Journal of Web Engineering, Vol.17, No. 1&2 (2018) 121-140

A GRAPH BASED TECHNIQUE OF PROCESS PARTITIONING

GANG XUE, JING LIU, LIWEN WU, SHAOWEN YAO

School of Software, Yunnan University, Kunming, Yunnan, China

hill@ynu.edu.cn, liujing@ynu.edu.cn, liwen_w@outlook.com, yaosw@ynu.edu.cn

Received November 6, 2017
Revised January 29, 2018

Web service is an important technology for constructing distributed applications. In order to provide more complex functionalities, services can be reused by applying service composition. A service composition can be designed and implemented through a centralization or decentralization strategy. When observing the decentralized service composition, several researchers found out that this kind of compositions has its own advantages. These findings promote the development of approaches for designing, implementing and applying decentralized service compositions. Process partitioning is a topic about dividing a process into a collection of small parts. The technique is applicable to partitioning a process in a centralized service composition, and the result can provide support to constructing a decentralized service composition. This paper presents a technique of process partitioning. The technique can be used for constructing decentralized service compositions, and it provides a graph transformation based approach to reorganizing a process which is represented as a process structure graph. Compared to existing approaches, the technique can partition well-structured and unstructured processes. Some issues about decentralized service compositions and performance tests of service compositions are discussed in this paper. Experimental results show that, when compared with the centralized service composition, the decentralized service composition can have lower average response time and higher throughput in runtime environment.

Key words: process partitioning, graph transformation based algorithm, typed directed graphs
Communicated by: B. White & O. Pastor

1 Introduction

Service-oriented architecture (SOA) is a paradigm for building distributed applications. The architecture requires that system components are built on web services. Service-based components in an application are assembled with little effort into a network of services that can be loosely coupled to create flexible dynamic business processes and agile applications [1]. When technicians design distributed applications which can be easily developed, managed and extended, services with smaller size are proposed and applied. These services are called as microservices, and they also promote the development of microservice architecture (MSA) [2]. Microservices in MSA-based applications have the following characteristics [3]: small in size, messaging enabled, bounded by contexts, autonomously developed, independently deployable, decentralized, built and released with automated processes.

Service composition encourages and enables reusing existing services for providing more complex functionalities. Service orchestration and service choreography are two basic forms of service

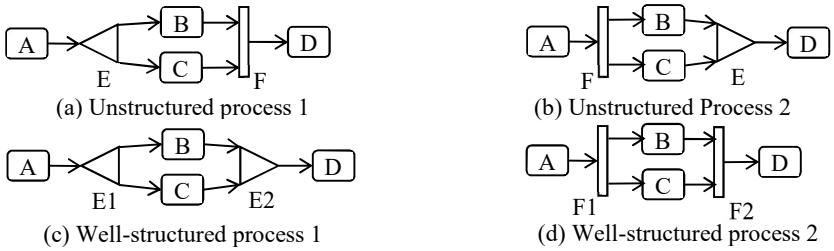
composition [4]. The composition of services in an orchestration is controlled by a central component. Compared to it, service choreography does not need a central controller. It describes participating services from a global viewpoint, and then implements composition by exchanging messages. For SOA, WS-BPEL [5] is a service orchestration language, and WS-CDL [6] is a famous solution for service choreography. Both composition forms can be applied in MSA-based systems. But in MSA world, service choreography may have more "playgrounds" than before, since decentralization is a core characteristic of microservices.

Figure 1 Service composition examples

Decentralized service composition is a special service composition technique, and it may cover the following topics [7, 8, 9, 10]: designing decentralized composite services, decentralizing service orchestrations, deploying participating services, executing compositions, and others. This technique improves modularity of application systems. In addition, for a MSA-based application, it supports applying multiple different smaller services, and it can make development easier. The configuration of a decentralized service composition is different from that of a centralized composition. One example is shown in Figure 1. For a given task, the composition in Figure 1 (a) uses "Service1" as a controller. After sending a request "o1", the controller requires message "i" for generating requests "o2" and "o3", which can initiate "Service3" and "Service4" respectively. As shown in Figure 1 (b), the composition can be organized without a control component. In the decentralized composition, "Service1'" only needs to send a request "o1" (to "Service2'"), and then "Service2'" can initiate "Service3" and "Service4". The example shows that centralized and decentralized compositions have different structures, and their participating services can have different functionalities. Existing research results show that if a decentralized service composition has reasonable structure, and all operational services are deployed properly, the composition may reduce network traffic [8], increase throughput of server [9], and reduce response time [10] when compared with the centralized service composition.

A process is a description about the execution of composite services in a composition. A process can be treated as an object in a process management system, or it can be implemented as a working component. When a process is represented as a directed graph, a *well-structured* process means that every node with multiple outgoing edges (a split) has a corresponding node with multiple incoming edges (a join), and vice versa, such that the set of nodes between the split and the join induces a single entry single exit (SESE) region [12]; in addition, the well-structuredness requires that a split and its corresponding join (or a join and its corresponding split) must have same type, i.e. if the split is an *or-split*, the join must be an *or-join*; and if the split is an *and-split*, the join must be an *and-join* [13]. If the *well-structured* condition cannot be satisfied, the process is *unstructured*. Figure 2 shows some process examples. Process in Figure 2 (a) is unstructured, since node "E" is an or-split (which is represented as a triangle), and node "F" is an and-join (which is represented as a thin rectangle). The structure will lead deadlock, since node "F" will never work when the process is running. Process in Figure 2 (b) is unstructured, since node "F" is an and-split (which is represented as a thin rectangle), and "E" is an or-

join (which is represented as a triangle). The structure will leave instances of "B" or "C" when the process is finished. Process in Figure 2 (c) is a well-structured process, since node "E1" and "E2" are or-split and or-join respectively, and node "B" or "C" can induce a SESE region. Process in Figure 2 (d) is a well-structured process, since node "F1" and "F2" are and-split and and-join respectively, and node "B" or "C" can induce a SESE region. Well-structured processes can be executed normally. However, not all processes in applications are well-structured, and researchers also have found out that not all unstructured processes can lead to deadlock or leave multiple instances of activity after they are finished [13].



(a) Unstructured process 1 (b) Unstructured Process 2

(c) Well-structured process 1 (d) Well-structured process 2

Figure 2 Well-structured and unstructured process examples

Process partitioning has been discussed in several researches. For example, paper [11] proposed an algorithm for transforming centralized workflow into decentralized work processes; and authors of paper [9] developed an approach for partitioning a composite web service written as a single BPEL program into an equivalent set of decentralized sub-programs. For different purposes, a process can be partitioned by using different methods, and the results also have different forms. Process partitioning can be applied in constructing decentralized service compositions, since partitioning a process will enforce reorganizing the configuration of a service composition. Many proposed approaches can be applied in partitioning well-structured processes for organizing decentralized service compositions. However, if a process is unstructured, it has to be restructured before applying those approaches.

This paper proposes a technique of partitioning process for constructing decentralized service compositions. Unlike other existing approaches, the technique can handle unstructured and well-structured processes. The technique represents a process as a typed directed graph, and forms a corresponding process structure graph (PSG) for the process model. The technique also has a graph transformation based algorithm for grouping nodes in a PSG. The output of the algorithm can provide a solution for partitioning the process. Technical issues about decentralized service compositions and performance tests are also discussed in this paper. Experimental results show that the decentralized composition can have lower average response time and higher throughput when compared to the centralized composition.

The rest of this paper is organized as follows. Section 2 summarizes some related works. Section 3 introduces basic tools and foundations of the technique. Section 4 presents the technique of process partitioning. Some issues about constructing decentralized service compositions are discussed in section 5, and this section also provides experimental results of the performance test of service compositions. Section 6 concludes the whole paper.

2 Related works

Techniques which are closely related to process partitioning and decentralized service compositions can be found in several researches. As mentioned before, a process can be partitioned by using different methods, and the results also have different forms for different purposes.

At the program level, program slicing is a source code analysis and manipulation technique, and it is used to identify sub-programs according to certain slicing criterions [23]. It's well known that program slicing can be transformed to a graph reachability problem, so reachability analysis in program dependence graphs (PDGs) [24] is a common method for program slicing [23]. Program slicing has been applied in the following fields [23]: testing and debugging, refactoring, reverse engineering, reuse, program optimization, and others.

