A Goal-Based Configurable Transaction Model of Artifact-Centric Workflows

Haihuan Qin
School of Electronic Information Engineering
Shanghai Dianji University
Shanghai, China

Leilei Chen
School of Information Engineering
Shanghai Open University
Shanghai, China

Abstract—**Ensuring process execution reliability and success is crucial for business process management. A mechanism is required to link high-level requirements to transaction management and utilize runtime information to remedy occurred problems. However, workflow transaction models have a gap with high-level requirements, and their limited forward recovery mechanisms lead to lots of unnecessary aborts. This paper proposes a relaxed configurable transaction model derived from the artifact-centric design principle. High-level requirements are described using goal model and linked to transaction management. Algorithms are provided to ensure workflow validity against user-specified failure atomicity through validity rules. With the support of configuration rules and enhanced failure handling mechanism, some failures can be avoided when their restored conditions are satisfied by the runtime data gathered in artifacts.**

I. INTRODUCTION

Business process management (BPM) is used in modern organizations to achieve their established business goals efficiently [2]. To stay competitive in the marketplace, business processes must be able to reflect changes of business goals, policies or operational routines, and allow users to specify their customized requirements on the model and execution behaviors. Meanwhile, process execution reliability must be ensured and the execution should be leaded to success as much as possible.

Transaction is a key mechanism to ensure execution reliability. However, the traditional ACID transaction semantics is too strict and needs to be relaxed before being applied to workflow transaction [3]. One kind of the failure atomicity relaxation takes user-specified accepted termination states (ATSs) as correctness criteria [1, 3]. Much work on failure recovery mechanisms uses compensation as the backward recovery mechanism and retry or alternative path as the forward one [1, 3, 4]. For non-compensatable activity, guaranteed termination [4] allows preserving its side-effects when process terminates after its completion. No matter what kinds of mechanisms above methods take, they are almost based on traditional activity-centric workflow models. Recently, artifact-centric workflow models [5] are promising to utilize runtime information to support better forward recovery since the runtime data and the instance status have already been gathered in artifacts.

Besides limited forward failure recovery mechanisms, traditional workflow transaction models have a gap with high-level requirements [10]. Thus changes of these requirements cannot be quickly reflected in a reliable way. To reduce the gap, transaction management should take these requirements into consideration. The challenge here is to provide a method to describe and map these requirements to transaction model. When the requirements change, their impact on the designed model and the running instances can be quickly reflected. We will discuss this challenge via an example

Fig. 1 illustrates a Short Term Visiting Program (STVP) of Fudan University represented by EZ-Flow [5] (see Section III). STVP is a program that Fudan U. provides for PH.D candidates to improve their research capability as well as to publish more high-quality papers through visiting famous professors abroad for four months. To get started, the applicant contacts professors to get an invitation letter, then submits the application and waits for the approval from an auditing process, which may take more or less than four months. To save time, the applicant may start collaboration soon after handing in the application. If he gets the approval (i.e., activity AS succeeds), he will apply for the visa, finish the visiting experience, and submit a summary afterward. Once AS fails, we hope there are alternative positive measures for achieving the goals of the program even without going abroad, rather than simply dropping the collaboration. The following cases are worthy of attention:

- **Case 1**: the activity *CO* has yielded some research results that are directly correlated with the process business goal. The side-effects is positive. It should be preserved rather than being aborted or compensated. Thus, it needs to support defining high-level requirements and customized failure atomicity in the transactional workflow model.

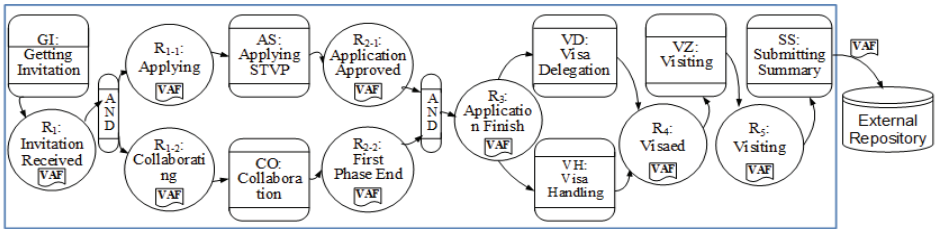

Figure 1. Short term visiting program application process (STVP)

This work is partially supported by "Academic Discipline Project of Shanghai Dianji University, Project Number: 16YSXK04" and "Shanghai Young Teachers' Training Program grant A1-5701-16-014-48"

- **Case 2:** side-effects yielded by *CO* may make the audit condition of *AS* to be true and earned the applicant a second chance, or meet conditions of alternative programs with higher requirements. Thus, flexible and positive failure handling measures considering runtime information, such as conditional retry or conditional substitution, should be provide to dynamically remedy the occurred problems.
- **Case 3:** if *AS* cannot be retried or substituted, whether the collaboration will continue or not should depend on the negotiation of the applicant and the professor; that is, continuing visiting, remote collaboration or dropping collaboration. Failure handling of such activity depends on the relationship between its goal and process goals, which may vary with companies' strategies. A flexible model with methods for validity verification and dynamical configuration, are required to settle such requirements.

