Compensable WorkFlow Nets

Fazle Rabbi, Hao Wang, and Wendy MacCaull*

Centre for Logic and Information
St. Francis Xavier University
{x2010mcf,hwang,wmaccaul}@stfx.ca

Abstract. In recent years, Workflow Management Systems (WfMSs) have been studied and developed to provide automated support for defining and controlling various activities associated with business processes. The automated support reduces costs and overall execution time for business processes, by improving the robustness of the process and increasing productivity and quality of service. As business organizations continue to become more dependant on computarized systems, the demand for reliability has increased. The language *t-calculus* [8] was developed to aid in the creation and verification of compensable systems. Motivated by this we define *Compensable WorkFlow nets* (CWF-nets) and introduce a graphical modeling language *Compensable Workflow Modeling Language* (CWML). We present a case study, using CWML to model a real world scenario, translate the resulting CWF-net into DVE (the input language of the DiVinE model checker) and verify properties of interest.

1 Introduction

A traditional system which consists of ACID (Atomicity, Consistency, Isolation, Durability) transactions cannot handle *long lived transaction* as it has only a flow in one direction. A long lived transaction system is composed of sub-transactions and therefore has a greater chance of partial effects remaining in the system in the presence of some failure. These partial effects make traditional rollback operations infeasible or undesirable. A *compensable transaction* is a type of transaction whose effect can be semantically undone even after it has committed [1]. A compensable transaction has two flows: a forward flow and a compensation flow. The forward flow executes the normal business logic according to system requirements, while the compensation flow removes all partial effects by acting as a backward recovery mechanism in the presence of some failure.

The concept of a compensable transaction was first proposed by Garcia-Molina and Salem [2], who called this type of long-lived transaction, a *saga*. A saga can be broken into a collection of sub-transactions that can be interleaved in any way with other sub-transactions. This allows sub-transactions to commit prior to the completion of the whole saga. As a result, resources can be released earlier and the possibility of deadlock is reduced. If the system needs to rollback in case of some failure, each sub-transaction executes an associated compensation

* Three authors contributed equally to this paper.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 122–137, 2010.

to semantically undo the committed effects of its own committed transaction. Bruni et al. worked on long running transactions and compared Sagas with CSP [3] in the modelling of compensable flow composition [4].

In recent years, He et al. [5-8] have developed a specification for compensable transactions in order to provide increased system reliability. Ideally, such compensable transactions would be used to model a larger computer system, by composing several compensable sub-transactions to provide more complex functionality. These transactions would provide backward recovery if an intermediate error were to occur in the larger system. The result of this research was the creation of a transactional composition language, the t-calculus [8].

Workflow management systems (WfMS) provide an important technology for the design of computer systems which can improve information system development in dynamic and distributed organizations. Motivated by the petri net [9] and t-calculus [8] formalisms and the graphical representations underlying the YAWL [11] and ADEPT2 [12] modeling languages, we define *Compensable WorkFlow nets* (CWF-nets) and develop a new workflow modeling language, the *Compensable Workflow Modeling Language* (CWML). In addition, we present a verification method for CWF-nets using model checkers. Using this approach, both requirements and behavioural dependencies of transactions (specified in a temporal logic) can be verified efficiently. DiVinE [14], a well known distributed model checker, is used in the case study.

Section 2 provides some background information. The CWF-nets and the graphical modeling language, CWML, are defined in section 3. Section 4 presents the verification method. A case study is provided in section 5, and section 6 concludes the paper.

2 Background

Petri nets [9], developed by Petri in the early 1960s, is a powerful formalism for modeling and analyzing process.

Definition 1. *A petri net is a 5-tuple,* $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of places,*
- $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
- W*:* $F \rightarrow \{1,2,3,...\}$ *is a weight function,*
- M_0*:* $P \rightarrow \{0,1,2,3,...\}$ *is the initial marking,*
- $P \cap T = \phi$ *and* $P \cup T \neq \phi$.

