be dealt with after the application of a graph transformation rule. Therefore, to handle all CaSPiS processes, an infinite number of rules would be needed. By contrast, our graph model is based on the DPO approach which enables us to define the behavior of all processes using a finite number of rules.

We introduce the calculus CaSPiS in the next section, and our graph model in Section 3. In Section 4 we give the graph representation of CaSPiS processes and define the graph transformation system, followed by a small example to illustrate the application of the graph transformation rules. In Section 5 we provide the soundness and completeness results of the system and give the whole proof of them.
2 The calculus CaSPiS

This section introduces the key notions of the service-oriented calculus CaSPiS [2]. Let $S$, $R$, and $V'$ be three disjoint infinite sets, respectively of service names, session names and variables. We also assume a set $\Sigma$ of constructors. Each constructor $f$ has a fixed arity $\text{ar}(f)$. We allow constants in CaSPiS. Each constant $c$ can be regarded as a constructor of arity 0.

We use $\vec{x}$ to denote a sequence, $\vec{x}[i]$, $|\vec{x}|$ and $\{\vec{x}\}$ to denote the $i$-th element, the length and the set of elements of the sequence, respectively. A sequence of length 0 is the empty sequence, denoted as $\emptyset$. We do not distinguish between an element $x$ and a sequence $(x)$ of length 1.

2.1 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 $(V)$ that generates a value $V$; a return $(V)^\uparrow$ that returns a value $V$ to the outside environment; or an abstraction $(F)$ that is ready to receive a value that matches the pattern $F$. Such a process is called a prefixed process. In a prefixed process, a value can be simply a variable $x$, or a constructed value $f(\vec{V})$ composed of a constructor and a sequence of values. Similarly, a pattern can be a variable $?x$ or a constructed pattern $f(\vec{F})$.

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 is represented by $r\triangleright P$, where $r$ is a session name, and $P$ is the protocol 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 [15] by $(\nu n)P$. This restricts all the occurrences of the name $n$ within $P$, and $P$ is called the scope of the restriction.

**Definition 1 (Basic processes)** A basic CaSPiS process is a term generated by the syntax:

- **Process**
  $$ P, Q ::= M \mid P|Q \mid s.P \mid r\triangleright P \mid P > Q \mid (\nu n)P $$

- **Sum**
  $$ M ::= 0 \mid (F)P \mid (V)^\uparrow P \mid (V)^\uparrow P \mid M + M $$

- **Pattern**
  $$ F ::= ?x \mid f(\vec{F}) $$

- **Value**
  $$ V ::= x \mid f(\vec{V}) $$

where $s \in S$, $r \in R$, $x \in V$, $f \in \Sigma$ and $n \in S \cup R \cup V'$.
We remark 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. We omit $0$ in a prefixed term and write, for example, $(?x)(x)0$ for $(?x)(x)0$. For a pattern $F$, we use $\text{bn}(F)$ to denote 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 $(F)$ with $n \in \text{bn}(F)$. Otherwise, it is a free name, and we use $\text{fn}(P)$ to denote the set of free names of $P$. For a value $V$, we also use the notation $\text{fn}(V)$ to denote the set of variables occurring in $V$. Notice that a variable always occurs free in a value.

**Congruence of processes.** As in the $\pi$-calculus [15], we do not distinguish between processes that are alpha-convertible, for example $(?x)(x)(z)$ and $(?y)(y)(z)$. We also have a set of structural congruence rules among processes. They are classified as basic rules of commutativity and associativity (shown in Fig. 1) and special rules for moving restrictions “forward”, out of a certain scope (shown in Fig. 2).

\[
\begin{align*}
(P|P')|P'' & \equiv_c P'(P'|P'') & M + 0 & \equiv_c M \\
 P|P' & \equiv_c P'|P & (\nu n)(\nu n')P & \equiv_c (\nu n')(\nu n)P \\
 (M + M') + M'' & \equiv_c M + (M' + M'') & (\nu n)0 & \equiv_c 0 \\
 M + M' & \equiv_c M' + M
\end{align*}
\]

Figure 1: Basic congruence rules

\[
\begin{align*}
P|(\nu n)Q & \equiv_c (\nu n)(P|Q) & \text{if } n \notin \text{fn}(P) \\
(\nu n)Q > P & \equiv_c (\nu n)(Q > P) & \text{if } n \notin \text{fn}(P) \\
r \triangleright (\nu n)P & \equiv_c (\nu n)(r \triangleright P) & \text{if } n \neq r
\end{align*}
\]

Figure 2: Special congruence rules

It can be inferred that congruent processes have the same set of free names.

### 2.2 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 $Q$. A step of such a change is called a pure reduction, denoted as $P \rightarrow_P Q$.

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 $\sigma.Q$ synchronize on the service $s$ and its corresponding invocation $\sigma$. 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 $\sigma$, $\sigma.Q$ becomes a session process $r \triangleright Q$ of the same session name $r$. For example, $s.P[\sigma, Q] \rightarrow r \triangleright P[r \triangleright Q]$. 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, $r \triangleright (\langle x \rangle P[r \triangleright (\langle y \rangle Q) \rightarrow r \triangleright P[y/x]r \triangleright Q)$.

A pipelined process $P \triangleright Q$ 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 $(\langle y \rangle P[r \triangleright (\langle x \rangle Q)]Q[y/x]$.

**Contexts.** The formal definition of a reduction needs the notion of *contexts* of processes. A context is a process expression with some “holes”. Specifically, a context with $k$ holes is a process term $C[X_1, \ldots, X_k]$ defined in Definition\[1\] but containing processes variables $X_1, \ldots, X_k$. For such a context, we can always omit its process variables and denote it as $C[\cdot, \ldots, \cdot]$. In most cases, we only need to consider contexts with one or two holes, i.e. $C[\cdot]$ or $C[\cdot, \cdot]$. 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 $\sigma.[\cdot]$, a sum $[\cdot] + M$ or $M + [\cdot]$, a prefix $π[\cdot]$ or the right-hand side of a pipeline $P \triangleright [\cdot]$.

A context is *session-immune* if its hole(s) does not occur in the scope of a session, and *restriction-immune* if its hole(s) does not occur in the scope of a restriction. Furthermore, a 2-hole context is called *restriction-balanced* if the holes occur in the same restriction environment. For example, $(\langle vn \rangle[\cdot]r \triangleright [\cdot]$ is not restriction-balanced, since the first hole is bound by the restriction $(\langle vn \rangle$ while the second is not. However, $(\langle vn \rangle[\cdot]r \triangleright [\cdot]$ is a static context.

**Reduction rules.** Following the discussion about the informal behavior of processes, we summarize the pure reduction rules for service definition, service invocation, session and pipelined processes in Fig.\[3\] where each rule shows a pair of processes $P$ and $Q$ such that $P \rightarrow_p Q$.

In Fig.\[3\] it is required $C_0[\cdot]$ is static; $C[\cdot, \cdot]$ is static and restriction-balanced; $C_1[\cdot]$ and $C_2[\cdot]$ are static, session-immune and restriction-immune. So, there is no rule that allows a pure reduction to take place in a non-static context. In these rules, $σ$ denotes the substitution $\text{match}(F;V)$ calculated from the pattern $F$ and the value $V$ if they match. For example, $\text{match}(f(\langle x, y \rangle); f(z, g(1))) = [z, g(1)/x, y]$, while $\text{match}(f(\langle x, y \rangle); h(1,2))$ is undefined, i.e. the pattern $f(\langle x, y \rangle)$ and value $h(1,2)$ can not interact with each other (or lead to a pure reduction) since they do not match. Formally, $σ = \text{match}(F;V)$ is the substitution such that $\text{dom}(σ) = \text{bn}(F)$ and $\hat{F}σ = V$, where $\hat{F}$ denotes the value obtained from $F$ by replacing each $?x$ with $x$.

A reduction is a generalization of a pure reduction by allowing congruences. Specifically, we say a process $P$ reduces to another process $Q$, denoted by $P \rightarrow Q$, if there are processes $P_0$, $Q_0$ such that $P \equiv_c P_0 \rightarrow_p Q_0 \equiv_c Q$. We use $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$. 
The calculus CaSPIS

\[\begin{align*}
\text{(Sync)} & \quad P = C[x.P_1, \bar{z}.P_2] \\
\text{Q} & = (\nu r)C[r \triangleright P_1, r \triangleright P_2] \quad r \text{ fresh for } C[\cdot, \cdot], P_1, P_2 \\
\text{(S-Sync)} & \quad P = C[r \triangleright (P'(v(V)P_1 + M_1)), r \triangleright C_2[(F)P_2 + M_2]] \\
\text{Q} & = C[r \triangleright (P'P_1), r \triangleright C_2[P_2\sigma]] \\
\text{(S-Sync-Ret)} & \quad P = C[r \triangleright (P'[(V)^\dagger P_1 + M_1]), r \triangleright C_2[(F)P_2 + M_2]] \\
\text{Q} & = C[r \triangleright (P'P_1), r \triangleright C_2[P_2\sigma]] \\
\text{(P-Sync)} & \quad P = C_0[(P'[(V)(P_1 + M_1)]) > ((F)P_2 + M_2)] \\
\text{Q} & = C_0[P_2\sigma)((P'(P_1) > ((F)P_2 + M_2))] \\
\text{(P-Sync-Ret)} & \quad P = C_0[(P'[\nu \bar{c}]C_1[(V)^\dagger P_1 + M_1])) > ((F)P_2 + M_2)] \\
\text{Q} & = C_0[P_2\sigma((P'[\nu \bar{c}]C_1[P_1])) > ((F)P_2 + M_2))] \\
\end{align*}\]

Figure 3: Reduction rules

**Example.** Let us consider the process \(Q \triangleright (\nu r)((\nu \bar{c}) (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P)\), where \(Q = \text{req}(\nu \bar{e})(\nu \bar{c}) (\nu \bar{y}) + (\nu \bar{y})\) is a service to allocate new resources (if available), \(CI = \text{req}((\nu \bar{c}) (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P)\) is a client of \(Q\) and \(P\) is a generic process. Then the above process can evolve as illustrated below.

\[\begin{align*}
Q \triangleright (\nu r)((\nu \bar{c}) (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P) & \rightarrow (\nu r)(r \triangleright (\nu \bar{c}) (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P) \mid (r \triangleright (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P)) \quad \text{(Sync)} \\
& \equiv \quad (\nu r)(\nu \bar{c}) (r \triangleright (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P) \mid (r \triangleright (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P)) \\
& \rightarrow (\nu r)(\nu \bar{c}) (r \triangleright 0 \mid (r \triangleright (\nu \bar{y}) (\nu \bar{e}) + (\nu \bar{y}) P)) \quad \text{(S-Sync)} \\
& \rightarrow (\nu r)(\nu \bar{c}) (r \triangleright 0 \mid (r \triangleright 0 \mid (r \triangleright (\nu \bar{y}) P) \mid P[\nu \bar{y}])) \quad \text{(P-Sync-Ret)}
\end{align*}\]

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

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

- **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 \(r \triangleright r \triangleright P\) are illegal.

- **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 \((\bar{x})P + (\bar{y})Q + \bar{0}\), and
– the right-hand side of a pipeline is 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 detailed conditions are used to rule out certain constructs that are not likely to occur in a service system. From now on, we always assume a process is well-formed unless it is stated otherwise.

2.3 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 constructor of processes, replication, is introduced into CaSPiS.

Definition 2 (CaSPiS processes) The syntax of CaSPiS processes is an extension of the basic one in Definition[7] given by:

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

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 constructor \( ![\cdot] \) 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.

\[
!P \equiv_e P|!P
\]

A replication can take part in a reduction (only) indirectly, i.e., after it is unfolded. For example, if \( P|Q \rightarrow R \), then \( !P|Q \equiv_e P|Q|!Q \rightarrow R|!Q \).

3 Algebra of hierarchical graphs

A CaSPiS process can be represented as a graph. For example in Fig. 4(a), the graph of \( P = \langle x \rangle \langle y \rangle \) shows that \( P \) generates a value \( x \), returns a value \( y \) and then becomes the nil process. The unnamed \( \bullet \) nodes represent the states of the control flow, and the nodes \( \triangleright x \) and \( \triangleright y \) show that \( x \) and \( y \) are values generated and returned by the concretion and return edges. The graph Fig. 4(b) shows process \( Q \) generates a value \( z \) but this value is restricted, thus invisible from outside. A restricted value node is therefore not named in the graph. These graphs are called hypergraphs, since an edge can be associated with one or more nodes. A hypergraph shows the control flow and data flow, as well as the structure of a process, through different types of nodes, such as \( \triangleright \) and \( \bullet \), and different types of edges, such as \( \text{Ret} \), \( \text{Res} \) and \( \text{Con} \).
3.1 Graph grammar

We use a graph algebra [5] to specify and study the algebraic properties of hypergraphs, in order to study CaSPiS processes. Let \( \mathcal{N} \) be a set of node names and \( \mathcal{L} \) be a set of edge labels.

**Definition 3 (Graph term)** A graph term is generated by the grammar

\[
G ::= 0 \mid x \mid l(\vec{x}) \mid G \mid G \mid (\nu x)G
\]

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

Term 0 represents the empty graph, \( x \) is the graph of only one node, \( l(\vec{x}) \) is the graph of an \( l \)-labeled edge attached to nodes \( \vec{x} \) through its tentacles, \( G_1|G_2 \) represents the composition of two graphs and \( (\nu x)G \) is a restriction that binds the name \( x \) in \( G \) so that it is invisible outside \( G \).

A process \( P \) of CaSPiS can be represented by a graph term, denoted by \([P]\). The graph terms of processes \( P \) and \( Q \) in Fig. 4 are respectively

\[
[P] = (\nu p)(\nu p_1)(\nu p_2)(\text{Con}(p,x,p_1)|\text{Ret}(p_1,y,p_2)|\text{Nil}(p_2))
\]

\[
[Q] = (\nu p)(\nu p_1)(\nu p_2)(\text{Res}(p,z,p_1)|\text{Con}(p_1,z,p_2)|\text{Nil}(p_2))
\]

For a graph term \( G \), we denote its hypergraph as \( \mathcal{H}(G) \) and called it the hypergraph of \( G \), and \( \mathcal{H}([P]) \) the underlying graph of process \( P \).

We say a node name in a graph term is free if it is not in the scope of a restriction. As shown in Fig. 4, free node names of \( G \) are shown in its hypergraph with its type symbol, but the node of a restricted name is labeled only by its type without the name. When an edge has more than one tentacle, we usually order them clockwise, with the first one being drawn as an incoming arrow and others as outgoing arrows. If necessary, we explicitly give their order by 1, 2, \ldots, \( k \). For an edge with only one tentacle, it is not significant whether it is shown as an incoming or outgoing edge.

3.1.1 Hierarchical graph terms

The graph grammar defined above only describes single CaSPiS processes that represent closed systems. However, we have to treat open systems and their compositions. For example, the graph term \([P|Q]\) of a
parallel composition can not be directly calculated from the graph terms of $P$ and $Q$. For this we need to define a mechanism of encapsulation, called design, to introduce a hierarchical structure into the graphs. Like simple edges, designs need to be labeled. To this end, we assume a set $\mathcal{D}$ of design labels, and extend the graph grammar of Definition 3 into the following definition.

**Definition 4 (Hierarchical graph term)** A hierarchical graph term is either a graph or a design generated by the grammar

\[
\begin{align*}
\text{Graph} & \quad G ::= \emptyset \mid x \mid I(\overline{x}) \mid G|G \mid (\forall x)G \mid D(\overline{x}) \\
\text{Design} & \quad D ::= \lambda_{\overline{x}}[G]
\end{align*}
\]

where $x \in \mathcal{X}$, $l \in L$ and $L \in \mathcal{D}$.

A design $D = \lambda_{\overline{x}}[G]$ exposes a sequence of free nodes $\overline{x}$ of its body graph $G$ as its interface nodes, interface for short. For a design $D$, the hypergraph of $D(\overline{x})$ is obtained from the hypergraph of $D$ by attaching its interface nodes to the nodes $\overline{x}$. This hypergraph is a simple design edge, and it has the same label as $D$. The hierarchical nature of hypergraphs is given by that the body graph $G$ in a design $\lambda_{\overline{x}}[G]$ may also contain design edges.

Recall that the hypergraphs of $P$ and $Q$ in Fig. 4 and their corresponding graph terms represent $P$ and $Q$ as closed systems. They cannot be composed. We can represent each of them, say $P$, as an open process by a design graph term. Instead of restricting, we expose the first control node $p$ as the interface of the design and label it by the design label $P$. Then we have the design graph terms of $P$ and $Q$.

\[
\begin{align*}
[P] &= P_p[(\forall p_1)(\forall p_2)(\text{Con}(p, x, p_1)|\text{Ret}(p_1, y, p_2)|\text{Nil}(p_2))] \\
[Q] &= P_p[(\forall p_1)(\forall p_2)(\text{Res}(p, z, p_1)|\text{Con}(p_1, z, p_2)|\text{Nil}(p_2))]
\end{align*}
\]

The hypergraphs of these two terms are re-depicted on the top and bottom of Fig. 5(a). These two hypergraphs can then be composed by linking their interface nodes with an edge $\text{Par}$ that also have a third node $p$ to interface with the outside. We then make this composed hypergraph as a design graph labeled by $P$ and exposing $p$ as its interface. This is the graph in Fig. 5(a), representing the parallel composition $P|Q$.

\[
[P|Q] = P_p[(\forall p_1)(\forall p_2)\langle\text{Par}(p, p_1, p_2)\rangle\langle P_p\rangle\langle p_1 \rangle\langle Q_p\rangle\langle p_2 \rangle]
\]

A design plays two roles in the graph representation of a process. First, it represents a service or a session as a hierarchical part of a whole process. In this case the edges $P_p^1$, $P_p^2$ and $P_p^3$ represent the designs, called abstract edges. The other role of a design is to represent the interface of a process through which the process communicates with its environment. When we are not interested in the hierarchical structure, a hypergraph can always be flattened by combining the nodes linked by internal abstract edges. For example, the hypergraph in Fig. 5(b) is the flatten version of the hypergraph in Fig. 5(a). The use of labels such as $P_p^1$ will become clarified by the formal interpretation rules of graph terms.
3.2 Interpretation of graph terms by hypergraphs

A hypergraph has different types of nodes for different modeling entities. In Fig. 4 for example, ⦊ nodes represent data while ⦋ nodes represent states of the control flow. We assume a set $T$ of node types and use $T(x)$ to denote the type of node $x$. Each edge label or design edge label $l$ has an arity $AR(l)$ and a type $T(l)$ that is the sequence of types of the nodes that the edge connects. Thus it is required that $|T(l)| = AR(l)$. An edge $l(\vec{x})$ is well-typed if $\vec{x}$ is of type $T(l)$; a design $D = L_2[G]$ is well-typed if the sequence of its interface nodes $\vec{x}$ is of type $T(L)$; while a design edge $D(\vec{y})$ is well-typed if $D$ is well-typed and $\vec{y}$ is of type $T(L)$.

To syntactically indicate in a graph term if a design edge is to be interpreted in a flatten graph, we assume a designated set of design edge labels $\mathcal{F} \subseteq \mathcal{D}$.

**Definition 5 (Interpretation of graph terms)** For a graph term $G$, we define its interpretation as a hypergraph $\mathcal{H}(G) = (N(G), E(G), AE(G), fn(G), ex(G))$, where $N(G)$ is the set of nodes names, $E(G)$ the set of edges, $AE(G)$ the set of abstract edges, $fn(G)$ the set of free node names, and $ex(G)$ the sequence of interface nodes.

\[
\begin{align*}
\mathcal{H}(\emptyset) &= \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
\mathcal{H}(x) &= \langle \{x\}, \emptyset, \emptyset, \emptyset \rangle \\
\mathcal{H}(l(\vec{x})) &= \langle \{\vec{x}\}, \{l(\vec{x})\}, \emptyset, \emptyset \rangle \\
\mathcal{H}((\nu x)G_1) &= \langle N(G_1), E(G_1), AE(G_1), fn(G_1) \setminus \{x\}, \emptyset \rangle \\
\mathcal{H}(G_1 \cap G_2) &= \langle N(G_1) \cup N(G_2), E(G_1) \cup E(G_2), AE(G_1) \cup AE(G_2), fn(G_1) \cup fn(G_2), \emptyset \rangle \\
\mathcal{H}(L_2[G_1]) &= \langle N(G_1) \cup \{\vec{y}\}, E(G_1), AE(G_1) \cup \{L_2^{i}[\vec{y}[i],\vec{y}'[i]]|1 \leq i \leq |\vec{y}|\}, fn(G_1) \setminus \{\vec{y}\}, \vec{y}' \rangle \\
&\quad (\vec{y}' \text{ fresh}, \ T(\vec{y}') = T(\vec{y}), \ \alpha \text{ fresh for } L) \\
\mathcal{H}(L_2[G_1] \setminus \vec{x}) &= \langle N(G_1) \setminus \{\vec{y}\}, E(G_1) \setminus \{\vec{y}\}, AE(G_1) \setminus \{\vec{y}\}, (fn(G_1) \setminus \{\vec{y}\}) \cup \{\vec{x}\}, \emptyset \rangle \quad (L \in \mathcal{F}) \\
\mathcal{H}(L_2[G_1] \setminus \langle \vec{x} \rangle) &= \langle N(D) \setminus \{\vec{y}\}, E(D) \setminus \{\vec{y}\}, AE(D) \setminus \{\vec{y}\}, fn(D) \setminus \{\vec{y}\} \cup \{\vec{x}\}, \emptyset \rangle \quad (L \notin \mathcal{F}, \ D = L_2[G_1])
\end{align*}
\]

As discussed in the previous subsection, the interpretation of a node, edge, restriction composition of a graph term is straightforward and easy to understand. A design is generally represented by a set of
binary edges, called abstract edges, linking each of the interface node of the design to a fresh node for interaction with the environment. For example, Fig. 5(a) has three abstract edges, and its flatten version Fig. 5(b) has only one, that is $P_1$.