In order to solve above issues, we propose a goal-based transactional artifact-centric workflow model, TxEEZ-Flow, in which high-level requirements are described by goal model and failure atomicity is user-specified. Effective forward recovery mechanisms are provided. Algorithms are provided to ensure workflow validity against user-specified failure atomicity through validity rules. Dynamic transactional configuration and validity verification are also supported at runtime.

The rest of this paper is organized as follows. Section II introduces TxEEZ-Flow model. Section III proposes methods of transactional validity verification and dynamic configuration. Implementation prototype is introduced in Section IV. Section V discusses related work and section VI concludes the paper.

II. TxEEZ-Flow Model

TxEEZ-Flow is a goal-based transactional artifact-centric workflow model, which supports dynamic configuration of processes' transactional behaviors at runtime.

A. EEZ-Flow:An Extended Artifact-Centric Workflow Model

EEZ-Flow is developed based on EZ-Flow [5], which uses artifacts, classes, tasks, repositories, and workflow schemas to represent a workflow. To support parallel execution, EEZ-Flow splits an artifact into disjointed pieces to be manipulated by multiple tasks in parallel. A special field *splitId* is added to artifact class to record split artifact. Two structured task types are subsequently introduced: AND-Split and AND-Join. AND-Split task fetches the key artifact, splits it into multiple copies, each with a new identifier and a disjointed attribute set, and sets the its original identifier to the field *splitId*. The corresponding AND-Join task fetches all the copies that have the same *splitId* value and combines them into one artifact again. Exclusive selection execution is potentially supported by artifact-centric workflow through guards on edges, e.g., activity *VD* and *VH* in Fig. 1. In this pattern, only one path is selected for execution.

An EEZ-Flow workflow schema W consists of the following elements: $C, \Gamma, E, \Sigma, F, R, L,$ where C is the *core* artifact class, Γ is a set of *auxiliary* artifact classes; E is a set of event types; Σ is a set of tasks including basic tasks and structured tasks; R is a set of repositories with subset R_C for C, R_Γ for Γ, and an external repository r_{ext}; L is a set of guards on edges; and F maps each task t in Σ to a pair $(e_i; e_o)$, where e_i is a logic expression consisting of event types and logic operations ($\wedge$,$\vee$) that triggers t; e_o is an event type produced by t. Each event type is produced by one task and can be used to trigger one or more tasks. Each task needs one or more event types to activate its execution.

B. Modeling Business Goals

Each process has one or more business goals: one is the main objective, and the others may be supportive or mandatory to it. In artifact-centric process, business goals are reflected in the artifact fields. We can use an expression building with artifact fields to describe a goal. Business goals can be presented as hierarchical tree [6].

Definition 3.1. A business goal is a tuple g = (name, expr, type, pri), where *name* is its unique name; *pri* is its priority, and

- *expr* is an expression consisting of (1) one or more business goal triples (artf, op, thr), where *artf* is a field of the core artifact; *op* is a comparison operator including ==, <, >, <=, >=; and *thr* is the threshold value of the artifact field which indicates whether the goal has been achieved; and (2) logical operations that are used to connect multiple business goals, including $\wedge$ *and* $\vee$;
- *type* describes whether the goal is positive or negative; positive means profitability, while negative means upfront investment has a certain risk and will be shifted to positive only after some business goals are achieved.

Relationships between business goals supported include: mandatory, support, and negative-positive shift dependency, denoted by gr(man, g_1, g_2), gr(sup, g_1, g_2) and gr(shft, g_1, g_2), resp. *gr(man, g_1, g_2)* means the workflow will fail to obtain g_1 if it fails to achieve g_2. *gr(sup, g_1, g_2)* means achieving g_1 helps achieving g_2; it still has chance to obtain g_2 on the failure of g_1. *gr(shft, g_1, g_2)* means that the transform of g_1 from a negative goal to a positive one relies on the achieving of g_2.

Next, we link business goals of a process with its activities. An activity is called goal-related if there exists an artifact field in its output set contained in a business goal of the containing process. We provide a function *getRelatedTasks(g)* to get goal-related activities of a goal g. Once a goal-related activity succeeds, its side-effect is expected to be preserved forever. On its failure, we hope that some alternative solutions can be found to fix the problem and continue the process to obtain its specified goals, rather than simply failing the whole process.

C. Transaction Model

A transactional workflow is a collection of activities and their transactional behaviors, which dictates what failure handling measures should be taken when activities fail. The main transactional behaviors we considered are: compensatable (cp), retriable (r), pivot (p), configurable (c) and terminable (t). In case of failure occurs, an activity t is *compensatable* if its effect can be semantically undone after its commitment; t is *retriable* if it can succeed after being invoked several times; t is *pivot* if its effect must be preserved after its completion and it offers an ending-process activity to terminate the process execution; t is *terminable* if it can be terminated during its running; t is configurable if, in case of its failure, there exist flexible failure handling mechanisms that guarantee reliable execution of the process, including: (1) t can be retried if its parallel branch

produces some positive side-effects and makes its execution conditions be true; (2) *t* can be substituted if there are alternative activities becoming available at runtime and their execution conditions are satisfied by current context; and (3) if (1) and (2) are not satisfied, resort to regular failure handling mechanism.