In enterprise applications, the technique of decentralized workflow is discussed for enabling distributed workflow applications and management. Paper [11] presented a scalable, rigorously founded approach to enterprise-wide workflow management, based on the distributed execution of state and activity charts. An algorithm of workflow partitioning is proposed in [11], and the algorithm can partition a workflow specification which is represented by using state and activity charts. In the research of [10], a decentralized workflow model is presented. In the model, a workflow is divided into partitions called self-describing workflows, and handled by a light weight workflow management component, called the workflow stub, located at each organization. A workflow partitioning algorithm is also proposed in [10], and the algorithm is designed on the basis of finding sub-graphs in a graph-based workflow model. Researchers also proposed a dependency table based methodology for transforming a centralized process specification into a form that is amenable to a distributed execution and to incorporate the necessary synchronization between different processing entities [22].

In the world of web services, proposals have been made to decentralize the service composition execution. An intuitive approach is proposed for partitioning a composite web service written as a single BPEL program into an equivalent set of decentralized sub-programs [9]. The work in paper [9] uses a threaded control flow graph (TCFG) to model a process, and generates a program dependence graph (PDG) of the process. Based on the PDG, an algorithm for merging portable tasks is proposed, and the decentralization of a web service composition can be formed on the output of the algorithm. A process can also be partitioned according to its working conditions. A method presented in [25] can form partitions of a process by selecting activities, and the selection takes the following properties into account: communication costs between partitions and Quality of Service (QoS) of services.

Compared to these works, the technique presented in this paper has its own characteristics. It is developed on the basis of typed graphs and graph transformation. The approaches in the technique can describe and partition a well-structured or unstructured process directly and intuitively.

3 Preliminaries

This section introduces typed directed graphs and graph transformation. For clarity, the concepts are introduced by using category-based tools. Another topic in this section is the program structure tree, and it will be used in Section 4.

3.1 The category of typed directed graphs

A typed directed graph is a directed graph, whose vertices and edges have different types. The definition of a directed graph is summarized as follows.

Definition 1 (*directed graph* and *digraph homomorphism*) [15]: a directed graph (or a digraph) G is a quadruple (V, E, src, tgt), where V is the vertex set of G; E is the directed edge set of G; $src: E \to V$ is the source function, and $tgt: E \to V$ is the target function.

If $G = (V, E, src, tgt)$ and $G' = (V', E', src', tgt')$ are two digraphs, a homomorphism from G to G' is a map $f: G \to G'$, which includes two functions $f_0: V \to V'$ and $f_1: E \to E'$ such that the following diagrams commute:

$$\begin{array}{ccc} E & \xrightarrow{src} & V \\ {\scriptstyle f_1}\downarrow & & \downarrow{\scriptstyle f_0} \\ E' & \xrightarrow{src'} & V' \end{array} \qquad \begin{array}{ccc} E & \xrightarrow{tgt} & V \\ {\scriptstyle f_1}\downarrow & & \downarrow{\scriptstyle f_0} \\ E' & \xrightarrow{tgt'} & V' \end{array}$$

■

Based on the digraph, a typed digraph is defined as follows.

Definition 2 (*typed digraph* and *typed digraph homomorphism*): a typed digraph $\tilde{G}$ is a triple (G, T, τ), where:

(1) G is the underlying directed graph;

(2) $T := (T_V, T_E)$ is the type set for G, and T_V, T_E are the type sets for vertices and edges in G respectively;

(3) $\tau := (\tau_V, \tau_E)$ is the typing function with $\tau_V: V \to T_V$ and $\tau_E: E \to T_E$.

For two typed digraphs $\tilde{G} = (G, T, \tau)$, $\tilde{G}' = (G', T', \tau')$ and a homomorphism $f = (f_0, f_1): G \to G'$, a homomorphism from $\tilde{G}$ to $\tilde{G}'$ is a map $\tilde{f}: \tilde{G} \to \tilde{G}'$ which includes function $\tilde{f}_0: \tau_V(V) \to \tau'_{V'}(f_0(V))$ and $\tilde{f}_1: \tau_E(E) \to \tau'_{E'}(f_1(E))$.

■

Graph transformation is a technique of rule-based modification of graphs [14]. The most common approaches can be double-pushout (DPO) based, single-pushout (SPO) based, and others. For two morphisms with one common domain in a category, a diagram can be formed on the basis of these morphisms, and a pushout is a colimit of the diagram. The definition is summarized in Definition 3.

Definition 3 (*pushout*) [15]: it supposes that two morphisms $f: A \to B$ and $g: A \to C$ can be given in a category $\boldsymbol{C}$, where A, B and C are objects of $\boldsymbol{C}$, i.e. $A, B, C \in Ob(\boldsymbol{C})$. The pushout of f and g includes an object D and two morphisms $i_0: B \to D$ and $i_1: C \to D$ in category $\boldsymbol{C}$ such that:

(1) $i_0 \circ f = i_1 \circ g$;

(2) for any object E, and two morphisms $h: B \to E$, $j: C \to E$ in category $\boldsymbol{C}$ with $h \circ f = j \circ g$, there is a unique morphism $i: D \to E$, and two morphisms $h = i \circ i_0$, $j = i \circ i_1$ can be get.

■

The SPO graph transformation is defined on a pushout diagram. Each transformation depends on a transformation rule, and the rule can contribute to generating a new graph from the original graph.

Definition 4 (*rule based graph transformation*) [16, 17]: a transformation rule is a partial morphism $r: L \to R$, where L and R are *left side* and *right side* of the rule r respectively. A *match* for the rule r into graph G is a total morphism $m: L \to G$.

For a given rule $r: L \to R$ and a *match* $m: L \to G$, if the graph G can be transformed into graph H, then H is an object in the pushout of r and m, i.e. the following diagram commutes:

$$\begin{array}{ccc} \boldsymbol{L} & \xrightarrow{r} & \boldsymbol{R} \\ {\scriptstyle m}\downarrow & & \downarrow{\scriptstyle m'} \\ \boldsymbol{G} & \xrightarrow{r'} & \boldsymbol{H} \end{array}$$

■

A category can be formed for typed digraphs, and the category contains a collection of typed digraphs and morphisms.

Definition 5 (*the category of typed digraphs*): the category of typed digraphs, which is denoted as $\boldsymbol{DGrph_T}$, consists of the following entities:

(1) a collection $Ob(\boldsymbol{DGrph_T})$, whose elements are typed digraphs;

(2) for any typed digraphs $\tilde{G}, \tilde{H} \in Ob(\boldsymbol{DGrph_T})$, a set $Hom(\tilde{G}, \tilde{H})$, which is a set of morphisms from $\tilde{G}$ to $\tilde{H}$; a morphism from $\tilde{G}$ to $\tilde{H}$ is a homomorphism from $\tilde{G}$ to $\tilde{H}$;

(3) for every $\tilde{G} \in Ob(\boldsymbol{DGrph_T})$, an identity morphism on $\tilde{G}$; the morphism is denoted as $\widetilde{id}_{\tilde{G}}$,and $\widetilde{id}_{\tilde{G}} \in Hom(\tilde{G}, \tilde{G})$;

(4) for every three typed graphs $\tilde{G}, \tilde{H}, \tilde{I} \in Ob(\boldsymbol{DGrph_T})$, a binary operation $\circ: Hom(\tilde{H}, \tilde{I}) \times Hom(\tilde{G}, \tilde{H}) \to Hom(\tilde{G}, \tilde{I})$, which is called as the composition of morphisms; a composition of $\tilde{m}: \tilde{G} \to \tilde{H}$ and $\tilde{n}: \tilde{H} \to \tilde{I}$ is written as $\tilde{n} \circ \tilde{m}$.

Based on the entities above, the following rules must be satisfied:

(a) for any morphism $\tilde{m}: \tilde{G} \to \tilde{H}$ and $\tilde{G}, \tilde{H} \in Ob(\boldsymbol{DGrph_T})$, $\tilde{m} \circ \widetilde{id}_{\tilde{G}} = \tilde{m}$ and $\widetilde{id}_{\tilde{H}} \circ \tilde{m} = \tilde{m}$ can be get;

(b) for any three morphisms $\tilde{m}: \tilde{G} \to \tilde{H}$, $\tilde{n}: \tilde{H} \to \tilde{I}$, $\tilde{o}: \tilde{I} \to \tilde{J}$ and $\tilde{G}, \tilde{H}, \tilde{I}, \tilde{J} \in Ob(\boldsymbol{DGrph_T})$, $\tilde{o} \circ (\tilde{n} \circ \tilde{m}) = (\tilde{o} \circ \tilde{n}) \circ \tilde{m}$ can be get.

