US 20050267765A1

(19) United States

(12) Patent Application Publication

Jeng et al.

(10) Pub. No.: US 2005/0267765 A1

(43) Pub. Date: Dec. 1, 2005

(54) APPARATUS AND METHOD FOR POLICY-DRIVEN BUSINESS PROCESS EXCEPTION HANDLING

(76) Inventors: **Jun-Jang Jeng**, Armonk, NY (US); **Hui Lei**, Scarsdale, NY (US); **Liangzhao Zeng**, Ossining, NY (US); **Hung-yang Chang**, Scarsdale, NY (US); **Santhosh Kumaran**, Croton on Hudson, NY (US); **Jen-Yao Chung**, Yorktown Heights, NY (US)

Correspondence Address:
WHITHAM, CURTIS & CHRISTOFFERSON, P.C.
11491 SUNSET HILLS ROAD
SUITE 340
RESTON, VA 20190 (US)

(21) Appl. No.: **10/853,503**

(22) Filed: **May 26, 2004**

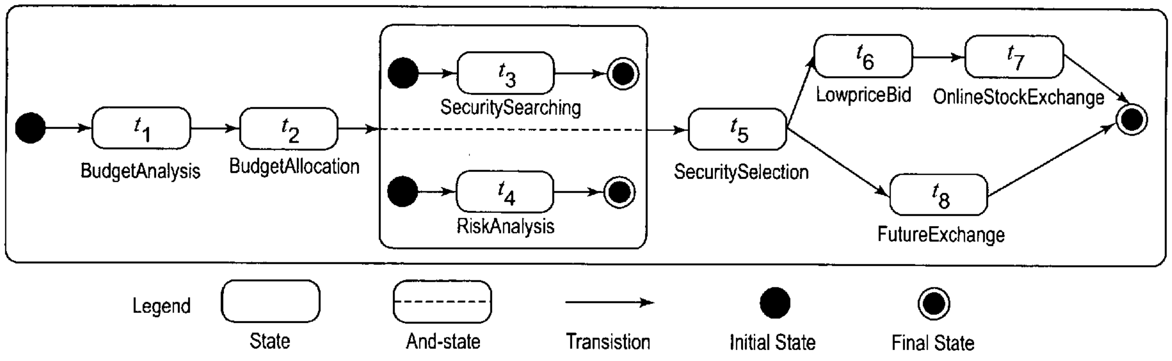
Publication Classification

(51) Int. Cl.[7] **G06F 17/60**

(52) U.S. Cl. **705/1**; 705/10

(57) **ABSTRACT**

A model-driven and QoS-aware infrastructure facilitates the scalable composition of Web services in highly dynamic environments. An exception management framework supports two modes of exception management for business processes, providing a novel policy-driven approach to exception management implemented in the system infrastructure. Exception management is implemented in the system infrastructure, with exception handling policies supplied by individual business processes. Using the exception management framework, developers define exception policies in a declarative manner. Before a business process is executed, the service composition middleware integrates the exception policies with normal business logic to generate a complete process schema. This policy driven-approach can significantly reduce the development time of business processes through its separation of the development of the business logic and the exception handling policies.



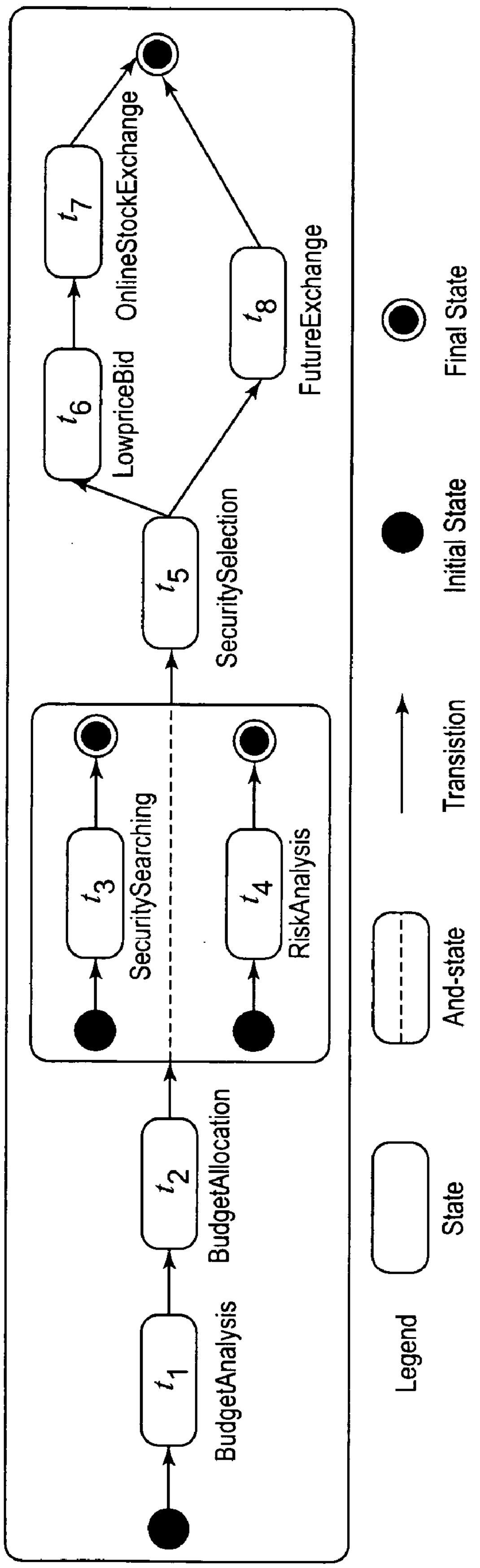

t_1
BudgetAnalysis
t_2
BudgetAllocation
t_3
SecuritySearching
t_4
RiskAnalysis
t_5
SecuritySelection
t_6
LowpriceBid
t_7
OnlineStockExchange
t_8
FutureExchange
Legend
State
And-state
Transistion
Initial State
Final State

Figure 1

```
inst oblig policyName "{"
on event-specification;
subject [<type>]domain-Scope-Expression;
[target [<type>]domain-Scope-Expression;]
do exception-management-action-list;
[when constraint-Expression ;]
"}"
```

Figure 2

```
inst oblig Timeout {
on $delayDuration >= 120 sec;
subject SecurityInvestment;
target ∀t ∈ {RiskAnalysis, SecuritySelection};
do timeout(t);
}
```

Figure 3

```
inst oblig Retry{
on E(timeout,t) ∨ E(failure,t);
subject SecurityInvestment;
target ∀t ∈ {FutureExchange, StockExchange };
do retry(t,2, alternative);
}
```

Figure 4

```
inst oblig Multiple Recovery {
subject SecurityInvestment;
target ∀t ∈ {SecuritySearching};
do multiple(t,3);
}
```

Figure 5

```
inst oblig Replacement Recovery {
on (E(timeout,t) ∨ E(failure,t)) ∧ ($retryTimes>=
   $retryUpperBound) ;
subject SecurityInvestment;
target ∀t ∈ {OnlineStockExchange};
do replace(OnlineStockExchange, ManualStockExchange);
}
```

Figure 6

```
inst oblig Skip Policy {
on E((timeout,t) ∨ E(failure,t)) ∧ ($retryTimes>=
   $retryUpperBound) ;
subject SecurityInvestment;
target ∀t ∈ {LowPriceBid};
do skip(t);
}
```

Figure 7

```
inst oblig Rollback {
on (E(timeout,t) ∨ E(failure,t)) ∧
   ($retryTimes>=$retryUpperBound) ;
subject SecurityInvestment;
target ∀t ∈ {FutureExchange,SecuritySearching};
do rollback(BudgetAnalysis);
}
```