The transactional behaviors can be combined, and all possible combination set is {r, cp, p, t, c, (r, cp), (r, p), (r, t), (cp, c), (p, c), (t, c), (cp, t, c), (p, t, c)}. To model transactional execution semantics of a task, we develop a model similar to Petri net based on [5]. A place, called *"state"* for convenience, represents the task execution state. Two types of token are used: "event" and "enactment", which contains the contents of an event that triggers the task and artifact information, resp. Fig. 2(a) shows the model of a *retriable* activity. When an activity instance is initiated, its state is *initial*. It can be activated by fetching artifact to generate an enactment token and transited to *ready*. Then it invokes a specified web service and transited to *finished* upon completion. If the invocation succeeds, it stores back the artifact and ends with *completed*; otherwise, it transits to *failed*, and the invocation will be retried. Note that if a task fails to obtain its goal, its state also transits to *failed*.

Execution semantics of other kinds of activities can be extended from these states/transitions, such as the *terminable* activity in Fig. 2(b), state *terminated* and transition *terminate()* are introduced. Fig. 2 then illustrates execution semantics of a *compensatable* activity (Fig.2(c)), a pivot activity (Fig. 2(d)) and a configurable activity (Fig. 2(e)) in turn. If an activity combines transactional behaviors, its behavior model should contain the union of corresponding semantics of behavior types.

Usually, a goal-related activity should be defined as *pivot* or *configurable*. As a pivot activity, its goal-related output can be preserved for ever once it succeeds. As a configurable activity, it will be configured to fix the problem so as to obtain the goals rather than simply failing the whole process if a failure occurs.

Transactional actions of a task, such as completion, abortion or compensation, may influence those of other tasks. We use transactional dependencies to represent such influences, mainly including: commitment, abortion, compensation, termination and configuration. Note that these dependencies may rely on the activation dependency between tasks, we will start with it.

Activation dependency. There is an activation dependency of task t_2 on t_1 if one event produced by t_1 upon its completion is required to activate t_2. We denotes activation conditions of task t by *ActCond(t)*. t_2 has activation dependency on t_1 iff for event e belongs to $t_1.completed$, $\exists e \in ActCond(t_2)$.

Commitment dependency. There is a commitment dependency of task t_2 on t_1 if the completion of t_2 requires that t_1 must be finally completed. We use tasks set C*mtSet(t)* to describe tasks on which task t have commitment dependency. t_2 has commitment dependency on t_1 iff $t_1 \in CmtSet(t_2)$. Note that if an activity is substituted, its commitment dependencies should be adjusted basing on the substituting activity.

Abortion dependency. There is an abortion dependency of task t_2 on t_1 if the abortion, termination or compensation of t_1 leads to the abortion of t_2. We denotes abortion events of task t by *AbtCond(t)*. There is an abortion dependency of t_2 on t_1 iff $t_1.aborted \in AbtCond(t_2) \vee t_1.terminated \in AbtCond(t_2) \vee t_1.compensated \in AbtCond(t_2)$.

Compensation dependency. There is a compensation dependency of task t_2 on t_1 if the abortion, termination or compensation of t_1 causes the compensation of t_2. We denotes compensation events of task t by *CpsCond(t)*. t_2 has compensation dependency on t_1 iff $t_1.aborted \in CptCond(t_2) \vee t_1.terminated \in CptCond(t_2) \vee t_1.compensated \in CptCond(t_2)$. When an event in *CpsCond(t_2)* arrives, it compensates t_2 if t_2 is compensatable, or it ends the process execution if t_2 is pivot.

Termination dependency. Task t_2 has termination dependency on t_1 if the abortion or termination of t_1 causes the termination of t_2. There are two modes: *must* and *may*. In *must* mode, t_2 must be terminated. In *may* mode, based on policies, t_2 may be terminated or continue to run. We denote termination events and dependency mode of task t by *TmtCond(t)=(event, mode)*. t_2 has termination dependency on t_1 iff $t_1.aborted \in TmtCond(t_2).event \vee t_1.terminated \in TmtCond(t_2).event$. Usually, termination dependencies are defined between goal-related tasks that have negative-positive shift dependencies.

Configuration dependency. There is a configuration dependency of task t_2 on t_1 if configuration conditions of t_2 relies on some side-effects of t_1. We use tasks set C*onfSet(t)* to describe tasks whose side-effects t may use as conditions to retry or substitute. There is a configuration dependency of t_2 on t_1 iff $t_1 \in ConfSet(t_2)$. Since side-effect of t_1 may be used by t_2, the side-effect should be preserved, or its compensation should result in termination or compensation of t_2.

Transactional dependencies should respect restrictions with respect to process structure and activation dependency between tasks, as follows:

R_1: A commitment dependency of t_2 on t_1 can't exist if t_2 and t_1 are exclusive.