■

In $\boldsymbol{DGrph_T}$, the following colimits can be found.

Proposition 1: $\boldsymbol{DGrph_T}$ has coproducts.

Proof: It supposes that $\tilde{G}_1 = (G_1, T_1, \tau_1)$ and $\tilde{G}_2 = (G_2, T_2, \tau_2)$ are two typed digraphs in $\boldsymbol{DGrph_T}$, where $G_1 = (V_1, E_1, src_1, tgt_1)$, $T_1 = (T_V^1, T_E^1)$, $\tau_1 = (\tau_V^1, \tau_E^1)$; and $G_2 = (V_2, E_2, src_2, tgt_2)$, $T_2 = (T_V^2, T_E^2)$, $\tau_2 = (\tau_V^2, \tau_E^2)$.

Let $\tilde{G}$ be the disjoin union of $\tilde{G}_1$ and $\tilde{G}_2$, i.e. $\tilde{G} \coloneqq \tilde{G}_1 \sqcup \tilde{G}_2$. For $\tilde{G}$, the underlying digraph G is $(V_1 \sqcup V_2, E_1 \sqcup E_2, \{src_1, src_2\}, \{tgt_1, tgt_2\})$, the type set T is $(T_V^1 \sqcup T_V^2, T_E^1 \sqcup T_E^2)$, and the typing function τ is $(\{\tau_V^1, \tau_V^2\}, \{\tau_E^1, \tau_E^2\})$. For $\tilde{G}_1$ and $\tilde{G}_2$, two inclusion morphisms $\tilde{\iota}_1: \tilde{G}_1 \to \tilde{G}$ and $\tilde{\iota}_2: \tilde{G}_2 \to \tilde{G}$ can be defined.

For any other cospan $\tilde{G}_1 \xrightarrow{\tilde{\iota}} \tilde{G}' \xleftarrow{\tilde{\jmath}} \tilde{G}_2$, there is a morphism $\tilde{s}_{i,j}: \tilde{G} \to \tilde{G}'$ such that $\tilde{\iota} = \tilde{s}_{i,j} \circ \tilde{\iota}_1$ and $\tilde{\jmath} = \tilde{s}_{i,j} \circ \tilde{\iota}_2$. If there is another morphism $\tilde{m}: \tilde{G} \to \tilde{G}'$ such that $\tilde{\iota} = \tilde{m} \circ \tilde{\iota}_1$ and $\tilde{\jmath} = \tilde{m} \circ \tilde{\iota}_2$, then $\tilde{m} = d \circ \tilde{s}_{i,j}$

and $\tilde{s}_{i,j} = d' \circ \tilde{m}$ can be defined. In the case of $\tilde{m} = d \circ \tilde{s}_{i,j}$, since $\tilde{s}_{i,j} \circ \tilde{\iota}_1 = \tilde{\iota} = \tilde{m} \circ \tilde{\iota}_1 = d \circ \tilde{s}_{i,j} \circ \tilde{\iota}_1$, so d is an identity morphism. Similarly, in the case of $\tilde{s}_{i,j} = d' \circ \tilde{m}$, since $\tilde{m} \circ \tilde{\iota}_1 = \tilde{\iota} = \tilde{s}_{i,j} \circ \tilde{\iota}_1 = d' \circ \tilde{m} \circ \tilde{\iota}_1$, so d' is an identity morphism.

Therefore, $\tilde{s}_{i,j}$ is unique, and $\tilde{G}$ is a coproduct of $\tilde{G}_1$ and $\tilde{G}_2$.

■

Proposition 2: ***DGrph*$_T$** has coequalizers.

Proof: It supposes that $\tilde{G}_1 = (G_1, T_1, \tau_1)$ and $\tilde{G}_2 = (G_2, T_2, \tau_2)$ are two typed digraphs in ***DGrph*$_T$**, and $\tilde{f}, \tilde{g}$ are two parallel morphisms from $\tilde{G}_1$ to $\tilde{G}_2$, i.e. $\tilde{f}, \tilde{g}: \tilde{G}_1 \to \tilde{G}_2$. Let $\tilde{G} = (G, T, \tau)$ be a typed diagraph and a morphism $\tilde{c}: \tilde{G}_2 \to \tilde{G}$ in ***DGrph*$_T$** such that $\tilde{c} \circ \tilde{f} = \tilde{c} \circ \tilde{g}$.

Since $\tilde{c}: \tilde{G}_2 \to \tilde{G}$, an equivalence relation $\tilde{c}(e) = \tilde{c}(e')$ can be found on $\tilde{G}_2$, where e and e' are typed edges in $\tilde{G}_2$. Considering that $\tilde{c} \circ \tilde{f} = \tilde{c} \circ \tilde{g}$, equivalence relation $\tilde{f}(x) \sim \tilde{g}(x)$ can be get, and the x is any one of typed edges in $\tilde{G}_1$. For the diagram, a new typed digraph $\tilde{G}'$ and $\tilde{q}: \tilde{G}_2 \to \tilde{G}'$ can be generated. Each typed edge in $\tilde{G}'$ is corresponding to an equivalence class of $\sim$, so $\tilde{q} \circ \tilde{f} = \tilde{q} \circ \tilde{g}$.

A morphism $\tilde{n}: \tilde{G}' \to \tilde{G}$ for $\tilde{c} = \tilde{n} \circ \tilde{q}$ can be defined. If there is another morphism $\tilde{n}': \tilde{G}' \to \tilde{G}$ that satisfies $\tilde{c} = \tilde{n}' \circ \tilde{q}$, then $\tilde{n} = d \circ \tilde{n}'$ and $\tilde{n}' = d' \circ \tilde{n}$ can be defined. For $\tilde{n} = d \circ \tilde{n}'$, d is an identity morphism, since $d \circ \tilde{n}' \circ \tilde{q} = \tilde{n}' \circ \tilde{q}$; and for $\tilde{n}' = d' \circ \tilde{n}$, d' is an identity morphism, since $d' \circ \tilde{n} \circ \tilde{q} = \tilde{n} \circ \tilde{q}$.

Therefore, $\tilde{n}$ is unique, and $\tilde{G}'$ together with morphism $\tilde{q}$ is a coequalizer of $\tilde{f}$ and $\tilde{g}$.

■

A pushout in ***DGrph*$_T$** can be formed on the basis of a coproduct and a coequalizer.

Proposition 3: ***DGrph*$_T$** has pushouts.

Proof: it supposes that $\tilde{f}: \tilde{A} \to \tilde{B}$ and $\tilde{g}: \tilde{A} \to \tilde{C}$ are two morphisms in ***DGrph*$_T$**, where $\tilde{A}, \tilde{B}, \tilde{C} \in Ob(\boldsymbol{DGrph_T})$. Let $\tilde{D} \coloneqq \tilde{B} \sqcup \tilde{C}$ be a coproduct of $\tilde{B}$ and $\tilde{C}$, and the inclusion morphisms are $\tilde{\iota}_1: \tilde{B} \to \tilde{D}$ and $\tilde{\iota}_2: \tilde{C} \to \tilde{D}$.

The pushout of $\tilde{f}$ and $\tilde{g}$ can be formed by using the coequalizer of $\tilde{\iota}_1 \circ \tilde{f}$ and $\tilde{\iota}_2 \circ \tilde{g}$. Let $\tilde{E}$ together with morphism $\tilde{e}: \tilde{D} \to \tilde{E}$ be the coequalizer of $\tilde{\iota}_1 \circ \tilde{f}$ and $\tilde{\iota}_2 \circ \tilde{g}$, the pushout of $\tilde{f}$ and $\tilde{g}$ consists of (1) the object $\tilde{E}$, (2) morphism $\tilde{e} \circ \tilde{\iota}_1: \tilde{B} \to \tilde{E}$ and $\tilde{e} \circ \tilde{\iota}_2: \tilde{C} \to \tilde{E}$.

■

Above propositions show that pushouts exist in ***DGrph*$_T$**, so the SPO graph transformation can be applied on typed digraphs.