Notice that terms $(\nu x)(\nu y)G$ and $(\nu y)(\nu x)G$ are interpreted as the same hypergraph. We thus extend the restriction operator to a set of nodes and write, for example, $(\nu \{x,y\})G$ for $(\nu x)(\nu y)G$ or $(\nu y)(\nu x)G$.

We show two more examples of the interpretation in Fig. 6, where the hypergraphs of the following terms are depicted, with $L \not\in F$.

$$G_1 = L_{(y_1,y_2)}[l(y_1,x)[y_2][x_1,x_2]|L_{(y_1,y_2)}[y_1][l(y_2,x)]|x_1,x_2]$$

$$G_2 = L_{(y_1,y_2)}[l(y_1,x)[l(y_2,x)])[x_1,x_2]|L_{(y_1,y_2)}[y_1[y_2]|x_1,x_2]$$

\[\text{Figure 6: Hypergraphs of terms}\]

Recall that 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. An abstract edge is represented as a dotted arrow with its label. The subscript $i$ of an abstract label $L_j^i$ indicates that the abstract edge links to the $i$-th interface node, see Fig. 6. The superscript $j$ of an abstract label $L_j^i$ are used to discriminate different occurrences of $L$-labeled designs. For example, the abstract edges in the upper and lower parts of $G_1$ are labeled by different superscripts as $(L_1^1, L_2^2)$ and $(L_2^1, L_2^2)$, respectively. Without these superscripts, the hypergraphs of $G_1$ and $G_2$ would be the same. However, graph terms $G_1$ and $G_2$ are not expected to be equivalent.

A hypergraph full of abstract edge labels looks complicated. So, we simplify its graphic representation by putting the body of each $L$-labeled design into a dotted box labeled by $L$ and removing all the abstract edge labels. We can take the dotted box as a special “edge” and the original abstract edges become its “tentacles”. We use the same convention for edges to order these tentacles. For example, $G_1$ and $G_2$ in Fig. 6 are re-depicted in Fig. 7. Notice that a free node is shared by different design instances, such as $x$ 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. 7 for example, the design $D$ (of label $L$) is simply depicted as a “double box” (with tentacles) since 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 6 (Morphism) A morphism \( m : G_1 \rightarrow G_2 \) is a mapping from one hypergraph \( G_1 \) to another hypergraph \( G_2 \) such that

1. \( m(e) \) has the same type as \( e \), where \( e \) is either a node, an edge or an abstract edge,
2. If \( m \) maps an edge or abstract edge \( l(\vec{x}) \) to \( l(\vec{y}) \), \( m \) maps \( \vec{x} \) to \( \vec{y} \), and
3. \( m \) maps the sequence of interface nodes of \( G_1 \) to those of \( G_2 \).

A morphism \( m : G_1 \rightarrow G_2 \) is 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 also 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 \( G_1 \equiv_d G_2 \), 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. 8.

3.3 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. Graph transformation rules are often defined in terms of the algebraic notions of pushout [7]. Intuitively, a pushout combines a pair of graphs by injecting them into a larger graph with certain common parts.
Definition 7 (Pushout) Let $G_0, G_1$ and $G_2$ be three graphs with two morphisms $m_1 : G_0 \rightarrow G_1$ and $m_2 : G_0 \rightarrow G_2$. A pushout of the pair $(m_1, m_2)$ is a graph $G_3$ together with two morphisms $m_{13} : G_1 \rightarrow G_3$ and $m_{23} : G_2 \rightarrow G_3$ (see Fig. 9), such that:

1. $m_{13} \circ m_1 = m_{23} \circ m_2$, and

2. if there exist a graph $G'_1$ and two morphisms $m'_{13}$ and $m'_{23}$ such that $m'_{13} \circ m_1 = m'_{23} \circ m_2$, there is a unique morphism $m_3 : G_3 \rightarrow G'_1$ such that $m'_{13} = m_3 \circ m_{13}$ and $m'_{23} = m_3 \circ m_{23}$.

We call $(m_2, m_{23})$ a pushout complement of $(m_1, m_{13})$, and vice versa.

![Figure 9: Pushout](image)

This definition shows that the pushout graph $G_3$ is the union of $G_1$ and $G_2$ with the images of $G$ by $m_1$ and $m_2$ being equalized.

A graph transformation rule is always formulated using a pair of pushouts, and such a rule is called a Double-Pushout (DPO) rule.

Definition 8 (Double-Pushout rule) A Double-Pushout rule $R : G_L \xrightarrow{m_L} G_I \xrightarrow{m_I} G_R$ is a pair of morphisms $m_L : G_L \rightarrow G_I$ and $m_I : G_I \rightarrow G_R$, where $m_I$ is injective. Graphs $G_L, G_I$ and $G_R$ are called the left-hand side, the interface and the right-hand side of the rule, respectively.
In many DPO rules, $m_l$ and $m_r$ are identity mappings or they only change a small part of the source graph. We thus simply represent a DPO rule by listing the three graphs as $GL|GI|GR$ when both of its morphisms are identity mappings, otherwise we add necessary annotations to indicate the mapping between nonidentical elements. Here, "|" is just used to separate the graphs, it does not represent a graph composition. We shown two examples of DPO rules $R_1$ and $R_2$ in Fig. 10. In Rule $R_1$, both morphisms are the identity mapping and thus no annotation is needed. For Rule $R_2$, however, we use $x/x' \rightarrow x$ to annotate that $m_r$ maps different nodes $x$ and $x'$ in the interface to the same one $x$ in the right-hand side.

Figure 10: Two DPO rules

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

**Definition 9 (Direct derivation)** Let $R : GL \xrightarrow{m_l} GI \xrightarrow{m_r} GR$ be a DPO rule. Given a graph $G$ and a morphism $m_1 : GL \rightarrow G$, $G'$ is a direct derivation of $G$ by $R$ (based on $m_1$), denoted as $G \Rightarrow_R G'$, if there exist the morphisms in Fig. 11 such that

- both squares are pushouts,
- $m'_l$ is strongly fn-preserving, and
- $m'_r$ is fn-preserving whose image includes all the free names of $G'$. This actually implies $\text{fn}(G') = \text{fn}(G'')$.

Figure 11: Direct derivation

In the definition, $m_1$ is called the match in the derivation as it actually matches graph $GL$ with the subgraph $m_1(GL)$ of $G$. With this match, Rule $R$ allows a graph $G''$ constructed with $\langle m_2, m'_l \rangle$ as a pushout complement of $\langle m_1, m_1 \rangle$. $G'$ is achieved from $G$ by removing the elements, i.e. nodes, edges.
and abstract edges, in $m_1(GL\setminus m_1(GL))$ and preserving the elements in $m_1(m_1(GL))$. Since $m_1$ is injective, there is a unique triple $(m_2, G'', m_1')$ such that $m_1'$ is fn-preserving. Then, a graph $G'$ is obtained from $G''$ by adding the elements corresponding to $GR \setminus m_r(GL)$. An example of direct derivation by Rule $R_1$ from Fig. 10 is shown in Fig. 12.

![Figure 12: A direct derivation by $R_1$](image)

It is not the case that for any match $m_1: GL \rightarrow G$, the pair of pushouts exist in the diagram of Definition 9. For them to exist, $m_1$ must satisfy the following conditions.

- **Identification condition.** The (same) image of two different elements of $GL$ by $m_1$ should be preserved: for elements $e_1 \neq e_2$ in $GL$, $m_1(e_1) = m_1(e_2)$ implies $e_1, e_2 \in m_1(GL)$.

- **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 $G \setminus m_1(GL)$ is attached to a node in $m_1(GL \setminus m_1(GL))$.

- **Combination condition.** Two free nodes can not be combined: for nodes $x, y$ of $GI$ such that $m_r(x) = m_r(y)$ and $m_1(m_l(x)) \neq m_1(m_l(y))$, either $m_1(m_l(x))$ or $m_1(m_l(y))$ is a bound node of $G$.

It is worth pointing out that a direct derivation actually transforms a sub-graph of the source graph $G$ to one of the target graph $G'$, and keeps the other part of nodes and edges unchanged. Therefore, direct derivation is preserved by all graph operators. For example, if $G \Rightarrow_R G'$ for a DPO rules $R$, we also have $G[H \Rightarrow_R G'H], (\exists x)G \Rightarrow_R (\exists x)G', L_{x\overline{x}}[G] \Rightarrow_R L_{x\overline{x}}[G']$ and further $L_{x\overline{x}}[G]/\langle x \rangle \Rightarrow_R L_{x\overline{x}}[G']/\langle x \rangle$, provided each graph is well-typed.

A graph transformation system is defined by a set $\delta$ of DPO rules, and a graph derivation is sequential applications of some DPO rules of the system. Formally, $G'$ is a derivation of $G$ in system $\delta$, denoted as $G \Rightarrow_\delta G'$, if there is a sequence of graphs $G_0, \ldots, G_k (k \geq 0)$ such that $G \equiv_d G_0 \Rightarrow_{R_1} G_1 \Rightarrow_{R_2} \ldots \Rightarrow_{R_k} G_k \equiv_d G'$ for $R_1, \ldots, R_k \in \delta$. As the case $k = 0$, $G \Rightarrow_\delta G'$ holds for any set of rules $\delta$ provided $G \equiv_d G'$. For a DPO rule $R$, we always use $G \Rightarrow_R G'$ as a shorthand for $G \Rightarrow_\{R\} G'$. 

Report No. 445, November 2010

UNU-IIST, P.O. Box 3058, Macao
4 Graph representation of CaSPiS

This section applies the graph algebra for representation of CaSPiS processes and uses graph transformations to study reductions of these processes. For this, we first define a direct representation of a CaSPiS process \( P \) by graph terms \([P]\). This representation is easy to understand, but it is hard to define its reductions for a representation of the reductions of the process. To overcome this problem, we define a tagged version \([P]^\dagger\) of \([P]\), and show \([P]^\dagger\) can be obtained by applying transformation rules to the untagged version \([P]\). Then we define reductions on tagged graphs by graph transformation rules that are consistent with the process reductions.

In order to represent a process as a hierarchical graph, we consider the following sets as our vocabularies.

- **Node types** \( T = \{\bullet, \triangleright, \lozenge\} \)
- **Edge labels** \( L = \{pv, vv, Nil, Abs, Con, Ret, Sum, Par, Def, Inv, Ses, Pip, Res, Rep\} \)
  \( \cup \{A, rv, C, PC, VC, RC, AS\} \cup \Sigma \)
- **Design labels** \( D = \{P, F, V, D, I, S, L, R\} \)
  with \( F = \{P, F, V\} \)

We define three node types \( \bullet, \triangleright \) and \( \lozenge \), 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 (Abs) for abstraction, (Par) for parallel composition and (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 (P), patterns (F), values (V), service definitions (D), invocations (I), sessions (S) and right-hand sides of pipelines (R), respectively. Design edges labeled with P, F and V are flat, so the hierarchy of a graph are introduced only by services, sessions and pipelines.

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, since a process is likely to contain several patterns and values.

**Definition 10 (Graph representation of patterns)** The graph representation of a pattern \( F \), denoted as \([F]_F\), is a design of label \( F \) and type \( \triangleright \), defined as follows (depicted in Fig. 13).

\[
\begin{align*}
[F]_F & \equiv Fv[pv(v,x)] \\
[f(F_1, \ldots, F_k)]_F & \equiv Fv[(v_1, \ldots, v_k)](f(v_1, \ldots, v_k)[[F_1]_F(v_1)] \ldots [[F_k]_F(v_k)])
\end{align*}
\]

A pattern variable is represented as an edge \( pv \) of type \( \langle \triangleright, \triangleright \rangle \) attached to the node of this variable; a constructor \( f \) is represented by an edge labelled by \( f \) which is of arity \( AR(f) = ar(f) + 1 \) and type \( \triangleright \) for each rank. It is straightforward to verify that \( fn([F]_F) = bn(F) \) for each pattern \( F \). The graph of an example pattern \( f(\langle x, g(\langle y, ?z \rangle) \rangle) \) is shown in Fig. 14.
Definition 11 (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. 15).

\[
\llbracket x \rrbracket_V \overset{\text{def}}{=} \forall v. \llbracket vv(v, x) \rrbracket_V
\]
\[
\llbracket f(V_1, \ldots, V_k) \rrbracket_V \overset{\text{def}}{=} \forall v. \llbracket v(v_1, \ldots, v_k) \rrbracket (f(v, v_1, \ldots, v_k) \llbracket V_1 \rrbracket_V(v_1) \ldots \llbracket V_k \rrbracket_V(v_k))
\]

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 $\triangleright$-labeled design. The interface of such a design consist of a $\bullet$ node $p$, together with an input $\diamond$ nodes $i$, an output $\diamond$ node $o$ and a return $\diamond$ node $t$.

Definition 12 (Graph representation of processes) The graph representation of a process $P$, denoted
as \([P]\), is defined by induction on the structure of \(P\) (depicted in Fig. [16]).

\[
\begin{align*}
[0] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[t]o[t] Nil(p) \\
[(F)P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[\{p_1, v\} \cup \text{bn}(F))(Abs(p, v, p_1, i)[\{F\} = (v)[P](p_1, i, o, t)) \\
[(V)P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[\{p_1, v\} (\text{Con}(p, v, p_1, o)[\{V\} = (v)[P](p_1, i, o, t)) \\
[\langle V \rangle P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[\{p_1, v\} (\text{Ret}(p, v, p_1, t)[\{V\} = (v)[P](p_1, i, o, t)) \\
[M + M'] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}(\{p_1, p_2\}) (\text{Sum}(p, p_1, p_2)[M][\{P\} = (p_1, i, o, t)] [M'][p_2, i, o, t]) \\
[P|Q] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}(\{p_1, p_2\}) (\text{Par}(p, p_1, p_2)[P][p_1, i, o, t]) [\{Q\} = (p_2, i, o, t)] \\
[s.P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[t][\{p_1, i, o, t\} = (\text{Def}(p, s, p_1, i, o, t)) [P](p_1, i, o, t)] [t, o, t)] \\
[z.P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[t][\{p_1, i, o, t\} = (\text{Im}(p, s, p_1, i_1, o_1)) [P](p_1, i, o, t)] [p, o] \\
[r \triangleright P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}[t][\{p_1, i, o, t\} = (\text{Inv}(p, r, p_1, i, o_1)) [P](p_1, i, o, t)] [p, o] \\
[(\text{vn})P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}(\{p_1, n\}) (\text{Res}(p, n, p_1)[P](p_1, i, o, t))] \\
[P > Q] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}(\{p_1, p_2, o_1\}) (\text{Rep}(p, p_1, p_2, o_1, i, o, t)] \\
[!P] & \overset{\text{def}}{=} P_{(p, \text{o}, \text{t})}(\{p_1, i, o, t\} (\text{Par}(p, p_1, i, t)[P](p_1, i, o, t)] [t, o, t)] \\
\end{align*}
\]

Figure 16: Graph representation of processes
The nil process $\mathbf{0}$ is represented as an edge $\mathit{Nil}$. An abstraction $(F)P$ is represented as a graph with an edge $\mathit{Abs}$ 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 $\mathit{Con}$ and a $\mathit{Ret}$ edge 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 $\mathit{Par}$ (or $\mathit{Sum}$) edge, 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 $\mathit{Ses}$. The $\mathit{Ses}$ edge is also connected with the input and output channels of $P$. This subgraph is then encapsulated by an $S$-labeled design. The graphs of a service definition and a service invocation are defined similarly.

A pipeline $P > Q$ is represented as an edge $\mathit{Pip}$ connected with the graphs of $P$ and $Q$, where the graph of the right-hand side of the pipeline $Q$ is encapsulated by a $R$-labeled design. A restriction $(vn)P$ and a replication $!P$ are respectively represented as an edge $\mathit{Res}$ and $\mathit{Rep}$, which is attached to the graph of $P$. While in the latter case, the channels of $P$ are invisible from outside.

It is worth pointing out that $\text{fn}([P]) = \text{fn}(P)$ for each process $P$, according to Definition 12.

### 4.2 Tagged graph and tagging rules

In the graph term $[P]$ of a process $P$, each control flow node (● node) $p$ actually represents the “start point” of one of its sub-processes $Q$. In this sense, $p$ corresponds to a context $C[\cdot]$ with $C[Q] = P$. Recall that in a process reduction only sub-processes occurring in static contexts are allowed to interact with each other (e.g. reductions cannot take place under a prefix). Therefore, 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 unary edges labeled by $A$ ($A$ means “active”), called tag edges.

**Definition 13 (Tagged graph of processes)** The tagged graph representation of $P$, denoted as $[P]^\dagger$, is defined by induction on the structure of $P$.

\[
\begin{align*}
[0]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[i|o|A(p)|\mathit{Nil}(p)] \\
[(F)P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,v\} \cup \mathit{bn}(F))](A(p)\mathit{Abs}(p,v,p_1,i)[[F]]v[[P]]\{p_1,i,o,t\})] \\
[(V)P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,v\}](A(p)\mathit{Con}(p,v,p_1,o)[[V]]v[[P]]\{p_1,i,o,t\})] \\
[(V')P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,v\}](A(p)\mathit{Ret}(p,v,p_1,t)[[V]]v[[P]]\{p_1,i,o,t\})] \\
[M + M']^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,p_2\}](A(p)\mathit{Sum}(p,p_1,p_2)[[M]]\{p_1,i,o,t\}[[M']]\{p_2,i,o,t\})] \\
[P|Q]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,p_2\}](A(p)\mathit{Par}(p,p_1,p_2)[[P]]\{p_1,i,o,t\}[[Q]]\{p_2,i,o,t\})] \\
[s.P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,i,o_1\}](A(p)\mathit{Def}(p,s,p_1,i_1,o_1)[[P]]\{p_1,i_1,o_1,t\})\{p,o\}] \\
[r.P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,i,o_1\}](A(p)\mathit{Inv}(p,s,p_1,i_1,o_1)[[P]]\{p_1,i_1,o_1,t\})\{p,o\}] \\
[(r \triangleright P)]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,i,o_1\}](A(p)\mathit{Ses}(p,r,p_1,i_1,o_1)[[P]]\{p_1,i_1,o_1,t\})\{p,o\}] \\
[(vn)P]^\dagger & \overset{\text{def}}{=} P_{(p,i,o,t)}[[v\{p_1,i,o_1\}](A(p)\mathit{Ses}(p,r,p_1,i_1,o_1)[[P]]\{p_1,i_1,o_1,t\})\{p,o\}]
\end{align*}
\]
We show that these rules are able to transform the untagged graph of any process to its tagged version.

\[
\begin{align*}

[P > Q]^\dagger & \overset{\text{def}}{=} P^\dagger\left[ (V\{p_1, p_2, o_1\}) (P_{\text{ip}}(p, p_1, p_2, o_1, i, o, t)) [P]^{\dagger}(p_1, i, o_1, t) \right] [Q](p_1, i, o_1, t) \langle p_2 \rangle \\
[!P]^\dagger & \overset{\text{def}}{=} P^\dagger\left[ (V\{p_1, i_1, a_1, t_1\}) (A(P) | R_{\text{ep}}(p, p_1, i, o, t)) [P]^{\dagger}(p_1, i_1, o_1, t_1) \right]
\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. We depict four representative tagged graphs in Fig. 17. Notice that in the case of a restriction, \( [\langle vn \rangle P]^\dagger \) is quite different from its untagged version. In \( [(\langle vn \rangle P)^\dagger \), a new value is generated and it is denoted by an \( rv \)-labeled edge, and original \( \text{Res} \)-labeled edge in the untagged version is not needed in the tagged version any more. Similar to the untagged version, the tagged graph of a process \( P \) always has the same set of free names as \( P \), i.e. \( \text{fn}([P]^\dagger) = \text{fn}(P) \).

![Figure 17: Tagged graphs of processes](image)

To obtained a tagged graph \( [P]^\dagger \) from its untagged version \( [P] \), we add a tag edge to the start of the control flow of \( [P] \), and then apply a sequence of graph transformation rules. These rules are called tagging rules, denoted as \( \Delta_T \). They are shown in Fig. 18.

The tagging starts with a tag \( A \) at the beginning of the control. It then moves along the flow of control through a session to its body, a pipeline to its left-hand side or a parallel composition to both of its branches, before it stops at a nil process or a dynamic operator. When the tag arrives at a restriction, the restriction edge is removed, its original control flow nodes are combined, and an \( rv \)-labeled edge is added.

We show that these rules are able to transform the untagged graph of any process to its tagged version.
Theorem 1 (Completeness of tagging rules) For any process $P$, $P_{(p,i,o,t)}[A(p)[Q|Q'|p,i,o,t]] \Rightarrow_{\Delta_T}^* [P]^\dagger$.

Proof. By induction on the structure of $P$. In the proofs of theorems, lemmas and propositions throughout this report, we always use “(IH)” as a shorthand for “induction hypothesis”.

