# Component-based workflow systems development

## Hai Zhuge*

*Knowledge Grid Group, Key Lab of Intelligent Information Processing, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, 100080, PR China*

## Abstract

Component-based development is a promising way to promote the productivity of large workflow systems development. This paper proposes a component-based workflow systems development approach by investigating the following notions, mechanisms, and methods: workflow component, workflow component composition, reuse–association relationship between workflow components, and workflow component repository. The proposed approach is supported by a set of development strategies and a development platform. Through application and comparison, we show the advantages of the component-based workflow systems and the effectiveness of the proposed approach.
© 2002 Elsevier Science B.V. All rights reserved.

## 1. Introduction

A workflow is the computerised facilitation or automation of a business process, in whole or part [10,23]. A workflow can be abstracted as a network with task (i.e., activity) nodes and flows (i.e., transitions between task nodes). Domain business processes can be modelled through the execution of the network, and thus can be controlled and managed by incorporating the domain business application into the execution process of the network. The benefits of workflow-based applications, like flexibility, integration, and reusability, were discussed in Ref. [11].

Research works on workflow mainly concern the following aspects: (1) the extensions of workflow models [30,37], e.g., topics on incorporating more application-relevant factors such as time restriction, resources scheduling, and cost-effective evaluation into the existing workflow model; (2) the establishment and updating of the workflow model standard [23–30], so as to build a common language and to improve the interoperability and the interchange between workflow products; (3) semantic research [1,6,8,21], e.g., topics on checking the deadlock and the loop of workflow execution, workflow equivalence, workflow optimisation, etc.; and (4) the workflow application system development [5,11,12] and application-relevant research, e.g., topics on transactional workflow, agent-based workflow, reflexive workflow, workflow interoperation, and enhancing the quality and productivity of large workflow systems development. The research scope of this paper is on the fourth aspect and focuses on the reuse-based workflow development methodology.

---

* Fax: +86-1062567724.
*E-mail address:* zhuge@ict.ac.cn (H. Zhuge).

A good workflow is not only a simple image of domain business process but also the abstraction of it. The current workflow system development approaches lack a built-in development methodology. The current WfMC standard only provides a workflow definition language [23], but it does not provide business analysis tools and the development methodology. To employ software development methodologies like OOM [18] (including UML [17]) for domain business analysis is a way to complement the lack of workflow development methodology [30]. OOM supports inheritance mechanism, but the class-based inheritance needs to be adjusted so as to meet the needs of the workflow process reuse. On the other hand, workflow system development is an experience-dependent process. To realise effective experience reuse, we need to use the abstraction and analogy method. Unfortunately, the existing workflow development approaches do not support abstraction and analogy. Each new workflow system development has to carry out from scratch and follows the stringent development steps even though the developers can fully understand the domain business and have the development experience. The efficiency and quality of workflow development vary with different developers' development skills and experiences. It is still troublesome to dealing with the correctness and the complexity in composing different workflows designed by different designers even though they are in a cooperative development team and work for the same project.

Component technology is a way to raise the efficiency and quality of system development. It is an interesting topic to apply the software component concept and method to workflow development so as to promote its development efficiency and quality. On one hand, the component-based development encourages the developers to make domain business analysis by referring to the analysis experience of similar domain business. On the other hand, the component-based reuse provides a possibility of reusing the existing workflow components to compose a new workflow. The motivation of this paper is to investigate the concept and mechanism of workflow components and to propose an approach that supports the component-based workflow development.

To realise the motivation, three main problems need to be solved: (1) the definition of workflow component, an ideal workflow component should have an interface for encapsulating the workflow process and be able to be executed and used through the interface; (2) the approach for composing workflow components; and, (3) the establishment of an effective reuse mechanism, a set of reuse strategies, and the reuse-based workflow development process.

The semantic of event-driven workflow execution based on reactive components was investigated in Ref. [6], where the reactive component refers to the task (activity) entity that takes part in the workflow, and a broker mechanism is used to represent the component. The issue of the modular and the shared transactional workflow was discussed in Ref. [15]. However, the current research has neither given a complete definition of workflow component that has the advantages of a software component, nor a complete approach for component-based workflow systems development. The commercial workflow management products do not support the component-based workflow development.

A workflow system usually consists of two parts. First, a set of data definitions about: the workflow control data, the role model data, the work list, and the application invocation definition data. Second, a set of software components including: a process definition component, an enactment component (Engine), a work list handler, the user interface components, and the application components. Five kinds of interfaces between these components are defined by WfMC [23]. We define the workflow component by considering both the conceptual level and the system level. The conceptual level is separated from the system level by establishing an execution interface, which consists of the following four parts (subinterfaces): (1) an access interface, the input flow and the output flow of the workflow component by viewing the workflow component as a single task; (2) the process (control) interface; (3) the role interface; and (4) the restrictions.

To meet the needs of diverse application domains like e-commerce, we enable the flows of the discussed workflow model to reflect not only the execution dependence (i.e., control flow), but also the data dependence (i.e., data flow) and the temporal order relationship between tasks. Task execution can only be triggered by the condition satisfaction. To simplify the discussion of workflow component composition,

we assume that the data structures of the corresponding interfaces of all the workflow components are the same. This implies that all the discussed components are developed under the same standard.

In the following section, we first discuss the time order and the time restriction, and then define the concept of workflow component based on the investigation of its characteristics. An implementation model for workflow component is presented. In Section 3, the basic composition approaches of workflow components are investigated. Based on the definition of workflow component and the method of workflow component composition, Section 4 defines the reuse–association relationship between the workflow components and then the workflow component repository. To promote the effectiveness of workflow component reuse, the repository incorporates the domain business abstraction architecture (i.e., inheritance hierarchy) as well as the mapping between the domain business and the workflow components as the development experience. Section 5 presents the strategies for business abstraction and establishes the analogy relationship between the new business and the previously developed business in the repository (i.e., business analysis reuse). Section 6 presents the workflow system development process based on the component repository. Section 7 presents the application of the proposed approach, the development platform, and the application implication. Section 8 first presents the advantages of the component-based workflow compared with the traditional workflow, and then presents the related works on inheritance and component reuse as well as the comparison between these works and the proposed approach. Section 9 provides a summary of the work.

## 2. Workflow component

### 2.1. Time order and time restriction

Traditional workflow models mainly concern the execution dependence between tasks. In many real applications (especially in Internet-based e-commerce applications), workflow is also required to manage the data dependence and the time order dependence. Otherwise, inconsistencies may happen during the workflow execution process. For example, a task can only start to execute after receiving a required data flow from the preceding task, and the flow may take a long time to be transmitted from one task site to another (may even be longer than the execution duration of the related tasks). If the arrival time of the flow is later than the begin time restriction of the task that is expecting the flow, then temporal inconsistency happens. Such a begin time restriction usually exists in routine applications, e.g., investors need to receive the profit information about a company from some accountants before the stock market opens.

The semantics of traditional workflow models need to be adapted after considering the time factor. First, the 'and–split' is no longer 'a single thread of control splits into two or more parallel activities' as defined in Ref. [23]; these tasks may execute sequentially in time order caused by the duration differences taken by these split flows or caused by the differences between the execution conditions of these tasks. Second, the 'and–join' can no longer be defined as 'two or more parallel executing tasks converged into a single common thread of control' as defined in Ref. [23]; these tasks may not be parallel in time order. Third, the 'sequential routing' no longer means 'two or more tasks execute sequentially' defined in Ref. [23]; these tasks may not be strict by sequential in time order. So any task needs an execution condition, which can be the satisfaction of time restrictions or the arrival of the flow (the control flow or the data flow). Flow also needs a starting condition (e.g., the end of the related task or the data transmission condition). Appendix A presents the detail approach for representing time and time order in the workflow.

**Definition 1**. A workflow is called *temporal consistent* if all of its tasks and flows satisfy the time order and the time restriction during its execution process.

### 2.2. Definition of workflow component

First, a workflow component should be *independent*. The independency requires the workflow component to reflect an independent business that does not functionally depend on another workflow component (or task) and to be able to execute independently. All

the tasks in the workflow component cooperate with each other to implement an entire business component, i.e., none of them implements the irrelevant business. This also implies that any two workflow components do not have common tasks.