A petri net structure $N = (P, T, F, W)$ *without any specific initial marking is denoted by N.*

Places may contain tokens and the distribution of tokens among the places of a petri net determine its *state* (or *marking*).

The coloured petri Net [10] is an extension to the petri net, where the tokens are valued and typed so that they can be distinguished.

Definition 2. *A coloured petri net is a 5-tuple,* $CPN = (P, T, C, IN, OUT)$ *where:*

- P *is a finite set of places,*
- T *is a finite set of transitions,*
- C *is a colour function such that* $C : P \cup T \rightarrow$ *Non-empty sets of colours,*
- *IN and OUT are functions with domain* $(P \times T)$ *such that for all* $(p,t) \in P \times T$, $IN(p,t)$, $OUT(p,t):C(p) \rightarrow [C(t) \rightarrow N]_f$, *where* $[C(t) \rightarrow N]_f$ *denotes the set of all total functions g from* $C(t)$ *to N with the support* $a \in C(t):g(a) \neq 0$ *finite.*

A compensable transaction refers to a transaction with the capability to withdraw its result after its commitment, if an error occurs. A compensable transaction is described by its external state. There is a finite set of eight independent states, called *transactional states*, which can be used to describe the external state of a transaction at any time. These transactional states include `idle` (`idl`), `active` (`act`), `aborted` (`abt`), `failed` (`fal`), `successful` (`suc`), `undoing` (`und`), `compensated` (`cmp`), and `half-compensated` (`hap`), where `idl`, `act`, etc are the abbreviated forms. Among the eight states, `suc`, `abt`, `fal`, `cmp`, `hap` are the terminal states. The transition relations of the states are illustrated in Fig. 1.

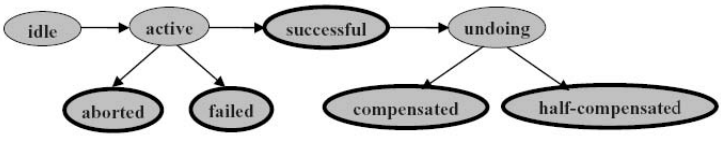

Fig. 1. State transition diagram of compensable transaction [1]

Before activation, a compensable transaction is in the `idle` state. Once activated, the transaction eventually moves to one of five terminal states. A `successful` transaction has the option of moving into the `undoing` state. If the transaction can successfully undo all its partial effects it goes into the `compensated` state, otherwise it goes into the `half-compensated` state. An ordered pair consisting of a compensable transaction and its state is called an *action*. Actions are the key to describing the behavioural dependencies of compensable transactions. In [1], five binary relations were proposed to define the constraints applied to actions on compensable transactions. Informally the relations are described as follows, where both a and b are actions:

1. $a < b$: only a can fire b.
2. $a \prec b$: b can be fired by a.
3. $a \ll b$: a is the precondition of b.
4. $a \leftrightarrow b$: a and b both occur or both not.
5. $a \nleftrightarrow b$: the occurance of one action inhibits the other.

The transactional composition language, t-calculus [8], was proposed to create reliable systems composed of compensable transactions. In addition, it provides flexibility and specialization, commonly required by business process management systems, with several alternative flows to handle the exceptional cases.

The syntax of t-calculus is made up of several operators which perform compositions of compensable transactions. Table 1 shows eight binary operators, where S and T represent arbitrary compensable transactions. These operators specify how compensable transactions are coupled and how the behaviour of a certain compensable transaction influences that of the other. The operators are discussed in detail in [1,5,6] and described in section 3.

Table 1. t-calculus syntax

Sequential Composition	$S \ ; \ T$	Parallel Composition	$S \ \| \| \ T$
Internal Choice	$S \sqcap T$	Speculative Choice	$S \otimes T$
Alternative Forwarding	$S \rightsquigarrow T$	Backward Handling	$S \trianglerighteq T$
Forward Handling	$S \triangleright T$	Programmable Composition	$S \circledast T$

3 Workflow with Compensable Transactions

A workflow consists of steps or tasks that represent a work process. We extend the task element with the concept of compensable transaction, and accordingly adapt t-calculus for the composition of a compensable workflow. In this section, first, we present the formal syntax and semantics of Compensable Workflow nets; second, we present the graphical representation of the language, and finally, we give formal analysis of the language.