1. $P = Q|Q'$.
   \[
   P_{(p,i,o,t)}[A(p)[Q|Q'|p,i,o,t]] \\
   \Rightarrow_{(Par-Tag)} P_{(p,i,o,t)}[(\forall \{p_1,p_2\})(\text{Par}(p,p_1,p_2)A(p_1)||Q|Q'|p_1,i,o,t)A(p_2)||Q'|p_2,i,o,t)] \\
   (IH) \Rightarrow_{\Delta_T}^* [Q|Q']^\dagger \\
   \equiv_d [Q|Q']^\dagger
   \]

2. $P = r\triangleright Q$.
   \[
   P_{(p,i,o,t)}[A(p)[r\triangleright Q|p,i,o,t]] \\
   \Rightarrow_{(Ses-Tag)} P_{(p,i,o,t)}[\{\{\{p_1,p_1,s_1\}\}\{\{p_1,p_1,i_1,o_1\}\}\{\{p,r,p_1,i_1,o_1\}\}\{\{p_1,i_1,o_1,t\}\}\{p,o\}] \\
   (IH) \Rightarrow_{\Delta_T}^* [r\triangleright Q']^\dagger \\
   \equiv_d [r\triangleright Q']^\dagger
   \]
Graph representation of CaSPiS

(3) \[ P = (vn)Q. \]

\[ P_{(p,i,o,t)}[A(p)[(vn)Q][p,i,o,t]] \]

⇒ (Res-Tag) \[ P_{(p,i,o,t)}[(vn)(rv(n))A(p)[[Q][p,i,o,t]]] \]

(4) \[ P = Q > Q'. \]

\[ P_{(p,i,o,t)}[A(p)[Q > Q'][p,i,o,t]] \]

⇒ (Par-Tag) \[ P_{(p,i,o,t)}[(\nu\{p_1,p_2,o_1\})(Pip(p,p_1,p_2,o_1,i,o,t))A(p_1)[[[Q][p_1,i,o_1,t]]] \]

\[ P_{(p,i,o,t)}[\nu\{i,o,t\}[Q'[p,i,o,t][p_2]] \]

4.3 Rules for congruence

We provide a set of graph transformation rules \( \Delta_C \) to “simulate” the congruence relation between CaSPiS processes. \( \Delta_C \) include rules of commutativity, associativity and unit, for moving restrictions forward, as well as rules for making a copy of a (sub-)process. We will introduce the former group of rules here, and leave the copy rules to the next subsection.

The basic congruence rules are for commutativity and associativity of sums and parallel compositions, shown in Fig. [19]. In the case of commutativity, we simply change the order of tentacles of the \( \text{Sum} \) and \( \text{Par} \) edges; while in the case of associativity, we rearrange the configuration of these edges.

![Graph representation of CaSPiS](image)

Figure 19: Rules for commutativity and associativity
In order to simulate the congruence relation $M + 0 = M$, we provide a few unit rules for sums, shown in Fig. 20. 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. 21. These rules includes 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 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 $(\bullet, \bullet, \circ, \circ)$, to copy the control flow of processes, as well as edges of label $PC, VC$ and $RC$, which are of type $(\triangleright, \triangleright)$, 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. 22.

---

**Figure 20: Unit rules for sums**

**Figure 21: Rules for restrictions**
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 cannot 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. 23, 24 and 25 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 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.

Figure 23: Control-copy rules (Part I)
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.
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. 26 and 27 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
Graph representation of CaSPiS

Figure 27: Data-copy rules (Part II)

achieve the graph of the target process. Rules for this purpose are provided in Fig. 28.

Figure 28: Rules for eliminating copy edges

We should be careful in applying these rules. Specifically, there is a priority order among them (and the other copy rules).
Graph representation of CaSPiS

- (VC-Elim-PC) > (PC-Elim) or (VC-Elim)
- (VC-Elim-RC) > (RC-Elim) or (VC-Elim)
- Any control-copy rule or data-copy rule > (PC-Elim) or (RC-Elim) or (VC-Elim)

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

4.5 Rules for reduction

We provide a set of graph transformation rules $\Delta_R$ to simulate 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. 29. 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.

![Figure 29: Rules for reduction (Part I)](image)

We also have a pair of rules for the reduction of a session, shown in Fig. 30. Rule (Ses-Sync) is for the interaction between a concretion and an abstraction of a session $r$. The shared channel node by the edges Con and 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. 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 can be removed by garbage collection, which will be introduced later on. 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 $Ret$ 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.

Figure 30: Rules for reduction (Part II)

We have two more rules for the reduction of a pipeline, shown in Fig. 31. Rule (Pip-Sync) is for the interaction between a concretion and an abstraction of a pipeline. The shared channel node by the edges $Con$ and $pip$ makes sure that the concretion belongs to left-hand side of the pipeline, so that it can communicate with the abstraction $Abs$ 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 $AS$ edge and a $PC$ 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 Ret occurs in body of an additional session. Due to the limit of space, we draw these rules vertically, i.e. from top to bottom.

We require that 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.

Figure 31: Rules for reduction (Part III)
It is also worth pointing out that each rule for the reduction of a session or a pipeline has variants. 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:

- on one session side is a concretion, on the other side is an abstraction,
- on one session side is a concretion, on the other side is a sum of abstractions, and
- on one session side is a sum of concretions, on the other side is an abstraction.

Fig. 32 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. \((F)P \equiv_c (F)P + 0, (V)P \equiv_c (V)P + 0\).

Figure 32: Two variants of Rule (Ses-Sync)
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 DPO rules \( \Delta_G \), called garbage collection rules, to remove these parts from the graph. These rules are shown in Fig. 33 and 34 covering all the cases of isolated process constructs, data and channels.

![Garbage collection rules diagram](image)

Figure 33: Garbage collection rules (Part I)
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 DPO rules $\Delta_D$ for data assignment, as well as the subsequent data normalization. They are called data assignment rules, shown in Fig 35. 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.

4.8 Summary of graph transformation rules

To sum up, we have provided

1. a set of tagging rules $\Delta_T$ (Fig. 18),
2. a set of rules for congruence $\Delta_C$ (Fig. 19, 20 and 21), including a subset $\Delta_P$ of copy rules (Fig. 22, Fig. 23, 24, 25, 26, 27, and 28),
3. a set of rules and reduction $\Delta_R$ (Fig. 29, 30 and 31),
4. a set of garbage collection rules $\Delta_G$ (Fig. 33 and 34), and
5. a set of data assignment rules $\Delta_D$ (Fig. 35).
Graph representation of CaSPiS

Let $\Delta_A$ be the union of them, i.e. the whole set of our graph transformation rules.

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 $P_0 = time.\langle T \rangle | time.\langle ?x \rangle \langle x \rangle^{\uparrow}$. The synchronization between $time$ and $time$ creates a session with a fresh name $r$, and $P_0$ evolves to $P_1 = (\nu r)(r \triangleright \langle T \rangle | r \triangleright \langle ?x \rangle \langle x \rangle^{\uparrow})$. 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 $P_2 = (\nu r)(r \triangleright \langle ?x \rangle \langle x \rangle^{\uparrow})$.

The same behavior can be simulated by graph transformations shown in Fig. 36. The left graph in the
Graph representation of CaSPIS

The first row is $[P_0]^\uparrow (p,i,o,t)$. It is transformed to $[P_1]^\uparrow (p,i,o,t)$ (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 $[P_2]^\uparrow (p,i,o,t)$ (the right graph in the last row) by applying the DPO rules (Ses-Sync), (PV-Assign) and (Ctr-Norm).

Figure 36: Application of Graph Transformation Rules
5 Soundness and completeness of graph transformation rules

In this section, we show that the graph transformation rules we have defined so far are sound and complete, with respect to both congruence and reduction of CaSPiS processes.

The soundness with respect to congruence means that two processes are congruent if the tagged graph of one can be transformed to that of the other, through applications of rules for congruence $\Delta_C$ as well as auxiliary tagging rules $\Delta_T$.

**Theorem 2 (Soundness w.r.t. congruence)** For two processes $P$ and $Q$, $[P]^\dagger \Rightarrow_{\Delta_C \cup \Delta_T}^\ast [Q]^\dagger$ implies $P \equiv_c Q$.

The soundness with respect to reduction means that a process is able to reduce to another process if the tagged graph of the former can be transformed to that of the latter, through applications of DPO rules $\Delta_A$, involving exactly one application of rules for reduction $\Delta_R$.

**Theorem 3 (Soundness w.r.t. reduction)** For two processes $P$ and $Q$, if $[P]^\dagger \Rightarrow_{\Delta_A}^\ast [Q]^\dagger$ 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 can be transformed to the tagged graph of some common process, through applications of rules for congruence $\Delta_C$ as well as necessary tagging rules $\Delta_T$.

**Theorem 4 (Completeness w.r.t. congruence)** For two processes $P$ and $P'$, $P \equiv_c P'$ implies $[P]^\dagger \Rightarrow_{\Delta_C \cup \Delta_T}^\ast [Q]^\dagger$ and $[P']^\dagger \Rightarrow_{\Delta_C \cup \Delta_T}^\ast [Q]^\dagger$ for some process $Q$.

It is worth pointing out that the completeness with respect to congruence does not mean “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. We know, for example, that $!P$ and $!P[P]$ are congruent, and we are able to “unfold” the graph $[[!P]^\dagger$ into $[[!P[P]]^\dagger$ by making a copy of $P$ using the copy rules. However, we can hardly transform from $[[!P[P]]^\dagger$ back to $[[!P]^\dagger$ 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 “for two processes such that one reduces to the other, the tagged graph of the former can always be transformed to that of the latter”. Instead, the tagged graph of the former can be transformed to that of some process congruent with the latter.

**Theorem 5 (Completeness w.r.t. reduction)** For two processes $P$ and $Q$, $P \rightarrow Q$ implies $[P]^\dagger \Rightarrow_{\Delta_A}^\ast [Q']^\dagger$ for some process $Q' \equiv_c Q$. 
5.1 Auxiliary processes

In order to prove these soundness and completeness theorems, we need to reason about all the possible intermediate states of graphs in the applications of DPO rules. Our approach is to introduce auxiliary processes into CaSPiS so that each intermediate state corresponds to an auxiliary process.

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.

\[
\begin{align*}
\text{Process} & \quad P \ ::= \ \ldots \mid l : P \mid (l : s).P \mid \overline{l : s}.P \mid (\nu l : n)P \\
& \quad \mid \text{Copy}(l) \mid \text{VC}(l).P \mid \text{VC}(l).P \mid (\nu \text{RC}(l,n))P \\
& \quad \mid \dagger P \mid \text{GB}(P; GI) \mid \text{AS}(\overline{V}; F)P \\
\text{Pattern} & \quad F \ ::= \ \ldots \mid l : F \mid \text{pv}(l : x) \mid \text{PC}(l) \mid \text{pv}(\text{PC}(l,x)) \\
\text{Value} & \quad V ::= \ \ldots \mid l : V \mid \text{vv}(l : x) \mid \text{VC}(l) \mid \text{vv}(\text{VC}(l)) \\
& \quad \mid \text{vv}(V) \mid L : x \mid L : \text{vv}(V) \mid \text{Sh}(L) \\
\text{Garbage Item} & \quad GI ::= s \mid \text{ch} \mid \text{var}(x) \mid F \mid V \mid P \mid S ::= \{ GI, \ldots, GI \}
\end{align*}
\]

where \( l \in \mathcal{L}C \), \( L \in \mathcal{L}S \), \( s \in \mathcal{S} \), \( n \in \mathcal{S} \cup \mathcal{R} \cup \mathcal{V} \), \( ch \) is a channel name and \( S \) 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 a service name as in \( (l : s).P \) or \( l : s.P \), to a restriction as in \( (\nu l : n)P \), or to the whole process as in \( l : P \). Their corresponding copy processes are \( \text{VC}(l).P \), \( \text{VC}(l).P \), \( \nu \text{RC}(l,n)P \) and \( \text{Copy}(l) \), respectively. Here, \( \text{VC}(l) ((\nu \text{RC}(l,n)) \) or \( \text{Copy}(l) \)) denotes a value (restriction of \( n \) or process) that aims at copying the value (restriction or process) labeled by \( l \) somewhere. 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 \( \text{pv}(l : x) \), where the label \( l \) applies to the pattern variable \( x \). Their corresponding copy patterns are \( \text{PC}(l) \) and \( \text{pv}(\text{PC}(l,x)) \), respectively. Here, \( \text{PC}(l) \) (or \( \text{pv}(\text{PC}(l,x)) \)) denotes a 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 \( \text{vv}(l : x) \), where the label \( l \) applies to the value variable \( x \). Their corresponding copy values are \( \text{VC}(l) \) and \( \text{vv}(\text{VC}(l)) \), respectively. (*)

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

A process can be in the pre-tagged version \( \dagger P \), representing the state that \( P \) is ready for tagging. A process can also be equipped with an assignment, as \( \text{AS}(\overline{V}; F)P \), where a name \( x \in \text{fn}(P) \) is bound by the patterns \( F \) if \( x \in \text{bn}(F) \) (thus it can be renamed through an alpha-conversion). It represents the state that we are ready to assign the values of \( \overline{V} \) to variables of \( P \) through the pattern \( F \). Besides, a process may contain some garbage item GI, denoted as \( \text{GB}(P; GI) \). 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 S :: \{GI_1, \ldots, GI_k\} consists of a set of garbage items GI_1, \ldots, GI_k and bound by a set of names S.

From now on, we use the terminology “process” to denote any process defined by the above syntax and normal processes to denote a process defined by the original CaSPiS syntax (given in Section 2). 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 \( l \in LC \) or \( L \in LS \).

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 \( \uparrow[\cdot] \) and \( AS(V;F)[\cdot] \). We require that each of them is dynamic, so that the notion of static context is still the same as before. Similar to processes, a context defined in Section 2 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 state 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 2). 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 \( \uparrow[\cdot] \), 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 \( L : \text{vv}(V) \) 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 \( L : \text{vv}(V) \) of \( P \). However, such a shared value can be matched by zero or more sharing values \( Sh(L) \).
(4) Conditions for garbage.

- 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 $\emptyset :: \{ GI \}$.

(5) Conditions for assignment.

- For each assignment $AS(V_1, \ldots, V_m; F_1, \ldots, F_{m'})Q$ of $P$, $m = m' \geq 1$, $V_1, \ldots, V_m$ are normal values, $F_1, \ldots, F_{m'}$ do not contain labeled sub-patterns, $\tilde{F}_1, \ldots, \tilde{F}_{m'}$ have distinct bound names, and none of these names occurs in a shared value of $Q$. For a pattern $F$, $\tilde{F}$ is the pattern obtained from $F$ by replacing each copy sub-pattern $PC(l)$ by $F'$, with $l : F'$ occurs elsewhere.
- $P$ is well-matched: for each assignment $AS(\tilde{V}; \tilde{F})Q$ of $P$, $\tilde{F}$ and $\tilde{V}$ match. We say a sequence of patterns $F$ and a sequence of value $V$ match if there is a substitution $\sigma$ such that $\text{dom}(\sigma) = \text{bn}(F)$ and $\tilde{F} \sigma = V$, where $\tilde{F}$ is the values obtained from $F$ by replacing each $?x$ with $x$. Such a substitution is denoted as $\text{match}(F; V)$.

It is worth pointing out that the naming of labels, both for copy and for value-share, is not important, according to Conditions (2) and (3). For example, we do not distinguish between processes $l : P|\text{Copy}(l)$ and $l' : P|\text{Copy}(l')$. In the following discussion, a process always means a well-formed one unless it is stated otherwise.

Graph representation. The graph representation of extended patterns and values are defined as follows.

\[
\begin{align*}
[l : F]_F & \overset{\text{def}}{=} F_v[[F]_F \langle v' \rangle] \\
[pv(l : x)]_F & \overset{\text{def}}{=} F_v[pv(v, x')] \\
[l : V]_V & \overset{\text{def}}{=} V_v[[V]_V \langle v' \rangle] \\
[vv(l : x)]_V & \overset{\text{def}}{=} V_v[vv(v, x')] \\
[vv(V)]_V & \overset{\text{def}}{=} V_v((vv_1)(vv(v, v_1))[[V]_V(v_1)]) \\
[L : vv(V)]_V & \overset{\text{def}}{=} V_v((vv_1)(vv(v, v_1))[[V]_V(v_1)]) \\
[Pc(l)]_F & \overset{\text{def}}{=} F_v[PC_v(v, v')] \\
Pc(l, x)]_F & \overset{\text{def}}{=} F_v[pv(v, x)]PC(x, v') \\
[Vc(l)]_V & \overset{\text{def}}{=} V_v[VC_v(v, v')] \\
[Vc(l)]_V & \overset{\text{def}}{=} V_v[(vx)(vv(v, x))VC(x, v')] \\
[L : x]_V & \overset{\text{def}}{=} V_v[vv(v, x)] \\
[Sh(L)]_V & \overset{\text{def}}{=} V_v[vv(v, v')] 
\end{align*}
\]

The representative ones are depicted in Fig. 37.
The graph representation of a garbage item GI, denoted as $[GI]_g$, is a graph term defined as follows.

$$
\begin{align*}
[s]_g & \overset{\text{def}}{=} s \\
[\text{var}(x)]_g & \overset{\text{def}}{=} x \\
[V]_g & \overset{\text{def}}{=} \{V\} [V](v) \\
[S :: \emptyset]_g & \overset{\text{def}}{=} 0 \\
\end{align*}
$$

$[\mathcal{S}]_g = \{ \mathcal{S}_1, \ldots, \mathcal{S}_k \}$ (k \geq 1)

A garbage item GI is called empty if $[GI]_g \equiv_d 0$. For example, $S :: \emptyset$ and $S :: \{S_1 :: \emptyset, S_2 :: \emptyset\}$ are empty garbage items.

The graph representation of extended processes are defined as follows.

$$
\begin{align*}
[l : P] & \overset{\text{def}}{=} P(p, i, o, t) \\
[\text{Copy}(l)] & \overset{\text{def}}{=} P(p, \cdot, i, o, t) \\
[(l : s).P] & \overset{\text{def}}{=} P(p, \text{IF}(p, [V](s', v, \cdot)[V](v))) \left( \text{Def}(p, s', p, i, o, t) \right) (P(p, i, o, t)) (p, v) \\
[\text{VC}(l).P] & \overset{\text{def}}{=} P(p, \text{IF}(p, [V](s', v, \cdot)[V](v))) \left( \text{Def}(p, s, p, i, o, t) \right) (P(p, i, o, t)) (p, v) \\
[\text{VC}(l).P] & \overset{\text{def}}{=} P(p, \text{IF}(p, [V](s', v, \cdot)[V](v))) \left( \text{Def}(p, s, p, i, o, t) \right) (P(p, i, o, t)) (p, v) \\
[(\nu l : n)P] & \overset{\text{def}}{=} P(p, n) (P(p, n, p)) (P(p, i, o, t)) \\
[(\nu RC(l, n))P] & \overset{\text{def}}{=} P(p, n) (P(p, n, p)) (P(p, i, o, t)) \\
[\uparrow P] & \overset{\text{def}}{=} P \\
[\text{GB}(P; GI)] & \overset{\text{def}}{=} P(p, i, o, t, \text{GB}(P; GI)) \\
[\text{AS}(\overline{V}; \overline{F})]P & \overset{\text{def}}{=} P(p, i, o, t, \text{AS}(\overline{V}; \overline{F})) \\
\end{align*}
$$

Figure 37: Graph representation of extended patterns and values
where $\mathcal{V} = V_1, \ldots, V_m$ and $\mathcal{F} = F_1, \ldots, F_m$. The representative ones are depicted in Fig. 38.

For each of these new process constructs $P_0$, we define its tagged graph as:

$$[[P_0]]^\dagger \overset{\text{def}}{=} \mathcal{P}(p,i,o,t)[A(p)][[P_0]]^\dagger \langle p,i,o,t \rangle.$$  

So, Theorem 1 actually means $[[P]]^\dagger \Rightarrow \delta_{\lambda} [[P]]^\dagger$, 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 states of graphs, we need to introduce a notion of congruence between auxiliary processes. For this purpose, we map all the auxiliary processes
into normal processes so that we can make use of the congruence relation between normal processes (given in Section 2).

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

\[
\begin{align*}
P(l : Q, Copy(l)) & \mapsto P(Q, Q) \\
P((l : s).Q, VC(l).Q') & \mapsto P(s.\bar{Q}, s.Q') \\
P(l : s.Q, VC(l).Q') & \mapsto P(\bar{s}.\bar{Q}, \bar{s}.Q') \\
P((v l : n)Q, (v RC(l, n))Q') & \mapsto P((\bar{v}n)\bar{Q}, (\bar{v}n)Q') \\
P(l : F, PC(l)) & \mapsto P(F, F) \\
P(pv(l : x), pv(PC(l, x))) & \mapsto P(\bar{x}, \bar{x}) \\
P(l : V, VC(l)) & \mapsto P(\bar{V}, V) \\
P(v v(l : x), v v(VC(l))) & \mapsto P(x, x) \\
P(L : x, Sh(L), \ldots, Sh(L)) & \mapsto P(x, x, \ldots, x) \\
P(L : v v(V), Sh(L), \ldots, Sh(L)) & \mapsto P(V, V, \ldots, V) \quad \text{if } V \text{ contains no shared value}
\end{align*}
\]

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 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 \( P \) that 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 \( \text{lf}(P) \). Especially, if \( P \) is label-free itself, \( \text{lf}(P) = P \).