Second, a workflow component should be *encapsulated* so that it can be used, executed, and viewed as a whole like a single task. The encapsulation requires an access interface to normally specify the input and output of the workflow. To simplify the access interface, a workflow component can be normalised as that with just a unique start task node, and a unique success-end task node or a unique fail-end node. We can normalise a workflow that has multiple begin tasks as one begin task by adding a common predecessor (called connector) before them. The access interface of a workflow component C consists of an input flow (denoted as In(−, C)) related to the start node, an output flow (denoted as Out(C, −)) related to the success-end node, and a fail-exit flow (denoted as Exit(C, −)) related to the fail-end node. Any outside flow can only access the workflow component through its access interface (i.e., the input flow or the output flow). The encapsulation also requires a workflow component to have an execution interface (denoted as EI). Two mappings should be made before executing a component because different components may work with different execution platforms and different applications. The first mapping is from the execution interface into the execution platform interface (denoted as EPI), called EI-to-EPI mapping. This mapping enables a workflow component to be executed on any engines. The second mapping is from the execution interface into the application interface (denoted as AI), called EI-to-AI mapping. All the tasks will be mapped to the relevant applications during the workflow execution. The encapsulation requires the roles that implement the tasks of a component to be effective only within the component.

Third, a workflow component should *satisfy internal process completeness*. The completeness concerns the definition completeness and the execution completeness. A workflow component is called internal process definition complete if it satisfies the following conditions: (1) every internal node has at least an input flow and an output flow; (2) every internal flow directs to an internal node except for the exit flow and the output flow; and (3) the end nodes are reachable

from the start task node (deadlock and loop should be checked and eliminated). The second aspect of completeness is the execution completeness. On one hand, it requires all the restrictions and execution conditions could be satisfied during the execution process. On the other hand, it requires the execution of a workflow component to be treated as a single task. This implies that a workflow component can only execute on one engine. So a component has the following execution characteristic.

**Characteristic 1**. Once a running component instance has exited, all its tasks instances should not be in the running status until the component (instance) is reactivated.

Fourth, a workflow component should satisfy *internal consistency*. The internal consistency consists of two aspects: (1) *type consistency*, i.e., a task node can only accept the flows with the required type; and (2) *temporal consistency* between the flows and the related task nodes. All the internal times of a workflow component can be set in terms of a begin time variant. The variant will be assigned a begin time value when the component executes. All the internal times and the time restrictions do not need to be repeatedly computed.

The notion of a workflow component can be summarized as follows.

**Definition 2**. A *workflow component* is a workflow process that describes a category of complete business process units. It has the characteristics of independency, encapsulation, completeness, and consistency.

### 2.3. An implementation model for workflow component

A build time workflow component can be specified at the conceptual level and the system level as shown in Fig. 1. The conceptual level is the conceptual modelling of the business process. It describes the conceptual business process, the type of the input and output flows, and the restrictions (e.g., time restrictions and resource restrictions). The system level refers to the *execution interface*, which further consists of the *access interface* (EI-AI), the *process control interface* (EI-PI), the *role interface* (EI-RI), and the *restrictions* (RES). The access interface
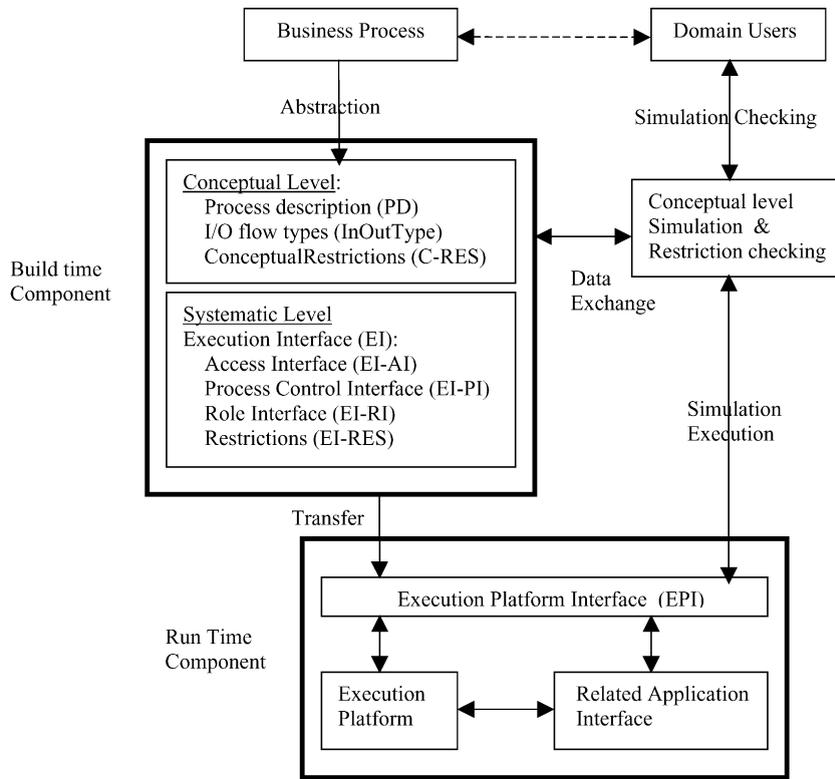
Fig. 1. The implementation model of workflow component.

defines the data structure of the input flow and output flow. The process control interface is the data structure for describing the workflow process and for controlling the status of the workflow process and its tasks. The role interface is the mapping from the tasks set into the task performers (roles). The basic roles can be the human participants, the client applications that interact with the work list handler, or the invoked applications (which have no interface).

The run time workflow component includes: *the execution platform*, the *execution platform interface*, and the related *application interface* (API). The execution platform consists of the workflow enactment service and the components like the work list handler and the administration and monitoring tools. The execution platform interface has the same content as the execution interface, but different platforms may have different data structures. The mapping from the execution interface to the execution platform interface

needs to be carried out before the execution of the component.

The execution platform also needs to provide a conceptual level simulation of the component execution for checking the run time satisfaction of the related restrictions and the process logic. Because checking by real execution of a workflow component could be very expensive in some areas, e.g., it may take several months to finish the execution process.

A set of functions is used to represent the data elements of the process control interface. First, OutSplit(C) is used to denote the split form of the output flows of C, $OutSplit(C) \in \{\vee, \wedge, None\}$. 'OutSplit(C) = None' means that there is no split output of C, so it just has one output flow. Second, InJoin(C) is used to denote the join form of the input flows of C, $InJoin(C) \in \{\vee, \wedge, None\}$. 'InJoin(C) = None' means that there is no join input of C, so it has just one input flow. Third, Append(C′, EI-PI(C)) is used to append a

task node C′ to the process control interface of the execution interface of C. Besides, $f(C_i, C_j)$ denotes the flow from $C_i$ to $C_j$, Type( $f$ ) denotes the types of flow $f$, and use Con(task$_i$) to denote the condition of task$_i$ execution.

The role set of component C, Roles(C), is a set of roles that implement all the tasks of component C and are only effective within C. The role structure of C can be represented by RoleStru(C)={role$_i$ → role$_j$ | role$_i$, role$_j$∈Roles(C)}, where role$_i$ → role$_j$ means that role$_i$ can participate in role$_j$. Each role$_k$∈Roles(C) has a unique identity within the component. Role interface (RI) is a mapping from the tasks set into Roles(C). Since a component C can be regarded as a single task, it should have a corresponding role (can be a virtual role), named as root(C), which satisfies root(C) → role$_i$ for any role$_i$∈Roles(C).

## 3. Composition of workflow components

### 3.1. General

The composition of existing components means to compose their conceptual level (i.e., the process description, the input/output flows types, and the restrictions) and system level (i.e., the execution interface), respectively. The connection between the existing execution interfaces means to generate a new data structure for the new component based on the existing interfaces and to make necessary data updating. Since we assume that all the existing workflow components use the same data structure, the new interface of the compound workflow component has the same data structure as the interfaces of the existing components. The new execution interface can be created by appending the data of the existing interfaces to the corresponding new data structure and adjusting the data for flow connection. For example, relational tables are eligible for representing the data structure of the execution interface. Different workflow components have the same set of table structures. New execution interfaces can be created by appending the data of the existing tables to the new tables, and then updating the data of the new table for flow connection.

Concerning the connection of the access interfaces, all the fail-exit flows of the existing components can be connected together as the fail-exit flow of the compound component, so we can focus on the connections of the input flows and the successful-end flows and the normalisation of the new process.