3.2 The program structure tree

A program structure tree (PST) is a hierarchical representation of a program structure, and it is created on the basis of single entry single exit (SESE) regions of a control flow graph [18]. In a PST, nodes are SESE regions, and directed edges represent the nesting relationships of SESE regions.

A SESE region is defined on the concepts of *dominate* and *post-dominate* of edges in a control flow graph. An edge *x dominates y* means that if *V* is the target vertex of the edge *y*, every path from *entry* vertex to *V* includes *x*; similarly, an edge *y post-dominates* edge *x* means that if *W* is the source vertex of the edge *x*, very path from *W* to the *exit* vertex includes *y*.

Definition 6 (*SESE region*) [18]: a single entry single exit (SESE) region in a control flow graph is an ordered edge pair (a, b) of distinct control flow edges a and b where

(1) a dominates b;

(2) b post-dominates a;

(3) every cycle containing a also contains b and vice versa.

■

A SESE region (x, y) is *canonical* means that y dominates z for any SESE region (x, z) and x post-dominates w for any SESE region (w, y) [18]. Two canonical regions are either nested or disjoint [18]. Moreover, A vertex v is contained within a SESE region (a, b) if v dominates b and post-dominates a.

4 Process partitioning

A technique of process partitioning is introduced in this section. The idea has the following steps: (1) modelling a process by using a typed digraph; (2) constructing a process structure graph for the process model; (3) grouping nodes in the process structure graph; (4) partitioning the process according to the node groups in the process structure graph.

4.1 Process modelling

A basic process model contains activities and dependences. The activities are functions and the dependencies are execution-order constraints between activities. Dependences in a process model can be classified as control dependences or data dependences. In a graph-based process model, activities are represented as vertices, and dependences are represented as directed edges. For complex control structures, *gateways* are controllers of diverging or converging control dependences. There are four kinds of gateways: *fork*, *split*, *join* and *merge*. A *fork* gateway has one incoming edge, and it can enable two or more outgoing edges; the gateway has to ensure that all outgoing edges are enabled concurrently. A *split* gateway has one incoming edge, and it can enable one edge from multiple alternative outgoing edges; the gateway has to ensure that only one outgoing edge can be enabled. A *join* gateway has two or more parallel incoming edges, and it can enable one outgoing edge; the gateway has to ensure that the outgoing edge can be enabled only after all incoming edges have been enabled. A *merge* gateway has two or more incoming edges, and it can enable one outgoing edge; the gateway has to ensure that the outgoing edge can be enabled after one of incoming edges has been enabled.

Definition 7 (*process model*): a process model $\tilde{P}$ is a typed digraph, i.e. $\tilde{P} := (PG, T, \tau)$, where

(1) $PG = (V, E, src, tgt)$ is the underlying directed graph of $\tilde{P}$; for PG, V is the set of vertices; E is the set of edges; $src: E \to V$ is the source function, and $tgt: E \to V$ is the target function.

An element of V is one of the following kinds: *activity*, *entry*, *exit*, *fork*, *join*, *split*, and *merge*. An *activity* is a vertex with exactly one incoming edge and exactly one outgoing edge. An *entry* vertex has exactly one outgoing edge and no incoming edges. On the contrary, if a vertex has exactly one incoming edge and no outgoing edges, then the vertex is an *exit*. Gateway *fork* or *split* is a vertex with exactly one incoming edge and more than one outgoing edges. Gateway *join* or *merge* is a vertex with more than one incoming edges and exactly one outgoing edge.

(2) $T = (T_V, T_E)$ is the type set for PG; for T, $T_V \coloneqq \{"fixed", "portable"\}$ is the type set for vertices, and $T_E \coloneqq \{"control", "data"\}$ is the type set for edges.

(3) $\tau = (\tau_V, \tau_E)$ is the typing function with $\tau_V: V \to T_V$ and $\tau_E: E \to T_E$.

Besides these, a process model should have only one *entry* and one *exit*.

■

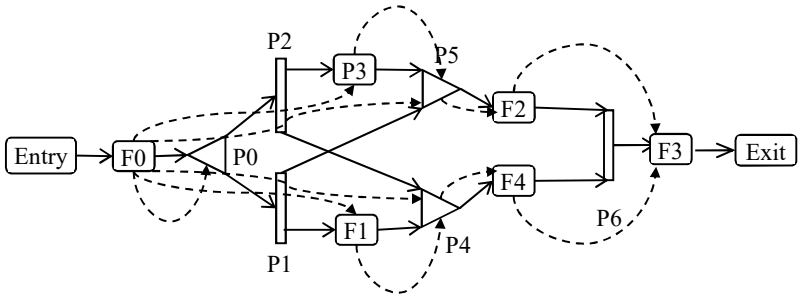
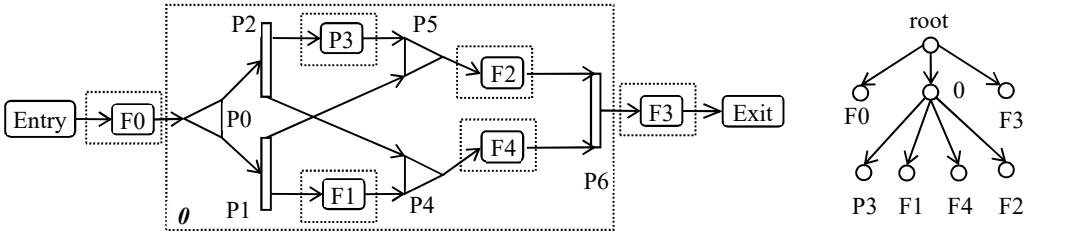


Figure 3 A process model

In Definition 7, a *fixed* vertex in a process means that the vertex can interact with outer environment; and a *portable* vertex means that it is an internal operation. Figure 3 shows a process model. The model has fixed vertices $\{F0, F1, \dots, F4\}$ and portable vertices $\{Entry, Exit, P0, \dots, P6\}$. Among these vertices, *P0* is a *split*; *P1*, *P2* are *forks*; *P4*, *P5* are two *merges*, and *P6* is a *join*. The solid arrows in the model are control dependences, and dashed arrows are data dependences. The structure of the model is *unstructured* according to the propositions in [12, 13], but it can be executed normally.

According to the work of [18], canonical SESE regions can be generated on Figure 3. Figure 4 (a) displays SESE regions of the process model in Figure 3. Since SESE regions are defined on the control dependences, data dependences in Figure 3 have been hidden in Figure 4 (a). All generated SESE regions can be represented as a PST, and the PST of Figure 4 (a) is summarized in Figure 4 (b).



(a) SESE regions of the process model

(b) The PST of the process model

Figure 4 SESE regions and the PST of the process model in Figure 3

Definition 8 (*process structure graph*): for a process model, a process structure graph (PSG) is a PST [18] based typed digraph. Besides the PST, a PSG has the following constitutions:

(1) all vertices in a SESE region; this means that a vertex in a PSG is a SESE region, or it is a vertex in the process model;

(2) all data dependences, and incoming, outgoing control flow edges of gateways in the process model.

Formally, a PSG $\tilde{S}$ is a typed digraph, i.e. $\tilde{S} := (S, T_S, \tau_S)$, where:

(i) $S = (V_S, E_S, src_S, tgt_S)$ is the underlying digraph of $\tilde{S}$; the vertices of S are SESE regions and some of vertices of a process model; the directed edges of S are branches of the PST, data dependences and some of control dependences in the process model; function $src_S: E_S \to V_S$ and $tgt_S: E_S \to V_S$ are the source and target functions;

(ii) $T_S = (T_V^S, T_E^S)$ is the type set for S; for T_S, T_V^S is the type set for vertices, and T_E^S is the type set for directed edges; the sets are defined as follows:

$$T_V^S := \{"fixed", "portable", "untyped"\}$$
$$T_E^S := \{"control", "data", "branch"\}$$

(iii) $\tau_S := (\tau_V^S, \tau_E^S)$ is the typing function with $\tau_V^S: V_S \to T_V^S$ and $\tau_E^S: E_S \to T_E^S$.

■

A PSG records the control dependences whose target vertices or source vertices are gateways in a process model. But for those control dependences whose target and source vertices are not gateways, they will be omitted in a PSG. According to Definition 8, a PSG can be generated easily on the basis of a PST. Figure 5 demonstrates a PSG example. In the Figure, a SESE region is represented as a solid box, and a process element is represented as a dashed box. Branches of the PST in Figure 5 are represented as solid arrows; data dependences are represented as dashed arrows; and control dependences are dotted arrows.