Figure 8

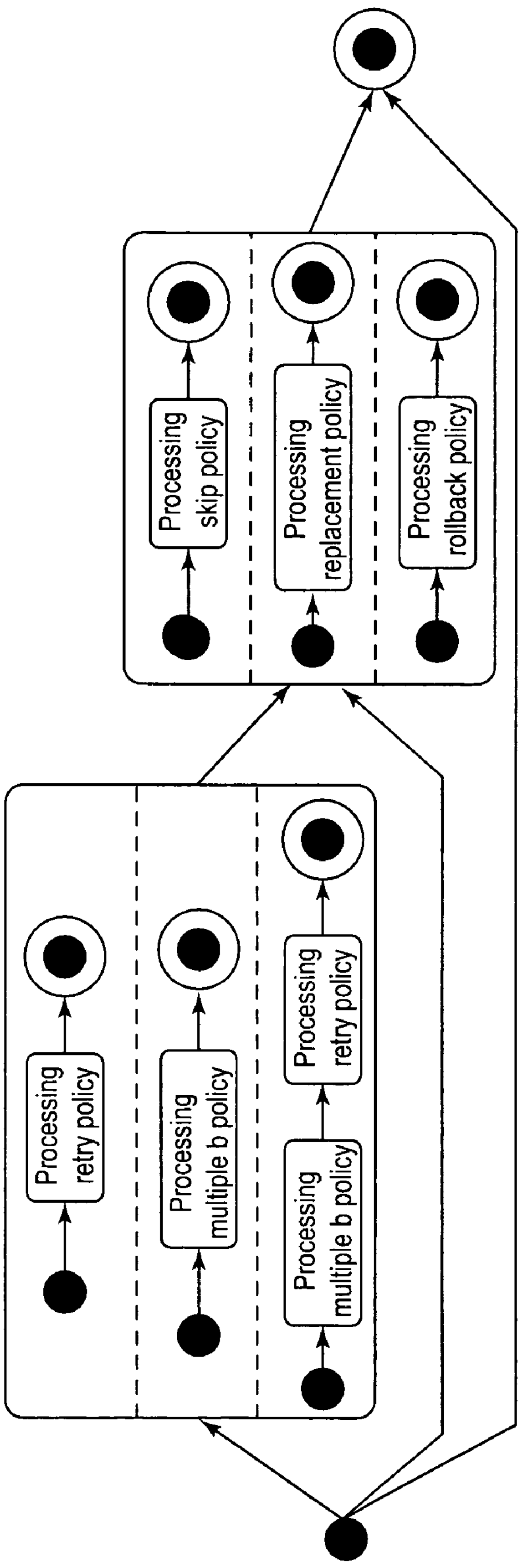

Processing retry policy
Processing multiple b policy
Processing multiple b policy
Processing retry policy
Processing skip policy
Processing replacement policy
Processing rollback policy

Figure 9

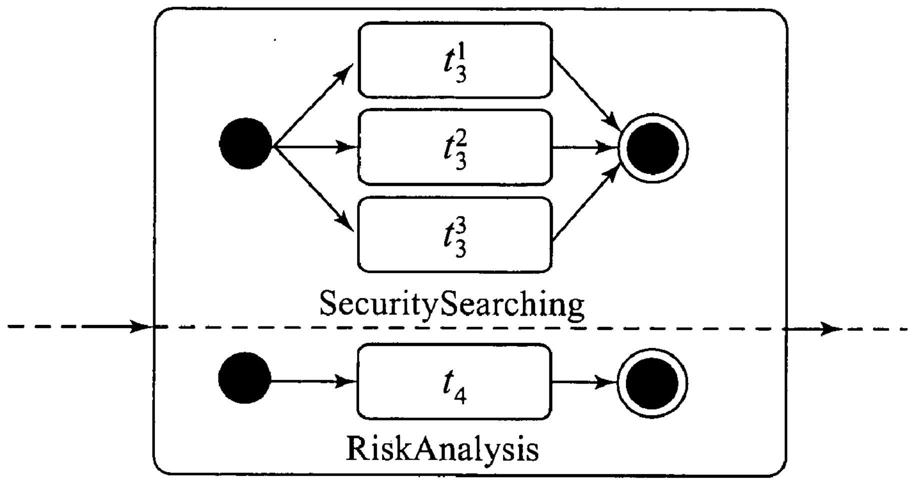


Figure 10

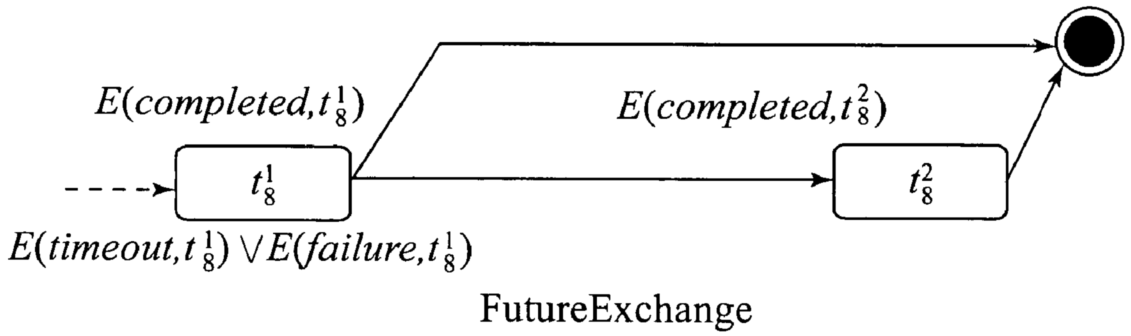


Figure 11

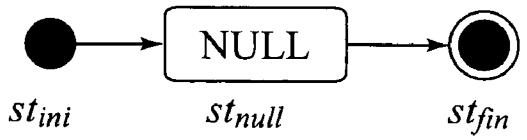


Figure 12

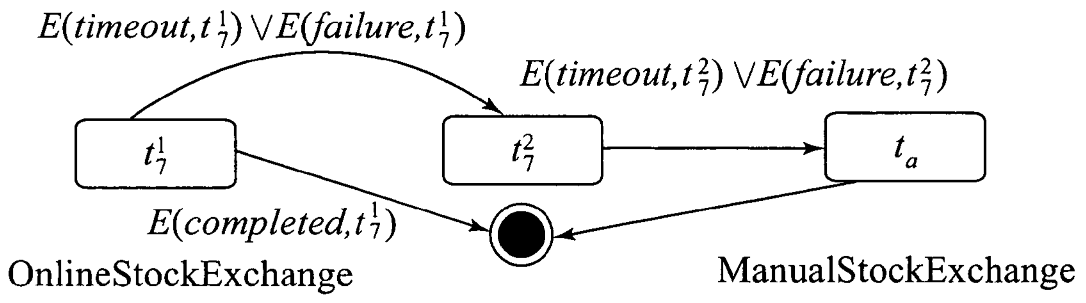

$E(timeout,t_7^1) \vee E(failure,t_7^1)$
$E(timeout,t_7^2) \vee E(failure,t_7^2)$
t_7^1
t_7^2
t_a
$E(completed,t_7^1)$
OnlineStockExchange
ManualStockExchange