Additional roles (denoted as a set AddRoles) can be added to the existing roles when carrying out connection. The new role set of C is defined by: Role(C) = Role(C$_1$)∪...∪Role(C$_n$)∪AddRoles. The new role interface can be described by: RI(C) = RI(C$_1$)∪...∪RI (C$_n$)∪{C → root(C)}. The roles of the compound component can be adjusted so as to enable a role to participate in the tasks that belong to different original components according to business requirements. The new role structure can be defined on the basis of the existing role structures: RoleStru(C) = RoleStru(C$_1$)∪...∪RoleStru(C$_n$)∪{root(C) → root(C$_i$) | C$_i$∈C} ∪NewStru, where root(C) is the role of C and New-Stru={AddRoles → Roles(TaskSet)| where tasks in TaskSet share the same role and are sequentially executed}.

Hence, our discussion can focus on the following three aspects: (1) *access interface connection*; (2) *data adjustment of the process interface*; and (3) *time restriction adjustment*.

### 3.2. Sequential connection

A new compound component can be generated by sequentially connecting one component's output flow to another's input flow according to the sequence of $n$ components: $\langle C_1, ..., C_n \rangle$ if these components satisfy the following two items:

1. Type(Out(C$_i$, −) = Type(In(−, C$_{i+1}$)) for $i = 1,...,$ $n − 1$; and,
2. Restrictions of the equations of the first item are compatible with each other.

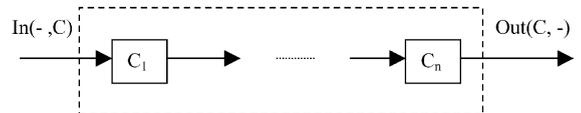The connection can be conceptually described as shown in Fig. 2.



Fig. 2. Sequential connection of $n$ components.

The generation of the new access interface consists of the following three steps.

1. The input flow of $C_1$ is as the input flow of the new compound component, i.e., $In(-, C) = In(-, C_1)$;
2. The output flow of $C_n$ is as the output flow of the new compound component, i.e., $Out(C, -) = Out(C_n, -)$; and,
3. $In(-, C_{i+1})$ and $Out(C_i, -)$ are updated as $f(C_i, C_{i+1})$, respectively, for $i = 1, \ldots, n-1$.

Once a compound workflow component is formed, users usually do not need to be concerned with its internal time restrictions anymore, for they are interested in the duration of the whole component and its begin time and end time. The run time duration of C can be computed by $D_r(C) = Time_E(C) - Time_B(C)$. In build time, we assume: $Time_B(f(C_i, C_{i+1})) = Time_E(C_i)$ and $Time_E(f(C_i, C_{i+1})) = Time_B(C_{i+1})$ hold. So the build time duration of C can be computed as follows:

$$D(C) = D(C_1) + \sum_{i \in [1, \; n-1]} [D(f(C_i, C_{i+1}))$$
$$+ D(C_{i+1})].$$

Any build time duration can be defined by a maximum duration and a minimum duration restriction. So the build time duration of C is a range from its minimum duration $D_{min}(C)$ to its maximum duration $D_{max}(C)$. Time restrictions of the compound component C are set in build time according to the build time duration of C, e.g., $Time_B(In(C)) = Time_B(In(C_1))$, $Time_E(Out(C)) < Time_B(In(C)) + D_{max}(C)$, and $D_r(C) \in [D_{min}(C), D_{max}(C)]$.

## 3.3. Parallel connection

The purpose of parallel connection is for shortening the execution duration of a set of related components or for synchronising between a set of components during execution. The parallel connection requires two connectors to normalise the access interface of the new component. The parallel connection of $n$ workflow components is conceptually described in Fig. 3. The responsibility of $Connector_1$ is to 'split' its input flow into the output flows required by $C_1, \ldots,$ and $C_n$, respectively. If all the flows only concern the control flow, then the function of the connector is very simple: its input flow type is the same as any of its output flow types. If the input flows of $C_i$ ($i = 1, \ldots, n$) concern multiple types, then the input flow type of the connector is the union of these types. The connector provides an integrated input interface for the compound component, the execution condition of the $Connector_1$ is to receive the flow that includes all of the input flows of $C_i$ ($i = 1, \ldots, n$). The responsibility of $Connector_2$ is to 'and−join' the $n$ flows output from $C_1, \ldots,$ and $C_n$, respectively. Its output flow is the union of the output flows of $C_i$ ($i = 1, \ldots, n$), and as the output of the compound component.

The connection between the $n$ access interfaces can be realised through the flow construction as follows: $Type(In(-, C)) = Type(In(-, Connector_1)) = Type(In(-, C_1)) \cup \ldots \cup Type(In(-, C_n))$; and, $Type(Out(C, -)) = Type(Out(Connector_2, -)) = Type(Out(C_1, -)) \cup \ldots \cup Type(Out(C_n, -))$. The data updating of the new execution interface consists of the following five steps:

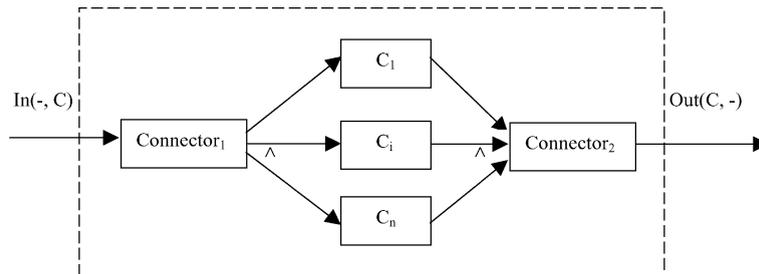1. $Append(Connector_1, EI\text{-}PI(C))$ and $Append(Connector_2, EI\text{-}PI(C))$;



Fig. 3. Parallel connection of $n$ components.

2. $OutSplit(Connector_1) = `\wedge$';
3. $Out(Connector_1, -) = \{C_1, \ldots, C_n\}$;
4. $Out(C_i, -) = f(C_i, Connector_2)$, $i \in \{1, \ldots, n\}$; and,
5. $InJoin(Connector_2) = `\wedge$'.

The build time duration of the new compound component C can be computed as follows: $D(C) = Max\{D(Connector_1) + D(f(Connector_1, C_1)) + D(C_1)\} + D(f(C_1, Connector_2)) + D(Connector_2), \ldots, D(Connector_1) + D(f(Connector_1, C_n)) + D(C_n) + D(f(C_n, Connector_2)) + D(Connector_2)\}$. In order to keep parallel execution between $n$ components, the following time restriction should be kept: $Max\{Time_B(C_1), \ldots, Time_B(C_n)\} < Min\{Time_E(C_1), \ldots, Time_E(C_n)\}$. The begin time and the end time are defined by $Time_B(C) = Time_B(Connector_1)$ and $Time_E(C) = Time_E(Connector_2)$, respectively. The run time duration of C can be computed by $D_r(C) = Time_E(Connector_2) - Time_B(Connector_1)$ and $D_r(C) \in [D_{min}(C), D_{max}(C)]$.

### 3.4. 'And–split' connection and 'Xor–split' connection

The 'and–split' connection between components $C_0$, $C_1$, $\ldots$, and $C_n$ is described in Fig. 4. The connector is responsible for carrying out the 'and–join' of the output flows of $C_i$ as its output flow, i.e., $Type(Out(C, -)) = Type(Out(Connector, -)) = Type(Out(C_1, -)) \cup \ldots \cup Type(Out(C_n, -))$. The access interface of the new compound component can be created by two assignments: $In(C, -) = In(C_1, -)$; and, $Type(Out(C, -)) = Type(Out(C_1, -)) \cup \ldots \cup$

$Type(Out(C_n, -))$. The updating of the data of the execution interface consists of the following steps:

1. $Out(C_0, -) = \{C_1, \ldots, C_n\}$;
2. $OutSplit = `\wedge$';
3. $Append(Connector, EI-PI(C))$;
4. $Out(C_i, -) = f(C_i, Connector)$;
5. $Out(C, -) = Out(Connector_2, -)$; and,
6. $InJoin = `\wedge$'.

The build time duration of the new component C can be computed as follows:

$$D(C) = Max\{D(C_0) + D(f(C_0, C_i)) + D(C_i) + D(f(C_i, Connector)) + D(Connector)\}$$

for $i \in \{1, \ldots, n\}$.