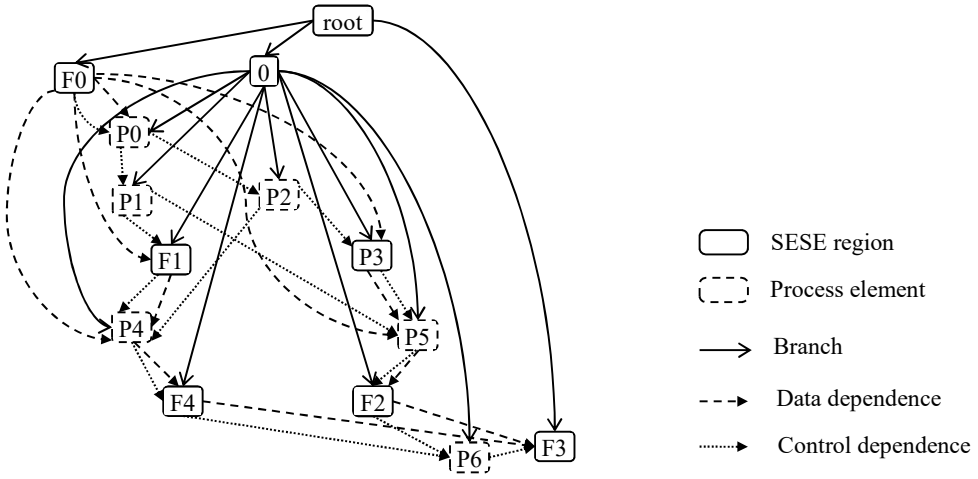


Figure 5 A PSG of the process model in Figure 3

In a PSG, if a directed edge is a PST branch, then its type is “branch”; if an edge is dependence in the process model, then the type can be “data” (dependence) or “control” (dependence). For an initiated PSG, if a PST vertex has multiple children nodes, the vertex is typed as “untyped”. Except for “untyped”, the type (“fixed” or “portable”) of a vertex is inherited from the type of the vertex in the process model. In Figure 5, *root* and *0* are *untyped* vertices; *F0*, *F1*, …, *F4* are *fixed* vertices; and *P0*, *P1*, …, *P6* are *portable* vertices.

4.2 An algorithm for grouping nodes in a PSG

Vertices in a PSG can be reorganized by grouping portable vertices and a fixed vertex together. Following the discussion in Section 3.1, the SPO graph transformation can be applied in the work of grouping. Figure 6 lists 9 transformation rules for grouping vertices in a PSG. As shown in the figure, the left side and the right side of a rule are restricted in cells with dashed lines. A branch in a rule is represented as a solid arrow. Data dependence in a rule is represented as a dashed arrow, and control dependence is represented as dotted arrow. The type of a vertex is marked with label ":". A fixed vertex in a rule has a type label ":f"; a portable vertex has a type label ":p"; and the label ":u" in a rule means that the corresponding vertex is an untyped vertex.

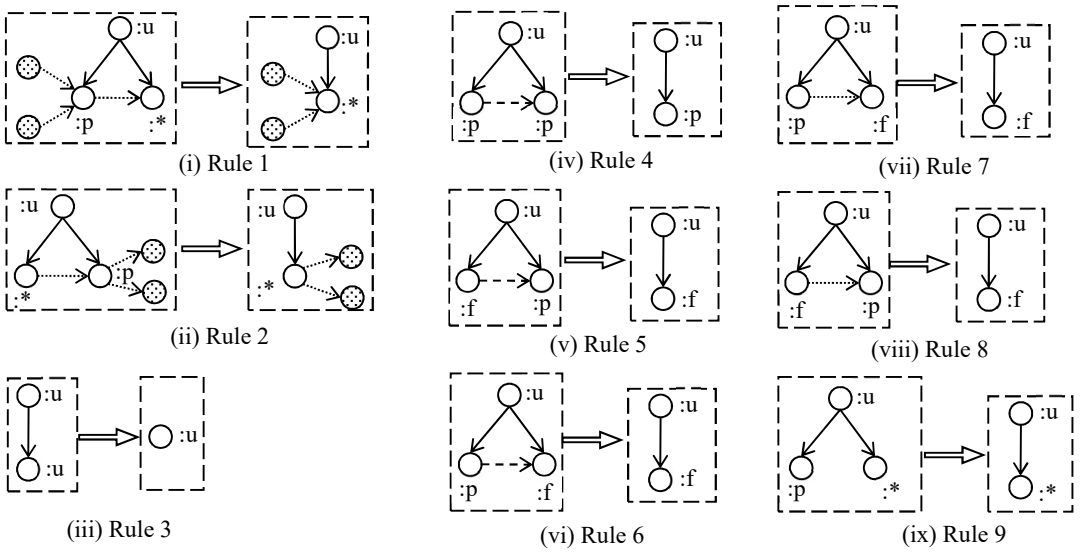


Figure 6 Transformation rules for a PSG

Some vertices in the rules of Figure 6 have label ":*". The label means that the type can be "fixed" or "portable". A concrete type that the label ":*" stands for can't be changed from the left side to the right side of the rule in a transformation. Vertices without labels in Rule 1 and Rule 2 are arbitrary vertices. These vertices and their types also can't be changed when the rule is being applied.

Considering that the homomorphism is a structure-preserved morphism, in Figure 6, the right sides of rules can have more elements except for Rule 9. Take the Rule 3 as an example. A fully detailed form of it is shown in Figure 7. Compared to the Rule 3 in Figure 6, the right side of Rule 3 in Figure 7 has a directed edge. However, the edge can be hidden safely, and it will not be used in the future working steps. Therefore, the rules in Figure 6 have already hidden the unnecessary edges.

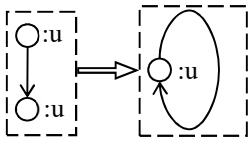


Figure 7 Fully detailed form of Rule 3 in Figure 6

For a SESE region (*a*, *b*), if the target node of *a* is a *fork* gateway, the region (*a*, *b*) is called as "fork-region" in this paper; and if the target node of *a* is not a *fork* gateway, the region (*a*, *b*) is called as "non-fork-region". According to the rules in Figure 6, an algorithm of grouping nodes can be designed as follows.

Algorithm 1 (the node grouping algorithm for a PSG):

Input: a process structure graph

Output: a typed digraph

The procedure:

(1) denote value *0* as the depth of node *root*;

(2) get the depth value *N* of the PST of the input PSG;

(3) set an indicator *cur* by using value *N-1*;

(4) for any node *v* with depth *cur*, all children nodes of *v* are merged by applying the following rules sequentially (each rule can be applied repeatedly until the rule is unable to be applied, and the merged nodes have to be recorded whenever a rule is applied): (i) Rule 1; (ii) Rule 2; (iii) Rule 4; (iv) Rule 5; (v) Rule 6.

(5) if *cur* > 0, then *none-fork-regions* with depth *cur* are merged into nodes with depth *cur-1* by applying Rule 3; after applying Rule 3, all children nodes of a *none-fork-region* should be recorded as children nodes of the generated node;

(6) *cur* = *cur-1*;

(7) if *cur* > *-1*, the procedure goes to step (4); otherwise, the procedure goes to next step;

(8) *cur* = *N-1*;

(9) if *cur* > 0, then *fork-regions* with depth *cur* are merged into nodes with depth *cur-1* by applying Rule 3; after applying Rule 3, all children nodes of a *fork-regions* should be recorded as children nodes of the generated node;

(10) for any node *w* with depth *cur-1*, all children nodes of *w* are merged by applying the following rules sequentially (each rule can be applied repeatedly until the rule is unable to be applied, and the merged nodes have to be recorded whenever a rule is applied): (i) Rule 1; (ii) Rule 2; (iii) Rule 4; (iv) Rule 5; (v) Rule 6.

(11) *cur* = *cur-1*;

(12) if *cur* > *0*, the procedure goes to step (9); otherwise, the procedure goes to next step;

(13) all portable children nodes of *root* are merged by applying the following rules sequentially (each rule can be applied repeatedly until the rule is unable to be applied, and the merged nodes have to be recorded whenever a rule is applied): (i) Rule 7; (ii) Rule 8; (iii) Rule 9.

(14) output the result.