For a well-formed and label-free process \( P \), we can always map it to a well-formed normal process. Such a process is called the normal form of \( P \), denoted as \( \text{nf}(P) \) and defined inductively as follows.

\[
\begin{align*}
\text{nf}(0) & \stackrel{\text{def}}{=} 0 \\
\text{nf}(GB(P; G)) & \stackrel{\text{def}}{=} \text{nf}(P) \\
\text{nf}(\bar{F}) & \stackrel{\text{def}}{=} (F) \text{nf}(P) \\
\text{nf}(\bar{V}) & \stackrel{\text{def}}{=} \langle \bar{V} \rangle \text{nf}(P) \\
\text{nf}(PQ) & \stackrel{\text{def}}{=} \text{nf}(P) | \text{nf}(Q) \\
\text{nf}(sP) & \stackrel{\text{def}}{=} s \text{nf}(P) \\
\text{nf}(\bar{v}n) & \stackrel{\text{def}}{=} (\bar{v}n) \text{nf}(P) \\
\text{nf}(\bar{v}P) & \stackrel{\text{def}}{=} !\text{nf}(P)
\end{align*}
\]

where \( \sigma = \text{match}(\bar{F}; \bar{V}) \), and \( \bar{V} \) is the value obtained from \( V \) by replacing each \( v v(V') \) by \( V' \). We can infer
from the above definition that \( \text{nf}(C[P]) = C[\text{nf}(P)] \) for any normal context \( C[\cdot] \) 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 also called the normal form of \( P \). Notice that \( \text{nf}(\text{lf}(P)) = \text{nf}(P) \) if \( P \) is label-free. So, the use of the terminology “normal form” does not lead to ambiguity.

\textbf{Nf-congruence.} With the notion of normal form, we can define the congruence relation between processes. Specifically, two processes \( P \) and \( Q \) are called \( \text{nf-congruent} \), 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. \( [P]^\dagger \equiv_d [Q]^\dagger \), in one of the following basic cases or their combinations.

- \( P \) and \( Q \) are alpha-convertible, or different only in the naming of labels. For example, \( P = \nu l : P' \mid \text{Copy}(l) \) and \( Q = \nu l' : P' \mid \text{Copy}(l') \).
- \( P = C[C'[\nu n]P'] \) and \( Q = C[(\nu n)C'[P']] \), or vice versa, where both \( C[\cdot] \) and \( C'[\cdot] \) are static.
- \( P = C[\dagger P'] \) and \( Q = C[P'] \), or vice versa, where \( P' \) is constructed through a dynamic process operator if \( C[\cdot] \) is static.
- \( P = P(L : x, Sh(L), \ldots, Sh(L)) \) and \( Q = P(x, x_1, \ldots, x) \), where \( Sh(L), \ldots, Sh(L) \) are all the occurrences of \( Sh(L) \) in \( P \), or vice versa.
- \( P = P(L : vv(V)) \) and \( Q = P(vv(V)) \), or vice versa, where no \( Sh(L) \) occurs in \( P \).
- \( P = C[\text{GB}(P'; GI)] \) and \( Q = C[\dagger P'] \), or vice versa, where \( GI \) is empty.
- \( P \) and \( Q \) are different only in the distribution of garbage item. For example, \( P = \text{GB}(P_1; GI)\mid P_2 \) and \( Q = P_1 \mid \text{GB}(P_2; GI) \).

In either case, \( P \) and \( Q \) are \( \text{nf-congruent} \). So, for any graph \( H \), the process \( P \) such that \( [P]^\dagger \equiv_d H \) is unique up to \( \text{nf-congruence} \), if it exists.

\section{5.2 Proof of soundness}

The outline of the proof is shown in Fig. 39. The target is to prove Theorem 2 and Theorem 3, i.e. the soundness of DPO rules with respect to congruence and reduction. For this purpose, we will introduce one lemma and a few sub-theorems. The casual dependency of the lemma and theorems is shown by arrows in the figure. To improve the readability of the figure, we manually draw some of the arrows as dotted ones.
To prove the soundness of the DPO rules, we first show that nf-congruence is preserved by any contexts, so that it is indeed a congruence relation.

**Lemma 1** For any processes $P$, $Q$ and context $C[\cdot]$, $\text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q))$ implies $\text{nf}(\text{lf}(C[P])) \equiv_c \text{nf}(\text{lf}(C[Q]))$.

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

If $C[\cdot]$ 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 $\text{nf}(\text{lf}(C[P])) = \text{nf}(\text{lf}(C[Q]))$.

In the rest of the proof, we assume $C[\cdot]$ is label-free and garbage-free, and make induction on its structure.

1. $C[\cdot] = [\cdot]$. This case is trivial.


   $\text{nf}(\text{lf}(C[P])) = (F) \text{nf}(\text{lf}(C'[P]))$

   **(IH) $\equiv_c (F) \text{nf}(\text{lf}(C[Q])) = \text{nf}(\text{lf}(C[Q]))$**

3. $C[\cdot] = \langle V \rangle C'[\cdot]$ (Case $C[\cdot] = \langle V \rangle C'[\cdot]$ is similar). 

   $\text{nf}(\text{lf}(C[P])) = \langle V \rangle \text{nf}(\text{lf}(C'[P]))$

   **(IH) $\equiv_c \langle V \rangle \text{nf}(\text{lf}(C[Q])) = \text{nf}(\text{lf}(C[Q]))$**


   $\text{nf}(\text{lf}(C[P])) = \text{nf}(\text{lf}(C'[P])) + \text{nf}(\text{lf}(M))$

   **(IH) $\equiv_c \text{nf}(\text{lf}(C'[Q])) + \text{nf}(\text{lf}(M)) = \text{nf}(\text{lf}(C[Q]))$**

Figure 39: Outline of the soundness proof
(5) \( C[\cdot] = C'[\cdot] \mid P_1 \) (Case \( C[\cdot] = P_1 \mid C'[\cdot] \) is similar).
\[
\text{nf}(\text{lf}(C'[P])) = \text{nf}(\text{lf}(C'[P])) \mid \text{nf}(\text{lf}(P_1)) \\
(\text{IH}) \equiv_c \text{nf}(\text{lf}(C'[Q])) \mid \text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}(C[Q]))
\]

(6) \( C[\cdot] = sC'[\cdot] \) (Case \( sC'[\cdot] \) is similar).
\[
\text{nf}(\text{lf}(C'[P])) = s \cdot \text{nf}(\text{lf}(C'[P])) \\
(\text{IH}) \equiv_c s \cdot \text{nf}(\text{lf}(C'[Q])) = \text{nf}(\text{lf}(C[Q]))
\]

(7) \( C[\cdot] = r \triangleright C'[\cdot] \).
\[
\text{nf}(\text{lf}(C'[P])) = r \triangleright \text{nf}(\text{lf}(C'[P])) \\
(\text{IH}) \equiv_c r \triangleright \text{nf}(\text{lf}(C'[Q])) = \text{nf}(\text{lf}(C[Q]))
\]

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

(9) \( C[\cdot] = (vn)C'[\cdot] \).
\[
\text{nf}(\text{lf}(C'[P])) = (vn) \cdot \text{nf}(\text{lf}(C'[P])) \\
(\text{IH}) \equiv_c (vn) \cdot \text{nf}(\text{lf}(C'[Q])) = \text{nf}(\text{lf}(C[Q]))
\]

(10) \( C[\cdot] = \lnot C'[\cdot] \) (Case \( C[\cdot] = \lnot C'[\cdot] \) is similar).
\[
\text{nf}(\text{lf}(C'[P])) = \lnot \text{nf}(\text{lf}(C'[P])) \\
(\text{IH}) \equiv_c \lnot \text{nf}(\text{lf}(C'[Q])) = \text{nf}(\text{lf}(C[Q]))
\]

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

(12) \( C[\cdot] = AS(\tilde{V}; F)C'[\cdot]. \)
\[
\text{nf}(\text{lf}(C'[P])) = \text{nf}(\text{lf}(C'[P])) \sigma \\
(\text{IH}) \equiv_c \text{nf}(\text{lf}(C'[Q])) \sigma = \text{nf}(\text{lf}(C[Q])) \quad \sigma = \text{match}(\tilde{F}; \tilde{V})
\]

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 a nf-congruent one.
Theorem 6 (Soundness of tagging rules) For a process $P$, a DPO rule $R \in \Delta_T$ and a graph $H$ such that $\llbracket P \rrbracket \uparrow \Rightarrow_K H$, there exists a process $Q$ such that $\llbracket Q \rrbracket \uparrow \equiv_d H$ and $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$.

Proof. We prove for each rule $R \in \Delta_T$.

1. $R = (\text{Ses-Tag})$.
   $P$ must be of the form $C[\ddagger r \triangleright P_1]$ for some static context $C[\cdot]$, in order that $R$ can be applied to $\llbracket P \rrbracket \uparrow$. In this case, $H \equiv_d \llbracket Q \rrbracket \uparrow$, where $Q = C[\ddagger r \triangleright P_1]$. In addition, we have $\mathsf{nf}(\mathsf{lf}(\ddagger r \triangleright P_1)) = \mathsf{nf}(\mathsf{lf}(\ddagger r \triangleright P_1)) = \mathsf{nf}(\mathsf{lf}(\ddagger r \triangleright P_1))$. According to Lemma 1, $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$.

2. $R = (\text{Par-Tag})$.
   $P$ must be of the form $C[\ddagger (P_1 > P_2)]$ for some static context $C[\cdot]$, in order that $R$ can be applied to $\llbracket P \rrbracket \uparrow$. In this case, $H \equiv_d \llbracket Q \rrbracket \uparrow$, where $Q = C[\ddagger (P_1 > P_2)]$. In addition, we have $\mathsf{nf}(\mathsf{lf}(\ddagger (P_1 > P_2))) = \mathsf{nf}(\mathsf{lf}(\ddagger (P_1 > P_2))) = \mathsf{nf}(\mathsf{lf}(\ddagger (P_1 > P_2)))$. According to Lemma 1, $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$.

   $P$ must be of the form $C[\ddagger (vn)P_1]$ for some static context $C[\cdot]$, in order that $R$ can be applied to $\llbracket P \rrbracket \uparrow$. In this case, $H \equiv_d \llbracket Q \rrbracket \uparrow$, where $Q = C[\ddagger (vn)P_1]$. In addition, we have $\mathsf{nf}(\mathsf{lf}(\ddagger (vn)P_1)) = \mathsf{nf}(\mathsf{lf}(\ddagger (vn)P_1)) = \mathsf{nf}(\mathsf{lf}(\ddagger (vn)P_1))$. According to Lemma 1, $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$.

Theorem 7 (Soundness of copy rules) For a process $P$, a DPO rule $R \in \Delta_P$ and a graph $H$ such that $\llbracket P \rrbracket \uparrow \Rightarrow_K H$, there exists a process $Q$ such that $\llbracket Q \rrbracket \uparrow \equiv_d H$ and $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$.

Proof. We prove for each rule $R \in \Delta_P$.

1. $R = (\text{Rep-Step})$.
   $P$ must be a normal process of the form $C[\ddagger ! P_1]$, in order that $R$ can be applied to $\llbracket P \rrbracket \uparrow$. In this case, $H \equiv_d \llbracket Q \rrbracket \uparrow$, where $Q = C[\ddagger ! P_1 \cdot \mathsf{Copy}(l)]$. Since $\mathsf{nf}(\mathsf{lf}(\ddagger ! P_1)) = \ddagger ! P_1 \equiv_c \ddagger ! P_1 = \mathsf{nf}(\mathsf{lf}(\ddagger ! P_1 \cdot \mathsf{Copy}(l))),$ we have $\mathsf{nf}(\mathsf{lf}(P)) \equiv_c \mathsf{nf}(\mathsf{lf}(Q))$, according to Lemma 1.
Soundness and completeness of graph transformation rules

(2) \( R = (\text{Nil-Copy}) \).
P must be of the form \( C[\text{Copy}(l), l : 0] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[0, 0] \) and thus \( \text{If}(Q) = \text{If}(P) \).

(3) \( R = (\text{Abs-Copy}) \) (Case \( R = (\text{Con-Copy}) \) or \( \text{Ret-Copy} \) is similar).
P must be of the form \( C[\text{Copy}(l), l : (F)P_1] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[(PC(l'))\text{Copy}(l), (l': F)l : P_1] \) and thus \( \text{If}(Q) = \text{If}(C[(F)P, (F)P_1]) = \text{If}(P) \).

(4) \( R = (\text{Par-Copy}) \) (Case \( R = (\text{Sum-Copy}) \) is similar).
P must be of the form \( C[\text{Copy}(l), l : (P_1|P_2)] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[\text{Copy}(l)\text{Copy}(l'), l : P_1|l' : P_2] \) and thus \( \text{If}(Q) = \text{If}(C[P_1|P_2, P_1|P_2]) = \text{If}(P) \).

(5) \( R = (\text{Def-Copy}) \) (Case \( R = (\text{Inv-Copy}) \) is similar).
P must be of the form \( C[\text{Copy}(l), l : (s.P_1)] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[\text{Copy}(l)\text{Copy}(l'), l : P_1] \) and thus \( \text{If}(Q) = \text{If}(C[s.P_1, s.P_1]) = \text{If}(P) \).

(6) \( R = (\text{Pip-Copy}) \).
P must be of the form \( C[\text{Copy}(l), l : (P_1 > P_2)] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[\text{Copy}(l) > \text{Copy}(l'), l : P_1 > l' : P_2] \) and thus \( \text{If}(Q) = \text{If}(C[P_1 > P_2, P_1 > P_2]) = \text{If}(P) \).

(7) \( R = (\text{Res-Copy}) \).
P must be of the form \( C[\text{Copy}(l), l : (vn)P_1] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[(v RC(l', n))\text{Copy}(l), (v l': n)l : P_1] \) and thus \( \text{If}(Q) = \text{If}(C[(vn)P_1, (vn)P_1]) = \text{If}(P) \).

(8) \( R = (\text{Rep-Copy}) \).
P must be of the form \( C[\text{Copy}(l), l : !P_1] \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = C[!\text{Copy}(l), !l : P_1] \) and thus \( \text{If}(Q) = \text{If}(C[!P_1, !P_1]) = \text{If}(P) \).

(9) \( R = (\text{PV-PCopy}) \).
P must be of the form \( P(PC(l), l : ?x) \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = P(pv(PC(l, x)), pv(l : x)) \) and thus \( \text{If}(Q) = \text{If}(P(?x, ?x)) = \text{If}(P) \).

(10) \( R = (\text{VV-VCopy}) \).
P must be of the form \( P(V(C(l), l : x) \), in order that \( R \) can be applied to \([P]\)\(^\dagger\). In this case, \( H \equiv_d [Q]\)\(^\dagger\), where \( Q = P(vv(V(C(l)), vv(l : x)) \) 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 \( P(PC(l), l : f(F_1, \ldots, F_k)) \), in order that \( R \) can be applied to \([P]^\top\). In this case, \( H \equiv_d [Q]^\top \), where \( Q = P(f(PC(l_1), \ldots, PC(l_k)), f(l_1 : F_1, \ldots, l_k : L_k)) \) and thus \( \text{If}(Q) = \text{If}(P) = \text{If}(P(f(F_1, \ldots, F_k), f(F_1, \ldots, F_k))) \).

(12) \( R = \text{(VC-Elim-PC)} \).
\( P \) must be of the form \( C([F_1 \text{(pv}(PC(l_1), x))])P_1(x, (F_2 \text{(pv}(l : x)))]P_2(x), \text{or} (F_2 \text{(pv}(l : x)))P_2(x) \), in order that \( R \) can be applied to \([P]^\top\). Without loss of generality, suppose \( x \) is never bound in \( P_1 \) or \( P_2 \). In this case, \( Q \) can always be achieved by alpha-conversions. In this case, \( H \equiv_d [Q]^\top \), where \( Q = C([F_1 \text{(pv}(PC(l_1), x))])P_1(x) \), \( (F_2 \text{(pv}(l : x)))P_2(x) \) and thus \( \text{If}(Q) = \text{If}(C([F_1 \text{(pv}(PC(l_1), x))])P_1(x), (F_2 \text{(pv}(l : x)))P_2(x)) = \text{If}(P) \).

(13) \( R = \text{(PC-Elim)} \).
\( P \) must be of the form \( C([F_1 \text{(pv}(PC(l_1), x))])P_1(x) \), \( (F_2 \text{(pv}(l : x)))]P_2(x) \), in order that \( R \) can be applied to \([P]^\top\), where \( P_2 \) does not contain any value \( v \) \( \in \text{fn}(V) \), or \( l' : V \) with \( x \equiv \text{fn}(V) \) or any sub-process \( l' : P' \) with \( x \equiv \text{fn}(P') \). In this case, \( H \equiv_d [Q]^\top \), where \( Q = C([F_1 \text{(pv}(PC(l_1), x))])P_1(x) \), \( (F_2 \text{(pv}(l : x)))P_2(x) \) and thus \( \text{If}(Q) = \text{If}(P) \).

(14) \( R = \text{(VC-Elim-RC)} \).
\( P \) must be of the form \( C([v RC(l, x)])P_1(x, (v l : s)]P_2(x) \), (or \( s \equiv \text{fn}(V) \)) for some variable \( x \), or \( C([v RC(l, s)])P_1(x, (v l : s)]P_2(x) \), (or \( s \equiv \text{fn}(V) \)) for some service \( s \), in order that \( R \) can be applied to \([P]^\top\). 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, \( H \equiv_d [Q]^\top \), where \( Q = C([v RC(l, x)])P_1(x) \), \( (v l : s)]P_2(x) \) 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, \( H \equiv_d [Q]^\top \), where \( Q = C([v RC(l, s)])P_1(s) \), \( (v l : s)]P_2(s) \) and we also have \( \text{If}(Q) = \text{If}(P) \).

(15) \( R = \text{(RC-Elim)} \).
\( P \) must be of the form \( C([v RC(l, n)])P_1(x, (v l : n)]P_2(n) \), in order that \( R \) can be applied to \([P]^\top\), where \( P_2 \) does not contain any value \( v \) \( \equiv \text{fn}(V) \), or \( l' : V \) with \( n \equiv \text{fn}(V) \) or any sub-process \( l' : P' \) with \( n \equiv \text{fn}(P') \). In this case, \( H \equiv_d [Q]^\top \), where \( Q = C([v RC(l, n)])P_1(n) \), \( (v l : n)]P_2(n) \) and thus \( \text{If}(Q) = \text{If}(P) \).

(16) \( R = \text{(VC-Elim)} \).
\( P \) must be of the form \( C[P_1(vv(VC(l))) \text{ or } P_1(VC(l), l : s)] \), in order that \( R \) can be applied to \([P]^\top\), where \( l \equiv s \) is never bound in \( P_1 \). In the former case, \( H \equiv_d [Q]^\top \), where \( Q = C[P_1(x, x)] \) and thus \( \text{If}(Q) = \text{If}(P) \). In the latter case, \( H \equiv_d [Q]^\top \), where \( Q = C[P_1(s, s)] \) and we also have \( \text{If}(Q) = \text{If}(P) \).

**Theorem 8 (Soundness of rules for congruence)** For a process \( P \), a DPO rule \( R \in \Delta_C \) and a graph \( H \) such that \([P]^\top \Rightarrow_R H \), there exists a process \( Q \) such that \([Q]^\top \equiv_d H \text{ and } \text{nf(If}(P)) \equiv_c \text{nf(If}(Q)) \).
Soundness and completeness of graph transformation rules

Proof. According to Theorem 7, the set of copy rules $\Delta_F$ are sound. So, we only need to prove the soundness of each rule $R \in \Delta_C \setminus \Delta_F$.

1. $R = (\text{Par-Comm})$ (Case $R = (\text{Sum-Comm})$ is similar).
   $P$ must be of the form $C[P_1[P_2]]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[P_1P_2]$. Since $\text{nf}(\text{If}(P_1|P_2)) = \epsilon \text{nf}(\text{If}(P_1|P_1))$, we have $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$ according to Lemma 1.

2. $R = (\text{Par-Assoc})$ (Case $R = (\text{Sum-Assoc})$ is similar).
   $P$ must be of the form $C[(P_1)[P_2]P_3]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[P_1P_2P_3]$. Since $\text{nf}(\text{If}((P_1|P_2)[P_3])) = \epsilon \text{nf}(\text{If}(P_1|(P_2P_3)))$, we have $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$ according to Lemma 1.

3. $R = (\text{Sum-Unit})$.
   $P$ must be of the form $C[M + 0]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[M]$. In addition, $\text{nf}(\text{If}(M + 0)) = \epsilon \text{nf}(\text{If}(M)) + 0 = \epsilon \text{nf}(\text{If}(M))$. According to Lemma 1, $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$.

4. $R = (\text{Nil-toSum})$.
   $P$ must be of the form $C[0]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[0 + 0]$. In addition, $\text{nf}(\text{If}(0 + 0)) = 0 + 0 = \epsilon \text{nf}(\text{If}(0))$. According to Lemma 1, $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$.

5. $R = (\text{Abs-toSum})$ (Case $R = (\text{Con-toSum})$ or (Ret-toSum) is similar).
   $P$ must be of the form $C[(F)P_1]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[(F)P_1 + 0]$. In addition, $\text{nf}(\text{If}((F)P_1)) = \epsilon \text{nf}(\text{If}((F)P_1)) + 0 = \epsilon \text{nf}(\text{If}((F)P_1 + 0))$. According to Lemma 1, $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$.

6. $R = (\text{Res-Unit})$ (Case $R = (\text{Res-Unit-A})$ is similar).
   $P$ must be of the form $C[(vn)0]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[0]$. In addition, $\text{nf}(\text{If}((vn)0)) = (vn)0 = \epsilon \text{nf}(\text{If}(0))$. According to Lemma 1, $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$.