The component $C_i$ ($i \in \{1, \ldots, n\}$) can be executed in either the time sequential order or the parallel order according to the time restriction. But they need to satisfy time restrictions $Time_B(C_0) < Time_B(C_i)$ for $i = 1, \ldots, n$. If we need to keep the parallel execution between $C_1, \ldots, C_n$ in time order, the time restriction can be set as: $Max\{Time_B(C_1), \ldots, Time_B(C_n)\} < Min\{Time_E(C_1), \ldots, Time_E(C_n)\}$. The begin time and the end time are defined by $Time_B(C) = Time_B(C_0)$ and $Time_E(C) = Time_E(Connector)$. The run time duration of C can be computed by $D_r(C) = Time_E(Connector) - Time_B(C_0)$, and $D_r(C) \in [D_{min}(C), D_{max}(C)]$.

The 'Xor–split' connection is similar to the 'and–split' connection except for the formation of the output flow of the connector: $Out(C, -) = Out(Con-$
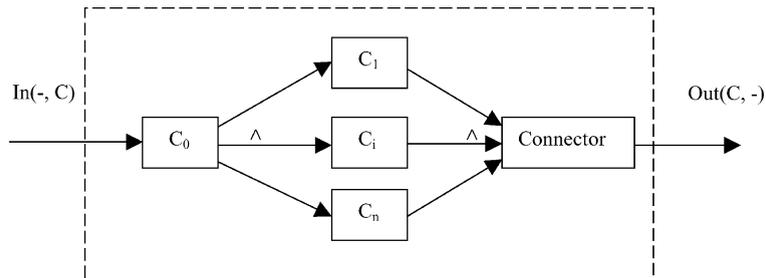


Fig. 4. 'And–split' connection.

nector, $-) = \mathrm{Out}(C_i, -)$ where $C_i$ $(i = 1, \ldots, n)$ is selected in 'or–split' case.

## 3.5. 'And–join' connection and 'Xor–join' connection

The 'and–join' connection requires a connector to carry out the 'and–split' of the input flow of the new compound component as shown in Fig. 5. Its function is the same as that of $\mathrm{Connector}_1$ in the parallel connection case. Its execution condition is to receive the flow that includes all (at least one in 'Xor–join' connection case) of the flows required by $C_i$ $(i = 1, \ldots, n)$.

The access interface of the new compound component can be created by two assignments: $\mathrm{Out}(C, -) = \mathrm{Out}(C_0, -)$, and $\mathrm{Type}(\mathrm{In}(-, C)) = \mathrm{Type}(\mathrm{In}(-, C_1)) \cup \ldots \cup \mathrm{Type}(\mathrm{In}(-, C_n))$. The data updating of the execution interface consists of the following five steps:

1. Append(Connector, EI-PI(C));
2. OutSplit(Connector) = '∧';
3. Out(Connector) = $\{C_1, \ldots, C_n\}$;
4. $\mathrm{Out}(C_i, -) = f(C_i, C_0)$ for $i = 1, \ldots, n$; and,
5. InJoin($C_0$) = '∧'.

The build time duration of the new component C can be computed as follows:

$$D(\mathrm{C}) = \mathrm{Max}\{D(\mathrm{Connector}) + D(f(\mathrm{Connector}, C_i)) \\ + D(C_i) + D(f(C_i, C_0)) + D(C_0)\},$$

for $i \in \{1, \ldots, n\}$.

The begin time and the end time are defined by $\mathrm{Time_B}(C) = \mathrm{Time_B}(\mathrm{Connector})$ and $\mathrm{Time_E}(C) = \mathrm{Time_E}(C_0)$, respectively. The run time duration of C can be computed by $D_r(C) = \mathrm{Time_E}(C_0) - \mathrm{Time_B}(\mathrm{Connector})$, and $D_r(C) \in [D_{\min}(C), D_{\max}(C)]$. $C_i$ $(i \in \{1, \ldots, n\})$ can be executed in time sequential order (or in parallel order) according to the time restriction. But they need to satisfy the following time restriction $\mathrm{Time_E}(C_i) < \mathrm{Time_E}(C_0)$ for $i = 1, \ldots, n$.

The 'Xor–join' connection is similar to the 'and–join' connection except for the formation of the input flow of the connector: $\mathrm{In}(-, C) = \mathrm{In}(-, \mathrm{Connector}) = \mathrm{Out}(-, C_i)$, where $i = 1, \ldots, n$.

## 3.6. Connection rules and component evolution

To keep the internal execution completeness of the new compound workflow component, the connector needs to connect the existing components according to the following four rules.

*C_Rule1*. In 'and–join' connection case, the connector can only split its output flows in 'and–split' way.

*C_Rule2*. In 'Xor–join' connection case, the connector can split its output flows in either 'and–split' way or 'or–join' way.

*C_Rule3*. In 'and–split' connection case, the connector can join its input flows in either 'and–join' way or 'or–join' way.

*C_Rule4*. In 'Xor–split' connection case, the connector can only join its input flows in 'Xor–join' way.

The pure 'sequential', 'parallel', 'and–split', 'and–join', 'or–split', and 'or–join' component connection are called the basic component composition. A more complex connection can be formed by replacing a component node with another component
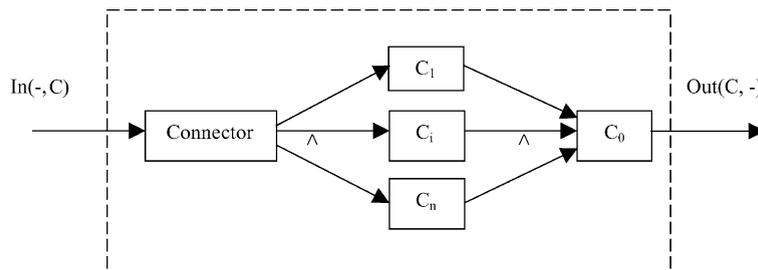


Fig. 5. 'And–join' connection.

whose interface satisfies the interface of the original node. For example, in Fig. 5, $C_1$ can be replaced by a compound component composed of several existing components in the sequential connection way. A component can be evolved through such a task node replacement. Since any component and any compound component can be regarded as a single task, and any replacement of the internal node of a component will not affect the other outside components, a new workflow generated through replacing a task node of a component with another (compound) component will not change the consistency and completeness of the original component (this can be proved by inductive method). This characteristic enables developers to carry out a top-down refinement (expansion) from the high conceptual level to the low implementation level when carrying out the workflow design.

**Characteristic 2**. The replacement of any node of a component does not change the internal consistency and completeness of the original component.

## 4. Workflow component repository

Workflow component repository is a repository that stores and manages workflow components for reuse purpose. A new component can be defined by reusing a certain form of an existing component. We define a reuse–association relationship to reflect such a reuse relationship between workflow components.

### 4.1. Reuse–association relationship between workflow components

The reuse–association relationship is a kind of reuse relationship between workflow components. If a workflow component C′ is defined by reusing another workflow component C, then we say C′ reuses C. C′ and C are called the descendant and the ancestor of the reuse–association, respectively. A workflow component reuse–association relationship includes the following three types.

(1) *Identical reuse*. This type of reuse enables a descendant to reuse its ancestor through a complete
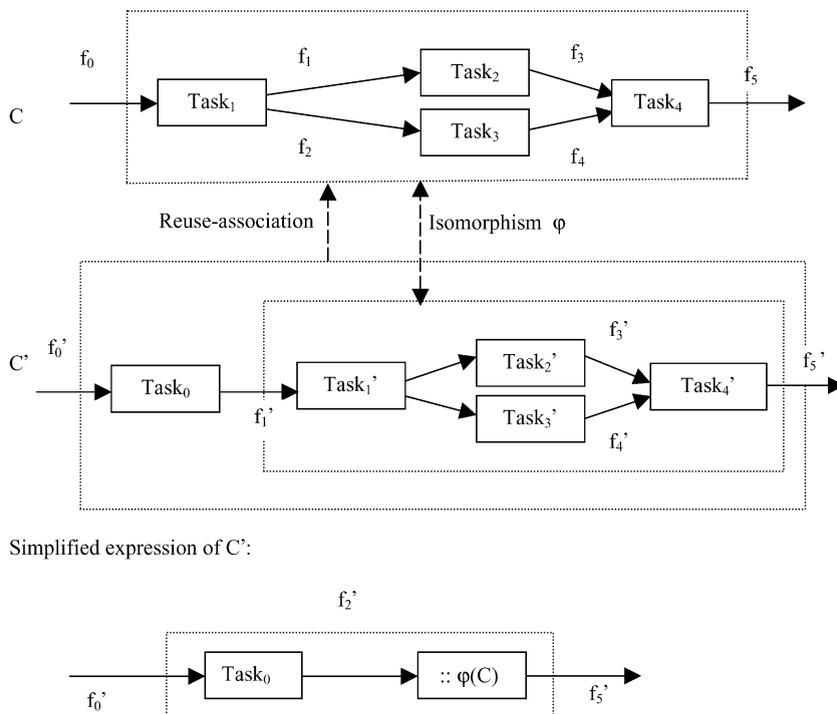


Fig. 6. An example of the reuse–association relationship.

copy. In this case, the reused part of the descendant can be represented by a mark ':: AncestorID'.