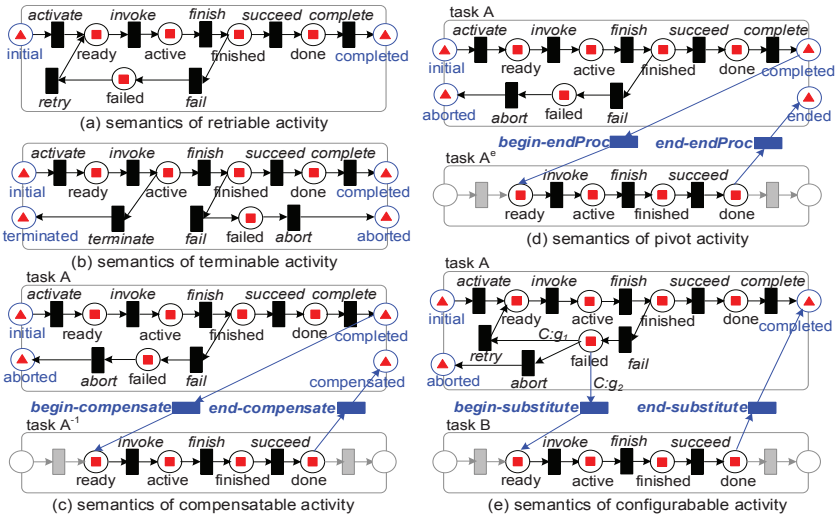

Figure 2. Execution semantics of different transactional behaviors of one task

R_2: An abortion dependency of t_2 on t_1 exists only if t_2 and t_1 are executed in parallel.

R_3: A compensation dependency of t_2 on t_1 exists only if (1) there is an activation dependency of t_1 on t_2, or t_2 and t_1 are executed in parallel, and (2) t_2 is compensatable.

R_4: A termination dependency of t_2 on t_1 exists only if (1) t_2 and t_1 are executed in parallel, (2) t_2 is a terminable goal-related task, and it has a negative-positive goal shift dependency on t_1.

R_5: A configuration dependency of t_2 on t_1 exists only if (1) t_2 and t_1 are executed in parallel, (2) t_1 is a goal-related task and its goal may make the retry condition or substitute condition of t_2 become true.

An EEZ-Flow potentially defines allowed transactional dependencies (ALDs) by its patterns, which include sequential, parallel and exclusive selection. User-defined transactional dependencies should be a subset of *ALDs*. Using aforementioned rules, we can deduce *ALDs* for each pattern (ignored for space limitations).

ATSs is a set of all accepted termination states of a transactional process *tp* as required by designers. So, we use ATSs to express the correctness of *tp* from the designer's point of view. ATSs must be carefully designed. Typically, it includes: (1) all composing activities reach *completed* state; (2) some activities fail, and their previous activities undo all the undesired side-effects; (3) activities fail after some pivot activities have completed: activities between the last pivot activity and the failed activity have been compensated, and the last pivot activity has been handled to *ended*. An execution is correct iff it terminates at an accepted termination state.

After a process p completes execution, each task may terminate at one of its possible termination states. Several different execution traces can be initiated according to the same schema and their termination states may vary from each other. The termination state of an execution trace of p composed of n tasks is a tuple $(s_1, s_2, \ldots, s_n)$, where s_i is the termination state of task t_i. The trace is correct if $(s_1, s_2, \ldots, s_n)$ belongs to p's ATSs.

We now incorporate business goals, transactional behaviors, transactional dependencies and ATSs into workflow schema to obtain a transactional workflow.

Definition 3.2. A TxEEZ-Flow schema is a six-tuple *TW=(W, B, BR, H, D, ATSs)* where *W* is a workflow schema; *B* is a set of business goals; *BR* is a set of relationships of goals in *B*; *ATSs* is a set of accepted termination states of *TW*; and

- *H* maps each task *t* in Σ to a tuple: h(t)=(type, [(hName, preCond, task, adjRule, pri)]), where *type* specifies the transactional behavior type of *t*, and the latter component describes failure handling mechanisms. *hName* represents handling method such as *retry* or *compensate*; *preCond* describes the condition under which the handling method can be applied; *task* describes additional task needed to implement the method; *adjRule* specifies the transactional adjustment of *t* if necessary; *pri* describes the priority;
- *D* maps each task *t* in Σ to a tuple which specifies its transactional dependencies: d(t)=(*CmtSet, AbtCond, CpsCond, TmtCond, ConfSet*), where *CmtSet* represents tasks with commitment dependency on *t*; *AbtCond* describes events that can cause abortion of *t*; *CpsCond* describes events that can lead to compensation of *t*; *TmtCond* describes termination mode and events that can lead to termination of *t*; and *ConfSet* represents tasks on which *t* has configuration dependency.

The most important issue here is how to ensure specified ATSs is correct against business goals, activity transactional behaviors and dependencies. Next, we will discuss this issue.