Figure 13

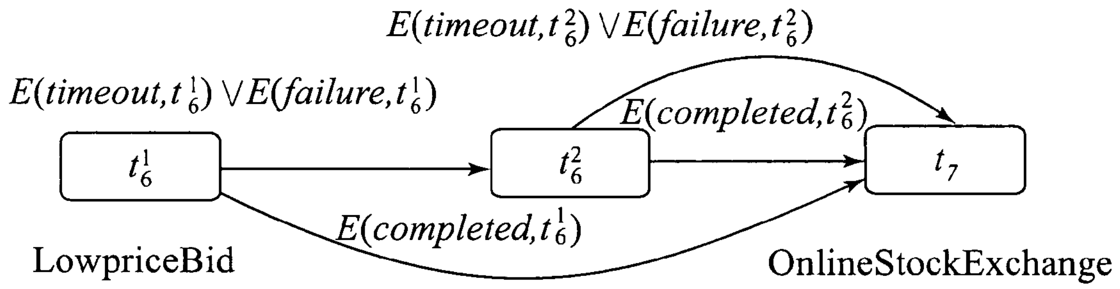


Figure 14

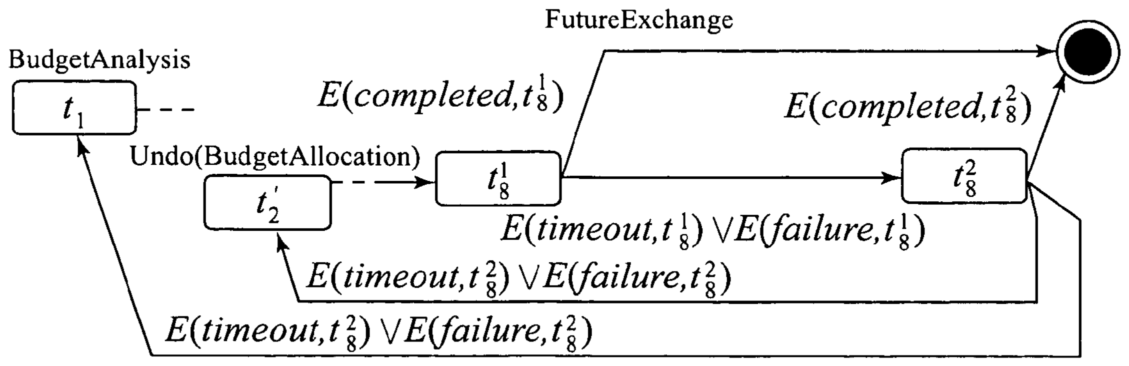
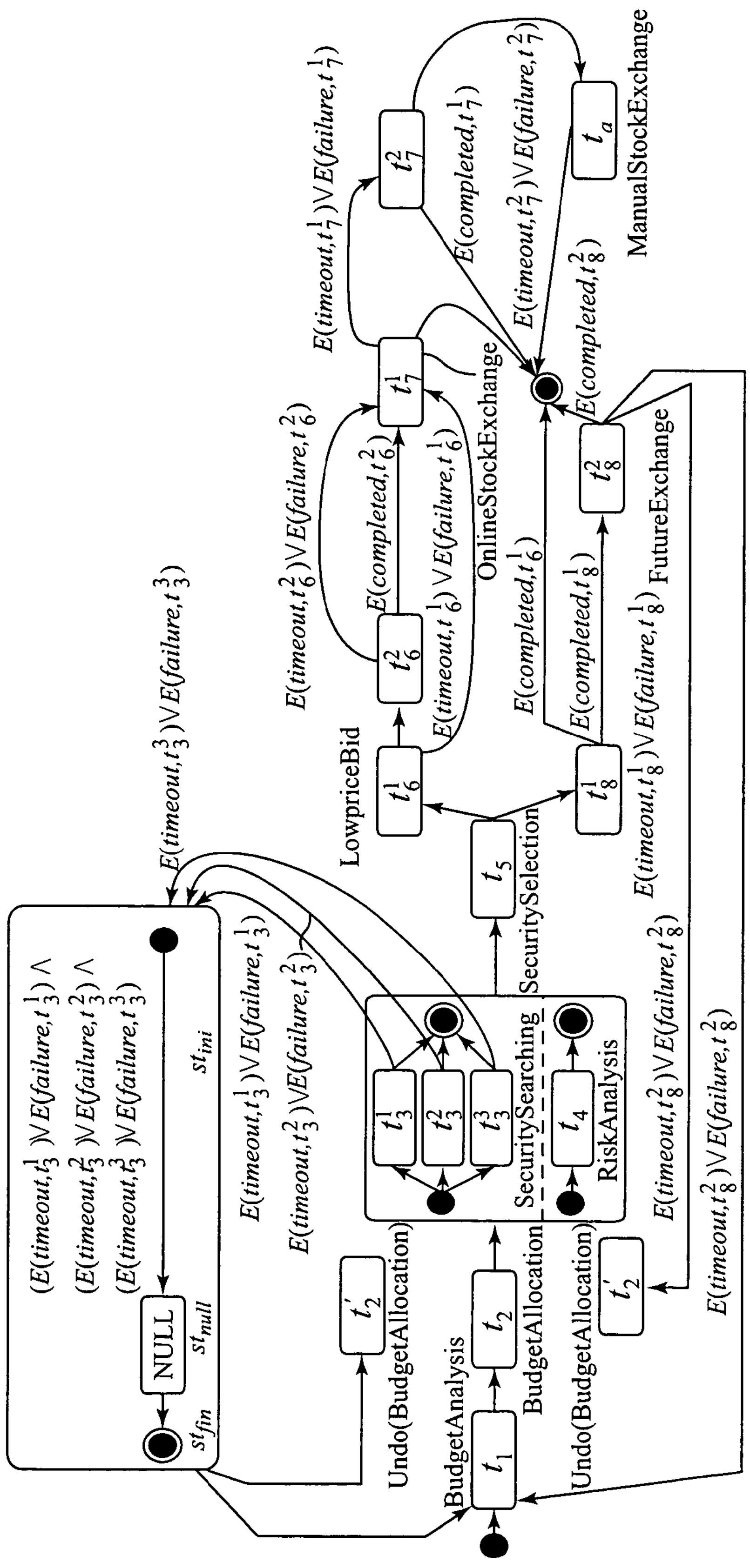

FutureExchange
BudgetAnalysis
t_1
$E(completed,t_8^1)$
$E(completed,t_8^2)$
Undo(BudgetAllocation)
t_8^1
t_8^2
t_2'
$E(timeout,t_8^1) \vee E(failure,t_8^1)$
$E(timeout,t_8^2) \vee E(failure,t_8^2)$
$E(timeout,t_8^2) \vee E(failure,t_8^2)$

Figure 15



$(E(timeout,t_3^1) \vee E(failure,t_3^1)) \wedge$
$(E(timeout,t_3^2) \vee E(failure,t_3^2)) \wedge$
$(E(timeout,t_3^3) \vee E(failure,t_3^3))$
NULL
st_{null}
st_{fin}
st_{ini}
$E(timeout,t_3^3) \vee E(failure,t_3^3)$
$E(timeout,t_3^1) \vee E(failure,t_3^1)$
$E(timeout,t_3^2) \vee E(failure,t_3^2)$
t_2'
Undo(BudgetAllocation)
BudgetAnalysis
t_1
t_2
BudgetAllocation
t_3^1
t_3^2
t_3^3
SecuritySearching
t_4
RiskAnalysis
Undo(BudgetAllocation)
t_2'
$E(timeout,t_8^2) \vee E(failure,t_8^2)$
$E(timeout,t_8^2) \vee E(failure,t_8^2)$
t_5
SecuritySelection
LowpriceBid
t_6^1
t_6^2
$E(timeout,t_6^2) \vee E(failure,t_6^2)$
$E(completed,t_6^2)$
$E(timeout,t_6^1) \vee E(failure,t_6^1)$
OnlineStockExchange
t_7^1
$E(timeout,t_7^1) \vee E(failure,t_7^1)$
t_7^2
$E(completed,t_7^1)$
$E(timeout,t_7^2) \vee E(failure,t_7^2)$
t_a
ManualStockExchange
$E(completed,t_6^1)$
t_8^1
$E(completed,t_8^1)$
$E(timeout,t_8^1) \vee E(failure,t_8^1)$
FutureExchange
t_8^2
$E(completed,t_8^2)$