7. $R = (\text{Nil-toRes})$ (Case $R = (\text{Nil-toRes-A})$ is similar).
   $P$ must be of the form $C[0]$, in order that $R$ can be applied to $[P]^\dagger$. In this case, $H \equiv_d [Q]^\dagger$, where $Q = C[(vn)0]$. In addition, $\text{nf}(\text{If}(0)) = 0 = (vn)0 = \epsilon \text{nf}(\text{If}((vn)0))$. According to Lemma 1, $\text{nf}(\text{If}(P)) = \epsilon \text{nf}(\text{If}(Q))$. 
Soundness and completeness of graph transformation rules

(8) \( R = (\text{Res-Comm}) \).

\( P \) must be of the form \( C[(vn)(vn')P_1] \) for some non-static context \( C[\cdot] \), in order that \( R \) can be applied to \([P]^\dagger\). In this case, \( H \equiv_d [Q]^\dagger \), where \( Q = C[(vn')(vn)P_1] \). In addition,

\[
\text{nf}(\text{lf}((vn)(vn')P_1)) = (vn)(vn)\text{nf}(\text{lf}(P_1)) \\
\equiv_c (vn')(vn)\text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}((vn')(vn)P_1)).
\]

According to Lemma[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 \( C[P_1|(vn)P_2] \) for some non-static context \( C[\cdot] \), in order that \( R \) can be applied to \([P]^\dagger\). Without loss of generality, suppose \( n \notin \text{fn}(P_1) \), which can always be achieved by alpha-conversions.

In this case, \( H \equiv_d [Q]^\dagger \), where \( Q = C[(vn)(P_1|P_2)] \). Since \( \text{nf}(\text{lf}((vn)(P_1|P_2))) \equiv_c \text{nf}(\text{lf}((vn)(P_1|P_2))) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \) according to Lemma[1].

(10) \( R = (\text{Nil-Res-Comm}) \).

\( P \) must be of the form \( C[(vn)P_1 > P_2] \) for some non-static context \( C[\cdot] \), in order that \( R \) can be applied to \([P]^\dagger\). Without loss of generality, suppose \( n \notin \text{fn}(P_2) \), which can always be achieved by alpha-conversions.

In this case, \( H \equiv_d [Q]^\dagger \), where \( Q = C[(vn)(P_1 > P_2)] \). Since \( \text{nf}(\text{lf}((vn)(P_1 > P_2))) \equiv_c \text{nf}(\text{lf}((vn)(P_1 > P_2))) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \) according to Lemma[1].

(11) \( R = (\text{Ses-Res-Comm}) \).

\( P \) must be of the form \( C[r\triangleright (vn)P_1] \) for some non-static context \( C[\cdot] \), 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, \( H \equiv_d [Q]^\dagger \), where \( Q = C[(vn)r\triangleright P_1] \). Since \( \text{nf}(\text{lf}(r\triangleright (vn)P_1)) \equiv_c \text{nf}(\text{lf}((vn)(r\triangleright P_1))) \), we have \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \) according to Lemma[1].

**Theorem 9 (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 \( Q \) 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 \( R \in \Delta_G \).

(1) \( R = (\text{Nil-GC}) \).

\( P \) must be of the form \( C[\text{GB}(P_1; GI)] \), where \( GI \) has a sub-item \( S :: \{0, GI_1, \ldots, GI_k\} \), in order that \( R \) can be applied to \([P]^\dagger\). We write \( GI \) as \( GL(S :: \{0, GI_1, \ldots, GI_k\}) \). In this case, \( H \equiv_d [Q]^\dagger \), where \( Q = C[\text{GB}(P_1; GI(S :: \{i,o,t, GI_1, \ldots, GI_k\}))] \). In addition,

\[
\text{nf}(\text{lf}(\text{GB}(P_1; GI(S :: \{0, GI_1, \ldots, GI_k\})))) \\
= \text{nf}(\text{lf}(P_1)) = \text{nf}(\text{lf}(\text{GB}(P_1; GI(S :: \{i,o,t, GI_1, \ldots, GI_k\})))).
\]

According to Lemma[1] \( \text{nf}(\text{lf}(P)) \equiv_c \text{nf}(\text{lf}(Q)) \).
Soundness and completeness of graph transformation rules

58

(2) \( R = \text{(Abs-GC)} \).
\( P \) must be of the form \( C[GB(P_1;GI((F)P_2))] \), in order that \( R \) can be applied to \( [P]^\dagger \). In this case, 
\( H \equiv_d\llbracket Q \rrbracket \), where \( Q = C[GB(P_1;GI(\{F, P_2\}) :: \{F, P_2\})] \). In addition,
\[
\begin{align*}
\operatorname{nf}(\operatorname{lf}(GB(P_1;GI((F)P_2))))
&= \operatorname{nf}(\operatorname{lf}(GB(P_1;GI(\{F, P_2\}) :: \{F, P_2\}))).
\end{align*}
\]
According to Lemma \[1\] \( \operatorname{nf}(\operatorname{lf}(P)) \equiv_c \operatorname{nf}(\operatorname{lf}(Q)) \).

(3) \( R = \text{(Con-GC)} \) (Case \( R = \text{(Ret-GC)} \) is similar).
\( P \) must be of the form \( C[GB(P_1;GI(S :: \{V, P_2, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( [P]^\dagger \).

We choose \( Q = C[GB(P_1;GI(S :: \{V, P_2, GI_1, \ldots, GI_k\}))] \), so that \( \llbracket Q \rrbracket \equiv_d H \). In addition,
\[
\begin{align*}
\operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{V, P_2, GI_1, \ldots, GI_k\}))))
&= \operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{V, P_2, GI_1, \ldots, GI_k\})))).
\end{align*}
\]
According to Lemma \[1\] \( \operatorname{nf}(\operatorname{lf}(P)) \equiv_c \operatorname{nf}(\operatorname{lf}(Q)) \).

(4) \( R = \text{(Par-GC)} \) (Case \( R = \text{(Sum-GC)} \) is similar).
\( P \) must be of the form \( C[GB(P_1;GI(S :: \{P_2, P_3, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( [P]^\dagger \).

We choose \( Q = C[GB(P_1;GI(S :: \{P_2, P_3, GI_1, \ldots, GI_k\}))] \), so that \( \llbracket Q \rrbracket \equiv_d H \). In addition,
\[
\begin{align*}
\operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{P_2, P_3, GI_1, \ldots, GI_k\})))))
&= \operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{P_2, P_3, GI_1, \ldots, GI_k\})))).
\end{align*}
\]
According to Lemma \[1\] \( \operatorname{nf}(\operatorname{lf}(P)) \equiv_c \operatorname{nf}(\operatorname{lf}(Q)) \).

(5) \( R = \text{(Def-GC)} \) (Case \( R = \text{(Inv-GC)} \) is similar).
\( P \) must be of the form \( C[GB(P_1;GI(S :: \{s, P_2, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( [P]^\dagger \).

We choose \( Q = C[GB(P_1;GI(S :: \{s, i, o, t, \{i, o, t\} :: \{P_2\}, GI_1, \ldots, GI_k\}))] \), so that \( \llbracket Q \rrbracket \equiv_d H \). In addition,
\[
\begin{align*}
\operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{s, P_2, GI_1, \ldots, GI_k\}))))
&= \operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{s, i, o, t, \{i, o, t\} :: \{P_2\}, GI_1, \ldots, GI_k\})))).
\end{align*}
\]
According to Lemma \[1\] \( \operatorname{nf}(\operatorname{lf}(P)) \equiv_c \operatorname{nf}(\operatorname{lf}(Q)) \).

(6) \( R = \text{(Pip-GC)} \).
\( P \) must be of the form \( C[GB(P_1;GI(S :: \{P_2 > P_3, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( [P]^\dagger \).

We choose \( Q = C[GB(P_1;GI(S :: \{o, \{o\} :: \{P_2\}, \{i, o, t\} :: \{P_3\}, GI_1, \ldots, GI_k\}))] \), so that \( \llbracket Q \rrbracket \equiv_d H \). In addition,
\[
\begin{align*}
\operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{P_2 > P_3, GI_1, \ldots, GI_k\}))))
&= \operatorname{nf}(\operatorname{lf}(GB(P_1;GI(S :: \{o, \{o\} :: \{P_2\}, \{i, o, t\} :: \{P_3\}, GI_1, \ldots, GI_k\})))).
\end{align*}
\]
According to Lemma \[1\] \( \operatorname{nf}(\operatorname{lf}(P)) \equiv_c \operatorname{nf}(\operatorname{lf}(Q)) \).

(7) \( R = \text{(Res-GC)} \).
\( P \) must be of the form \( C[GB(P_1;GI(\{vn\}P_2))] \), in order that \( R \) can be applied to \( [P]^\dagger \). If \( n \) is a variable name, we choose \( Q = C[GB(P_1;GI(\{n\} :: \{var(n), P_2\}))] \), so that \( \llbracket Q \rrbracket \equiv_d H \) and
According to Lemma 1, \( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(\{n\}) :: \{\text{var}(n), P_2\})))) \)
must be of the form
\( \text{nf}(\text{If}(P_1)) = \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(\{n\}) :: \{\text{var}(n), P_2\})))) \).
Otherwise, \( n \) is a service name. We choose \( Q = C[\text{GB}(P_1; \text{GI}(\{n\}) :: \{n, P_2\})) \], so that \( \lbrack Q \rbrack \overset{d}{=} H \) and \( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(\{n\}) :: \{n, P_2\}))) \).
In either case, we have \( \text{nf}(\text{If}(P_1)) \equiv_c \text{nf}(\text{If}(Q)) \), according to Lemma 1.

(8) \( R \) = (Rep-GC).
\( P \) must be of the form \( C[\text{GB}(P_1; \text{GI}(S :: \{P_2, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( \lbrack P \rbrack \overset{d}{=} \). We choose \( Q = C[\text{GB}(P_1; \text{GI}(S :: \{i, o, t, \{i, o, t\} :: \{P_2, GI_1, \ldots, GI_k\}))] \), so that \( \lbrack Q \rbrack \overset{d}{=} H \). In addition,
\( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(S :: \{P_2, GI_1, \ldots, GI_k\}))) \).
According to Lemma 1, \( \text{nf}(\text{If}(P_1)) \equiv_c \text{nf}(\text{If}(Q)) \).

(9) \( R \) = (Ch-GC).
\( P \) must be of the form \( C[\text{GB}(P_1; \text{GI}(S :: \{FV_1, \ldots, FV_m, GI_1, \ldots, GI_k\}))] \) in order that \( R \) can be applied to \( \lbrack P \rbrack \overset{d}{=} \). We choose \( Q = C[\text{GB}(P_1; \text{GI}(S :: \{FV_1, \ldots, FV_m, GI_1, \ldots, GI_k\}))] \), so that \( \lbrack Q \rbrack \overset{d}{=} H \). In addition,
\( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(S :: \{FV_1, \ldots, FV_m, GI_1, \ldots, GI_k\}))) \).
According to Lemma 1, \( \text{nf}(\text{If}(P_1)) \equiv_c \text{nf}(\text{If}(Q)) \).

(10) \( R \) = (PV-GC) \( R \) = (VV-GC) is similar.
\( P \) must be of the form \( C[\text{GB}(P_1; \text{GI}(\{\text{var}(x)\}))] \), in order that \( R \) can be applied to \( \lbrack P \rbrack \overset{d}{=} \). We choose \( Q = C[\text{GB}(P_1; \text{GI}(\{\text{var}(x)\}))] \), so that \( \lbrack Q \rbrack \overset{d}{=} H \). In addition,
\( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(\{\text{var}(x)\}))) \).
According to Lemma 1, \( \text{nf}(\text{If}(P_1)) \equiv_c \text{nf}(\text{If}(Q)) \).

(11) \( R \) = (D-GC).
\( P \) must be of the form \( C[\text{GB}(P_1; \text{GI}(S :: \{\text{var}(x), GI_1, \ldots, GI_k\}))] \) or \( C[\text{GB}(P_1; \text{GI}(S :: \{\text{var}(x), GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( \lbrack P \rbrack \overset{d}{=} \). In either case, we choose \( Q = C[\text{GB}(P_1; \text{GI}(S :: \{GI_1, \ldots, GI_k\}))] \), so that \( \lbrack Q \rbrack \overset{d}{=} H \). In addition,
\( \text{nf}(\text{If}(\text{GB}(P_1; \text{GI}(S :: \{\text{var}(x), GI_1, \ldots, GI_k\}))) \).
According to Lemma 1, \( \text{nf}(\text{If}(P_1)) \equiv_c \text{nf}(\text{If}(Q)) \).

(12) \( R \) = (Ch-GC).
\( P \) must be of the form \( C[\text{GB}(P_1; \text{GI}(S :: \{\text{ch}, GI_1, \ldots, GI_k\}))] \), in order that \( R \) can be applied to \( \lbrack P \rbrack \overset{d}{=} \). We
choose \( Q = C[GB(P_1; GI(S :: \{ GI_1, \ldots, GI_k \})), \text{so that } [Q]^\dagger \equiv_d H. \) In addition, 
\[ \text{nf}(\text{If}(GB(P_1; GI(S :: \{ ch, GI_1, \ldots, GI_k \})))) = \text{nf}(\text{If}(P_1)) = \text{nf}(\text{If}(GB(P_1; GI(S :: \{ GI_1, \ldots, GI_k \})))) \].

According to Lemma 1, \( \text{nf}(\text{If}(P)) \equiv_c \text{nf}(\text{If}(Q)). \)

**Theorem 10 (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 \( \text{nf}(\text{If}(P)) \equiv_c \text{nf}(\text{If}(Q)). \)

**Proof.** We prove for each rule \( R \in \Delta_D. \)

1. \( R = (\text{PV-Assgn}). \)

\( P \) must be of the form \( C[AS(V_1, \ldots, V_m; V; F_1, \ldots, F_m, ?x)P_1] \) for some context \( C[\cdot] \) and \( m \geq 0 \), in order that \( R \) can be applied to \([P]^\dagger\). Assume the bound names of \( F_1, \ldots, F_m \) do not occur in \( V \). This can always be achieved by alpha-conversions.

If \( x \notin \text{fn}(P_1) \), there are two cases.

- If \( m = 0 \), \( H \equiv_d [Q]^\dagger \), where \( Q = C[GB(P_1; \emptyset :: \{ V \})] \). In this case, we have:
  \[ \text{nf}(\text{If}(AS(V; ?x)P_1)) = \text{nf}(\text{If}(P_1[V/x])) = \text{nf}(\text{If}(P_1)) = \text{nf}(\text{If}(GB(P_1; \emptyset :: \{ V \}))). \]

- Otherwise, \( H \equiv_d [Q]^\dagger \), where \( Q = C[GB(AS(V_1, \ldots, V_m; F_1, \ldots, F_m)P_1; \emptyset :: \{ V \})] \). In this case, we have:
  \[ \text{nf}(\text{If}(AS(V_1, \ldots, V_m; V; F_1, \ldots, F_m, ?x)P_1)) = \text{nf}(\text{If}(P_1[V/x] \sigma)) \]
  \[ = \text{nf}(\text{If}(P_1 \sigma)) = \text{nf}(\text{If}(GB(AS(V_1, \ldots, V_m; F_1, \ldots, F_m)P_1; \emptyset :: \{ V \}))) \]
  where \( \sigma = \text{match}(F_1, \ldots, F_m; V_1, \ldots, V_m) \).

In either case, \( \text{nf}(\text{If}(P)) = \text{nf}(\text{If}(Q)) \), according to Lemma 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 cases.

- If \( m = 0 \), \( H \equiv_d [Q]^\dagger \), where \( Q = C[P_1(L :: vv(V), Sh(L), \ldots, Sh(L))] \). In this case,
  \[ \text{nf}(\text{If}(AS(V; ?x)P_1)) = \text{nf}(\text{If}(P_1[V/x])) \]
  \[ = \text{nf}(\text{If}(P_1(V, V, \ldots, V))) = \text{nf}(\text{If}(P_1(L :: vv(V), Sh(L), \ldots, Sh(L)))). \]

- Otherwise, \( Q = C[AS(V_1, \ldots, V_m; F_1, \ldots, F_m)P_1(L :: vv(V), Sh(L), \ldots, Sh(L))] \), so that \([Q]^\dagger \equiv_d H\).

In this case,
\[ \text{nf}(\text{If}(AS(V_1, \ldots, V_m; V; F_1, \ldots, F_m, ?x)P_1)) = \text{nf}(\text{If}(P_1[V/x] \sigma)) \]
\[ = \text{nf}(\text{If}(P_1(V, V, \ldots, V) \sigma)) \]
\[ = \text{nf}(\text{If}(AS(V_1, \ldots, V_m; F_1, \ldots, F_m)P_1(L :: vv(V), Sh(L), \ldots, Sh(L)))) \]
where \( \sigma = \text{match}(F_1, \ldots, F_m; V_1, \ldots, V_m) \).

In either case, \( \text{nf}(\text{If}(P)) = \text{nf}(\text{If}(Q)) \), according to Lemma 1.
(2) \( R = (VV\text{-Assign}). \)

\( P \) must be of the form \( C[AS(V_1, \ldots, V_m, f(V'_1, \ldots, V'_k); F_1, \ldots, F_m, f(F'_1, \ldots, F'_k)]P_1 \), where \( m \geq 0 \) and \( k \geq 0 \), in order that \( R \) can be applied to \( [P] \). There are two cases.

- If \( m = k = 0 \), \( H \equiv_d [Q] \), where \( Q = C[P_1] \). In this case, we have:
  \[ \text{nf}(\text{lf}(AS(f; f)P_1)) = \text{nf}(\text{lf}(P_1)) \]
- Otherwise, \( H \equiv_d [Q] \), where \( Q = C[AS(V_1, \ldots, V_m, V'_1, \ldots, V'_k; F_1, \ldots, F_m, F'_1, \ldots, F'_k)]P_1 \). In this case, we have:
  \[ \text{nf}(\text{lf}(AS(V_1, \ldots, V_m, f(V'_1, \ldots, V'_k); F_1, \ldots, F_m, f(F'_1, \ldots, F'_k)]P_1)) = \text{nf}(\text{lf}(P_1)) \]

where \( \sigma = \text{match}(F_1, \ldots, F_m, \tilde{F}'_1, \ldots, \tilde{F}'_k; V_1, \ldots, V_m, V'_1, \ldots, V'_k) \).

In either case, \( \text{nf}(\text{lf}(P)) = \text{nf}(\text{lf}(Q)) \), according to Lemma [1]

(3) \( R = (VV\text{-Norm}). \)

\( P \) must contain a value \( vv(V) \), i.e. \( P \) is of the form \( P(vv(V)) \), in order that \( R \) can be applied to \( [P] \). According to \( R, H \equiv_d [Q] \), where \( Q = P(V) \) and thus \( \text{nf}(\text{lf}(P)) = \text{nf}(\text{lf}(Q)) \).

There is only one special case that \( V \) is a shared value \( L : V_0 \) and the value \( vv(L : V_0) \) is also shared as \( L' : vv(L : V_0) \), i.e. \( P \) is of the form \( P(L' : vv(L : V_0)) \). In this case, we choose \( Q = P(L : V_0)[Sh(L)/Sh(L')] \), so that \( [Q] \equiv_d H \) and \( \text{lf}(P) = \text{lf}(Q) \).

(4) \( R = (Ctr\text{-Split}). \)

\( P \) must be of the form \( P(L : vv(f(V_1, \ldots, V_k)), Sh(L)) \), in order that \( R \) can be applied to \( [P] \). We choose \( Q = P(L : vv(f(L_1 : vv(V_1), \ldots, L_k : vv(V_k))), f(Sh(L_1), \ldots, Sh(L_k))) \), where \( L_1, \ldots, L_k \) are fresh, so that \( [Q] \equiv_d H \) and \( \text{lf}(P) = \text{lf}(Q) \).

(5) \( R = (Ctr\text{-Norm}). \)

\( P \) must be of the form \( P(vv(f(V))) \), in order that \( R \) can be applied to \( [P] \). We choose \( Q = P(f(V)) \), so that \( [Q] \equiv_d H \) 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 process to the tagged graph of one it reduces to (up to \( \text{nf}-\text{congruence} \)).

**Theorem 11 (Soundness of rules for reduction)** For a normal process \( P \), a DPO rule \( R \in \Delta_R \) and a graph \( H \) such that \( [P] \Rightarrow_R H \), there exists a process \( Q \) (which may not be well-matched) such that \( [Q] \equiv_d H \). If \( Q \) is well-matched, it is also well-formed and \( P \rightarrow \text{nf}(\text{lf}(Q)) \).

**Proof.** We prove for each rule \( R \in \Delta_R \).
Soundness and completeness of graph transformation rules

1. \( R = (\text{Ser-Sync}) \)

\( P \) must be of the form \( C[\sigma, P_1, \bar{x}, P_2] \) for some static and restriction-balanced context \( C[\cdot, \cdot] \), in order that \( R \) can be applied to \([P]^\uparrow\). In this case, \( H \equiv_d [Q]^\uparrow \), where \( Q = (\nu r)C[\bar{x}, P_1; \theta :: \{s\}, \bar{x} \cdash P_2] \) with \( r \) fresh. As a result, \( P \rightarrow (\nu r)C[r \cdash P_1, r \cdash P_2] = \text{nf}(\text{If}(Q)) \).