(2) *Specialisation reuse*. This type of reuse enables a descendant to reuse a specialisation form of its ancestor. The tasks and flows of the descendant (component) are the subtypes of the corresponding tasks and flows of the ancestors. In this case, the descendant will specialise the data types of the access interface and the execution interface.

(3) *Isomorphism reuse*. This type of reuse enables a descendant to reuse the structure of its ancestor. The types of tasks and flows can be changed through a mapping $\varphi$. In this case, the descendant will redefine the data types of the execution interface.

To guarantee the completeness and the integration of the descendant component, the descendant can only be allowed to reuse a complete workflow process of its ancestor (i.e., partially reuse a basic component is not allowed). A new workflow component can be created by reusing an existing workflow component and then making necessary extension, which can be realised by connecting it with the other components in the basic connection ways. Fig. 6 graphically shows an example that component C′ reuses C in an isomorphism way.

### 4.2. Basic structure of workflow component repository

The basic structure of the workflow component repository is a set of workflow component reuse–association hierarchies. Each reuse hierarchy consists of several levels. The top level of each hierarchy is a basic workflow component. The reuse relationship does not exist between the basic workflow components. A low-level workflow component is related to a high-level workflow component through a proper type of reuse–association. A workflow component is specified in the workflow component repository by the following frame:

ComponentName:: <AncestorName, ReuseAssociationType>;
[ ConceptualLevel: Process: $<N, F>$;
                  InOutType: <In: Type, Out: Type>;
                  CRes: RestrictionRepresentations;
                  BusinessLink: <BusinessId, CategoryDescription>;
  SystemLevel: EI-AI: InDataStru→OutDataStru;
               EI-PI: ExecutionDataStru;
               EI-RI: <RoleInterface, RoleStru>;
               SRes: RestrictionRepresentations;
               ApplicationLink: ApplicationInterfaceDataStru]

In the above frame, 'ComponentName' is a coded string reflecting the classification of the workflow component, 'AncestorName' and 'ReuseType' describe the relationship between the component and the reuse ancestor, and the reuse–association type. At the conceptual level, $\langle N, F \rangle$ represents the conceptual workflow process of the workflow component, where $N$ denotes the task node set, and $F$ denotes the flow set. BusinessId represents the business category identity that corresponds to the workflow component. With the BusinessId, a workflow component can be related to a business classification hierarchy. 'CRes' specifies the conceptual general restriction. At the system level, the access interface (EI-AI) defines the mapping from the input data structures into the output flow data structures. 'SRes' defines the system level restrictions.

To facilitate the workflow component reuse, the basic structure of the workflow component repository can be extended according to the application requirement. In our approach, the workflow component repository includes the following three structures:

(1) A set of 'is–part–of' hierarchies of domain businesses (denoted as B_IsPartOf);

(2) A set of 'is–a' inheritance hierarchies of domain business components (denoted as BC_IsA); and,

(3) An analogy table (denoted as AnalogyTable) for recording the known mapping from the domain business components into the corresponding work-

flow components. The analogy table includes three fields: domain business name that reflects its category; the workflow component name and its access interface corresponding to the domain business; and, the instance names of the workflow component. A new correspondence relationship should be appended to the table when a new analogy has been successfully used.

### 4.3. Component retrieval

The retrieval operation is a frequently used operation that retrieves the required node (i.e., the business component or the workflow component) in terms of users' query requirement. When the retrieval mechanism searches the 'is–part–of' hierarchy, the 'is–part–of' relationship between the requirement and the candidate node should be checked first. If the candidate is a part of the requirement, then the retrieval mechanism should carry out a bottom-up search, otherwise should carry out a top-down search. When the retrieval mechanism searches the inheritance hierarchy, the retrieval mechanism first needs to check the specialisation or the generalisation relationship between the target and the current node. If the current visiting node is more general than the target, then the retrieval mechanism carries out a searching from the ancestor to the descendant. If the current visiting node is more special than the target node, then the retrieval mechanism carries out a searching from the descendant to the ancestor. The 'is–part–of' relationship and the specialisation relationship between the current visiting node and the requirement specification can be informally determined through man–machine interoperations, or by a set of heuristic rules as introduced in Ref. [17]. The select operation can have the following three interfaces: (1) view browse, browse the components (names) that satisfied a given condition within a view of the whole component repository by moving a scroll bar; (2) key words input, retrieve in terms of several key words describing the characteristics of the required node; and (3) SQL-like command, users' requirement is represented by an SQL-like command like the model retrieval mechanism discussed in Ref. [17].

The other basic operations of the workflow component repository include 'append', 'delete', and 'modify'. To append a new workflow component to

the workflow component repository is to define its framework and to establish the inheritance relationship between the new workflow component and the existing workflow components in the repository. The function Append($C' :: C$, repository) is responsible for appending a component $C'$ that inherits from C to the repository. Any modification of a workflow component may change the inheritance relationship between the workflow component and its ancestor. To delete a workflow component means to delete the workflow component and all of its descendants.

## 5. Strategies for workflow component reuse

The experienced developers can use past experience to solve new problems. Similarly, the workflow component repository enables developers (i.e., repository users) to focus on: the understanding of the new domain business; the establishment of an analogy between the new domain business and a previously investigated domain business (i.e., repository business); and, the establishment of the link to the corresponding workflow components in the repository. Such an analogy determines the effectiveness of workflow component reuse. So strategies are required to support different developers to make an effective workflow component reuse.

### 5.1. Strategy 1: Domain business abstraction

Different developers have different experiences and different skills as well as different understandings of the domain business. The different understandings can lead to reuse different workflow components in the workflow component repository. The abstraction strategy can guide developers to make a proper domain business abstraction. The abstraction strategy consists of the following three steps:

(1) *Top-down business division*. Based on the analysis of domain organisation structure, the top-level domain business should be described first (can be as simple as the main purpose of the business), and then carry out top-down division. The division process carries out until reaching the basic business. As the result of the top-down division, the 'is–part–of' business hierarchy can be established. The hierarchy is similar to the domain organisation architecture.

(2) *Bottom-up business abstraction*. Domain business at the same level of the 'is−part−of' hierarchy can be classified as the business categories. A general domain business component (called root) can be formed for each domain business category. The category members are the instances of the business component. According to the 'is−part−of' relationship between business instances, the 'is−part−of' hierarchy can be established between business components. The 'is−part−of' hierarchy of the business components reflects a kind of abstraction architecture of the domain business.

(3) *Establish domain business component inheritance hierarchy*. Classify the business components into categories, and then generate a representative business component (can be virtual) for each category. The category members are called the 'is−a' descendant of the representative component. Then, continue to classify the representative components until a general representative component (can be null) is formed. As the result, the domain business component inheritance ('is−a' relationship) hierarchy is established. It reflects developers' domain business abstraction, which is useful in identifying new domain business.

## 5.2. Strategy 2: Analogy

Analogy is a mapping association based on abstraction. It solves a new problem through establishing the mapping association between the new problem and a previously solved similar problem [18]. The suggested analogy strategy consists of the following two steps.

(1) *Establish analogy between the new domain business and a repository domain business*. This step is to establish the analogy relationship between the new domain business (i.e., the new problem) and a repository domain business component through comparing the instances of the business component and the new business. Analogy enables a new business component to match an existing business component even though they have some differences in appearance.