Figure 16

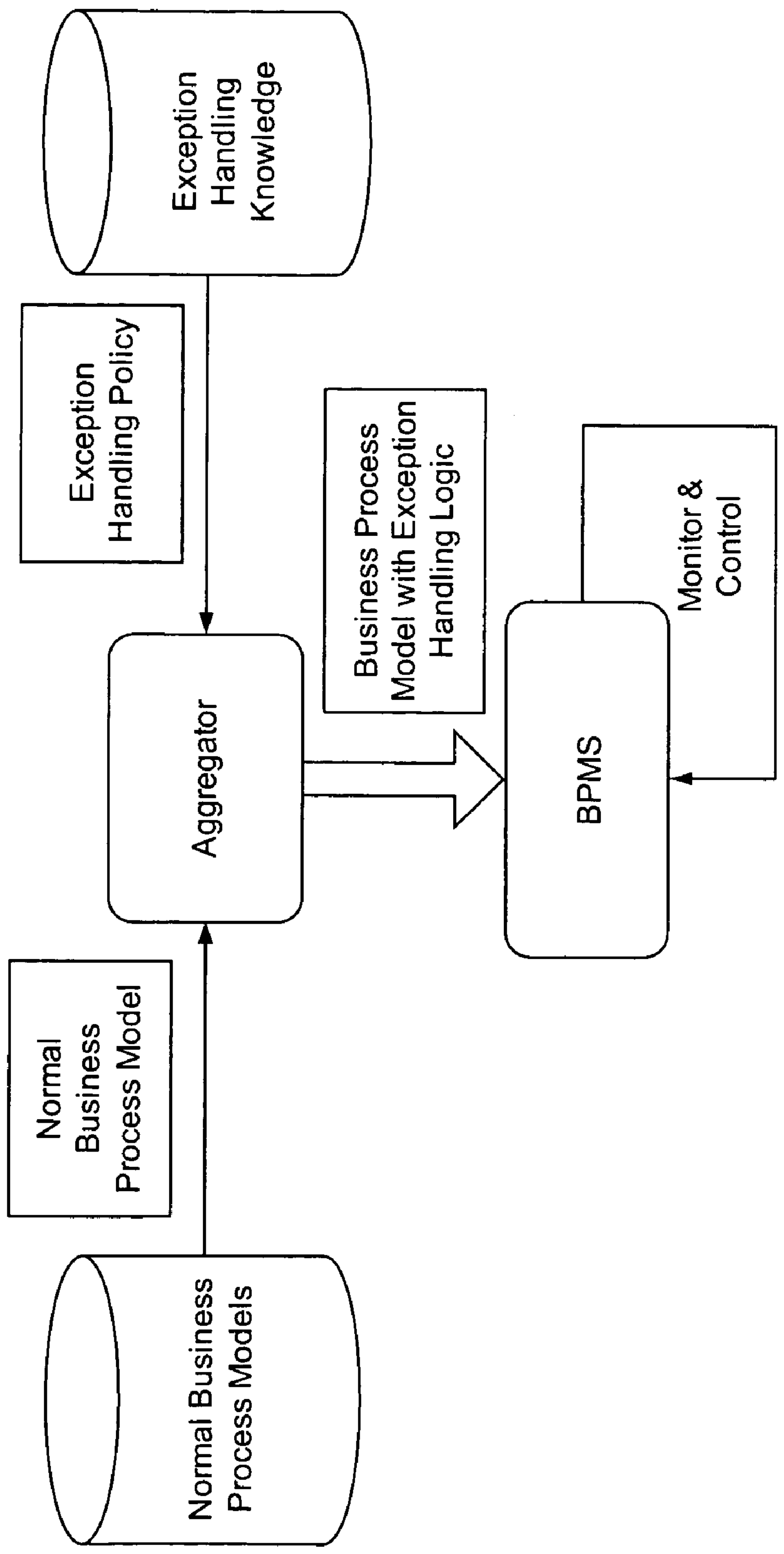

Exception Handling Knowledge
Exception Handling Policy
Business Process Model with Exception Handling Logic
Monitor & Control
Aggregator
BPMS
Normal Business Process Model
Normal Business Process Models

Figure 17

APPARATUS AND METHOD FOR POLICY-DRIVEN BUSINESS PROCESS EXCEPTION HANDLING

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] The present invention generally relates to a model-driven and QoS-aware infrastructure for facilitating the scalable composition of Web services in highly dynamic environments and, more particularly, to an exception management framework which supports two modes of exception management for business processes, providing a novel policy-driven approach to exception management implemented in the system infrastructure.

[0003] 2. Background Description

[0004] Process-based composition of Web services has recently gained significant momentum in the implementation of business processes. A critical and time-consuming part of modeling any business process is the detection and handling of exceptions that may occur during process execution. The Web services paradigm promises to take across-network application interactions one step further by enabling programmatic access to applications over the Web. Recently, process-based composition of Web services has emerged as the technology of choice for integrating heterogeneous and loosely coupled applications (see Boualem Benatallah and Fabio Casati, editors, *Distributed and Parallel Database, Special Issue on Web Services*, Springer-verlag, 2002). As such, process-based integration of services has been the subject of intense research and standardization efforts. This approach provides an attractive alternative to hand-coding the interactions between applications using general-purpose programming languages. An example of a business process would be a "Security Investment" system that aggregates multiple component services for security selection, budget analysis, market analysis, share exchange and future exchange, which are executed sequentially or concurrently.

[0005] Modeling business processes would be easier if all the activities could be completed successfully without any occurrence of exceptions (see Paul Greenfield et al., "Compensation is not enough", 7th *IEEE International Enterprise Distributed Object Computing Conference,* September 2003). Unfortunately experiences show (see Chris Peltz, "Web Services Orchestration: a review of emerging technologies, tools and standards", Technical Report, Hewlett-Packard Company, 2003) that a large amount of effort in the development of a business process is spent on exception management. In particular, Web services may operate in a highly dynamic environment, e.g., new services may become available at any time, existing services may become obsolete or temporarily unavailable, and services may offer different QoS (Quality of Service) properties or withdraw of advertised QoS properties. Such highly dynamic environment increases the probability of deviation situations during the execution of a business process and an increased complexity in exception handling logic. Therefore, it is important to provide support for exception management in the infrastructure so that developers can focus on defining the business logic, or normal flow, of a business process and delegate exception handling to the system infrastructure.

SUMMARY OF THE INVENTION

[0006] It is therefore an object of the present invention to provide a novel policy-driven approach to exception management, which can substantially simplify the development of business processes.

[0007] According to the invention, exception management is implemented in the system infrastructure, with exception handling policies supplied by individual business processes. Using the exception management framework, developers define exception policies in a declarative manner. Before a business process is executed, the service composition middleware integrates the exception policies with normal business logic to generate a complete process schema. Our initial experiments show that our policy driven-approach can significantly reduce the development time of business processes through its separation of the development of the business logic and the exception handling policies.