3.1 Compensable Workflow Nets

We build a compensable workflow using a group of tasks connected with operators; a task handles a unit of work and operators indicate the nature of the flow (forward and/or backward) relations between tasks. In particular, we define a colored petri net part for each task, therefore a CWF-net corresponds to a complete colored petri net. We use the colored petri net because in reality, a workflow handles several job instances (with different progress) at the same time so that the token of each color can be used to represent a specific job instance.

An atomic task [13] is an indivisible unit of work. Atomic tasks can be either compensable or uncompensable. We follow the general convention [13] of assuming that if activated, an atomic uncompensable task always succeeds.

Definition 3. *An atomic uncompensable task t is a tuple* (s, P_t) *such that:*

- P_t *is a petri net, as shown in Fig. 2;*
- s *is a set of unit states* $\{$*idle, active, successful*$\}$*; the unit state idle indicates that there is no token in* P_t*; the unit states active and successful indicate that there is a token in the place p_act and p_succ respectively;*

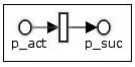

Fig. 2. Petri net representation of an uncompensable task

Remark that the unit states of a task are different from the state (marking) of a petri net. In addition, the unit state of a task is token-specific, i.e., a task is in a unit state for token(s) of a specific color and it may be in a different state for token(s) of another color.

Definition 4. *An atomic compensable task t_c is a tuple (s_c, P_{t_c}) such that:*

- *P_{t_c} is a petri net; as shown in Fig. 3;*
- *s is a set of unit states {idle, active, successful, undoing, aborted, failed}; the unit state idle indicates that there is no token in P_{t_c}; the other unit states indicate that there is a token in the relevant place;*

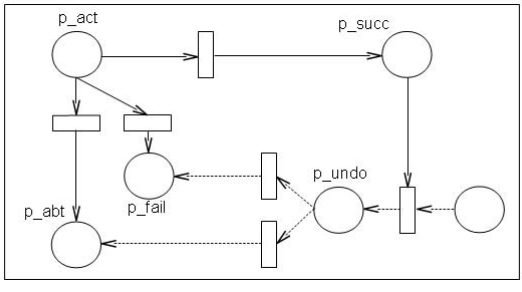

Fig. 3. Petri net representation of a compensable task

The compensation flow is denoted by the dotted arcs. Note that an atomic compensable task does not have the *compensated* and *half-compensated* states. We drop these states as their semantics overlap with the aborted and failed states. The *compensated* state means successful compensation, and corresponds to *aborted* after *successful*; the *half-compensated* state means error during compensation, resulting in data inconsistencies (i.e., remained partial effects); it corresponds to *failed* after *successful*.

The task t_c transits to the unit state of *active* after getting a token in `p_act`. The token can move to either `p_succ`, `p_abt` or `p_fail` representing unit states *successful*, *aborted* or *failed* respectively. The unit state *aborted* indicates an error occurred performing the task and the effects can be successfully removed. The backward (compensation) flow is started from this point. On the other hand, the unit state *failed* indicates that the error cannot be removed successfully and the partial effect will remain in the system unless there is an exception handler. Note that t_c can transit to the unit states *aborted* or *failed* either before or after the unit state *successful*.

A compensable task can be composed with other compensable tasks using t-calculus operators.

Definition 5. *A compensable task (ϕ_c) is recursively defined by the following well-formed formula: (Adapted from [1])*

$$\phi_c = t_c \mid (\phi_c \odot \phi_c)$$

where t_c is an atomic compensable task, and $\odot \in \{;, ||, \sqcap, \otimes, \rightsquigarrow, \trianglerighteq, \rhd, \divideontimes\}$ is a t-calculus operator defined as follows:

- ϕ_{c_1} ; ϕ_{c_2}*:* ϕ_{c_2} *will be activated after the successful completion of* ϕ_{c_1}*,*
- $\phi_{c_1} || \phi_{c_2}$*:* ϕ_{c_1} *and* ϕ_{c_2} *will be executed in parallel. If either of them (*ϕ_{c_1} *or* ϕ_{c_2}*) is aborted, the other one will also be aborted,*
- $\phi_{c_1} \sqcap \phi_{c_2}$*: either* ϕ_{c_1} *or* ϕ_{c_2} *will be activated depending on some internal choice,*
- $\phi_{c_1} \otimes \phi_{c_2}$*:* ϕ_{c_1} *and* ϕ_{c_2} *will be executed in parallel. The first task that reaches the goal will be accepted and the other one will be aborted,*
- $\phi_{c_1} \rightsquigarrow \phi_{c_2}$*:* ϕ_{c_1} *will be activated first to acieve the goal, if* ϕ_{c_1} *is aborted,* ϕ_{c_2} *will be executed to achieve the goal,*
- $\phi_{c_1} \trianglerighteq \phi_{c_2}$*: if* ϕ_{c_1} *fails during execution,* ϕ_{c_2} *will be activated to remove the partial effects remaining in the system.* ϕ_{c_2} *terminates the flow after successfully removing the partial effects,*
- $\phi_{c_1} \rhd \phi_{c_2}$*: if* ϕ_{c_1} *fails,* ϕ_{c_2} *will be activated to remove the partial effects.* ϕ_{c_2} *resumes the forward flow to achieve the goal,*
- $\phi_{c_1} \divideontimes \phi_{c_2}$*: if* ϕ_{c_1} *needs to undo its effect, the compensation flow will be redirected to* ϕ_{c_2} *to remove the effects.*

Any task can be composed with uncompensable and/or compensable tasks to create a new task.

Definition 6. *A task (ϕ) is recursively defined by the following well-formed formula:*

$$\phi = t \mid (\phi_c) \mid\mid (\phi \ominus \phi)$$

where t is an atomic task, ϕ_c is a compensable task, and $\ominus \in \{\wedge, \vee, \times, \bullet\}$ is a control flow operator defined as follows:

- $\phi_1 \wedge \phi_2$*:* ϕ_1 *and* ϕ_2 *will be executed in parallel,*
- $\phi_1 \vee \phi_2$*:* ϕ_1 *or* ϕ_2 *or both will be executed in parallel,*
- $\phi_1 \times \phi_2$*: exclusively one of the task (either* ϕ_1 *or* ϕ_2*) will be executed,*
- $\phi_1 \bullet \phi_2$*:* ϕ_1 *will be executed first then* ϕ_2 *will be executed.*

A subformula of a well-formed formulae is also called a *subtask*. We remark that if T_1 and T_2 are compensable tasks, then $T_1;T_2$ denotes another compensable task while $T_1 \bullet T_2$ denotes a task consisting of two distinct compensable subtasks. Any task which is built up using any of the operators $\{\wedge, \vee, \times, \bullet\}$ is deemed as uncompensable.

In order for the underlying petri net to be complete, we add a pair of split and join routing tasks for operators including $\wedge$, $\vee$, $\times$, $||$, $\sqcap$, $\otimes$, and $\rightsquigarrow$ and we give

their graphical representation in section 3.2. Each of these routing tasks has a corresponding petri net representation, e.g., for the speculative choice operator $\phi_{c_1} \otimes \phi_{c_2}$, the split routing task will direct the forward flow to ϕ_{c_1} and ϕ_{c_2}; the task that performs its operation first will be accepted and the other one will be aborted.

Now we can present the formal definition of Compensable WorkFlow nets (CWF-nets):

Definition 7. *A Compensable Workflow net (CWF-net) C_N is a tuple (*$\mathtt{i}$, $\mathtt{o}$, T, T_c, F*) such that:*

- $\boldsymbol{i}$ *is the input condition,*
- $\boldsymbol{o}$ *is the output condition,*
- T *is a set of tasks,*
- $T_c \subseteq T$ *is a set of compensable tasks, and* $T \backslash T_c$ *is a set of uncompensable tasks,*
- $F \subseteq (\{\boldsymbol{i}\} \times T) \cup (T \times T) \cup (T \times \{\boldsymbol{o}\})$ *is the flow relation (for the net),*
- *the first atomic compensable task of a compensable task is called the initial subtask; the backward flow from the initial subtask is directed to the output condition,*
- *every node in the graph is on a directed path from* $\boldsymbol{i}$ *to* $\boldsymbol{o}$.