III. Transactional Validity and Configuration

We use ATSs as the correct criteria of the transactional workflow. There must be a corresponding transactional mechanism to ensure the workflow execution to comply with specified ATSs. The overview of our approach is shown in Fig. 3. It includes three phases: (1) Transactional workflow definition; (2) workflow validity verification; (3) dynamic transactional configuration and re-verification.

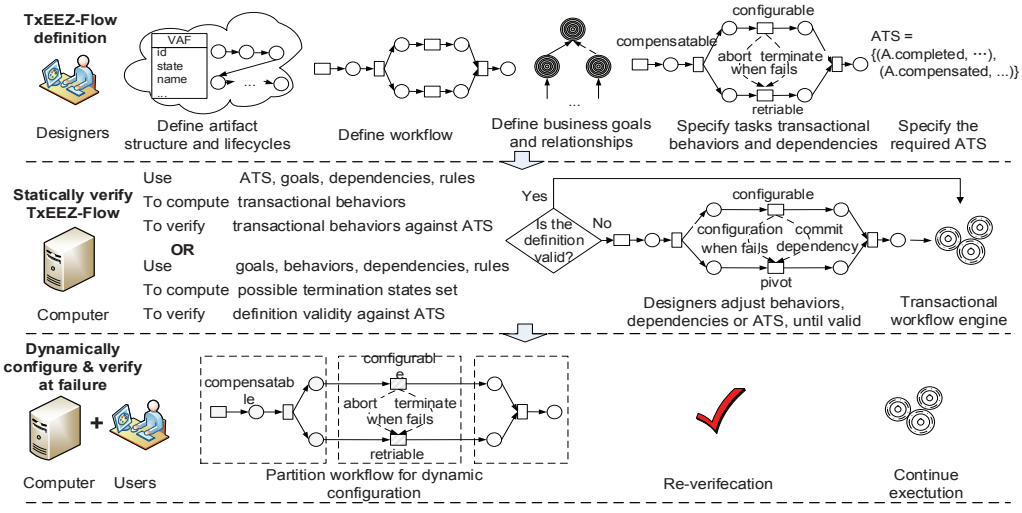

Figure 3. Approach Overview

Transactional workflow definition. Designers firstly define business goals and their relationships after finishing workflow definition, then add transactional behaviors and dependencies allowed by the patterns, and finally specify ATSs.

Workflow validity verification. Two verification methods are provided: (1) using ATS to compute behaviors of tasks according to a set of rules, and then verifying the consistency of computed behaviors with specified ones; (2) using transactional behaviors to compute possible termination

state sets, and then verifying the consistency of the computed sets with specified ATSs. If the schema is invalid, designers should adjust transactional behaviors, dependencies or ATSs. Once it becomes valid, it can be deployed and executed.

Dynamic transactional configuration and re-verification. In case the execution encounters failures, we may need to configure the workflow and re-verify its validity.

A. Static ATSs Verification

We use *STVP* to illustrate the transactional validity rules and how they are used to verify workflow validity. Tab.I shows an example of specified ATSs. In order to ensure the validity of a workflow, we need to verify: (1) ATSs is reasonable. ATSs must be consistent with workflow pattern semantics; (2) ATSs is consistent. There is no conflict among all *atss* in ATSs; (3) ATSs is valid. ATSs must comply with requirements of business goals, transactional behaviors and dependencies.

Reasonability verification of ATSs checks if there exist allowed transactional dependencies that can be satisfied by each *ats* in ATSs. ATSs consistency verification requires the state of $ats_i(t)$ and $ats_j(t)$ must be the same if states of all other tasks in ats_i and ats_j are the same, except that $ats_i(t)$ or $ats_j(t)$ is *terminated*. Because termination dependency has *may-mode*, state of *t* can be *terminated* or others (such as *ended*) when the termination conditions are satisfied. After transactional behaviors of tasks from ATSs and goal dependencies are calculated, conflicts between different *atss* can be detected. ATSs validity verification checks if there exist specified transactional dependencies (include those introduced by goal dependencies) that can be satisfied by each *ats* in ATSs, which can be summarized as validity rules.

We identify transactional aspects of each task *t* by: (1) behavior types; (2) transactional dependency conditions, including: *CpsCond(t)* for compensation conditions and *TmtCond(t)* for termination conditions; (3) transactional dependency task set, including: *CmtSet(t)* for commitment dependency and *ConfSet(t)* for configuration dependency. All possible termination states accepted by *t* in ATSs, are denoted as *ATS(t)*. We can deduce the following validity rules:

TABLE I. SPECIFIED ATSS OF THE *STVP* PROCESS

tasks / ats	GI	AS	CO	VD	VH	VZ	SS
ats_1	completed	completed	completed	completed	-	completed	completed
ats_2	completed	completed	completed	-	completed	completed	completed
ats_3	completed	completed	completed	aborted	completed	completed	completed
ats_4	completed	completed	completed	-	completed	ended	aborted
ats_5	completed	compensated	ended	aborted	aborted	-	-
ats_6	completed	aborted	ended	-	-	-	-
ats_7	compensated	compensated	aborted	-	-	-	-
ats_8	compensated	aborted	aborted	-	-	-	-
ats_9	aborted	-	-	-	-	-	-