[0008] The novel policy-driven exception-management framework for business processes according to the invention is characterized by the following:

> **[0009]** In our framework, the development of business processes is substantially simplified by separating the development of the business logic and the exception handling policies.
>
> **[0010]** In order to capture the exception management knowledge, we identify a set of exception handling policy templates or patterns for declaratively defining deviation situations and associated exception handlers. At run time, the service composition middleware dynamically integrates the exception handling policies with the business logic to generate complete process schemas that specify both the normal and exceptional behaviors of the business processes.
>
> **[0011]** The framework supports two modes of exception management in business processes, namely, centralized and distributed. In the centralized mode, since a generated process schema contains all the necessary exception handlers, the exception management can be supported using the exception handling capabilities of the underlying process execution engine. In the distributed mode, our exception management framework dynamically binds exception policies with a specific execution plan (i.e., a business process instance) at runtime to generate control tuples. When these control tuples are deployed to component services, they enable local exception detecting and handling which is able to react to an exception faster than the centralized approach.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] The foregoing and other objects, aspects and advantages will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

[0013] **FIG. 1** is a statechart of a business process "security investment";

[0014] **FIG. 2** is a listing of the exception handling policy syntax;

[0015] **FIG. 3** is a listing of the timeout policy;

[0016] **FIG. 4** is a listing of the retry policy;

[0017] **FIG. 5** is a listing of the multiple binding policy;

[0018] **FIG. 6** is a listing of the replacement policy;

[0019] **FIG. 7** is a listing of the skip policy;

[0020] **FIG. 8** is a listing of the rollback policy;

[0021] **FIG. 9** is a block diagram of the procedure of processing exception handling policy;

[0022] **FIG. 10** is a block diagram of the processing result of a multiple binding policy;

[0023] **FIG. 11** is a block diagram of the processing result of a retry policy;

[0024] **FIG. 12** is a block diagram illustrating a compound state with a null state;

[0025] **FIG. 13** is a block diagram of the processing result of a replacement policy;

[0026] **FIG. 14** is a block diagram of the processing result of a skip policy;

[0027] **FIG. 15** is a block diagram of processing a rollback policy;

[0028] **FIG. 16** is a statechart of the business process "security investment" after policy based reconstructing; and

[0029] **FIG. 17** is a block diagram of the architecture of the prototype of the invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT OF THE INVENTION

[0030] We begin by briefly describing the main concepts of the service composition model we use in this disclosure. A composite Web service is an aggregation of multiple other elementary and composite Web services, which interact with each other according to a process model. We choose to specify the process model of a business process as a statechart which is a platform independent modeling tool, as shown in **FIG. 1**. The choice of statecharts for specifying business process is motivated by two main reasons: (i) statecharts have a well-defined semantics; and (ii) they offer the basic flow constructs found in contemporary process modelling languages (i.e., sequence, conditional branching, structured loops, concurrent threads, and inter-thread synchronization). It should be noted that exception management techniques developed in the context of statecharts, can be adapted to other process modeling languages (e.g., BPEL4WS, WSCI, BPML).

[0031] A basic state of a statechart describing a business process can be labelled with an invocation to either of the following:

[0032] An elementary Web service, i.e., a service which does not transparently rely on other Web services.

[0033] A business process aggregating several other services.

[0034] A Web service community, i.e., a collection of Web services with a common functionality although different non-functional properties (e.g., with different providers, different QoS parameters, reputation, etc.)

[0035] Web services in a community share a common service ontology. Service communities provide descriptions of a desired functionality (e.g., flight booking) without referring to any actual service (e.g., Qantas flight booking Web service). The set of members of a community can be fixed when the community is created, or it can be determined through a registration mechanism, thereby allowing service providers to join, quit, and reinstate the community at any time.

[0036] In general, by selecting a candidate service (see Liangzhao Zeng et al., "Quality Driven Web Services Composition", *Proceedings of the* 12th *International Conference on World Wide Web* (*WWW*), Budapest, Hungary, ACM Press, May 2003) (see Definition 1) for each task in the business process, an execution plan (see Definition 2) can be generated to create execution instance of business processes.

[0037] Definition 1 (Candidate services) Let us assume that $T=\{t_1, t_2, \ldots, t_n\}$ represents the set of all tasks in the business process CS. Function $S(t_i)$ gives a set of candidate services that can be used to execute task t_i, where

$$S(t_i)=\{s_{1i}, s_{2i}, \ldots, s_{mi}\} \tag{1}$$

[0038] Definition 2 (Execution plan). A set of pairs $p=\{<t_1, s_1>, <t_2, s_2>, \ldots, <t_n, s_n>\}$ is an execution plan of a business process if:

[0039] $\{t_1, t_2, \ldots, t_n\}$ is the set of tasks in the business process.

[0040] For each 2-tuple $<t_i, s_i>$ in p, the service si is assigned the execution of task t_i.

Exception Handling Policies

[0041] The syntax of an exception handling policy is shown in **FIG. 2**. In this syntax, every word in bold is a token in the language and optional elements are specified with square brackets []. Note the required exception event specification following the on keyword. The event can be a primitive event or a compound event which is composed from primitive events using event combination operators (see Sharma Chakravarthy et al., "Composite events for active databases: Semantics, contexts and detection", *Proceedings of* 20th *International Conference on Very Large Data Bases,* Sep. 12-15, 1994, Santiago de Chile, Chile, pp. 606-617, Morgan Kaufmann, 1994. A primitive event can be an execution exception (e.g., E(failure)) or a QoS exception event. The term subject refers to either composite, communities or component services. The target object indicates the scope of the exception handling policy, or the entire business process. The do-clause specifies exception management actions that include primitive and compound actions. See table 1 for examples. We distinguish between six types of exception handling policies, namely: timeout, retry, multiple binding, replacement, skip and rollback policies. It should be noted that the types of policy in our framework is extensible: new types of policies can be added without fundamentally altering the exception handling techniques built on top of the framework.

TABLE 1

Exception Handling Action

Action	Brief Explanation
timeout( )	Timouts a task execution
retry( )	Retries task execution by same services or different services
multiple( )	Invokes multiple services at the same time and returns the results of the service that completes execution first.
replace( )	Replaces the execution of a task (resp., segment) with other task (resp., segment)
skip( )	Skips the execution of tasks or a segment
rollback( )	Rollbacks the task execution to a task and then resumes the task execution

[0042] Timeout Policy. A timeout policy specifies the condition when service execution should be timeout. For example, in **FIG. 3**, the timeout policy is associated with the tasks RiskAnalysis and SecuritySelection. In this policy, if the completion of the execution of the task is delayed more than 120 seconds, then the execution will be timeout.

[0043] Retry Policy. A retry policy specifies how many times the service binding in sequence is allowed for a certain task. There are two approaches to retry the service binding, namely alternative and repeat. The alternative approach indicates to attempt the service binding by assigning the task to different services in sequence, while the repeat method indicates to retry the service binding using the same services. The service binding is retried until either the service execution is successful or the number of retries reaches the specified upper bound. For example, in **FIG. 4**, a retry policy is associated with the task FutureExchange and StockExchange, which sets the upper bound of retry to be two. It should be noted that the retry policy is applied whenever the execution of a task timeout or fails.

[0044] Multiple Binding Policy. A multiple binding policy specifies the condition that allows the concurrent invocation of multiple services for a task execution. For example, in **FIG. 5**, a multiple bind policy is associated with the task SecuritySearching, which allows three concurrent service execution bindings. Therefore, three services can now be invoked at the same time to execute the task SecuritySearching, which completes when any of the services have completed their execution of the task.