(2) *Find the suitable workflow components from repository*. Through referring to the correspondence relationship between domain business components (selected by the first step) and their workflow components, developers can find the suitable workflow components from the repository for composing the solution to the new problem.

As human intelligent problem-solving skills, abstraction and analogy are usually used in a combination way when solving problems. For example, a new domain business may not be able to be directly analogous to an existing domain business, while it sometimes could after abstraction. The combination can take either the abstraction first form or the analogy first form [18]. Developers are encouraged to learn from the workflow component repository about the corresponding relationship between the domain business and the workflow components before carrying out the design. Abstraction and analogy are the keys to form and to reuse workflow components at different abstraction levels.

## 5.3. Strategy 3: Workflow component refinement

Workflow component refinement is to generate more abstract workflow components from the existing workflow components through human analogy and abstraction. The refinement process consists of two steps. The first step is to establish the process isomorphism between two candidate workflow components, which can be selected by establishing the analogy between their businesses. The second step is to check whether the mapping can satisfy the restrictions of the two workflows. If the mapping cannot satisfy these restrictions, then the refinement is not successful and the refinement process will return to the first step, otherwise generate a new workflow component that is isomorphism to the existing two workflow components, and name the new workflow component and all its task nodes and flows by more abstract names in domain ontology. The existing two components is the isomorphism reuse−association descendant of the new workflow component. For example, the 'patient see doctor' workflow process of different hospitals can be abstracted as the same workflow component. Through abstraction and analogy, we can further establish the analogy relationship between the 'patient see doctor' workflow process and the 'test product' workflow process, and then generate a more abstract workflow component named as 'object test object', where the 'object' is the abstraction of 'product' and 'human' (either patient or doctor).

## 6. Development process

### 6.1. Component-based workflow definition process

A workflow definition process can be regarded as a problem-solving process. The new domain business is regarded as the new problem to be solved, and the workflow system corresponding to the new problem is the solution. The problem-solving process for the new workflow definition as described in Fig. 7 consists of the following steps:

(1) *Problem analysis through abstraction and analogy.* This step establishes the 'is−a' business component inheritance hierarchy and the 'is−part−of' business component hierarchy for the problem as stated in Strategy 1.

(2) *Component retrieval.* By referring to the analogy table, developers can retrieve the existing domain business components corresponding to the new business components from the workflow component repository, and then find the workflow components corresponding to these existing business components.

(3) *Design.* To design those special workflow components that the repository does not support, and then to append them to the corresponding business component hierarchy and the workflow component hierarchy. The analogy table needs to be updated after design.

(4) *Composition.* To form the solution by composing the available workflow components according to the 'is−part−of' relationship between business components. The composed workflow can be normalised as a big component and then appended to the workflow component repository. Its business component inheritance hierarchy needs to be appended to the repository, and the analogy table needs to be updated.

(5) *Set restrictions.* New restrictions (e.g., time restriction) should be set and attached to the new compound workflow according to the whole business requirement.

### 6.2. Execution platform components

The execution platform is to perform a workflow (component) in terms of the mapping from the
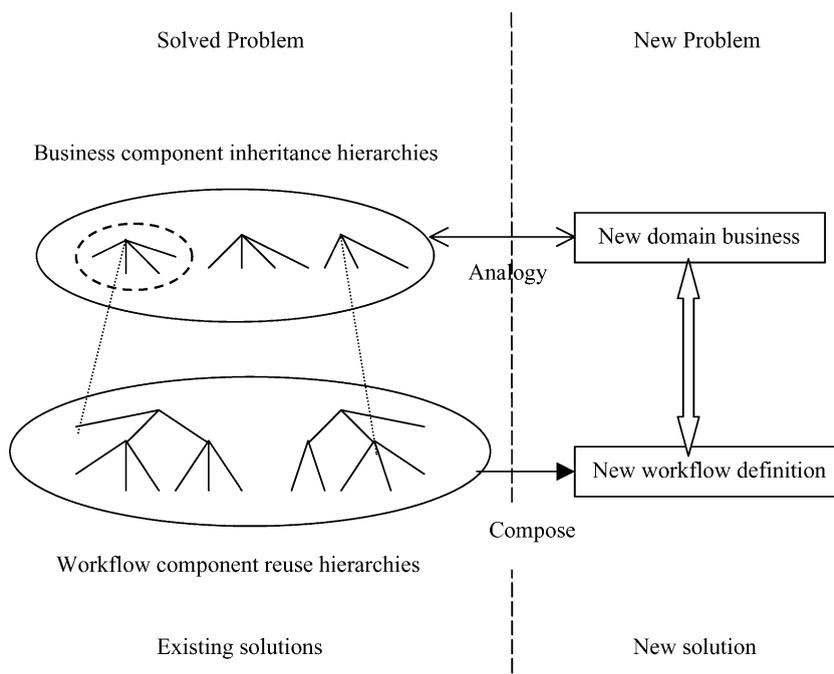


Fig. 7. Problem-solving-based workflow system definition process.

component execution interface into the execution platform interface. The platform mainly consists of four components: (1) a router that is responsible for determining the execution order between tasks in terms of the workflow process definition; (2) a process handler that is responsible for managing all the process instances (including their creation, starting, suspending, and termination); (3) a task handler that is responsible for managing all the task instances (including their creation, starting, abortion, suspending, and assignment); and (4) an application handler that is responsible for invoking and terminating applications, as well as requesting the status of applications. The other related components include: the time manager, the transaction manager, the security manager, and the work list handler. These software components can be organised as an execution support component repository. New components can be appended to the repository when necessary. The execution platform of a workflow component can be composed by selecting the suitable components from the repository in terms of different kinds of application requirements. For example, the repository can include multiple routers that are suitable for multiple kinds of application requirements. This can increase the flexibility of the execution platform.

### 6.3. Component-based workflow execution process

A component-based workflow is a nest workflow component hierarchy, where the task node at the high level can be either a simple task or a nest node that corresponds to a low-level workflow component. The top-level workflow can be regarded as the general workflow process that reflects the general domain business. When a high-level nest task node is activated, it passes the input flow to the corresponding low-level workflow component as its input flow and triggers the component. When terminating, the component passes its output flow back up to the corresponding high-level nest task node as its output flow.

The nest workflow component hierarchy is isomorphism to the 'is–part–of' business component hierar-chy, which is similar to the domain business organisation hierarchy. *The nest workflow component hierarchy is different from the traditional subprocess*

*nest workflow.* First, a traditional workflow subprocess is not encapsulated, and any of its tasks are accessible by any external flow, so more cost is involved to guarantee execution completeness when it is shared by different processes, e.g., some of its tasks have been suspended by one process when another process requires them to run. Second, a subprocess does not have the abstraction and reuse support mechanism, while a workflow component is supported by the domain inheritance rela-tionship and the reuse–association relationship. A workflow component can be regarded as the abstraction of several workflow subprocesses.

From the users' point of view, the nest workflow enables users with different ranks to work at different levels. Users at higher levels can only concern the input flow and the output flow of the nest task node at the current working level, and let the users with lower ranks work on the low-level workflows corresponding to the nest task node.

## 7. Application

### 7.1. Purpose and method

The proposed approach has been applied to the development of the MISs for a large number of different financial businesses in some cities of China [36] and the HISFLOW project in Singapore [37]. This application includes two purposes: one is to enhance the development efficiency of this particular application field, and the other is to test the effectiveness of the proposed approach through the application.

The financial businesses of this application field can be classified into several general categories through domain business abstraction as discussed in Ref. [36]. The businesses within the same category are similar to each other. At the first stage, we chose one category to develop the workflow-based MISes. We use the proposed approach to design the related workflow components and the application relevant components. At the second stage, we developed a development support mechanism, where the component resources came from the first stage development. At the third stage, we developed the workflow-based MISes for the other categories by using the support mechanism.

## 7.2. Support system development and application

The development platform consists of five main parts as shown on the left part of Fig. 8: (1) *a workflow component repository*; (2) *a data structure dictionary* for recording all the data elements and structures of the access interface and the execution interface of workflow components; (3) *an execution support component repository* that includes all the necessary components for constituting the execution platform; (4) *an application support component repository* that includes all the application software components; and (5) *a management mechanism* that is responsible for managing the former four parts. This system can assist developers to generate the new workflow system and the related application as shown in the middle part and the right part of Fig. 8. According to the mass development characteristic of this application, all the interfaces of the execution platform are based on the same data structure, so EI-to-EPI mapping is not necessary herein.