■

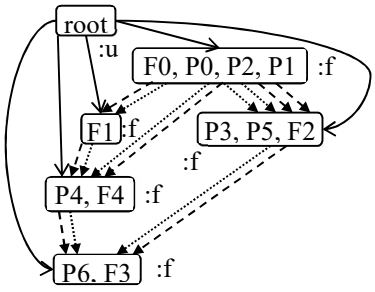
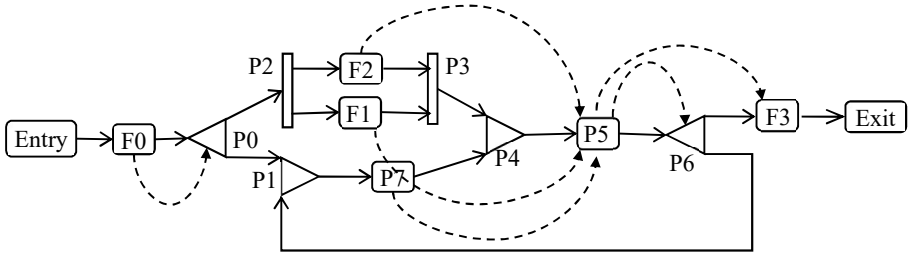
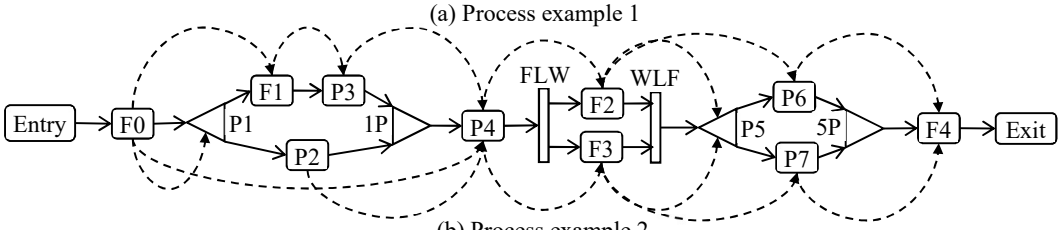


Figure 8 Node groups of the PSG in Figure 5

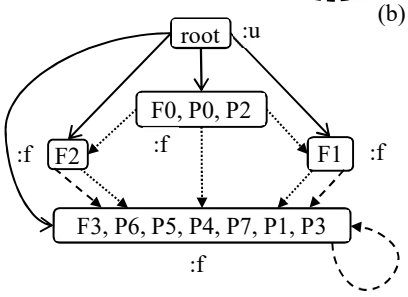
By applying Algorithm 1, the PSG in Figure 5 are reorganized in the following order: (1) $N = 2$; (2) *cur* = 1; (3) node *P4* and *F4* are merged as node "P4, F4" (according to Rule 1); (4) node *P5* and *F2* are merged as node "P5, F2" (according to Rule 1); (5) node *P0* and *P2* are merged as node "P0, P2" (according to Rule 2); (6) node "P0, P2" and *P1* are merged as node "P0, P2, P1" (according to Rule 2); (7) node *P3* and "P5, F2" are merged as node "P3, P5, F2" (according to Rule 6); (8) node *0* and *root* are merged as node *root* (according to Rule 3); (9) *cur* = 0; (10) node *P6* and *F3* are merged as node "P6, F3" (according to Rule 1); (11) node *F0* and "P0, P2, P1" are merged as node "F0, P0, P2, P1" (according to Rule 2); (12) *cur* = -1; (13) the result is generated, and the program exits.



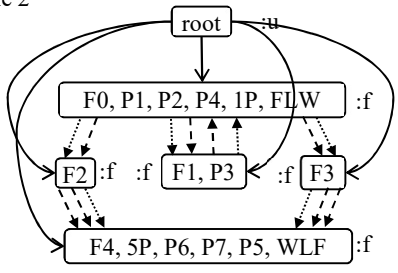
(a) Process example 1



(b) Process example 2



(c) Node groups of process example 1



(d) Node groups of process example 2

Figure 9 Two process examples and their node groups

Figure 8 shows the result after applying the Algorithm 1 on the PSG in Figure 5. The result contains six nodes: "root", "F0, P0, P2, P1", "F1", "P4, F4", "P3, P5, F2" and "P6, F3".

Besides the example above, Algorithm 1 can be applied on other processes. Figure 9 demonstrates two more samples. Figure 9 (a) shows a process with a control loop, and the output of the algorithm is demonstrated in Figure 9 (c). Algorithm 1 can also handle *well-structured* process models. Figure 9 (b) shows a process example which is designed and demonstrated in paper [9]. The structure of the example is *well-structured* according to the propositions in [12, 13]. Figure 9 (d) displays the result of Algorithm 1, and the result is same as one of the results in discussion of paper [9].

4.3 Partitions of process

Following Definition 8, a process structure graph can be generated for a process model. After applying Algorithm 1, the PSG is reorganized. The reorganized PSG includes nodes and dependences. Each non-root node in the reorganized PSG can sketch out a partition in the original process model, and the dependences in the reorganized PSG are dependences between partitions. Therefore, a result of Algorithm 1 can provide a solution for partitioning a process. Take the process model in Figure 3 as an example. The process can be divided into five different parts, since there are five non-root nodes in Figure 8. For the node "F0, P0, P2, P1" in Figure 8, the corresponding part in the process has four components: *F0*, *P0*, *P2* and *P1*; for the node "F1", the corresponding part in the process has one component: *F1*; and components in other parts can be summarized by observing "P4, F4", "P3, P5, F2" and "P6, F3". According to Figure 8, the partitioned process is shown in Figure 10.

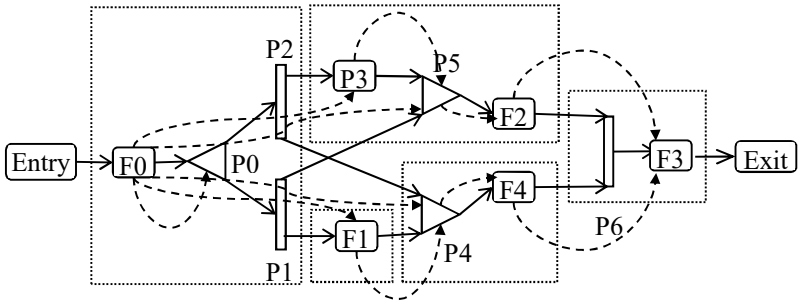


Figure 10 Partitions of the process model in Figure 3

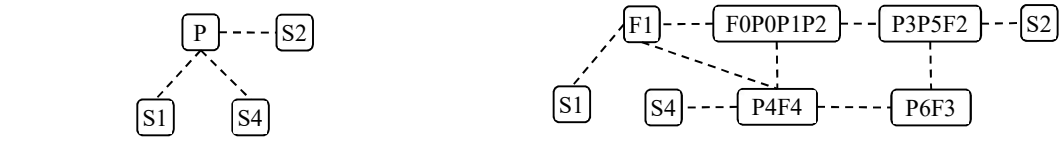
5 Configurations and performance evaluation of partitioned processes

In a centralized service composition, a process is implemented as or executed in a central controller component. After partitioning, the central controller can be simplified, and the composition should be reorganized in runtime environment. This section discusses configuration, implementation, deployment and performance issues about decentralized service compositions.

5.1 Partition configurations

Web services are basic components of service compositions. A centralized service composition contains working services and controller. The controller composes working services according to a process. After partitioning the process, the role of controller is replaced by a collection of partitions. The partitions

could be implemented as services, and this forces the whole composition to be reorganized. A service composition is shown in Figure 11 (a) for the process model in Figure 3. The composition has four components. The component "P" is a controller who can cooperate with other services. The component "S1", "S2" and "S4" are working services, and they work with "P" through activity "F1", "F2" and "F4" of the process respectively. The dashed lines in Figure 11 are message links between components. In Figure 11 (b), the component "P" is replaced by a collection of partitions (the partitions are shown in the Figure 10, and the name of each partition is generated by concatenating the names of participating components), and the composition becomes a decentralized composition.



(a) Centralized composition for the model in Figure 3

(b) Decentralized composition for the model in Figure 10

Figure 11 Service compositions for the example

When deploying a service composition, it may have different configurations for different runtime environments. Figure 12 (a), (b) and (c) show examples that the service composition (which is demonstrated in Figure 11 (a)) is deployed on two, three and four servers respectively. A decentralized service composition can have more configurations. Figure 12 (d), (e) and (f) demonstrate three different configurations for deploying the decentralized composition in different environments.