[0045] Replacement Policy. A replacement policy specifies what other task can be used to replace a certain task in a business process when the execution of the task fails or has timeout and the number of retries had reach the upper bound as specified in the retry policy. For example, in **FIG. 6**, the replacement is associated with the task OnlineStockExchange, which indicates that when the number of retry has reached a given upper bound, the task OnlineStockExchange is to be replaced by the task ManualStockExchange.

[0046] Skip Policy. A skip policy specifies a condition when a business process needs to skip the execution of a certain task. For example, in **FIG. 7**, a skip policy is associated with the task LowPriceBid, which specifies that if the task execution has failed or timeout, then the execution LowPriceBid is to be skipped.

[0047] Rollback Policy. A rollback policy specifies the point at which execution of business process has to rollback when the execution of task has failed or timeout and the maximal allowed number of retries had been reached. For example, in **FIG. 8**, the rollback policy is associated with the tasks SecuritySearch and RiskAnalysis specifies that if the execution has failed or timeout and the maximal allowed number of retries had been reached, the business process execution is rollbacked to the task BudgetAnalysis. It should be noted that when a rollback policy is specified, for affected tasks (e.g., the task of BudgetAllocation), the services selected to execute the tasks are required to support the cancellation or compensation of a given operation within a certain time interval from its completion (see Boualem Benatallah et al., ibid.).

Exception Handling Policy Binding

[0048] In our framework, an exception handling policy is separated of any individual business processes. It regulates the behavior of business processes when execution exceptions are raised during runtime. Such an approach requires dynamically binding the exception handling policies with process schemas.

[0049] In order to implement policy binding, the system identifies the exception handling policies that are associated with each task in a business process and then uses these policies to reconstruct the process schema. It should be noted that all types of policies except timeout policies can modify the structure of the process schema of a business process. **FIG. 9** illustrates the procedure for applying policies to generate exception handling aware process schemas. In this procedure, we assume that: (i) A task is associated with only one instance of each type of policy. (ii) If a task is associated with both retry and multiple binding policies, then the action in the retry policy can only be retry (t, n, repeat). (iii) A task can only associate with either skip, replacement or rollback policy at a business process instance. The procedure starts with processing the multiple and retry policies, and then processes any skip, replacement or rollback policies.

[0050] In the remainder of this section, we present the algorithms for generating exception-aware process schemas. Here, we assume that the process schema is a 2-tuple <ST, TR>, where ST is the set of states and TR is the set of transitions between these states; a transition tr is a 3-tuple <initialState, targetState, r> and r is an Event Condition Action rule.

[0051] Processing a multiple binding policy. The processing of a multiple binding policy for task t_i is done by duplicating t_i and enabling both t_i and its duplicates at the same time (see Algorithm 1 below). For example, in the business process Security Investment, the task Security-Searching is associated with a multiple binding policy. The processing result is shown in **FIG. 10**.

Algorithm 1: processing a multiple binding policy

```
input  : Task $t_i$, Process schema CS, Multiple Binding Policy p
begin
  N is multiple binding upper bound; $t_{i+1}$ is the next task of $t_i$, the
     transition between $t_i$ and $t_{i+1}$ is $tr_i$; $t_{i-1}$ is the previous task of $t_i$,
     the transition between $t_{i-1}$ and $t_i$ is $tr_{i-1}$
  for k $\leftarrow$ 1 to N do
```

-continued

Algorithm 1: processing a multiple binding policy

```
    $t_i^k \leftarrow clone(t_i^k)$
    if k = = 1 then
        replace $t_i$ with $t_i^k$ in CS
    else
        ST $\leftarrow$ ST $\cup$ $\{t_i^k\}$
        tr $\leftarrow$clone($tr_{i-1}$)
        $tr_{i-1}^k \leftarrow$ replace $t_i$ with $t_i^k$ in tr; TR $\leftarrow$ TR $\cup$ $\{tr_{i-1}^k\}$
        tr $\leftarrow$clone($tr_i$)
        $tr_i^k \leftarrow$ replace $t_i$, with $t_i^k$ in tr; TR $\leftarrow$ TR $\cup$ $\{tr_i^k\}$
end
```

[0052] Processing a retry policy. The processing of a retry policy for task ki needs to consider two cases (see Algorithm 2 below): (i) Task t_i is not associated with any multiple binding policy. This is a simple case which can be implemented by duplicating ti and enabling both t_i and its duplicates in sequence. (ii) Task t_i is also associated with a multiple binding policy. In this case, after processing the multiple binding policy, for each concurrent thread k the algorithm duplicates the task t_i^j and enables both t_i^j and its duplicates in sequence. For example, in the business process Security Investment, the task FutureExchange is associated with a retry policy but no multiple binding policies, which corresponds to case 1 of algorithm 2. The policy processing result is shown in **FIG. 11**.

Algorithm 2: Processing a retry policy

```
input  : Task $t_i$, Process schema CS, Retry Policy p
begin
    N is multiple binding upper bound; $t_{i+1}$ is the next task of $t_i$, the
        transition between $t_i$ and $t_{i+1}$ is tr
    if $t_i$ is not associated with any binding policies then
        for k $\leftarrow$ 1 to N do
            $t_i^k \leftarrow clone(t_i)$
            if k = = 1 then
                rename $t_i$ with $t_i^k$ in CS
                tr $\leftarrow$< $t_i^k$, $t_{i+1}$, E(completed, $t_i^k$)
                TR $\leftarrow$TR $\cup$ {tr}
            else
                ST $\leftarrow$ ST $\cup$ $\{t_i^k\}$
                tr $\leftarrow$< $t_i^{k-1}$, $t_i^k$, E(failure, $t_i$,) $\vee$E(timeout, $t_i$)<
                TR $\leftarrow$ TR $\cup$ {tr}
                tr $\leftarrow$< $t_i^k$, $t_{i+1}$, E(completed, $t_i^k$)>
            TR $\leftarrow$ TR $\cup$ {tr}
    else
        M $\leftarrow$ multiple binding upper bound
        for j $\leftarrow$1 to M do
            for k $\leftarrow$1 to N do
                $t_i^{j,k} \leftarrow$ clone ($t_i^j$)
                if k = = 1 then
                    rename $t_i$ with $t_i^k$ in CS
                    tr $\leftarrow$< $t_i^j$, $t_i^{j,k}$, E(completed, $t_i^{j,k}$)>
                    TR $\leftarrow$ TR $\cup$ {tr}
                else
                    ST $\leftarrow$ ST $\cup$ $\{t_i^{j,k}\}$
                    tr $\leftarrow$< $t_i^{j,k-1}$, $t_i^{j,k}$, E(failure, $t_i^{j,k-1}$)$\vee$
                        E(timeout, $t_i^{j,k-1}$)>
                    TR $\leftarrow$TR $\cup$ {tr}
                    tr $\leftarrow$< $t_i^{j,k}$, $t_{i+1}$, E(completed, $t_i^{j,k}$)>
                    TR $\leftarrow$ TR $\cup$ {tr}
end
```

[0053] Processing a replacement policy. The processing of a replacement policy for task ti needs to consider four cases (see Algorithm 3 below):

[0054] (i) Task t_i is associated with both a retry policy and a multiple binding policy. In this case, a compound state (see **FIG. 12**) is created to synchronize the concurrent threads when all of the tasks in all of concurrent threads fail. The compound state is also used to enable the alternative task.

[0055] (ii) Task t_i is only associated with a retry and possibly other types of policies but not with any multiple binding policy. In this case, the alternative task is enabled when the last duplicated task fails.