Four kinds of generations will be carried out (as shown in the block arrows in Fig. 8) during the

development process: (1) to generate the conceptual level of the new workflow based on the composition of the conceptual level of the existing workflow components in the workflow component repository; (2) to generate the execution interface based on the data dictionary; (3) to generate the execution engine based on the execution support component repository; and (4) to generate the application of the workflow system.

The developers and domain users are working at the cognitive level. Developers are responsible for analysing the domain business and for operating the management mechanism to assist the analysis and the system development. Domain users work on the user interface of the application system like a virtual working environment.

## 7.3. Application implication

The first implication of this application is that the effectiveness of using the proposed development approach mainly depends on three aspects: (1) the rate of the repository components overlapping the domain business (the effectiveness will rise with the
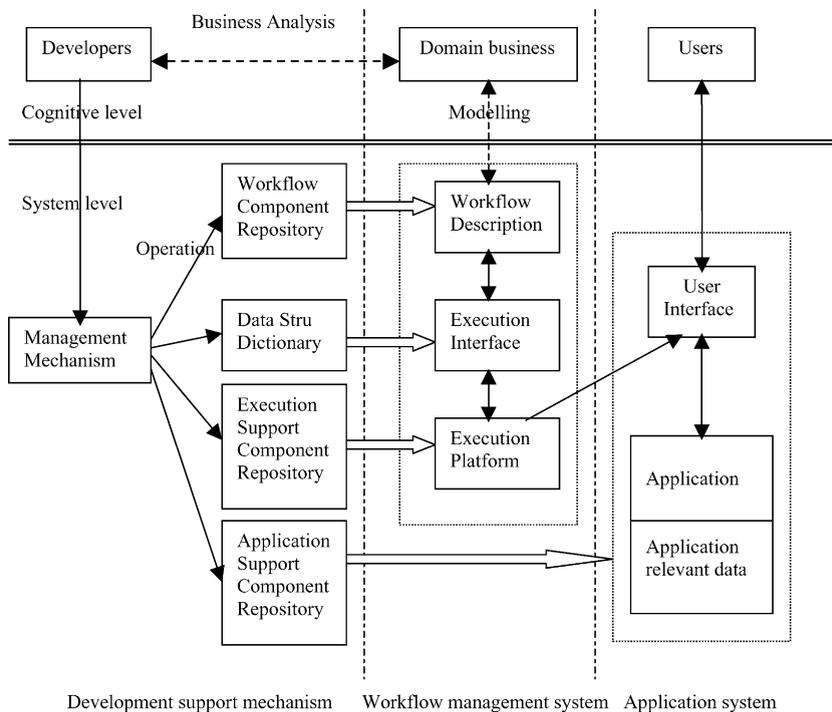


Fig. 8. Component-based workflow development platform.

increasing number of the developed workflow systems); (2) the degree of developers' familiarity with the repository content (the newly joined developers will slow down the development efficiency); and (3) the skills of using the abstraction and analogy strategies. The other factors that influence the development efficiency, e.g., the team cooperation and the project management, are beyond the scope of this discussion.

The second implication is that the quality of the developed component-based workflow system depends on the design quality of the repository workflow components and the correctness of component composition. All repository components should be carefully designed and checked by domain experts or by simulation.

The third implication is that the component-based workflow can simplify the complexity in realising the interorganisational workflow because of the encapsulation and the autonomous characteristic of workflow components.

## 8. Discussions and comparisons

### 8.1. Advantages of the component-based workflow

Compared with the traditional workflow, the component-based workflow system has the following advantages:

(1) *Decrease the complexity of workflow process*. A complex workflow can be transformed to a workflow component hierarchy, where each workflow component has the lower complexity than the whole workflow, and the complexity of the low level is hidden at the higher level of the hierarchy.

(2) *Increase the reusability and the connectivity*. A workflow component can be safely reused by any other workflow component or tasks through its access interface. Users do not need to be concerned about its internal implementation and execution. It also enables different components to be easily connected.

(3) *Error localisation and autonomous business*. Workflow definition errors and execution errors are localised within the component. It enables the checking mechanism to localise errors. Workflow component also enables its business to be performed autonomously.

(4) *Increase adaptability*. Domain business usually needs to be adjusted according to market changes. In a traditional workflow, a small change of domain business may lead to redesign of the whole workflow or thoroughly check the whole workflow after modification. The component-based workflow can adapt to the domain business change, because any modification of one component or any component addition will not influence the other components.

(5) *Run time maintainable*. Since a component can be independently executed, it can be maintained in run time if the maintenance can be finished before its execution. This can increase the adaptability of the workflow.

(6) *Users' acceptability*. Firstly, a workflow component can localise domain business, so it can be more easily checked and used. Secondly, the component-based workflow hierarchy enables users with different ranks (or different experience) to work on different levels. Thirdly, the workflow hierarchy is similar to the domain business hierarchy, so it can be more easily understood.

### 8.2. Related works on reuse and comparison

Software component reuse has been widely studied [2,9,13,14,16]. The traditional solutions include three aspects: (1) methods based on formal semantics like algebraic, operational, and denotation semantic methods; (2) informal methods like information retrieval based on the storage and the retrieval context of components, the topological methods, and the knowledge-based methods; and (3) hybrid methods. The cognitive aspect of software reuse was also investigated [19,36]. A cognitive-based software process model is proposed to unify the software process and the developers' cognition process based on abstraction and analogy [36]. The model provides a way to improve the software development process by enhancing the developers' cognitive skill. At the system level, component standards like CORBA and COM/DCOM are suggested and have been widely used [3].

Inheritance is an important concept of OOM and OOP [9,18,22]. The black box and the white box inheritance and the inheritance compatibility were discussed in Refs. [4,9,22]. Inheritance has been regarded as the 'is–a' relationship between two con-

cepts in the AI field [7,20]. The transitivity of the 'is–a' relationship forms a kind of common sense reasoning. A multivalued model inheritance was proposed for promoting the flexibility of model retrieval based on a two-level rule-based mathematical model base system RMMBS [31].

Compared with software component reuse approach, the proposed workflow component reuse has two characteristics. First, it is a grey box reuse. Designers can reuse a workflow component by not only checking its interface and the textual (language) description but also by knowing its process logic. One reason is that we not only need to guarantee the integration and the correctness of a workflow component through the black box reuse, but we also need to enable users to understand its semantics through the process scenario and the interface so as to make a proper analogy of the new business. Second, the proposed reuse approach integrates the characteristics of software component, the component repository mechanism, and the problem-solving strategies. It includes both the problem aspect (i.e., the domain business) and the solution aspect (i.e., the workflow component) of an entire reuse process, while the other reuse approaches mainly focus on the solution aspects of reuse (i.e., how to use a solution). The abstraction and analogy strategies also increase the creativity in the reuse process.

## 9. Concluding remarks

The paper proposes a component-based workflow development approach for raising the efficiency of large workflow systems development through integrating the characteristics of software component, the component repository mechanism, and the problem-solving strategies. The approach includes: the definition of workflow component, the methods for workflow component composition, and the workflow component reuse–association relationship, a component repository mechanism, a set of abstraction and analogy strategies, and a component-based development process. The approach provides a methodology, a tool, and the guidelines for different developers either in different teams or in the same team to design and to reuse workflow components in a more effective and efficient way.

Compared with traditional workflow development approaches, the proposed approach emphasizes on the accumulation of reusable experiences (i.e., the business analysis experiences) and the reusable resources (i.e., the workflow components and the application support components) during the development process, so its development effectiveness will keep on improving with the increasing number of the developed projects. The development efficiency will be better than the traditional approaches when the reusable experience and resources can overlap a bigger percentage of the new domain business. It is especially effective when used in similar domains or used for resource sharing in team development. Besides, the component-based workflow has the following advantages: lower complexity, reusability, adaptability, connection ability, maintainability, error localization ability, and users acceptability.

The ongoing work concerns two aspects. The first is to develop a tutorial mechanism that can help developers understand the concept of workflow components, the method of workflow composition, the workflow component repository, and the development strategies before carrying out the development. The second is to apply the proposed approach to develop the knowledge flow management mechanism based on the Knowledge Grid VEGA-KG [32–35] and to develop the service flow management mechanism based on the Service Grid. Up-to-date progress is shown at http://kg.ict.ac.cn.