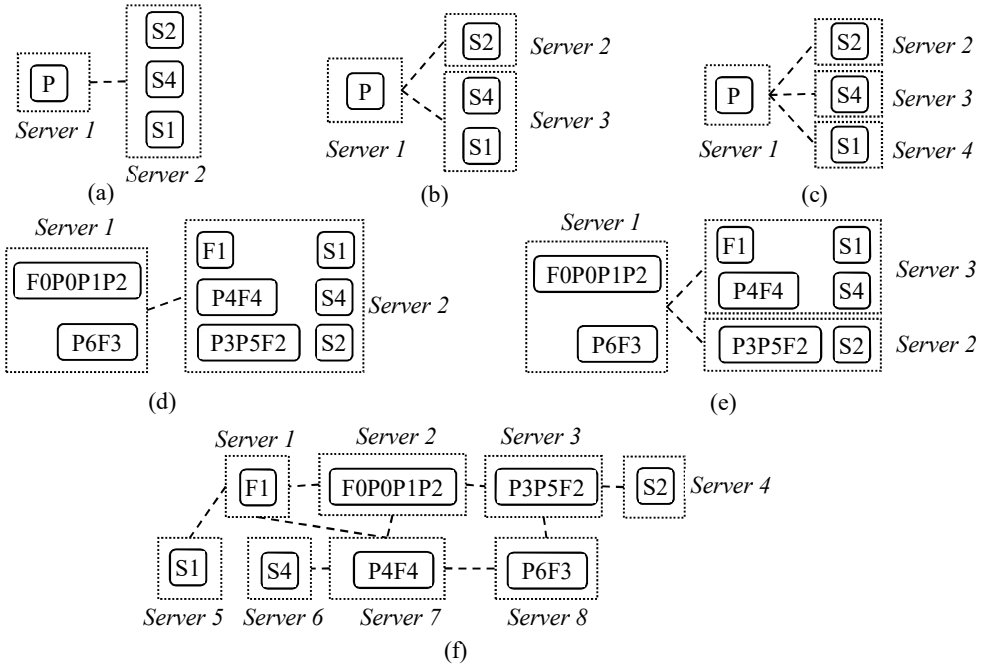


Figure 12 Component configurations of service compositions

As shown in Figure 12 (f), services in a decentralized composition can be deployed on totally different servers. But this configuration may raise some problems, e.g. additional network load, more testing and maintenance works, and others. Therefore, a process partition could be deployed in the same environment of the cooperating service. Take the Figure 11 (b) as an example. "F1" can be deployed in the working environment of "S1"; "P4F4" can be deployed on the server who contains "S4"; and similarly, the "P3P5F2" can be arranged to the server of "S2".

Process model has data and control dependences. All dependences should be tracked in a decentralized service composition. Considering that services are driven by messages, dependences among services are implemented as message links.

5.2 Runtime performance

A decentralized service composition does not dependent on a central controller, and this may simplify interactions among the services. Therefore, a decentralized composition is possible to improve runtime performance when compared with a centralized service composition. In order to show the real situation, this section sets up an experiment for evaluating performance of the process models in Figure 3 and Figure 10.

The centralized composition adopts the structure in Figure 11 (a), and all components are implemented as follows:

- Service "S1" collects data and returns a list of numbers.
- Service "S4" can sort numbers in a list and then selects three data from the list as outputs (for a list with length n, the selection is based on the following indices of the list: n-$n/2$, $n/2$, n+$n/2$). If the input does not contain data, the service uses a string constant as output.
- Service "S2" can sort numbers in a list and then selects the minimal, the maximal, and the middle numbers of the list as outputs. If the input does not contain data, the service uses a string constant as output.
- Controller "P" is an implementation of process model in Figure 3. Activities in the process is designed as follows: "F0" is used to receive requests from clients; "F1", "F2" and "F4" are used to invoke service "S1", "S2" and "S4" respectively; "F3" is used to send responses to clients; and "P3" is set as a data generator.

The decentralized composition adopts the structure in Figure 11 (b). New components "F0P0P1P2", "F1", "P3P5F2", "P4F4" and "P6F3" are implemented as services.

The experiment uses three computers as servers and one computer as a client. Services are implemented as RESTful web services by using Flask-RESTful [19]. Flask-RESTful is an extension for Flask framework [20] that adds support for quickly building REST services. The client adopts Apache JMeter [21] as a load testing tool for analyzing and measuring the performance of services. The whole experimental environment is summarized in Table 1.

Table 1 Experimental environment

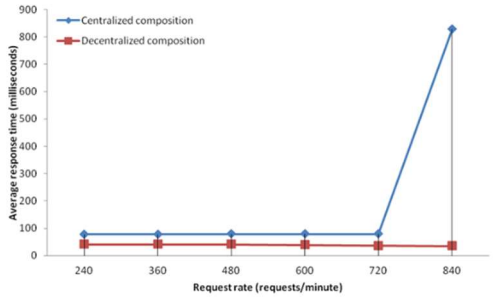
Participant	Hardware condition	Software platform	Network
Server1	Intel Core i7-7700 CPU, 16GB RAM	Flask-RESTful v0.3.6 Flask v0.12.2 (Runtime Environment: Python v2.7.13)	100Mb LAN
Server2	Intel Core i5-2410M CPU, 4GB RAM		
Server3	Intel Core i5-3317U CPU, 4GB RAM		
Client	Intel Celeron E3300 CPU, 2GB RAM	JMeter v3.3 (Runtime Environment: JDK v1.8.0)	

In the centralized setup (which is following the configuration in Figure 12 (b)), service "P" is deployed on Server1; service "S2" is deployed on Server2; and service "S1", "S4" are deployed on Server3. In the decentralized setup (which is following the configuration in Figure 12 (e)), service "F0P0P1P2", "P6F3" are deployed on Server1; service "P3P5F2", "S2" are deployed on Server2; and service "F1, "P4F4", "S1", "S4" are deployed on Server3. Service "S1", "S2" and "S4" are deployed only once in the whole experiment, and they are reused in the different compositions. In addition, all technical configurations or settings of runtime environments and software platforms are fixed in the whole experiment.

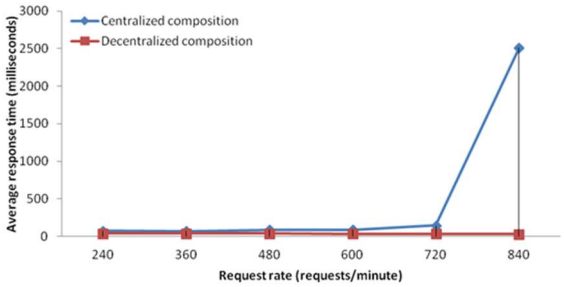
For the testing, sorting functions in "S2" and "S4" are implemented by using *sort()* method of List structure in Python (language). Seven different data sets are selected for the tests, and the size of data sets varies from 30 to 60k bytes (30, 10k, 20k, 30k, 40k, 50k and 60k bytes). The client uses multiple threads to generate HTTP requests. Considering that gateway "P0" in service "P" or service "F0P0P1P2" can lead to different results, each thread in tests creates two requests for two branches of "P0" respectively. The number of client threads is fixed in each test, and the ramp-up period of each test is set as 60 seconds. The tests vary the number of threads from 120 to 420, i.e. the average request rates vary from 4 to 14 requests per second.