If a compensable task aborts, the system starts to compensate. After the full compensation, the backward flow reaches the initial subtask of the compensable task and the flow terminates, as the backward flow of an initial task of compensable tasks is connected with the output condition.

The reader must distinguish between the flow relation (F) of the net, as above and the internal flows of the atomic (uncompensable and compensable) tasks.

A CWF-net such that $T_c = T$ is called a *true Compensable workflow net* (CWF$_t$-net).

3.2 Graphical Representation of CWF-Nets

We first present graphical representation of t-calculus operators, then present the contruction principles for modeling a compensable workflow. Fig. 4 gives graphical represetation of tasks. Some of the operators are described in this section.

Sequential Composition. Two compensable tasks ϕ_{c_1} and ϕ_{c_2} can be composed with sequential composition as shown in Fig. 4, which represents the formula $\phi_{c_1} \; ; \phi_{c_2}$. Task ϕ_{c_2} will be activated only when task ϕ_{c_1} finishes its operations successfully. For the compensation flow, when ϕ_{c_2} is aborted, ϕ_{c_1} will be activated for compensation, i.e., to remove its partial effects.

Recall that in CWF-nets, we drop the *compensated* and *half-compensated* states because their semantics overlap with the *aborted* and *failed* states; therefore, we do not consider the two states in the behavioural dependencies. The following two basic dependencies [1] describe behaviour of sequential composition: **i)** $(\phi_{c_1}, suc) < (\phi_{c_2}, act)$; **ii)** $(\phi_{c_2}, abt) \prec (\phi_{c_1}, und)$;.

Theorem 1. *The above dependencies hold in the petri net representation (as in Fig. 3) of sequential composition.*

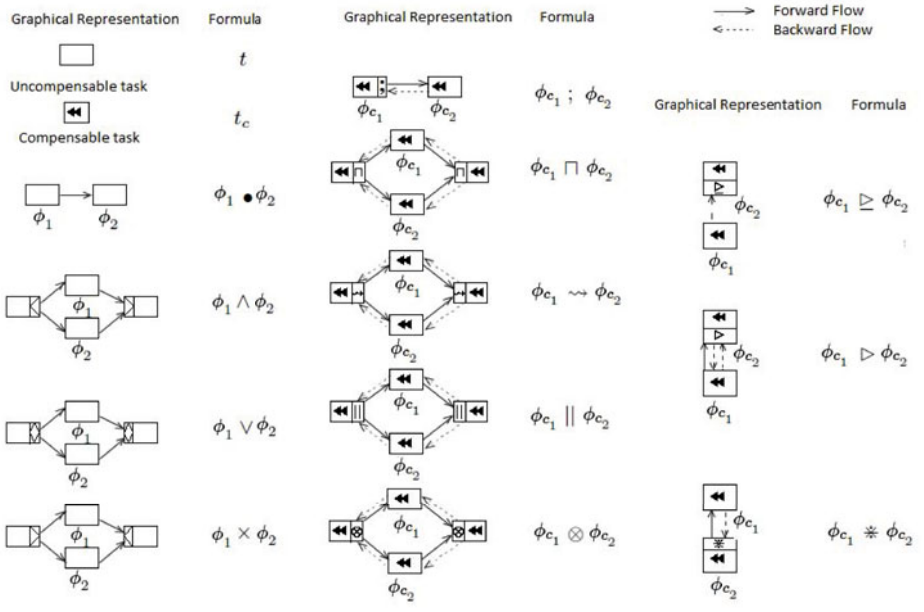

Fig. 4. Graphical Representation of Tasks

Proof: It is straight forward when we interpret the two tasks using the petri net in Fig. 3.