## Appendix A. Represent time factor in workflow

In order to represent time and time order, we use $D(x)$ and $D_r(x)$ to represent the static (build time) duration and the run time duration of $x$, respectively ($x$ can be either a task or a flow), and use $Time_B(x)$ and $Time_E(x)$ to represent the begin time and the end time (restriction) of $x$, respectively, and use ' $\leq$ ' and ' $<$ ' to denote the temporal order between two time variants,

e.g., $\text{Time}_B(x) < \text{Time}_E(x)$ represents that the begin time of $x$ is before its end time.

Three types of time restriction can be set according to the domain business requirement. The first type is the time point restriction, which takes the following three forms: (1) TimeVariant = TimePoint; (2) Time-Variant > TimePoint; and (3) TimeVariant < TimePoint. 'TimePoint' denotes an exact time value with a certain time granularity like date and hour. The second type is the time interval, i.e., the time region between two time points, e.g., from 10:00 a.m. to 3:00 p.m. The third type is the duration restriction, which takes the following three forms: (1) Duration = TimeDuration; (2) Duration > TimeDuration; and (3) Duration < Time-Duration. 'Duration' denotes the time duration between two time checkpoints, and 'TimeDuration' denotes an exact time duration value like 3 h and 3 days, etc. Usually, time checkpoints can be set in the following places: the begin time of a task (or component), the end time of a task (or component), the begin time of a flow, and the end time of a flow.

With the time restriction, workflow can be more flexible to suit real application. For example, $f_1$ and $f_2$ are two flows 'and–joining' $\text{task}_j$. The execution condition of $\text{task}_j$ is the arrival of $f_1$, but $f_2$ is allowed to delay until some time during the execution process (assume a certain time unit $\delta$ can be delayed after the task execution). Hence, time restriction in this case can be relaxed as: (1) $\text{Time}_E(f_1) < \text{Time}_B(\text{task}_j)$; and (2) $\text{Time}_E(f_2) < \text{Time}_B(\text{task}_j) + \delta$.

## References

[1] A. Basu, R.W. Blanning, Metagraphs in workflow support systems, Decision Support Systems 25 (1999) 199–208.

[2] D. Batory, S. O'Malley, The design and implementation of hierarchical software systems with reusable components, ACM Transactions on Software Engineering and Methodology 1 (4) (Oct. 1992) 355–398.

[3] CORBA, The OMG Management Architecture, The Object Management Group, http://www.omg.org/corba.

[4] S.H. Edwards, Representation inheritance: a safe form of "White Box" code inheritance, IEEE Transactions on Software Engineering 23 (2) (Feb. 1997) 83–92.

[5] D. Georgakopoulos, M.F. Hornick, F. Manola, Customising transaction models and mechanisms in a programmable environment supporting reliable workflow automation, IEEE Transactions on Knowledge and Data Engineering 8 (4) (Aug. 1996) 630–649.

[6] A. Geppert, D. Tombros, K.R. Dittrich, Defining the semantics of reactive components in event-driven workflow execution with event histories, Information Systems 23 (3/4) (1998) 235–252.

[7] J.F. Horty, R.H. Thomason, D.S. Touretzky, A skeptical theory of inheritance in nonmonotonic semantic networks, Artificial Intelligence 42 (1990) 311–348.

[8] K. Jensen, Colored Petri Nets, Springer-Verlag, Berlin, 1991.

[9] M. Lattanzi, S. Henry, Software reuse using C++ class, the question of inheritance, Journal of Systems and Software 41 (1998) 127–132.

[10] P. Lawrence (Ed.), Workflow Handbook, Wiley, New York, 1997.

[11] F. Leymann, D. Roller, Workflow-based applications, IBM Systems Journal 36 (1) (1997) 102–122.

[12] F. Malamateniou, G. Vassilacopoulos, P. Tsanakas, A workflow-based approach to virtual patient record security, IEEE Transactions on Information Technology in Biomedicine 2 (3) (Sept. 1998) 139–145.

[13] R. Mili, A. Mili, R.T. Mittermeir, Storing and retrieving software components: a refinement based system, IEEE Transactions on Software Engineering 23 (7) (July 1997) 445–460.

[14] G.S. Novak Jr., Software reuse by specialisation of generic procedures through views, IEEE Transactions on Software Engineering 23 (7) (July (1997) 401–417.

[15] J. Puustjarvi, H. Tirri, J. Veijalainen, Reusability and modularity in transactional workflows, Information Systems 22 (2/3) (1997) 101–120.

[16] V. Rajlich, J.H. Silva, Evolution and reuse of orthogonal architecture, IEEE Transactions on Software Engineering 22 (2) (Feb. 1996) 153–157.

[17] Rational, UML Document, Version 1.1, http://www.rational.com/uml/resources/documentaion/, Sept. 1997.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, N.J., 1991.

[19] A. Sen, The role of opportunism in the software design reuse process, IEEE Transactions on Software Engineering 23 (7) (July 1997) 418–436.

[20] L.A. Stein, Resolving ambiguity in nonmonotonic inheritance hierarchies, Artificial Intelligence 55 (1992) 259–310.

[21] W.M.P. Van der Aalst, Petri-net-based workflow management software, in: A. Sheth (Ed.), Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions, Georgia, http://lsdis.cs.uga.edu/activities/NSF-workflow. (May 1996).

[22] P. Wegner, Concepts and paradigms of object-oriented programming, OOPS Messenger 1 (1) (Aug. 1990) 7–87.

[23] WfMC, The Workflow Reference Model, http://www.wfmc.org.

[24] WfMC, Interface1: Process Definition Interchange Process Model, http://www.wfmc.org.

[25] WfMC, Workflow Standard—Interoperability Abstract Specification, http://www.wfmc.org.

[26] WfMC, Workflow Standard—Interoperability Wf-XML Binding, http://www.wfmc.org.

[27] WfMC, Workflow Standard—Interoperability Internet e-mail MIME Binding, http://www.wfmc.org.

[28] WfMC, Workflow Management Application Programming Interface Specification, http://www.wfmc.org.

[29] WfMC, WAPI Naming Conventions, http://www.wfmc.org.

[30] WIDE Consortium, WIDE Workflow Development Methodology, http://dis.sema.es/projects/WIDE/Documents/.

[31] H. Zhuge, Inheritance rules for flexible model retrieval, Decision Support Systems 22 (4) (1998) 379–390.

[32] H. Zhuge, A knowledge grid model and platform for global knowledge sharing, Expert Systems with Applications 22 (4) (2002) 313–320.

[33] H. Zhuge, A knowledge flow model for peer-to-peer team knowledge sharing and management, Expert Systems with Applications 23 (1) (2002) 23–30.

[34] H. Zhuge, VEGA-KG: A Way to the Knowledge Web, 11th International World Wide Web Conference, poster proceeding, available at http://www2002.org, Hawaii, May 2002.

[35] H. Zhuge, Distributed Team Knowledge Management by Incorporating Knowledge Flow with Knowledge Grid, the 2nd International Conference on Knowledge Management, http://www.i-know.at, Graz, Austria, July 2002.

[36] H. Zhuge, J. Ma, X. Shi, Abstraction and analogy in cognitive space: a software process model, Information and Software Technology 39 (7) (1997) 463–468.

[37] H. Zhuge, T.Y. Cheung, H.K. Pung, A timed workflow process model, Journal of Systems and Software 55 (3) (2001) 231–243.

**Hai Zhuge** is a professor at the Institute of Computing Technology, Chinese Academy of Sciences. He received a PhD in computer science from Zhejiang University, China. His current research interests include: semantic-web-based grid and its application, knowledge flow management, problem-oriented model base systems, component reuse mechanism, intelligent process model, interoperation model for group decision, and web-based workflow model. He is now the leader of the China Knowledge Grid project SVEGA-KG which has 20 team members. He is the author of one book and over 40 papers appeared mainly in leading international conferences and the following international journals: *IEEE Transactions on Systems, Man, and Cybernetics*; *Information and Management*; *Decision Support Systems*; *Journal of Systems and Software*; *International Journal of Cooperative Information Systems*; *Knowledge-based Systems*; *Expert Systems with Applications*; *Information and Software Technology*; and *Journal of Software*.