Table 2 Experimental result 1

Data size (bytes)	Service composition		Request rate (requests/minute)					
			240	360	480	600	720	840
40k	Centralized	Average response time (milliseconds)	69	69	70	70	70	73
		Throughput (requests/second)	4.0	6.0	8.0	10.0	12.0	14.0
	Decentralized	Average response time (milliseconds)	41	41	41	41	37	36
		Throughput (requests/second)	4.0	6.0	8.0	10.0	12.0	14.0
50k	Centralized	Average response time (milliseconds)	78	78	79	79	79	828
		Throughput (requests/second)	4.0	6.0	8.0	10.0	12.0	13.2
	Decentralized	Average response time (milliseconds)	41	41	41	39	36	35
		Throughput (requests/second)	4.0	6.0	8.0	10.0	12.0	14.0
60k	Centralized	Average response time (milliseconds)	74	71	88	87	146	2510
		Throughput (requests/second)	4.0	6.0	8.0	10.0	11.9	12.2
	Decentralized	Average response time (milliseconds)	40	40	40	36	33	32
		Throughput (requests/second)	4.0	6.0	8.0	10.0	12.0	14.0



(a) Data size condition: 50k bytes



(b) Data size condition: 60k bytes

Figure 13 Response time variations with different request rates

Table 2 records results of the tests. The results show that, when handling 40k, 50k and 60k bytes data, the decentralized composition has lower average response time and higher throughput than the

centralized one for the different request rates. Specially, when the size of data set is greater than 50k, increasing request rate has a very different impact on compositions. In the case of centralized composition, increasing the request rate from 600 to 840 requests per minute causes the average response time to rise dramatically due to a decrease in processing capacity; along with the rising of the average response time, throughput of the composition degrades. But the impact of request rate is low for the decentralized composition. These conclusions are clearly shown in Table 2 and Figure 13.

Under the conditions of request rates, the changes of average response time and throughput with different data sizes are listed in Table 3. The results show that, under the condition of a high request rate, the average response time of the centralized composition increases, and the throughput decreases when data size increases. But the decentralized composition has better performance when compared with the centralized composition.

Table 3 Experimental result 2

Request rate (requests/minute)	Service composition		Data size (bytes)						
			30	10k	20k	30k	40k	50k	60k
840	Centralized	Average response time (milliseconds)	32	41	50	62	73	828	2510
		Throughput (requests/second)	14.0	14.0	14.0	14.0	14.0	13.2	12.2
	Decentralized	Average response time (milliseconds)	40	38	38	36	36	35	32
		Throughput (requests/second)	14.0	14.0	14.0	14.0	14.0	14.0	14.0
720	Centralized	Average response time (milliseconds)	32	41	51	61	70	79	146
		Throughput (requests/second)	12.0	12.0	12.0	11.9	12.0	12.0	11.9
	Decentralized	Average response time (milliseconds)	42	41	40	38	37	36	33
		Throughput (requests/second)	12.0	12.0	12.0	12.0	12.0	12.0	12.0
600	Centralized	Average response time (milliseconds)	32	41	51	61	70	79	87
		Throughput (requests/second)	10.0	10.0	10.0	10.0	10.0	10.0	10.0
	Decentralized	Average response time (milliseconds)	42	41	41	42	41	39	36
		Throughput (requests/second)	10.0	10.0	10.0	10.0	10.0	10.0	10.0

Figure 14 shows a comparison of response times of compositions. When request rate is fixed at 600 requests per minute, the average response time of the decentralized composition is higher than the average response time of the centralized composition for handling 30 bytes data. But the situation changes when the data size is increased. The average response time of the centralized composition rises with the increase of data size; however, the increase has a little impact on the average response time of the decentralized composition.

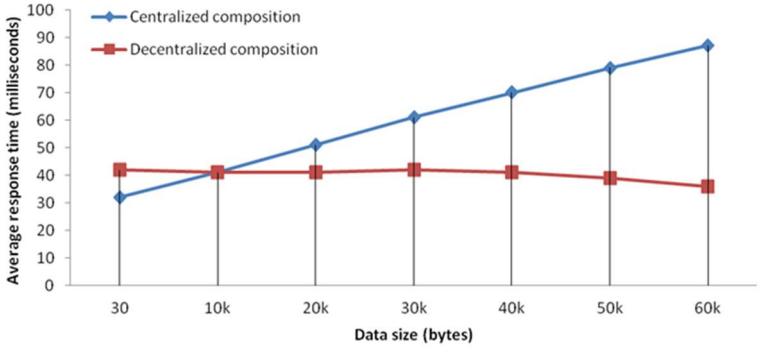


Figure 14 Response time variations with different data sizes (request rate condition: 600 requests per minute)

Above results show that, if components are deployed properly, the decentralized service composition for the process model in Figure 3 has better performance than the centralized service composition of the process model.

6 Conclusions

For constructing decentralized service compositions, this paper proposes a graph based technique that is applicable to partitioning a process in a centralized service composition. The technique is developed on typed digraphs and graph transformation, and it contains tools and approaches for constructing process models, representing the structures of process models, and partitioning the processes. This paper also discusses issues about the decentralized service composition, and provides experimental results of the performance test of service compositions. The experimental results show that the decentralized service composition can have better performance than the centralized composition.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. 61363084) and the Open Foundation of Key Laboratory in Software Engineering of Yunnan Province (No. 2015SE101).

References

1. Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., Service-Oriented Computing: A Research Roadmap. International Journal of Cooperative Information Systems, 2008, 17 (02). 223-255.
2. Dragoni, N., Giallorenzo, S., Lafuente, A. L., et al, Microservices: yesterday, today, and tomorrow. eprint arxiv: 1606.04036, 2016. Available online: https://arxiv.org/abs/1606.04036.
3. Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M., Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, 2016.
4. Peltz, C., Web services orchestration and choreography. IEEE Computer, 2003, 36(10). 46–52.
5. OASIS Standard, Web Services Business Process Execution Language Version 2.0. 2007. Available online: https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.
6. W3C Standard, Web Services Choreography Description Language Version 1.0. 2005. Available online: https://www.w3.org/TR/ws-cdl-10/.
7. Barker, A., Besana, P., Robertson, D. and Weissman, J. B., The Benefits of Service Choreography for Data-Intensive Computing. in Proceedings of the 7th international workshop on Challenges of large applications in distributed environments (CLADE '09), (Garching, Germany, June 09 - 10, 2009). 1-10.
8. Binder, W., Constantinescu, I. and Faltings, B., Decentralized Orchestration of Composite Web Services. in Proceedings of 2006 IEEE International Conference on Web Services (ICWS '06), (Chicago, Illinois, USA, September 2006). 869-876.
9. Nanda, M. G., Chandra, S. and Sarkar, V., Decentralizing Execution of Composite Web Services. in Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), (Vancouver, BC, Canada, October 24-28, 2004). 170-187.
10. Atluri, V., Chun, S. A., Mukkamala, R. and Mazzoleni, P., A decentralized execution model for inter-organizational workflows. Distributed and Parallel Databases, 2007, 22 (1). 55 - 83.
11. Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A. K. and Weikum, G., From Centralized Workflow Specification to Distributed Workflow Execution. Journal of Intelligent Information Systems -

Special issue on workflow management systems, 1998, 10 (2). 159-184.
12. Polyvyanyy, A., Structuring Process Models. PhD thesis, University of Potsdam, Germany, 2012.
13. Kiepuszewski, B., ter Hofstede, A. H. M. and Bussler, C., On structured workflow modelling. in Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE '00). 431–445.
14. Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G., Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer, 2010.
15. Spivak, D. I., Category Theories for Sciences, 1st edition. The MIT press, 2014.
16. Löwe, M., Algebraic approach to single-pushout graph transformation. Theoretical Computer Science, 1993, 109 (1-2). 181-224.
17. Müller, J., On Termination of Single-Pushout Graph Rewriting. Technical report 96-38, Technical University of Berlin, 1996. Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.4711.
18. Johnson, R., Pearson, D. and Pingali, K., The program structure tree: computing control regions in linear time. in Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation (PLDI '94), (Orlando, Florida, USA, June 20 - 24, 1994). 171-185.
19. Flask-RESTful (open source software). Available online: https://flask-restful.readthedocs.io/en/latest/.
20. Flask (open source software). Available online: http://flask.pocoo.org/.
21. Apapche JMeter (open source software). Available online: https://jmeter.apache.org/.
22. Fdhila, W., Yildiz, U. and Godart, C., A flexible approach for automatic process decentralization using dependency tables. in Proceedings of IEEE 7th International Conference on Web Services (ICWS 2009), (Los Angeles, United States, 2009). 847-855.
23. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J. and Tratt, L., State-based model slicing: a survey. ACM Computing surveys, 2013, 45 (4). Paper No. 53.
24. Ferrante, J., Ottenstein, K. J. and Warren, J. D., The program dependence graph and its use in optimization. ACM Transactions on Programming Language and Systems, 1987, 9 (3). 319-349.
25. Fdhila, W., Dumas, M., Godart, C. and Garciabanuelos, L., Heuristics for composite Web service decentralization. Software and Systems Modeling, 2014, 13 (2). 599-619.