(2) \( R = (\text{Ses-Sync}) \)

\( P \) must be of the form \( C[r \cdash C'[\mathcal{V}P_1 + M_1], r \cdash C_2[(F)P_2 + M_2]] \), in order that \( R \) can be applied to \([P]^\uparrow \), where \( C[\cdot, \cdot] \) is static and restriction-balanced, \( C_2[\cdot] \) and \( C'[\cdot] \) are static, session-immune and restriction-immune, and the hole of \( C'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( C'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( C'[Q'] \equiv_c P'[Q'] \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]^\uparrow \), where \( Q = C[r \cdash C'[\mathcal{V}P_1; \theta :: \{M_1\}], r \cdash C_2[\mathcal{V}P_2; \theta :: \{M_2\}] \). If \( Q \) is well-matched, the match \( \sigma = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and

\[
\begin{align*}
P & \equiv_c C[r \cdash (P'[\mathcal{V}P_1 + M_1]), r \cdash C_2[(F)P_2 + M_2]] \\
& \equiv_c C[r \cdash C'[P_1, r \cdash C_2[P_2\sigma]] \\
& = \text{nf}(\text{If}(Q)).
\end{align*}
\]

3. \( R = (\text{Ses-Sync-Ret}) \)

\( P \) must be of the form \( C[r \cdash C'[\mathcal{V}P_1 + M_1], r \cdash C_2[(F)P_2 + M_2]] \), in order that \( R \) can be applied to \([P]^\uparrow \), where \( C[\cdot, \cdot] \) is static and restriction-balanced, \( C_1[\cdot], C_2[\cdot] \) and \( C'[\cdot] \) are static, session-immune and restriction-immune, and the hole of \( C'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( C'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( C'[Q'] \equiv_c P'[Q'] \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]^\uparrow \), where \( Q = C[r \cdash C'[\mathcal{V}P_1; \theta :: \{M_1\}], r \cdash C_2[\mathcal{V}P_2; \theta :: \{M_2\}] \). If \( Q \) is well-matched, the match \( \sigma = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and

\[
\begin{align*}
P & \equiv_c C[r \cdash (P'[\mathcal{V}P_1 + M_1]), r \cdash C_2[(F)P_2 + M_2]] \\
& \equiv_c C[r \cdash C'[\mathcal{V}P_1], r \cdash C_2[P_2\sigma]] \\
& = \text{nf}(\text{If}(Q)).
\end{align*}
\]

4. \( R = (\text{Pip-Sync}) \)

\( P \) must be of the form \( C_0C'[\mathcal{V}P_1 + M_1] > ((F)P_2 + M_2)] \), in order that \( R \) can be applied to \([P]^\uparrow \), where \( C_0[\cdot] \) is static, \( C'[\cdot] \) is static, session-immune and restriction-immune, and the hole of \( C'[\cdot] \) does not occur in the scope of a pipeline. In other words, the hole of \( C'[\cdot] \) can occur only in the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( C'[Q'] \equiv_c P'[Q'] \) for any normal process \( Q' \). In this case, \( H \equiv_d [Q]^\uparrow \), where \( Q = C_0[\mathcal{V}P_1; \theta :: \{M_1\}] > ((F)l : P_2 + M_2)] \). If \( Q \) is well-matched, the match \( \sigma = \text{match}(PC(l'); V) \) exists, so that \( Q \) is well-formed and
Soundness and completeness of graph transformation rules

\[ P \equiv_c C_0((P'[r \triangleright C_1[(V)P_1 + M_1]]) > ((F)P_2 + M_2))] \\
\rightarrow C_0[P_2\sigma((P'[r \triangleright C_1[P_1]]) > ((F)P_2 + M_2))] \\
\equiv_c C_0[(C'[r \triangleright C_1[P_1]]) > ((F)P_2 + M_2))]P_2\sigma] \\
= \text{nf}(\text{lf}(Q)).

(5) \( R = (\text{Pip-Sync-Ret}). \)

\( P \) must be of the form \( C_0[C'[r \triangleright C_1[(V)P_1 + M_1]] > ((F)P_2 + M_2)] \), in order that \( R \) can be applied to \( [P] \), where \( C_0 \cdot \) is static, \( C_1[] \) and \( C'[] \) are static, session-immune and restriction-immune, and the scope of parallel compositions, i.e. there exists a normal process \( P' \) such that \( C'(P') \equiv P'(Q) \) for any normal process \( Q \). In this case, \( H \equiv_d [Q] \), where \( Q = C_0[C'[r \triangleright C_1[GB(P_1; 0 :: [M_1])]]) > ((l' : F)l : P_2 + M_2)]\)AS(V;PC(l')COPY(l')). If \( Q \) is well-formed, the match \( \sigma = \text{match}(PC(l'); V) = \text{match}(F; V) \) exists, so that \( Q \) is well-formed and

\[ P \equiv_c C_0((P'[r \triangleright C_1[(V)P_1 + M_1]]) > ((F)P_2 + M_2))] \\
\rightarrow C_0[P_2\sigma((P'[r \triangleright C_1[P_1]]) > ((F)P_2 + M_2))] \\
\equiv_c C_0[(C'[r \triangleright C_1[P_1]]) > ((F)P_2 + M_2))]P_2\sigma] \\
= \text{nf}(\text{lf}(Q)).

With the soundness of each individual set of rules, it is straightforward to prove the soundness of all the rules as a whole, with respect to congruence and reduction.

5.2.1 Proof of Theorem 2

Proof. \( [P] \vdash_{\Delta_r \cup \Delta_T} [Q] \) means \( [P] \vdash_{R_1, H_1 \Rightarrow R_2, H_2 \Rightarrow \ldots \Rightarrow R_k, H_k \equiv_d [Q] \), for some graphs \( H_1, \ldots, H_k \) and rules \( R_1, \ldots, R_k \in \Delta_r \cup \Delta_T \). According to Theorems 6 and 9, there exist a sequence of processes \( P_1, \ldots, P_k \) such that \( [P_i] \equiv_d H_i \) for \( 1 \leq i \leq k \) and \( P \equiv_c \text{nf}(\text{lf}(P_1)) \equiv_c \ldots \equiv_c \text{nf}(\text{lf}(P_k)) \). Since \( [P_i] \equiv_d H_i \equiv_d [Q] \), \( \text{nf}(\text{lf}(P_k)) \equiv_c Q \). As a result, \( P \equiv_c Q \).

5.2.2 Proof of Theorem 3

Proof. \( [P] \vdash_{\Delta_r} [Q] \) means \( [P] \equiv_d H_0 \Rightarrow R_1, H_1 \Rightarrow R_2 \ldots \Rightarrow R_k, H_k \equiv_d [Q] \), for some graphs \( H_0, H_1, \ldots, H_k \) and rules \( R_1, \ldots, R_k \in \Delta_r \). Suppose \( R_m (1 \leq m \leq k) \) is the only one among these rules that belongs to \( \Delta_r \), i.e. each of the others belongs to \( \Delta_T \cup \Delta_C \cup \Delta_G \cup \Delta_D \). According to Theorems 6, 8, 9, 10 and 11 there exist a sequence of processes \( P_0 = P, P_1, \ldots, P_m, \) where \( P_m \) may not be well-formed, such that \( [P_i] \equiv_d H_i \) for \( 0 \leq i \leq m \) and \( P \equiv_c \text{nf}(\text{lf}(P_1)) \equiv_c \ldots \equiv_c \text{nf}(\text{lf}(P_m-1)) \). 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 \( [P'] \equiv_d H_{m-1} \equiv_d [P_{m-1}] \), thus \( P' \equiv_c \text{nf}(\text{lf}(P_{m-1})) \). Notice that \( [P_m] \equiv_d H_m \Rightarrow_{\Delta_r} [Q] \), \( P_m \) must be well-formed, since no rule in \( \Delta_r \) 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 11 \( P_m \) is well-formed and \( P' \rightarrow \text{nf}(\text{lf}(P_m)) \). Then, according to Theorems 6, 8, 9 and 10 there exist a sequence of processes \( P_{m+1}, \ldots, P_k \) such that \( [P_i] \equiv_d H_i \) for
5.3 Variants of congruence and reduction

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. These variant relations are defined only between normal processes. In this subsection, therefore, a process and a context always mean a normal process and a normal context, respectively.

**Strict congruence.** For two processes $P$ and $Q$, we say $P$ is one-step congruent with $Q$, denoted as $P \equiv_c Q$, if there is a congruence rule $P' \equiv_c Q'$ (see Section 2.1) such that $P = C[P']$ and $Q = C[Q']$, or $P = C[Q']$ and $Q = C[P']$, for some context $C[\cdot]$. As a result, the congruence relation $\equiv_c$ is the reflexive and transitive closure of $\equiv_c^*$. For two processes $P$ and $Q$, we say $P$ is one-step strictly congruent with $Q$, denoted as $P \equiv_c^* Q$, if there is a basic congruence rule $P' \equiv_c Q'$ (see Section 2.1) such that $P = C[P']$ and $Q = C[Q']$, or $P = C[Q']$ and $Q = C[P']$, for some context $C[\cdot]$. 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 $P \triangleright^e Q$, if there is a special congruence rule $P' \equiv_c Q'$ (see Section 2.1) such that $P = C[P']$ and $Q = C[Q']$, or $P = C[Q']$ and $Q = C[P']$, for some context $C[\cdot]$. Furthermore, if the congruence rule is one of the three for moving restrictions forward (Section 2.1), we say $Q$ is a one-step res-forwardness of $P$, denoted as $P \triangleright^c Q$. Otherwise, the congruence rule is the one for unfolding replications (Section 2.3). In this case, we say $Q$ is a one-step unfolding of $P$, denoted as $P \triangleright^u Q$. For two processes $P$ and $Q$, we say $Q$ is a flexible unfolding of $P$, denoted as $P \triangleright_f Q$, if $P = C[\lambda P_1, \ldots, \lambda P_k]$ and $Q = C[P_1, \ldots, P_k]$, for some $k$-hole context $C[\cdot, \ldots, \cdot]$ 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 $\lambda P_1, \ldots, \lambda P_k$, respectively. Notice that the order of these $k$ applications is 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. $P \triangleright^u Q$ implies $P \triangleright_f Q$.

For two processes $P$ and $Q$, we say $Q$ is a one-step generalization of $P$, denoted as $P \Rightarrow^e Q$, if either $P \equiv_s^* Q$ or $P \triangleright^e Q$. As a result, $P \Rightarrow^e Q$ if and only if $P \Rightarrow^c Q$ or $Q \Rightarrow^e P$. Let $\Rightarrow_c$ be the reflexive and transitive closure of $\Rightarrow_c^*$. We say $Q$ is a generalization of $P$ if $P \Rightarrow_c Q$. For two processes $P$ and $Q$, we say $Q$ is a one-step reorganization of $P$, denoted as $P \Rightarrow^r Q$, if either $P \Rightarrow^c Q$ or $P \equiv_s^* Q$. As a result, $P \Rightarrow^r Q$ if and only if $P \Rightarrow^c Q$ or $P \Rightarrow^c Q$. Let $\Rightarrow_c$ be the reflexive and...
transitive closure of \( Q \). We say \( Q \) is a reorganization of \( P \) if \( P \Rightarrow_r Q \).

By applying the congruence rules provided in Section 2.1, we can move the restrictions of a process \( P \) to the front, as much as possible. The result process is unique for \( P \) up to strict congruence \( \equiv_s \). We call it the \emph{res-prefixed form} of \( P \), denoted as \( \text{rp}(P) \). For example, the res-prefixed form of \( (\nu y)(y) > (\exists x)(x)^\dagger \) is \( (\nu y)(y) > (\exists x)(x)^\dagger \). It is worth pointing out that the rep-prefixed from of a process does not change with (one-step) reorganizations, i.e. \( P \Rightarrow_r Q \) implies \( \text{rp}(P) \equiv_s \text{rp}(Q) \). Another fact is that the rep-prefixed from preserves the generalization relation, i.e. \( P \Rightarrow_c Q \) implies \( \text{rp}(P) \Rightarrow_c \text{rp}(Q) \).

**Strict reduction.** For two processes \( P \) and \( Q \), we say \( P \) \emph{strictly reduce} to \( Q \), denoted as \( P \rightarrow_s Q \), if \( P \Rightarrow_r P_0 \rightarrow_p Q_0 \Rightarrow_r Q \) for some processes \( P_0 \) and \( Q_0 \). Similar to the generalization relation, a strict reduction is preserved by the rep-prefixed from of processes, i.e. \( P \rightarrow_s Q \) implies \( \text{rp}(P) \rightarrow_s \text{rp}(Q) \). Let \( \rightarrow_s^+ \) be the reflexive and transitive closure of \( \rightarrow_s \). So, \( P \rightarrow_s^+ Q \) means \( P \) can be transformed to \( Q \) through a sequence of strict reductions.

### 5.4 Proof of completeness

The outline of the proof is shown in Fig. 40. The target is to prove Theorem 4 and Theorem 5, i.e. the completeness of DPO rules with respect to congruence and reduction. For this purpose, we will introduce a few lemmas and propositions. The casual dependency of these lemmas, propositions and theorems is shown by arrows in the figure. To improve the readability of the figure, we manually draw some of the arrows as dotted ones.

![Figure 40: Outline of the completeness proof](image_url)

**Separation of concern.** With the new notions of congruences, we are able to separate Theorem 4 into two propositions.
Proposition 1 For two processes $P$ and $P'$, $P \equiv_c P'$ implies $P \Rightarrow c Q$ and $P' \Rightarrow c Q$ for some process $Q$.

Proposition 2 For two processes $P$ and $Q$, $P \Rightarrow c Q$ implies $\Delta_\ast \Rightarrow Q$.

Similarly, with the new notions of reductions, we are able to separate Theorem 5 into the following propositions.

Proposition 3 For two processes $P$ and $Q$, $P \rightarrow s Q$ implies $P \rightarrow s Q'$ for some processes $P'$ and $Q'$.

Proposition 4 For two processes $P$ and $Q$, $P \rightarrow s Q$ implies $\Delta_\ast \Rightarrow Q$.

We give the proof of these four propositions in the subsequent four subsections, respectively.

5.5 Proof of Proposition 1

In this subsection, we only consider normal processes and normal contexts. To prove Proposition 1 we first introduce a couple of lemmas.

Lemma 2 If $P \triangleright Q$ and $P \triangleright f P'$, then $P' \triangleright Q'$ and $P \triangleright f Q'$ for some process $Q'$ (see Fig. 41).

```
\begin{center}
\begin{tikzpicture}
\node (P) at (0,0) {$P$};
\node (Q) at (1,0) {$Q$};
\node (P') at (0,-1) {$P'$};
\node (Q') at (1,-1) {$Q'$};
\draw[->] (P) -- (Q) node[midway,above]{$\triangleright$};
\draw[->] (P') -- (Q') node[midway,above]{$\triangleright f$};
\end{tikzpicture}
\end{center}
```

Figure 41: Idea of Lemma 2

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. 42).

1) $P_0$ is irrelevant with $P_1, \ldots, P_k$. So, $P = C[P_0 || P_0, \ldots, P_k]$ for some context $C[C, \ldots, C]$. As a result, $Q = C[P_0 || P_0, \ldots, P_k]$ and $P' = C[P_0, P_1 || P_1, \ldots, P_k || P_k]$. In this case, we choose $Q' = C[P_0 || P_0, P_1 || P_1, \ldots, P_k || P_k]$ so that $P' \Rightarrow Q'$ and $Q \triangleright f Q'$.
(2) $!_0$ is included in one of $!_1, \ldots, !_k$. Without loss of generality, suppose it is included in $!_k$. So, $P = C[!_1, \ldots, !_k]$ and $P_k = C[!_0]$ for some contexts $C[\cdot, \ldots, \cdot]$ and $C[\cdot]$. Thus $Q = C[!_1, \ldots, !_k]$ and $P' = C[!_1, \ldots, !_k, P_k]$, where $P_k = C[!_0]$. In this case, we choose $Q' = C[!_1, \ldots, !_k]$, $P_k[!_k]$ so that $P' \Rightarrow^e Q'$ and $Q \triangleright_f Q'$.

(3) Part of $!_1, \ldots, !_k$ is included in $!_0$. Without loss of generality, suppose $!_0$ contains $!_1, \ldots, !_m$ for some $m \leq k$. So, $P = C[!_0, \ldots, !_k]$ and $P_0 = C[!_1, \ldots, !_m]$ for some contexts $C[\cdot, \ldots, \cdot]$ and $C[\cdot]$. As a result, $Q = C[!_0, !_m, \ldots, !_k]$ and $P' = C[!_0, !_m, \ldots, !_k, !_k]$, where $P_0'$ is a shorthand for $C[!_1, !_k, \ldots, !_m]$. Let $Q' = C[!_0, !_m, \ldots, !_k]$. Then $P' \Rightarrow^e Q'$ and $Q \triangleright_f Q'$.

![Figure 42: Cases in the proof of Lemma 2](image)

Lemma 3 If $P \Rightarrow^* Q$ and $P \triangleright_f P'$, then $P' \Rightarrow^e Q'$ and $Q \triangleright_f Q'$ for some $Q'$ (see Fig. 43).

![Figure 43: Idea of Lemma 3](image)

Proof. Suppose $P = C[!_1, \ldots, !_k]$ and $P' = C[!_1, \ldots, !_k]$. Note that $P \Rightarrow^* Q$. There are two cases (see Fig. 44).
(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. $Q = C[P_1',!P_2, \ldots,!P_k]$ and $P_1 \Rightarrow^* P_1'$. In this case, we choose $Q' = C[P_1'|!P_1',!P_2, \ldots,!P_k]$, so that $P' \Rightarrow c Q'$ and $Q \Rightarrow_f Q'$.

(2) None of $P_1, \ldots, P_k$ is changed when $P$ transforms into $Q$. Note that each replication is simply preserved by any congruence rule (except the one $!P \equiv P$). There must be a $k$-hole context $C_1[\ldots,!,\ldots]$ such that $Q = C_1[!P_1,\ldots,!P_k]$ and that for any processes $X_1, \ldots, X_k$, $C[X_1,\ldots,X_k] \Rightarrow^* C_1[X_1,\ldots,X_k]$. Let $Q'$ be $C_1[P_1'|!P_1,\ldots,P_k]$ then we have $P' \Rightarrow c Q'$ and $Q \Rightarrow_f Q'$.

A direct deduction of Lemma 3 and Lemma 4 is as follows. Recall that each step of a generalization ($\Rightarrow^*$) is either an unfolding ($\Rightarrow^\triangledown$) or a reorganization ($\Rightarrow^\triangledown$).

**Lemma 4** If $P \Rightarrow_c Q$ and $P \Rightarrow_f P'$, then $P' \Rightarrow_c Q'$ and $Q \Rightarrow_f Q'$ for some $Q'$ (see Fig. 45).

![Figure 44: Cases in the proof of Lemma 3](image)

![Figure 45: Idea of Lemma 4](image)

Now, we can prove the following.

**Lemma 5** If $P \Rightarrow_c Q$ and $P \Rightarrow^* P'$, then $P' \Rightarrow_c Q'$ and $Q \Rightarrow c Q'$ for some $Q'$ (see Fig. 46).

**Proof.** There are two cases for $P \Rightarrow^* P'$ (see Fig. 47).

(1) $P \Rightarrow^\triangledown P'$. It is a special case of $P \Rightarrow_f P'$. According to Lemma 4 there exists a process $Q'$ such that $P' \Rightarrow c Q'$ and $Q \Rightarrow_f Q'$. Note that $Q \Rightarrow_f Q'$ implies $Q \Rightarrow c Q'$.

(2) $P \Rightarrow^* P'$, which implies $rp(P) \equiv_s rp(P')$. In this case, we choose $Q' = rp(Q)$, so that $Q \Rightarrow c Q'$. Also note that $P \Rightarrow c Q$ implies $rp(P) \Rightarrow c rp(Q)$. We have $P' \Rightarrow c rp(P') \equiv_s rp(P) \Rightarrow c rp(Q) = Q'$. 

Report No. 445, November 2010

UNU-IIST, P.O. Box 3058, Macao
Soundness and completeness of graph transformation rules

Figure 46: Idea of Lemma 5

Figure 47: Cases in the proof of Lemma 5

With Lemma 5, Proposition 1 can be proved by induction on the number \( k \) of one-step congruences from \( P \) to \( Q \), i.e. \( P = P_0 \equiv^* P_1 \equiv^* \ldots \equiv^* P_k = Q \).

5.6 Proof of Proposition 2

In order to prove Proposition 2, we only need to prove the following proposition, as a generalization is composed of a sequence of one-step generalizations.

**Proposition 5** \( P \xrightarrow{c} Q \) implies \( [P] \uparrow \Rightarrow_{\delta} [Q] \uparrow \).

We first show that derivations of graphs are preserved by process contexts.

**Lemma 6** Let \( \delta \) be a set of DPO rules, \( P, Q \) be two processes such that \( \|P\| \Rightarrow_{\delta} \|Q\| \) and \( \|P\| \uparrow \Rightarrow_{\delta} \|Q\| \uparrow \). Then, for any context \( C[\cdot] \), \( \|C[P]\| \Rightarrow_{\delta} \|C[Q]\| \) and \( \|C[P]\| \uparrow \Rightarrow_{\delta} \|C[Q]\| \uparrow \).

**Proof.** For any context \( C[\cdot] \) and process \( X \), \( \|C[X]\| \) is constructed based on \( \|X\| \), i.e. \( \|C[X]\| \) is of the form \( G(\|X\|) \). As a result, \( \|C[P]\| \equiv_d G(\|P\|) \Rightarrow_{\delta} \|C[Q]\| \equiv_d \|C[Q]\| \).