VR_1: if state *aborted* does not belong to *ATS(t)*, task *t* should be *retriable*;

VR_2: if state *compensated* does not belong to *ATS(t)*, task *t* should be *pivot*;

VR_3: if state *compensated* belongs to *ATS(t)*, say ats_i, then (1) task *t* should be compensatable; and (2) there should exist a condition in *CpsCond(t)* to be satisfied in ats_i;

VR_4: if state *ended* belongs to *ATS(t)*, say ats_i, then (1) task *t* should be pivot; and (2) there should exist a condition in *CpsCond(t)* to be satisfied in ats_i;

VR_5: if state *terminated* belongs to *ATS(t)*, say ats_i, then (1) task *t* should be terminable; and (2) there should exist a condition in *TmtCond(t)* to be satisfied in ats_i;

VR_6: if state *completed* belongs to *ATS(t)*, say ats_i, states in ats_i of all tasks contained in *CmtSet(t)* should be *completed*;

VR_7: if task *t* is goal-related, say g_1, and g_1 has negative-positive shift dependency on g_2, then (1) *t* should be terminable; (2) tasks related with g_2 should be contained in *TmtCond(t)*; and (3) conditions in step (2) should be satisfied in ATSs;

VR_8: if task *t* is configurable, then (1) all tasks belonging to *ConfSet(t)* should be pivot; or (2) compensation of the compensatable tasks in *ConfSet(t)* are contained in *CpsCond(t)* and they are satisfied in ATSs.

Algorithm 1 *algTaskValidity* is developed to verify task validity using validity rules, returning useful suggestions to designers to adjust transactional specification including lacking behavior types, dependencies and unreasonable parts of ATSs. Function *Check* in the algorithm checks reasonability and validity of various types of tasks according to *ats*, task's state in the *ats*, transactional dependencies, behavior type and allowed dependencies, verifying if there exist specified transactional dependencies and allowed transactional dependencies can be satisfied by the *ats*. A task is valid if its validity verification result is an empty array, otherwise, adjustment suggestions will be returned. A workflow is valid if all its composing tasks are valid. Suppose the number of *ats* in ATSs is *m*, task number in the workflow is *n*, then complexity of algorithm *algTaskValidity* is *O(m)* and that of the whole workflow verification is *O(m*n)*. Task may combine behaviors *terminable*, *configurable* with *pivot* or *compensatable* and its possible accepted termination states may be four; however, only state *completed* can continue the execution and the other three states are execution termination states. In XOR pattern, accepted states number will be multiplied by its branches number. Suppose total XOR branches as *a*, *m* can be estimated as $n < m < a * (3n + 1)$. The complexity of the whole workflow verification comes to $O(n^2)$.

ATSs design of a workflow becomes complicated when its composing tasks increase. In this case, ATSs can firstly be computed from business goals, transactional behaviors and dependencies abiding by validity rules, then filtered and optimized by the workflow designer according to their specific business. The approach is beyond the scope of this paper.

Algorithm 1 Algorithm for task validity verification

Input:
ATS: designer's ATSs
h(t): failure behaviors of task t, h(t).type represents behavior types
d(t): transactional dependencies of t, such as CmtSet, CpsCond, etc.
alD(t): allowed dependencies of *t*: AlCpsCond, AlTmtCond
Output:
val: a string array containing: (1)be: lacking behavior types; (2)de: lacking dependencies; (3)atsRs: unreasonability of ATSs
if $t.aborted \notin ATS(t)$ and $retriable \notin h(t).type$ **then**
$val.be \leftarrow val.be \cup$ "$retriable$"
if t.compensated ∉ ATS(t) and pivot ∉ h(t).type **then**
$val.be \leftarrow val.be \cup$ "$pivot$"
ats←next *ats* in *ATS*
while *ats* is not null **do**
if *ats(t)* is *compensated* **then** //check compensation condition
cpsChk = Check(*ats, "compensatable", d(t).CpsCond, h(t).type, alD(t).AlCpsCond*)
if *ats(t)* is *ended* **then** //check end condition of a pivot task
pvtChk = Check(*ats, "pivot", d(t).CpsCond, h(t).type, alD(t).AlCpsCond*)
if *ats(t)* is *terminated* **then** //check termination condition
tmtChk = Check(*ats, "terminable", d(t).TmtCond, h(t).type, alD(t).AlTmtCond*)
if *ats(t)* is *completed* and exist a t_i in *d(t).CmtSet* satisfies $ats(t_i) \neq completed$ **then**
$val.atsRs \leftarrow val.atsRs +$ "(" + t_i + " → " + t + ")" + "$commit\ dependency\ violated$" //commitment
ats←next *ats* in *ATS*
end while
if exist shft(getRelatedTasks^{-1}(t), g_2)) **then**
shftChk = CheckShft2Tmt (*ATS, h(t).type, getRelatedTasks(g_2), d(t).TmtCond*)
if d(t).ConfSet ≠ null **then** //check configuration condition
confChk = CheckCofig(*ATS,d(t).ConfSet, h(t).type*)
val←*val + cpsChk + pvtChk + tmtChk + shftChk + confChk*
return *val*