[0056] (iii) Task t_i is associated with a multiple binding policy and possibly other types of policies but not with any retry policy. In this case, the policy processing is similar to case 1.

[0057] (iv) Task t_i is not associated with any retry or multiple binding policy. The alternative task is enabled when the execution of t_i fails. For example, in the business process Security Investment, the task OnlineStockExchange is associated with a both replacement policy a retry policy, which corresponds to case 2 of algorithm 3. The processing result is shown in **FIG. 13**.

Algorithm 3: Processing a replacement policy

```
input  : Task $t_i$, Process Schema CS, Replacement Policy p
begin
    $t_a$ is the replacement task of $t_i$; ST $\leftarrow$ST $\cup$ $\{t_a\}$
    if $t_i$ is associated with both Retry and Multiple Binding Policy then
        M $\leftarrow$ multiple binding upper bound; N $\leftarrow$ retry upper bound
        create a compound state $st_{co}$ with a NULL state
        ST $\leftarrow$ ST $\cup$ $\{st_{co}\}$
        tr is the transition between $st_{ini}$ and $st_{null}$ in $st_{co}$
        tr $\leftarrow$< $st_{ini}$, $st_{null}$, (E(failure, $t_i^{1,N}$) $\vee$E(timeout, $t_i^{1,N}$)) $\wedge \ldots \wedge$
            (E(failure, $t_i^{M,N}$) $\vee$ E(timeout, $t_i^{M,N}$))>
        for j $\leftarrow$ 1 to M do
            tr $\leftarrow$< $t_i^{j,N}$, $st_{co}$, E(failure, $t_i^{j,N}$) $\vee$ E(timeout, $t_i^{j,N}$)>
            TR $\leftarrow$ TR $\cup$ {tr}
        tr $\leftarrow$< $st_{co}$, $t_a$ >; TR $\leftarrow$ TR $\cup${tr}
    if $t_i$ is associated with Retry but not Multiple binding Policy then
        N $\leftarrow$ retry upper bound
        tr $\leftarrow$< $t_i^N$, $t_a$, E(failure, $t_i^N$) $\vee$ E(timeout, $t_i^N$)>
        TR $\leftarrow$ TR $\cup$ {tr}
    if $t_i$ is associated with Multiple Binding but not Retry Policy then
        M $\leftarrow$ multiple binding upper limit
        create a compound state $st_{co}$ with a NULL state
        ST $\leftarrow$ ST $\cup$ $\{st_{co}\}$
        tr is the transition betwen $st_{ini}$ and $st_{null}$ in $st_{co}$
        tr $\leftarrow$< $st_{ini}$, $st_{null}$, (E(failure, $t_i^1$) $\vee$ E(timeout, $t_i^1$)) $\wedge \ldots \wedge$
            (E(failure, $t_i^1$) $\vee$ E(timeout, $t_i^M$))>
        for j $\leftarrow$ 1 to M do
            tr $\leftarrow$< $t_i^j$, $st_{co}$, E(failure, $t_i^j$) $\vee$ E(timeout, $t_i^j$)>
            TR $\leftarrow$ TR $\cup$ {tr}
        tr $\leftarrow$< $st_{co}$, $t_a$ >; TR $\leftarrow$ TR{tr}
    if $t_i$ is not associated with either Retry or Multiple Binding Policy then
        tr $\leftarrow$< $t_i$, $t_a$, E(failure, $t_i$) $\vee$ E(timeout, $t_i$)>
        TR $\leftarrow$ TR $\cup$ {tr}
    tr $\leftarrow$< $t_a$, $t_{i+1}$, E(completed, $t_a$) >; TR $\leftarrow$ TR $\cup$ {tr}
end
```

[0058] Processing a skip policy. The processing of a skip policy for task t_i also needs to consider four cases (see Algorithm 4 below):

[0059] (i) Task t_i is associated with both a retry and a multiple binding policy. In this case, a compound state (see **FIG. 12**) is created to synchronize the concurrent threads when all of the tasks in all of concurrent threads fail. The compound state is also used to enable the next task so that t_i can be skipped.

[0060] (ii) Task t_i is only associated with a retry policy and possibly other types of policies but not with any multiple binding policy. In this case, the next task is enabled when the execution of t_is last duplicate fails.

[0061] (iii) Task t_i is associated with a multiple binding policy and possibly other types of policies but not with any retry policy. In this case, the policy processing is similar to case 1.

[0062] (iv) Task t_i is not associated with any retry or multiple binding policy. The next task is enabled when the execution of t_i fails. For example, in the business process Security Investment, the task LowPriceBid is associated with both a skip policy and a retry policy, which corresponds to case 2 of algorithm 4. The processing result is shown in **FIG. 14**.

Algorithm 4: Processing a skip policy

```
input  : Task $t_i$, Process Schema CS, Skip Policy p
begin
  if $t_i$ is associated with both Retry and Multiple Binding Policy then
    $M \leftarrow$ multiple binding upper bound; $N \leftarrow$ retry upper bound
    create a compound state $st_{co}$ with a NULL state
    $ST \leftarrow ST \cup \{st_{co}\}$
    tr is the transition between $st_{ini}$ and $st_{null}$ in $st_{co}$
    $tr \leftarrow < st_{ini}, st_{null}, (E(failure, t_i^{1,N}) \vee E(timeout, t_i^{1,N})) \wedge \ldots \wedge$
      $(E(failure, t_i^{M,N}) \vee E(timeout, t_i^{M,N}))>$
    for $j \leftarrow$ 1 to M do
      $tr \leftarrow < t_i^{j,\ N}\ st_{co}, E(failure, t_i^{j,N}) \vee E(_{timeout}, t_i^{j,N})>$
      $TR \leftarrow TR \cup \{tr\}$
    $tr \leftarrow < st_{co}, t_{i+1} >$
    $TR \leftarrow TR \cup \{tr\}$
  if $t_i$ is associated with Retry Policy but not Multiple Binding Policy
    then
    $N \leftarrow$ retry upper bound
    $tr \leftarrow < t_i^N, t_{i+1}, E(failure, t_i^N) \vee E(timeout, t_i^N)>$
    $TR \leftarrow TR \cup \{tr\}$
  if $t_i$ is associated with Multiple Binding Policy but not Retry Policy
    then
    $M \leftarrow$ multiple binding upper bound
    create a compound state $st_{co}$
    $ST \leftarrow ST \cup \{st_{co}\}$
    tr is the transition between $st_{ini}$ and $st_{null}$
    $tr \leftarrow < st_{ini}, st_{null}, (E(failure, t_i^1) \vee E(timeout, t_i^1)) \wedge \ldots \wedge$
      $(E(failure, t_i^1) \vee E(timeout, t_i^M))>$
    for $j \leftarrow$ 1 to M do
      $tr \leftarrow < st_{ini,}st_{null}, (E(failure, t_i^j) \vee E(timeout, t_i^j)>$
      $TR \leftarrow TR \cup \{tr\}$
    $tr \leftarrow < st_{co}, t_{i+1}, >; TR \cup \{tr\}$
  if $t_i$ is not associated with either Retry or Multiple Binding Policy then
    $tr \leftarrow < t_i, t_{i+1}, E(failure, t_i) \vee E(timeout), t_i)>$
    $TR \leftarrow TR \cup \{tr\}$
end
```

[0063] Processing a rollback policy. Processing of a rollback policy for task t_i also needs to consider four cases (see Algorithm 5 below):

[0064] (i) Task t_i is associated with both a retry and multiple binding policy. In this case, a compound state (see **FIG. 12**) is created to synchronize the concurrent threads when all the tasks in all concurrent threads fail. The compound state is also used to enable the rollback of the execution of the business process to the re-entry point.