If \( C[\cdot] \) is a static context, \( \|C[X]\| \uparrow \) is constructed based on \( \|X\| \uparrow \) for any process \( X \), i.e. \( \|C[X]\| \uparrow \) is of the form \( G'(\|X\|) \). In this case, \( \|C[P]\| \uparrow \equiv_d G'(\|P\|) \Rightarrow_{\delta} G'(\|Q\|) \equiv_d \|C[Q]\| \). If \( C[\cdot] \) is non-static, \( \|C[X]\| \uparrow \) is constructed based on the untagged graph \( \|X\| \) for any process \( X \), i.e. \( \|C[X]\| \uparrow \) is of the form \( G''(\|X\|) \). In this case, \( \|C[P]\| \uparrow \equiv_d G''(\|P\|) \Rightarrow_{\delta} G''(\|Q\|) \equiv_d \|C[Q]\| \).

To prove Proposition 5, we also need to study the completeness of copy rules.
5.6.1 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 7 For a normal pattern $F(x_1, \ldots, x_k)$, $[P(I : F(x_1, \ldots, x_k))|Q(PC(I))] \Rightarrow_{\Delta_P}^* [P(F(pv(l_1 : x_1), \ldots, pv(l_k : x_k)))|Q(F(pv(PC(l_1, x_1)), \ldots, pv(PC(l_k, x_k))))]$, where $x_1, \ldots, x_k$ are all the pattern variables of $F$.

Proof. By induction on the structure of $F = F(x_1, \ldots, x_k)$.

1. $F = ?x$.
   
   $[P(I : ?x)|Q(PC(I))] \Rightarrow_{\Delta_P}^* [P(pv(I : ?x))]|Q(pv(PC(I,x)))].$

2. $F = f(F_1, \ldots, F_m)$.
   For $1 \leq i \leq m$, let $F_i$ be of the form $F_i(x_1^i, \ldots, x_{k_i}^i)$, where $x_1^i, \ldots, x_{k_i}^i$ are all its pattern variables. So, $x_1^1, \ldots, x_{k_1}^1, \ldots, x_1^m, \ldots, x_{k_m}^m$ are all the pattern variables of $F$.

   $[P(I : f(F_1, \ldots, F_m))|Q(PC(I))]$

   $\Rightarrow_{\text{Copy}} (IH) \Rightarrow_{\Delta_P}^* [P(f(F_1(pv(l_1^1, x_1^1), \ldots, pv(l_{k_1}^1, x_{k_1}^1)), \ldots, F_m(pv(l_1^m, x_1^m), \ldots, pv(l_{k_m}^m, x_{k_m}^m))))|Q(f(F_1(pv(PC(l_1^1, x_1^1), \ldots, pv(PC(l_{k_1}^1, x_{k_1}^1))), \ldots, F_m(pv(PC(l_1^m, x_1^m), \ldots, pv(PC(l_{k_m}^m, x_{k_m}^m)))))]$

$\Rightarrow_{\Delta_P}^* [P(F(pv(l_1^1, x_1^1), \ldots, pv(l_{k_m}^m, x_{k_m}^m)))|Q(F(pv(PC(l_1^1, x_1^1), \ldots, pv(PC(l_{k_m}^m, x_{k_m}^m))))]$

Lemma 8 For a normal value $V(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are all the occurrences of its variables, $[P(I : V(x_1, \ldots, x_k))|Q(PC(I))] \Rightarrow_{\Delta_P}^* [P(V(pv(l_1 : x_1), \ldots, pv(l_k : x_k)))|Q(V(pv(VC(l_1)), \ldots, pv(VC(l_k))))].$

Proof. By induction on the structure of the value $V$, similar to Lemma 7.

Lemma 9 Let $P(x_1, \ldots, x_k, s_1, \ldots, s_t)$ be a normal process without sessions, where $x_1, \ldots, x_k$ and $s_1, \ldots, s_t$ are all the occurrences of its free variables and free service names, respectively.

$[C_1 | l : P(x_1, \ldots, x_k, s_1, \ldots, s_t)]|C_2 [\text{Copy}(l)] = \Delta_P^* [C_1 | P(vv(l_1 : x_1), \ldots, vv(l_k : x_k), l_1^1 : s_1, \ldots, l_t^1 : s_t)]|C_2 [\text{P(vv}(VC(l_1)), \ldots, vv(VS(l_k)), VC(l_1^1), \ldots, VC(l_t^1)))]$ for any contexts $C_1[\cdot]$ and $C_2[\cdot]$.

Proof. By induction on the structure of $P = P(x_1, \ldots, x_k, s_1, \ldots, s_t)$. 

Report No. 445, November 2010

UNU-IIST, P. O. Box 3058, Macao
(1) \( P = 0 \).
\[
[C_1[l : 0]|C_2[\text{Copy}(l)]] \Rightarrow (\text{Nil-Copy}) [C_1[0]|C_2[0]]
\]

(2) \( P = (F)Q \).
Let \( F \) be of the form \( F(\varphi_1, \ldots, \varphi_m) \), where \( \varphi_1, \ldots, \varphi_m \) are all its pattern variables. Let \( Q \) be of the form \( Q(x_1, \ldots, x_k, x'_1, \ldots, x'_l, s_1, \ldots, s_t) \), where \( s_1, \ldots, s_t \) are all the occurrences of its free service names and \( x_1, \ldots, x'_l \) are the all the occurrences of its free variables, of which \( x'_1, \ldots, x'_l \) are bounded by \( F \). So, \( x_1, \ldots, x_k \) are all the occurrences of free variables of \( P \).
\[
[C_1[l : (F)Q]|C_2[\text{Copy}(l)]] \Rightarrow_{(\text{Abs-Copy})} [C_1[(l' : F)l : Q]|C_2[(PC(l'))\text{Copy}(l)]]
\]
\[
(C_{\text{IH}} \Rightarrow \lambda_p) \Rightarrow_{(\text{VC-Elim-PC})} [C_1[(F(pv(\varphi_1'), \ldots, pv(\varphi_m')))(vv(VC(l_1)) \ldots vv(VC(l'_1)), VC(l'_1), \ldots, VC(l'_2))]]
\]
\[
(C_{\text{IH}} \Rightarrow \lambda_p) \Rightarrow_{(\text{PC-Elim})} [C_1[(F)(vv(VC(l_1)) \ldots vv(VC(l'_1)), VC(l'_1), \ldots, VC(l'_2))]]
\]
\[
(C_{\text{IH}} \Rightarrow \lambda_p) \Rightarrow_{d} [C_1[P(vv(VC(l_1)) \ldots vv(VC(l'_1)), VC(l'_1), \ldots, VC(l'_2))]]
\]

(3) \( P = (V)Q \). (Case \( P = (V)^\dagger Q \) is similar.)
Let \( V \) be of the form \( V(x_1, \ldots, x_m) \), where \( x_1, \ldots, x_m \) are all the occurrences of its value variables. Let \( Q \) be of the form \( Q(y_1, \ldots, y_k, s_1, \ldots, s_t) \), where \( y_1, \ldots, y_k \) and \( s_1, \ldots, s_t \) are all the occurrences of its free variables and free service names, respectively. So, \( x_1, \ldots, x_m, y_1, \ldots, y_k \) are all the occurrences of free variables of \( P \).
\[
[C_1[l : (V)Q]|C_2[\text{Copy}(l)]] \Rightarrow (\text{Con-Copy}) [C_1[(l' : V)l : Q]|C_2[(VC(l'))\text{Copy}(l)]]
\]
\[
(C_{\text{IH}} \Rightarrow \lambda_p) \Rightarrow_{(\text{VC-Elim})} [C_1[(F)(vv(VC(l_1)) \ldots vv(VC(l'_1)), VC(l'_1), \ldots, VC(l'_2))]]
\]
\[
(C_{\text{IH}} \Rightarrow \lambda_p) \Rightarrow_{d} [C_1[P(vv(VC(l_1)) \ldots vv(VC(l'_1)), VC(l'_1), \ldots, VC(l'_2))]]
\]

(4) \( P = Q'Q \). (Case \( P = Q + Q' \) or \( P = Q > Q' \) is similar.)
Let \( Q \) be of the form \( Q(x_1, \ldots, x_k, s_1, \ldots, s_t) \), where \( x_1, \ldots, x_k \) and \( s_1, \ldots, s_t \) are all the occurrences of its free variables and free service names, respectively. Similarly, let \( Q' \) be of the form \( Q'(x'_1, \ldots, x'_{l'}, s'_1, \ldots, s'_{l'}) \). So, \( x_1, \ldots, x_k, x'_1, \ldots, x'_{l'} \) and \( s_1, \ldots, s_t, s'_1, \ldots, s'_{l'} \) are all the occurrences of free variables and free service names of \( P \), respectively.
Soundness and completeness of graph transformation rules

\[ [C_1[l: (Q')]]]_{C_2[Copy(l)]]}

⇒ (Par–Copy)
\[ [C_1[l_1: Q[l_2: Q']]]_{C_2[Copy(l_1)][Copy(l_2)]]}

(IH) ⇒ \( A P \)
\[ [C_1(Q[vv(l_1_1: x_1),...,vv(l_1_k: x_k), l_1_1': s_1,\ldots,l_1_k': s_k] | Q'[vv(l_2_1: x_2'),...,vv(l_2_k': x_2'), l_2_1': s_2',\ldots,l_2_k': s_2'] | C_2[Q[VC(l_1_1),...,vv(VC(l_1_k)), VC(l_1_1'),\ldots,VC(l_1_k')]] | Q'[vv(VC(l_2_1),...,vv(VC(l_2_k)), VC(l_2_1'),\ldots,VC(l_2_k')]]])

(5) \( P = s . Q \). (Case \( P = \pi . Q \) is similar.)
Let \( Q \) be of the form \( Q(x_1,\ldots,x_k,s_1,\ldots,s_t) \), where \( x_1,\ldots,x_k \) and \( s_1,\ldots,s_t \) are all the occurrences of its free variables and free service names, respectively. So, \( s,s_1,\ldots,s_t \) are all the occurrences of free service names of \( P \).

⇒ (Def–Copy)
\[ [C_1[l: (s.Q)]][C_2[Copy(l)]]

(IH) ⇒ \( A P \)
\[ [C_1([l': s).(l: Q)]][C_2[VC(l').Copy(l)]]

\[ [C_1([l': s].Q[vv(l_1_1: x_1),...,vv(l_1_k: x_k), l_1_1': s_1,\ldots,l_1_k': s_k]) | C_2[VC(l').Q[vv(VC(l_1_1)),...,vv(VC(l_1_k)), VC(l_1_1'),\ldots,VC(l_1_k')]]]

(6) \( P = (v\pi).Q \).
There are two cases: \( n \) is a variable or a service name. If \( n \) is a variable \( y, P = (v\pi)Q \). Let \( Q \) be of the form \( Q(x_1,\ldots,x_k,y,\ldots,y,s_1,\ldots,s_t) \) (\( y \) may occur more than once in \( Q \)), where \( x_1,\ldots,x_k,y,\ldots,y \) and \( s_1,\ldots,s_t \) 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 \( P \).

⇒ (Rec–Copy)
\[ [C_1[(v l': y).l: Q][C_2[(v RC(l', y)).Copy(l)]]]

(IH) ⇒ \( A P \)
\[ [C_1[(v l': y).Q[vv(l_1_1: x_1),...,vv(l_1_k: x_k), l_1_1': y,\ldots,vv(l_1_m: y), l_1_1': s_1,\ldots,l_1_k': s_k]) | C_2[(v RC(l', y)).Q[vv(VC(l_1)),...,vv(VC(l_k)), vv(VC(l_1)'),\ldots,vv(VC(l_k')), VC(l_1'),\ldots,VC(l_k')]]]

⇒ (VC–Elim–RC)
\[ [C_1[(v l': y).Q[vv(l_1_1: x_1),...,vv(l_1_k: x_k), y,\ldots,y, l_1_1': s_1,\ldots,l_1_k': s_k])]
\[ [C_2[(v RC(l', y)).Q[vv(VC(l_1)),...,vv(VC(l_k)), y,\ldots,y, VC(l_1'),\ldots,VC(l_k')]]]
\[ \equiv,\ A_d \]
\[ [C_1[P[vv(l_1_1: x_1),...,vv(l_1_k: x_k), l_1_1': s_1,\ldots,l_1_k': s_k]]]
\[ [C_2[P[vv(VC(l_1)),...,vv(VC(l_k)), VC(l_1'),\ldots,VC(l_k')]]]

Otherwise, \( n \) is a service name \( s \). Let \( Q \) be of the form \( Q(x_1,\ldots,x_k,s,\ldots,s,s_1,\ldots,s_t) \) (\( s \) may occur more than once in \( Q \)), where \( x_1,\ldots,x_k \) and \( s,s,s_1,\ldots,s_t \) are all the occurrences of its free variables and free service names, respectively. So, \( s_1,\ldots,s_t \) are all the occurrences of free service names of \( P \).
free variables and free service names, respectively.

\[ P = !Q. \]

Let \( Q \) be of the form \( Q(x_1, \ldots, x_k, s_1, \ldots, s_t) \), where \( x_1, \ldots, x_k \) and \( s_1, \ldots, s_t \) are all the occurrences of its free variables and free service names, respectively.

\[ C_1 \equiv \Delta_s \]

\[ \Rightarrow \text{(Rep−Copy)} \]

\[ C_1 !Q | C_2 \text{Copy}(l) \]

\[ (\text{IH}) \Rightarrow \Delta_s \]

\[ C_1 [Q(vv(l_1 : x_1), \ldots, vv(l_k : x_k), l_1 : s_1, \ldots, l_t : s_t)] \]

\[ C_2 [Q(vv(l_1 : x_1), l_1 : s_1, \ldots, l_t : s_t)] \]

Then, we can draw the conclusion that the set of copy rules are complete.

**Theorem 12 (Completeness of copy rules)** For any normal process \( !P \), \([!P] \Rightarrow \Delta_s [!P]P\).

**Proof.** Let \( P \) be of the form \( P(x_1, \ldots, x_k, s_1, \ldots, s_t) \), where \( x_1, \ldots, x_k \) and \( s_1, \ldots, s_t \) are all the occurrences of its free variables and free service names, respectively.

\[ \Rightarrow \text{(Rep−Step)} \]

\[ !P \text{Copy}(l) \]

\[ \text{(Lemma 9)} \Rightarrow \Delta_s \]

\[ !P(vv(l_1 : x_1), \ldots, vv(l_k : x_k), l_1 : s_1, \ldots, l_t : s_t) \]

\[ P(vv(l_1 : x_1), l_1 : s_1, \ldots, l_t : s_t) \]

\[ \Rightarrow \text{(VC−Elim)} \]

\[ !P(x_1, \ldots, x_k, s_1, \ldots, s_t) | P(x_1, \ldots, x_k, s_1, \ldots, s_t) \]

\[ \equiv_d \]

\[ !P \]

**5.6.2 Proof of Proposition 5**

With the completeness of copy rules, we are able to prove that each congruence rule can be simulated by graph transformation rules.

**Lemma 10** For each basic congruence rule \( LS \equiv_{c} RS \), \([LS] \Rightarrow \Delta_s [RS] \Rightarrow \Delta_s [LS] \) and \([LS] \Rightarrow \Delta_s [RS] \Rightarrow \Delta_s [LS] \).
Soundness and completeness of graph transformation rules

Proof. Straightforward for each rule.

(1) \( LS = P|P', RS = P'|P. \)
\[
\text{[LS]} \Rightarrow (\text{Par–Comm}) [RS] \Rightarrow (\text{Par–Comm}) [LS]. [LS]^+ \Rightarrow (\text{Par–Comm}) [RS]^+ \Rightarrow (\text{Par–Comm}) [LS]^+.
\]

(2) \( LS = (P|P')|P'', RS = P|(P'|P''). \)
\[
\text{[LS]} \Rightarrow (\text{Par–Assoc}) [RS]
\Rightarrow (\text{Par–Assoc}) [P'(P''|P)] \Rightarrow (\text{Par–Assoc}) [P''|P(P')] \\
\Rightarrow (\text{Par–Assoc}) [LS].
\]
In the same way, \([LS]^+ \Rightarrow (\text{Par–Assoc}) [RS]^+ \Rightarrow (\text{Par–Assoc}) [LS]^+.\)

(3) \( LS = M + M', RS = M' + M. \)
\[
\text{[LS]} \Rightarrow (\text{Sum–Comm}) [RS] \Rightarrow (\text{Sum–Comm}) [LS]. \text{ And } [LS]^+ \Rightarrow (\text{Sum–Comm}) [RS]^+ \Rightarrow (\text{Sum–Comm}) [LS]^+.
\]

(4) \( LS = (M + M') + M'', RS = M + (M' + M''). \)
\[
\text{[LS]} \Rightarrow (\text{Sum–Assoc}) [RS]
\Rightarrow (\text{Sum–Assoc}) [M' + (M'' + M)] \Rightarrow (\text{Sum–Assoc}) [M'' + (M' + M)] \\
\Rightarrow (\text{Sum–Assoc}) [LS].
\]
In the same way, \([LS]^+ \Rightarrow (\text{Sum–Assoc}) [RS]^+ \Rightarrow (\text{Sum–Assoc}) [LS]^+.\)

(5) \( LS = M + 0, RS = M. \)
\[
\text{[LS]} \Rightarrow (\text{Sum–Unit}) [RS]. [LS]^+ \Rightarrow (\text{Sum–Unit}) [RS]^+. \text{ In order to prove } [RS] \Rightarrow _\Delta^c [LS] \text{ and } [RS]^+ \Rightarrow _\Delta^c [LS]^+, \text{ we make induction on the structure of } M.
\]

- Case \( M = 0. \)
\[
\]
- Case \( M = (F)P \) (Case \( M = (V)P \text{ or } (V)^+P \text{ is similar).} \)
\[
[RS] \Rightarrow (\text{Abs–toSum}) [LS]. [RS]^+ \Rightarrow (\text{Abs–toSum}) [LS]^+.
\]
- Case \( M = M_1 + M_2. \)
\[
\text{IH, Lemma} \Rightarrow _\Delta^c [M_1 + (M_2 + 0)] \\
\text{(conclusion of (4))} \Rightarrow _\Delta^c [(M_1 + M_2) + 0] \\
\Rightarrow (\text{Sum–Unit}) [LS].
\]
In the same way, \([RS]^+ \Rightarrow _\Delta^c [LS]^+.\)
(6) \( LS = (vn)(vn')P, \) \( RS = (vn')(vn)P. \)
\[
\]

(7) \( LS = (vn)0, \) \( RS = 0. \)
\[
\]

**Lemma 11** 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|\( vn \)Q, \) \( RS = (vn)(P|Q). \)
\[
[LS] \Rightarrow_{(Par-Res-Comm)} [RS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger.
\]

2. \( LS = (vn)Q > P, \) \( RS = (vn)(Q > P). \)
\[
[LS] \Rightarrow_{(Pip-Res-Comm)} [RS]. \quad [LS]^\dagger \equiv_d [RS]^\dagger.
\]

3. \( LS = r \triangleright (vn)P, \) \( RS = (vn)(r \triangleright P). \)
\[
\]

4. \( LS = !P, \) \( RS = P|!P. \)
According to Theorem [12] \( [!P] \Rightarrow_{\Delta_p} [!P|P]. \) As a result, \( [LS] \Rightarrow_{\Delta_c} [!P|P] \Rightarrow_{(Par-Comm)} [RS]. \) And
\[
[LS]^\dagger \equiv_d \left( F_{(p,i,o,t)}(A(p))[LS]_{(p,i,o,t)} \right)^\dagger \Rightarrow_{\Delta_c} \left( F_{(p,i,o,t)}(A(p))[RS]_{(p,i,o,t)} \right)^\dagger.
\]
(\text{Theorem[11]} \Rightarrow_{\Delta_T} [RS]^\dagger.

Now, we are ready to prove Proposition[5] \( P \Rightarrow^* Q \) means \( P \equiv^*_c Q \) or \( P \triangleright^* Q. \)

- If \( P \equiv^*_c Q, \) there exist a basic congruence rule \( LS \equiv_c RS \) such that \( P = C[LS], Q = C[RS] \) or \( P = C[RS], Q = C[LS] \) for some context \( C[\cdot \cdot]. \) According to Lemma[10] we have \( [LS] \Rightarrow_{\Delta_c} [RS] \Rightarrow_{\Delta_c} [LS]^\dagger \Rightarrow_{\Delta_c} [RS]^\dagger \Rightarrow_{\Delta_c} [LS]^\dagger. \) Then, according to Lemma[6] we have \( [P]^\dagger \Rightarrow_{\Delta_c} [Q]^\dagger \) (and also \( [Q]^\dagger \Rightarrow_{\Delta_c} [P]^\dagger \)).

- If \( P \triangleright^* Q, \) there exist a special congruence rule \( LS \equiv_c RS \) such that \( P = C[LS] \) and \( Q = C[RS] \) for some context \( C[\cdot \cdot]. \) According to Lemma[11] we have \( [LS] \Rightarrow_{\Delta_c} [RS] \) and \( [LS]^\dagger \Rightarrow_{\Delta_c \cup \Delta_T} [RS]^\dagger. \) Then, according to Lemma[6] we have \( [P] \Rightarrow_{\Delta_c \cup \Delta_T} [Q]^\dagger. \)
5.7 Proof of Proposition\textsuperscript{3}

In this subsection, we only consider normal processes and normal contexts. In order to prove Proposition\textsuperscript{3} we need the following lemma.