B. Dynamic Transactional Configuration and Validity

Static ATSs verification of a workflow is taken at design time. Its validity may be violated at runtime, e.g., a failed task is substituted by a service with different transactional behaviors. Thus, the workflow needs to be re-configured to keep valid against ATSs. Consider a workflow consisting of a set of tasks T={t_1,t_2,…,t_n}. *T* can be partitioned into four regions based on tasks' execution states: (1) completed region R_c: containing completed tasks; (2) active region R_a: containing tasks that are being executed; (3) to-be executed region R_t: containing tasks that are inactive but may be executed later; (4) unselected region R_u: containing tasks in conditional branches (e.g., XOR pattern) that are not selected for the current execution. Situations that a workflow needs to be dynamically re-configured and re-verified can be concluded as follows.

1. Configurable task contained in R_a encounters exceptions during its execution and it is substituted by a service with different transactional behaviors. Exceptions can be errors occurrence or failing to achieve business goal.

2. Negative-positive shift dependencies between business goals have been changed because of changes in enterprise strategies or relative national policies. Meanwhile there exist goal-related tasks of business goals involved in these dependencies contained in R_a or R_t.

3. Transactional dependencies are changed because of changes of business logics. The change need to be re-verified only if (1) none of the changed tasks in *CmtSet* (or *CpsCond*) is contained in R_c or R_u, or (2) it is a termination (or configuration) dependency and none of the involved tasks is contained in R_c or R_u.

In case 1, configurable task substitution is re-verified using validity rule VR_8. The re-verification only involves compensation dependencies between *t* and *ConfSet(t)*, and between *ConfSet(t)* and substituting task of *t*, and transactional behaviors of *t*. It only takes a little time and is easy to implement.

In case 2, changes in negative-positive shift dependencies is re-verified using validity rule VR_7. Suppose a termination dependency *tmt*, in which task *t* depends on t_d, is introduced by the changed shift dependency *nps*. When *nps* is deleted, *tmt* will be removed, and *atss* that contain $ATS(t_d)$ with value *terminated* or *aborted* will be adjusted according to the left termination dependencies of t_d. When *nps* is newly added, $t_d.aborted$ and $t_d.terminated$ will be added to *TmtCond(t).event,* and *atss* that contain $ATS(t_d)$ with value *terminated* or *aborted* will be adjusted according to new termination dependency set of t_d. Modification of *nps* can be viewed as first deleting the old dependency and then add the new one.

Consider a changes of commitment dependency in case 3, we will recheck *atss* whose state of task in changed *CmtSet* is *completed*. For an added task *t*, only reserve *atss* whose states of *t* are *completed* and remove all others. For a removed task *t*, we check allowed states of *t* and add other *atss* if necessary. Similarly, it needs to re-compute the *atss* whose state of the added (or removed) task is *compensated* in compensation dependency change. The re-verification of termination dependency change is similar to that of case 2.

IV. IMPLEMENTATION

We are currently developing a prototype (written in C#) that supports this work. We describe the implementation of two major components of the TxEEZ-Flow architecture (shown in Fig. 4): transaction manager and transactional engine.

Transaction manager is the core module which provides transaction definition, verification and transactional execution

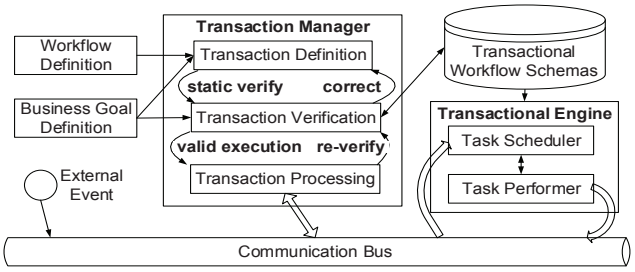

Figure 4. Architecture of TxEEZ-Flow

monitoring. Transaction verification module verifies workflows validity against their specified ATSs statically or dynamically. Transaction Processing (TP) module guarantees the reliable execution of workflows. It can handle the following exceptions: task termination, execution error and goal-achieving failure.

The transactional engine executes the transactional workflow. It is event-driven and consists of two main components: task scheduler and task performer. Task scheduler arranges next task to be executed according to received events, including: internal events, external events, and transactional events sent from TP in such form: *task@place*, *enactment*, *command* and *operation set*. When the scheduler receives a transactional event, it sends command to the executing task performer and passes event content to a new task performer instance if exists an operation in the operation set, such as *substitution, task*. Task performer reliably executes transactional tasks assigned by the scheduler. If exceptions occur, the performer will generate a transactional exception event in such form: *task@place*, *enactment*, *exception type*. Once receiving response from the scheduler, it performs the command of the transactional event. After the performer completes a task execution, it generates events according to the definition of the task.