[0065] (ii) Task t_i is associated with a retry policy and possibly other types of policies but not with any multiple binding policy. In this scenario, the rollback is enabled when the execution of t_i's last duplicated task fails.

[0066] (iii) Task t_i is only associated with a multiple binding policy and possibly other types of policies but not with any retry policy. In this case, the policy processing is similar to case 1.

[0067] (iv) Task t_i is not associated with any retry or multiple binding policy. The rollback is enabled when the execution of t_i fails.

Algorithm 5: Processing a rollback policy

```
input  : Task ti, Process Schema CS, Rollback Policy p
begin
  the rollback entry point is $t_a$
  if $t_i$ is associated with both Retry and Multiple Binding Policy then
    $M \leftarrow$ multiple binding upper bound; $N \leftarrow$ retry upper bound
    create a compound state $st_{co}$ with a NULL state
    $ST \leftarrow ST \cup \{st_{co}\}$
    tr is the transition between $st_{ini}$ and $st_{null}$ in $st_{co}$
    $tr \leftarrow < st_{ini}, st_{null}, (E(failure, t_i^{1,N}) \vee E(timeout, t_i^{1,N})) \wedge \ldots \wedge$
      $(E(failure, t_i^{M,N}) \vee E(timeout, t_i^{M,N}))>$
    for $j \leftarrow$ 1 to M do
      $tr \leftarrow < t_i^{j,\ N}, st_{co}, E(failure, t_i^{1,N}) \vee E(timeout, t_i^{1,N})>$
      $TR \leftarrow TR \cup \{tr\}$
    for task $t_k$ between $t_a$ and $t_i$ (includes $t_a$) do
      if task $t_k$ needs to be compensated then
        $ST \leftarrow ST \cup \{undo(t_k)\}$
        $tr \leftarrow < st_{co}, undo(t_k)) >; TR \leftarrow TR \cup \{tr\}$
  if $t_i$ is associated with Retry but not Multiple Binding Policy then
    $N \leftarrow$ retry upper bound
    for task $t_k$ between $t_a$ and $t_i$ (includes $t_a$) do
      if task $t_k$ needs to be compensated then
        $ST \leftarrow ST \cup \{undo(t_k)\}$
        $tr \leftarrow < t_i^N, undo(t_k)) >; TR \leftarrow TR \cup \{tr\}$
  if $t_i$ is associated with Multiple Binding but not Retry Policy then
    $M \leftarrow$ multiple binding upper bound
    create a compound state $st_{co}$ with a NULL state
    $ST \leftarrow ST \cup \{st_{co}\}$
    tr is the transition between $st_{ini}$ and $st_{null}$ in $st_{co}$
    $tr \leftarrow < st_{ini}, st_{null}, (E(failure, t_i^1) \vee E(timeout, t_i^1)) \wedge \ldots \wedge$
      $(E(failure, t_i^1) \vee E(timeout, t_i^M)) >$
    for $j \leftarrow$ to M do
      $tr \leftarrow < t_i^j, st_{co}, E(failure, t_i^j) \vee E(timeout, t_i^j) >$
      $TR \leftarrow TR \cup \{tr\}$
    for task $t_k$ between $t_a$ and $t_i$ (includes $t_a$) do
      if task $t_k$ needs to be compensated then
        $ST \leftarrow ST \cup \{undo(t_k)\}$
        $tr \leftarrow < st_{co}, undo(t_k)) >; TR \leftarrow TR \cup \{tr\}$
  if $t_i$ is not associated with either Retry or Multiple Binding Policy then
    for task $t_k$ between $t_a$ and $t_i$ (includes $t_a$) do
      if task $t_k$ needs to be compensated then
        $ST \leftarrow ST \cup (undo(t_k)\}$
        $tr \leftarrow < t_i, undo(t_k), E(failure, t_i) \vee E(timeout, t_i) >;$
          $TR \leftarrow TR \cup \{tr\}$
  $tr \leftarrow < undo(t_a), t_a >; TR \leftarrow TR \cup \{tr\}$
end
```

[0068] It should be noted that some completed tasks need to be undone if the compensation process is required. For example, in the business process Security Investment, the task FutureExchange is associated with both a rollback policy and a retry policy, which corresponds to case 2 of algorithm 5. The processing result is shown in **FIG. 15**.

[0069] After the reconstruction of the process schema is completed, the extended process schema (see the example in **FIG. 16** in which the processing result of the business process Security Investment is based on the exception handling policies given above) that contains exception management knowledge is used to conduct the execution planning. In our previous work (see Liangzhao Zeng et al., ibid.), we adopted Multiple Criteria Decision Making (see Valerie

Belton and Theodor Stewart, *Multiple Criteria Decision Analysis: An Integrated Approach*, Kluwer Academic Publishers, 2002) techniques and used the integer programming (IP) approach to select an optimal execution plan for a business process. It should be noted that based on the extended process schema and the selected execution plan, a fully executable description of the business process can be generated. For example, a BPEL4WS script that contains exception handlers can be generated to execute the business process by a process execution engine.

Implementation

[0070] The preferred implementation of the policy-driven exception-management framework for business processes is shown in **FIG. 17**. In the implementation, we use BPEL4WS as the language to describe process schemas. BPEL4WS allows process modelers to create complex processes by creating and writing together different activities that can perform Web services invocations, manipulate data, throw fault, or terminate a process. In our implementation framework, we separate normal processes that are specified in BPEL4WS and the knowledge of exception management. There are two types of persistent storage, the Normal Business Process Schema Models and the Exception Handling Knowledge. The former stores normal process schemas that are defined by business analysts and the latter manages exception handling policies that are defined by exception handling experts.

[0071] The Aggregator exploits the schema re-construction algorithms described in previous sections and transforms such BPEL4WS-based process schema into an extended process schema that is also presented by BPEL4WS but contains more constructs reflecting the requirement of exception handling policies. The BPEL4WS script is deployed into the BPMS (Business Process Management System) that is essentially an execution engine for BPEL4WS. In our case, we are using IBM's DragonFly engine for this purpose. The system utilizes the orchestration and exception handling mechanism provided by the DragonFly engine to manage business processes. While the invention has been described in terms of a single preferred embodiment, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is as follows:

1. A system for executing business process and handling runtime exceptions, comprising:

an aggregator that is able to integrate normal business logic with exception handling knowledge to generated extended business process definition; and

a business process engine that is able to execute business process definitions that contains knowledge on runtime exception handling.

2. The system in claim 1, further comprising an exception handling policy repository that contains exception handling policies.

3. The system in claim 2, wherein said exception handling policies includes said runtime exception specifications and exception handling action specifications

4. The system in claim 2, wherein said business process engine can understand runtime exception specifications, detect runtime exceptions and execute exception handling actions.

5. A method for executing business processes and handling runtime exceptions, comprising the steps of:

specifying exception handling knowledge from normal business process logic;

integrating normal business logic with exception handling knowledge to generate extended business process definitions;

executing business process definition that contains runtime exception handling knowledge;

detecting runtime exceptions; and

executing exception handling actions.

6. The method in claim 5, further comprising the step of defining exception specification.

7. The method in claim 5, further comprising the step of defining exception handling knowledge as exception handling policies.

8. The method in claim 5, further comprising the step of checking the compatibility between the exception handling policies and normal business logic.

9. The method in claim 5, further comprising the step of checking the conflicts among the exception handling policies.

* * * * *