**Lemma 12** If $P \xrightarrow{\epsilon} Q$ and $P \xrightarrow{c} P'$, then $P' \xrightarrow{\epsilon} Q'$ and $Q \xrightarrow{c} Q'$ for some $Q'$ (see Fig. 48).

\[ P \xrightarrow{\epsilon} Q \]
\[ \xrightarrow{} \]
\[ P' \xrightarrow{c} Q' \]

Figure 48: Idea of Lemma 12

**Proof.** There are two cases for $P \xrightarrow{c} P'$ (see Fig. 49).

1. \( P \xrightarrow{c} P' \). Suppose $P = C[P_0]!P_0$ and $P' = C[P_0]!P_0$ for some context $C$ and process $P_0$. Note that $!P_0$ cannot take part in the strict reduction $P \xrightarrow{s} Q$. (strict reduction = pure reduction + reorganization) It will be either preserved or simply deleted by the reduction.

   1.1 If it is preserved, then there exists a context $C_1$ such that $Q = C_1[P_0]$, and $C[X] \xrightarrow{\epsilon} C_1[X]$ for any process $X$. In this case, we can choose $Q' = C_1[P_0]!P_0$, so that $P' \xrightarrow{s} Q'$ and $Q \xrightarrow{c} Q'$.

   1.2 If $!P_0$ is deleted by the reduction, then for any process $X$, $C[X] \xrightarrow{\epsilon} Q$. In this case, we choose $Q' = Q$ so that $P' \xrightarrow{s} Q'$ and $Q \xrightarrow{c} Q'$.

2. $P \xrightarrow{c} P'$, which implies $\text{rp}(P) \equiv \text{rp}(P')$. In this case, we choose $Q' = \text{rp}(Q)$, so that $Q \xrightarrow{c} Q'$. Also note that $P \xrightarrow{s} Q$ implies $\text{rp}(P) \xrightarrow{s} \text{rp}(Q)$. We have $P' \equiv \text{rp}(P') \equiv \text{rp}(P) \xrightarrow{s} \text{rp}(Q) = Q'$.

\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]
\[ \text{rp} \]
\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]

(1.1)

\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]
\[ \text{rp} \]
\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]

(1.2)

\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]
\[ \text{rp} \]
\[ C[P_0]!P_0 \xrightarrow{c} C[P_0]!P_0 \]

(2)

Figure 49: Cases in the proof of Lemma 12

As a direct deduction, we have:

**Lemma 13** If $P \xrightarrow{s} Q$ and $P \xrightarrow{c} P'$, then $P' \xrightarrow{s} Q'$ and $Q \xrightarrow{c} Q'$ for some $Q'$ (see Fig. 50).
With Lemma 13 Proposition 3 can be proved. The idea of the proof is shown in Fig. 51.

Proof. $P \rightarrow Q$ means $P \equiv P_0 \rightarrow \_ \equiv Q$ for some $P_0$ and $Q_0$. According to Proposition 1 there exists a process $P'$ such that $P \Rightarrow_c P'$ and $P_0 \Rightarrow_c P'$. Then, according to Lemma 13 there exists a process $Q'$ such that $P' \rightarrow \_ Q'$ and $Q_0 \Rightarrow_c Q'$. From $Q_0 \Rightarrow_c Q'$, we know that $Q' \equiv Q_0 \equiv Q$.

5.8 Proof of Proposition 4

In order to prove Proposition 4 we only need to prove:

**Proposition 6** $P \rightarrow p Q$ implies $\llbracket P \rrbracket \uparrow \Rightarrow_{\Delta} \llbracket Q \rrbracket \uparrow$.

For this purpose, we study the completeness of garbage collection rules and data assignment rules.

5.8.1 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 by applications of these rules. For this purpose, we propose
a notion of size for garbage. We are going to show that after the application of a garbage collection rule, the total size of garbage decreases strictly, 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.

\[
sz(GI) \begin{cases} 
1 & \text{if } GI = s, ch \text{ or } var(x) \\
2 & \text{if } GI = \bar{x} \text{ or } x \\
sz(F_1) + \ldots + sz(F_k) + 1 & \text{if } GI = f(F_1, \ldots, F_k) \\
sz(V_1) + \ldots + sz(V_k) + 1 & \text{if } GI = f(V_1, \ldots, V_k) \\
4 & \text{if } GI = 0 \\
sz(F) + sz(P) + 1 & \text{if } GI = (F)P \\
sz(V) + sz(P) + 1 & \text{if } GI = \langle V \rangle P \text{ or } \langle V \rangle^\uparrow P \\
sz(P_1) + sz(P_2) + 1 & \text{if } GI = P_1|P_2 \text{ or } P_1 + P_2 \\
sz(P) + 5 & \text{if } GI = s.P \text{ or } \bar{s}.P \\
sz(P_1) + sz(P_2) + 2 & \text{if } GI = P_1 > P_2 \\
sz(P) + 2 & \text{if } GI = (\nu n)P \\
sz(P) + 4 & \text{if } GI = !P \\
0 & \text{if } GI = S :: \emptyset \\
sz(GI_1) + \ldots + sz(GI_k) & \text{if } GI = S :: \{GI_1, \ldots, GI_k\} 
\end{cases}
\]

With this definition, we can show that a garbage item can always be removed.

**Lemma 14** For any process $P$ and composite garbage item $GI$, $[GB(P;GI)] \Rightarrow^{\Delta_G} [P]$.

**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 $GI(S :: \{GI_0, GI_1, \ldots, GI_m\})$. We consider different cases of $GI_0$, and prove in either case $[GB(P;GI)] \Rightarrow^{\Delta_G} [P]$.

1. $GI_0 = s$ (Case $GI_0 = ch$ or $var(x)$ is similar).
   If $s$ is a common node of $[GB]$ and $[P] \langle p, i, o, t \rangle$, $[GB(P;GI)] \equiv_d [GB(P;GI')]$, where $GI' = GI(S :: \{GI_1, \ldots, GI_m\})$. Otherwise, $[GB(P;GI)] \Rightarrow (P_{vc}) [GB(P;GI')]$. Since $sz(GI') < sz(GI)$, we have $[GB(P;GI')] \Rightarrow^{\Delta_G} [P]$, according to (IH).

2. $GI_0 = \bar{x}$ (Case $GI_0 = x$ is similar).
   $[GB(P;GI)] \Rightarrow (P_{vc}) [GB(P;GI')]$, where $GI' = GI(S :: \{var(x), GI_1, \ldots, GI_m\})$. Since $sz(GI') < sz(GI)$, we have $[GB(P;GI')] \Rightarrow^{\Delta_G} [P]$, according to (IH).
(3) \(Gl_0 = f(F_1, \ldots, F_k)\) (Case \(Gl_0 = f(V_1, \ldots, V_k)\) is similar).
\[GB(P; GI) \Rightarrow (\text{Ctor-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{F_1, \ldots, F_k, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(4) \(Gl_0 = 0\).
\[GB(P; GI) \Rightarrow (\text{Nil-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{i, o, t, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(5) \(Gl_0 = (F)P_1\).
\[GB(P; GI) \Rightarrow (\text{Add-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{bn(F) :: \{F, P_1\}, GI_1, \ldots, GI_m\})\). Notice that \(sz(GI') < sz(GI)\). We have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(6) \(Gl_0 = (V)P_1\) (Case \(Gl_0 = (V)^1P_1\) is similar).
\[GB(P; GI) \Rightarrow (\text{Cont-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{V, P_1, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(7) \(Gl_0 = P_1 | P_2\) (Case \(Gl_0 = P_1 + P_2\) is similar).
\[GB(P; GI) \Rightarrow (\text{Par-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{P_1, P_2, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(8) \(Gl_0 = s \cdot P_1\) (Case \(Gl_0 = s \cdot P_1\) is similar).
\[GB(P; GI) \Rightarrow (\text{Def-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{s, i, o, t, \{i, o, t\} :: \{P_1\}, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(9) \(Gl_0 = P_1 \triangleright P_2\).
\[GB(P; GI) \Rightarrow (\text{Pr-GC}) [GB(P; GI')]\], where \(GI' = GI(S :: \{o, \{o\} :: \{P_1\}, \{i, o, t\} :: \{P_2\}, GI_1, \ldots, GI_m\})\). Since \(sz(GI') < sz(GI)\), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\), according to (IH).

(10) \(Gl_0 = (\text{vn})P_1\).
If \(n\) is a variable \(x\), \([GB(P; GI)] \Rightarrow (\text{Res-GC}) [GB(P; GI')]\), where \(GI' = GI(S :: \{x\} :: \{\text{var}(x), P_1\}, GI_1, \ldots, GI_m}\). Otherwise, \(n\) is a service name \(s\), and \([GB(P; GI)] \Rightarrow (\text{Res-GC}) [GB(P; GI')]\), where \(GI' = GI(S :: \{s\} :: \{s, P_1\}, GI_1, \ldots, GI_m}\). In either case, \(sz(GI') < sz(GI)\). According to (IH), we have \([GB(P; GI')] \Rightarrow \Delta_0 [P]\).
We would expect to prove that the set of rules

5.8.2 Completeness of data assignment rules

For this purpose, we need to consider the form in which a (normal) value can be shared. Formally, the transformed into that of

According to Theorem 1,

For any process $P$ and normal value $V$, $[GB(P; GI)] \Rightarrow^{\Delta_{G}} [P]$ and $[GB(P; GI)]' \Rightarrow^{\Delta_{G}} [P]'$. According to (IH).

As a natural deduction of Lemma [14], the set of garbage collection rules are complete.

**Theorem 13 (Completeness of garbage collection rules)** For any process $P$ and composite garbage item $GI$, $[GB(P; GI)] \Rightarrow^{\Delta_{G}} [P]$ and $[GB(P; GI)]' \Rightarrow^{\Delta_{G}} [P]'$.

**Proof.** According to Lemma [14], $[GB(P; GI)] \Rightarrow^{\Delta_{G}} [P]$. As a result,

$$[GB(P; GI)]' \equiv_{d} P_{(p, i, o, t)}[A(P)] [GB(P; GI)] (p, i, o, t) \Rightarrow^{\Delta_{G}} P_{(p, i, o, t)} [A(P)] [P] (p, i, o, t).$$

According to Theorem [1], $P_{(p, i, o, t)} [A(P)] [P] (p, i, o, t) \Rightarrow^{\Delta_{G}} [P]'$.

5.8.2 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 $\sigma = match(F; V)$.

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.

$$sf(V) \overset{def}{=} \begin{cases} V & \text{if } V \text{ is a variable } x \\ vv(V) & \text{if } V \text{ is a constructed value } f(V_{1}, \ldots, V_{k}) \end{cases}$$

We claim that the graph of the sharing form of a value can be transformed to the graph of the value through applications of data assignment rules.

**Lemma 15** 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 $f(V_{1}, \ldots, V_{k})$, we have $[P(sf(f(V_{1}, \ldots, V_{k})))] \equiv_{d} [P(vv(f(V_{1}, \ldots, V_{k})))] \Rightarrow^{\text{Ctr-Norm}} [P(f(V_{1}, \ldots, V_{k}))]$.

Then, we show that we can make a copy of each shared value through applications of data assignment rules.

**Lemma 16** 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 $V$. 
For any process $P$ and normal value $V$, as a natural deduction of Lemma 16, we have the following lemma.

**Proof.** Let $V = x$.

\[
\begin{align*}
\lbrack P(L:vv(x), Sh(L)) \rbrack & \Rightarrow_{(VV-Norm)} \lbrack P(L:x, Sh(L)) \rbrack \equiv_d \lbrack P(L:x,x) \rbrack \\
\end{align*}
\]

(2) $V = f(V_1, \ldots, V_k)$.

\[
\begin{align*}
\Rightarrow_{(Ctr-Split)} &\hspace{1cm} \lbrack P(L:vv(f(V_1, \ldots, V_k), Sh(L)) \rbrack \\
& \Rightarrow_{(IH)} \equiv_d \lbrack P(\sigma(V_1, \ldots, V_k), f(V_1, \ldots, V_k)) \rbrack \\
& \hspace{1cm} \equiv_{\Delta_0} \lbrack P(L:vv(f(V_1, \ldots, V_k), f(V_1, \ldots, V_k)) \rbrack \\
& \hspace{1cm} \equiv_f \lbrack P(L:vv(V_1, \ldots, V_k), f(V_1, \ldots, V_k)) \rbrack \\
\end{align*}
\]

As a natural deduction of Lemma 16 we have the following lemma.

**Lemma 17** For any process $P$ and normal value $V$, $\lbrack P(L:vv(V), Sh(L), \ldots, Sh(L)) \rbrack \Rightarrow_{\Delta_0} \lbrack P(V,V, \ldots, V) \rbrack$, where $Sh(L), \ldots, Sh(L)$ are all the occurrences of $Sh(L)$ in $P$.

**Proof.** If $V$ is a variable $x$,

\[
\begin{align*}
\lbrack P(L:vv(x), Sh(L), \ldots, Sh(L)) \rbrack & \Rightarrow_{(VV-Norm)} \lbrack P(L:x, Sh(L), \ldots, Sh(L)) \rbrack \\
& \equiv_d \lbrack P(x,x, \ldots, x) \rbrack. \\
\end{align*}
\]

If $V$ is a constructed value $f(V_1, \ldots, V_k)$,

\[
\begin{align*}
& \Rightarrow_{(IH)} \equiv_d \lbrack P(\sigma(V_1, \ldots, V_k), f(V_1, \ldots, V_k)) \rbrack \\
& \equiv_{\Delta_0} \lbrack P(L:vv(V_1, \ldots, V_k), f(V_1, \ldots, V_k)) \rbrack \\
& \Rightarrow_{(Ctr-Norm)} \lbrack P(V,V, \ldots, V) \rbrack.
\end{align*}
\]

With these notations and lemmas, we are ready to prove the completeness of data assignment rules.

**Theorem 14 (Completeness of data assignment rules)** For any normal process $P$, normal pattern $F$ and normal value $V$ such that $\sigma = match(F;V)$ exists, $\lbrack AS(V:F)P \rbrack \Rightarrow_{\Delta_0} \lbrack P_\sigma \rbrack$ and $\lbrack AS(V:F)P \rbrack \Rightarrow_{\Delta_0 \cup \Delta_T} \lbrack P_\sigma \rbrack$.

**Proof.** Let $F$ be of the form $F(\alpha_1, \ldots, \alpha_k)$, where $\alpha_1, \ldots, \alpha_k$ are all its pattern variables. In order that $\sigma = match(F;V)$ exists, $V$ must be of the form $F(V_1, \ldots, V_k)$ for some values $V_1, \ldots, V_k$. That is, $\sigma = [V_1, \ldots, V_k / \alpha_1, \ldots, \alpha_k]$. Let $P = P(x_1, x_1, \ldots, x_k)$, where $x_1, x_1, \ldots, x_k$ are all the occurrences of its free variables bound by $F$.

\[
\begin{align*}
& \Rightarrow_{(Ctr-Assign)} \lbrack AS(V_1, \ldots, V_k ; \alpha_1, \ldots, \alpha_k)P(x_1, x_1, \ldots, x_k, \ldots, x_k) \rbrack \\
& \Rightarrow_{(PV-Assign)} \equiv_d \lbrack P(L_1:uu(V_1), \ldots, L_k:uu(V_k), f(V_1, \ldots, V_k)) \rbrack \\
& \equiv_{\Delta_0} \lbrack P(V_1, V_1, \ldots, V_k, V_k, \ldots, V_k) \rbrack \\
& \equiv_{\Delta_0} \lbrack P_\sigma \rbrack.
\end{align*}
\]
5.8.3 Proof of Proposition 6

With Theorem 13 and Theorem 14 we are able to prove Proposition 6.

Proof. Straightforward for each case of $P \rightarrow P\sigma$.

1. $P = C[s_p, s_p, \bar{s}_p], Q = (\forall r)C[r \triangleright P_1, r \triangleright P_2]$, where $C[\cdot, \cdot]$ is static and restriction-balanced.

$$[P]$$

$$\Rightarrow_{(Spec-Sync)} [\forall r \triangleright GB(P_1; \emptyset :: \{s\})], r \triangleright P_2]$$

(\text{Theorem 13} \Rightarrow_{\Delta_p \cup \Delta_T} [\forall r \triangleright P_1, r \triangleright P_2]$$

(\text{Theorem 1} \Rightarrow_{\Delta_T} [Q]$$

2. $P = C[r \triangleright (P'((V)P_1 + M_1)), r \triangleright C_2[(F)P_2 + M_2]], Q = C[r \triangleright (P'P_1), r \triangleright C_2[P_2\sigma], where $\sigma = match(F;V)$, $C[\cdot, \cdot]$ is static and restriction-balanced, $C_2[\cdot]$ is static, session-immune and restriction-immune.

$$[P]$$

$$\Rightarrow_{(Spec-Sync)}[C[r \triangleright (P'GB(P_1; \emptyset :: \{M_1\})), r \triangleright C_2[GB(AS(V;F)P_2; \emptyset :: \{M_2\})]]$$

(\text{Theorem 13} \Rightarrow_{\Delta_p \cup \Delta_T} [C[r \triangleright (P'P_1), r \triangleright C_2[AS(V;F)P_2]]$$

(\text{Theorem 14} \Rightarrow_{\Delta_T} [Q]$$

3. $P = C[r \triangleright (P'P' \triangleright C_1[(V)P_1 + M_1]), r \triangleright C_2[(F)P_2 + M_2]], Q = C[r \triangleright (P'P' \triangleright C_1[P_1]), r \triangleright C_2[P_2\sigma], where $\sigma = match(F;V)$, $C[\cdot, \cdot]$ is static and restriction-balanced, $C_1[\cdot]$ and $C_2[\cdot]$ are static, session-immune and restriction-immune.

$$[P]$$

$$\Rightarrow_{(Spec-Sync-Red)} [C[r \triangleright (P'P' \triangleright C_1[GB(P_1; \emptyset :: \{M_1\})]), r \triangleright C_2[GB(AS(V;F)P_2; \emptyset :: \{M_2\})]]$$

(\text{Theorem 13} \Rightarrow_{\Delta_p \cup \Delta_T} [C[r \triangleright (P'P' \triangleright C_1[P_1]), r \triangleright C_2[AS(V;F)P_2]]$$

(\text{Theorem 14} \Rightarrow_{\Delta_T} [Q]$$

4. $P = C_0[(P'((V)P_1 + M_1)) > ((F)P_2 + M_2)], Q = C_0[P_2\sigma((P'P_1) > ((F)P_2 + M_2))], where $\sigma = match(F;V)$, $C_0[\cdot]$ is static.

Let $F$ be of the form $F(y_1, \ldots, y_m)$, where $y_1, \ldots, y_m$ are all its pattern variables. Let $P_2$ be of the form $P_2(x_1, \ldots, x_k, x'_1, \ldots, x'_{k'}, s_1, \ldots, s_t)$, where $s_1, \ldots, s_t$ 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 \(F\).

\[
\Rightarrow \text{(Pip-Sync}) [P] \uparrow
\]

(\text{Theorem}\ [13] \text{Lemma}\ [6] \Rightarrow \Rightarrow \Delta_P) \Rightarrow \Delta_P

\[
\Rightarrow \text{(VC-Elim-PC)} [Q] \uparrow
\]

(5) \(P = C_0[\{P' \mid r \triangleright C_1(\{V\} \mid P_1 + M_1)\}] > ((F)P_2 + M_2)\), \(Q = C_0[P_2\sigma[\{P' \mid r \triangleright C_1[P_1]\}] > ((F)P_2 + M_2)\]),

where \(\sigma = \text{match}(F; V)\), \(C_0[\cdot]\) is static, \(C_1[\cdot]\) is static, session-immune and restriction-immune.

Let \(F\) be of the form \(F(?y_1, \ldots, ?y_m)\), where \(y_1, \ldots, y_m\) are all its pattern variables. Let \(P_2\) be of the form \(P_2(x_1, \ldots, x_k, x'_1, \ldots, x'_k, s_1, \ldots, s_l)\), where \(s_1, \ldots, s_l\) 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 \(F\).

\[
\Rightarrow \text{(Pip-Sync-Rel)} [P] \uparrow
\]

(\text{Theorem}\ [13] \text{Lemma}\ [6] \Rightarrow \Rightarrow \Delta_P) \Rightarrow \Delta_P

\[
\Rightarrow \text{(VC-Elim-PC)} [Q] \uparrow
\]
6 Conclusion

We propose a graph characterization of structured service programming with sessions and pipelines. This is done by translating a CaSPiS process term to a graph term of a graph algebra, and giving the graph algebra a model of hypergraphs. A reduction semantics of CaSPiS is then defined by a suitable graph transformation system.

The advantage of this approach is gained from the intuitive understanding of graphs, in terms of the concepts and structures of graphs more than the graphic representation itself, as well as the mathematical elegance and large body of theory available on graphs and graph transformations. The graph representation of a process characterizes both the hierarchical structure and the dynamic behavior of the process. The hypergraph model is new compared with the one given in [5] in that hierarchy is modeled by proper combinations of abstract edges between nodes and edges of different designs. This is a key nature that enables us to define graph transformations in the DPO form.

We provide a few set of graph transformation rules, including basic rules for congruence and reduction relation between graphs (and thus for processes), and also those for auxiliary purposes such as tagging, copy, data assignment and garbage collection. We proved that these graph transformation rules are indeed consistent, i.e. sound and complete, with the congruence and reduction rules of CaSPiS processes.

For future work, 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 need to consider possible optimizations in the implementation so as to reduce the computation scale as well as the consumption of computer resources. Future work also include the application of our graph model to a more substantial case study, and further exploration of the power of the theory of graphs and graph transformations for analysis of service-oriented programs.

References