V. Related Work

Business process transaction modeling and management have been studied over a decade. In recent years, transaction adaptability has been introduced in more and more approaches to meet designers' customized transactional requirements. Our work is in particular related to (1) customized failure atomicity of workflow; (2) effective forward recovery mechanism, and (3) reliable goal-oriented process model.

Research in the area of customized atomicity allows designers to specify failure atomicity according to different requirements. In [1, 6-7], ATSs is used to express user-specified failure atomicity requirements. The issue how to ensure every execution satisfies the designed ATS is discussed, and a mechanism is proposed to ensure the correctness of composite web service against specified ATS at design time. Runtime issues such as failure recovery are not discussed. In [3], authors use accepted configuration as relaxed atomicity. Execution correctness against designed atomicity is enforced by an algorithm. However, all these efforts have not addressed the influence of business goals on transaction management or run-time configuration in terms of improving execution success rate.

To the best of our knowledge, there is not much research on enhancing forward recovery using runtime data. Work in [8] provide two forward recovery approaches in logistics area: change execution order of the failed instance and retry it later; and substitute a failed activity using information carried by the activity. Work in [9] proposes an *error prevention* mechanism and supports correctness and robustness in dynamic process changes. However, the side-effects of other activities can used to recover the failed one, and transactional changes introduced by changes in business logic are not considered in these work.

Research efforts on high level business transaction requirements [10] is also rare. Some work has been done in managing goal-oriented business processes automatically [2], and improving process agility [11] or behavioral customization [12]. However, reliability execution issues are not discussed.

The above work has provided insights in customized atomicity ensuring methods and goal-oriented process managing techniques, which can be used as a foundation for our work. However, our work focuses on a transactional management solution that supports high-level transactional requirements and provides correctness ensuring techniques.

VI. Conclusions

Ensuring process execution reliability and success is crucial for BPM and remains challenging, especially for high-level transactional requirements management and positive failure handling mechanism. In this paper we proposed a goal-based transactional artifact-centric workflow model TxEEZ-Flow, in which high-level requirements can be described using goal model. The reflection of high-level requirements on transaction management, positive failure handling, and static as well as dynamic process validity verification method, are specified. Effectiveness and flexibility of TxEEZ-Flow are analyzed using a short term visit program of high school. The approach is also suitable for other areas, such as project application in iron and steel industry and order processing in manufacture enterprises.

References

[1] S. Bhiri, O. Perrin, and C. Godart, "Ensuring required failure atomicity of composite Web services," In Proc. Int. conf. World Wide Web (WWW), ACM, Chiba, 2005, pp. 138–147.

[2] D. Greenwood, G. Rimassa, "Autonomic goal-oriented business process management," In Proc. 3rd Int. Conf. on Autonomic and Autonomous Systems (ICAS), 2007, pp. 43–43.

[3] X. Ding, J. Wei, T. Haung, "User-defined atomicity constraints: a more flexible transaction model for reliable service composition," LNCS, vol. 4260, Springer, 2006, pp. 168–184.

[4] H. Schuldt, G. Alonso, C. Beeri, and H. J. Schek, "Atomicity and Isolation for Transactional Processes," TODS. 27, 2002, pp. 63–116.

[5] W. Xu, J. Su, Z. Yan, J. Yang, and L. Zhang, "An Artifact-centric approach to dynamic modification of workflow execution," In OTM 2011, R. Meersman, Dillon, and P. Herrero, eds. LNCS, vol. 7044, Heidelberg: Springer, 2011, pp. 256–273.

[6] I. Markov, M. Kowalkiewicz, "Linking business goals to process models in semantic business process modeling," In Proc. 12th Int. Conference on Enterprise Distributed Object Computing (EDOC), 2008, pp. 332–338.

[7] M. Rusinkiewicz, A. Sheth, "Specification and execution of transactional workflows," In Modern Database Systems: The Object Model, Interoperability, and Beyond. W. Kim Eds., New York: ACM Press and Addison-Wesley, 1995.

[8] S. Rinderle, M. Reichert, "Data-driven process control and exception handling in process management systems," In LNCS, vol. 4001, E. Dubois, and K. Pohl, eds. CAiSE 2006. Heidelberg: Springer, 2006, pp. 273–287.

[9] A. Lanz, M. Reichert, and P. Dadam, "Robust and flexible error handling in the AristaFlow BPM Suite," In CAiSE Forum 2010, P. Soffer and E. Proper, eds. LNCS, Heidelberg: Springer, 2010, pp. 174–189.

[10] J. Yang, C. Liu, and Y. Zhang, "Collaborative business transaction management: issues and challenges," In ICWS, 2005, Tutorial.

[11] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa, "Bdi-agents for agile goal-oriented business processes," In AAMAS'08, Berger, Burg, Nishiyama eds. 2008, pp. 37-44.

[12] S. Liaskos, M. Litoiu, M. Jungblut, and J. Mylopoulos, "Goal-based behavioral customization of information systems," In CAiSE 2011, H. Mouratidis, C. Rolland eds. LNCS, Heidelberg: Springer, 2011, pp. 77–92.