Theorem 2. *The above dependencies hold in the DVE code of sequential composition*

Proof: It is straight forward when we look at the DVE code for sequential composition which can be found in section 4.

Parallel Composition. Compensable tasks that are composed using parallel composition are executed in parallel. If one of the parallel tasks or branches fails or aborts then the entire composed transaction will fail or abort, as a composed transaction cannot reach its goal if a sub-transaction fails. The petri net representation of parallel composition of two compensable tasks ϕ_{c_1} and ϕ_{c_2} ($\phi_{c_1} \mid\mid \phi_{c_2}$) is shown in Fig. 5. Furthermore, parallel composition requires that if one branch either fails or aborts then the other branch should be stopped to save time and resources. This is achieved by an internal mechanism called *forceful abort* (not shown in Fig. 5), which forcefully aborts a transaction and undos its partial effects. Details of forceful abort can be found from our website [16]. To sum up, when compensating, tasks which are composed in parallel are required to be compensate in parallel.

The related behavioural dependencies are formalized as: **i)** $(\phi_{c_1}, act) \leftrightarrow (\phi_{c_2}, act)$; **ii)** $(\phi_{c_1}, und) \leftrightarrow (\phi_{c_2}, und)$; **iii)** $(\phi_{c_1}, suc) \leftrightarrow (\phi_{c_2}, suc)$. Analogous to the dependencies of the sequential composition, these basic dependencies also

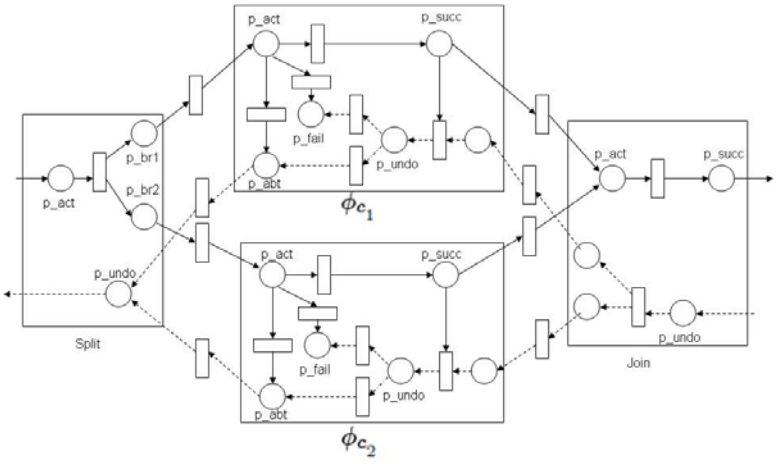

Fig. 5. Petrinet representation of parallel composition

holds in the petri net representation and in the DVE code of the parallel composition.

There can be more than two compensable tasks in parallel composition; for this composition all the branches will be activated and executed in parallel.

Alternative Forwarding. The alternative forwarding composition is used to decide between two or more equivalent tasks with the same goals. Alternative forwarding implies a preference between the tasks, and it does not execute all branches in parallel. Therefore if, for example, the alternative forwarding composition is used to buy air tickets, one airline may be preferred to the other and an order is first placed to the preferred airline. The other airline will be used to place an order only if the first order fails.

In Fig. 4, we can see the two tasks ϕ_{c_1} and ϕ_{c_2} which are composed by alternative forwarding. It represents the formula $\phi_{c_1} \rightsquigarrow \phi_{c_2}$. In this composition, task ϕ_{c_1} has higher priority and it will be executed first. Task ϕ_{c_2} will be activated only when task ϕ_{c_1} has been aborted or failed. In other words, ϕ_{c_1} runs first and ϕ_{c_2} is the backup of ϕ_{c_1}. The petri net representation of alternative forwarding composition of two compensable tasks ϕ_{c_1} and ϕ_{c_2} is shown in Fig. 6.

The basic dependency is described by: $(\phi_{c_1}, abt) < (\phi_{c_2}, act)$. Analogous to the dependencies of the sequential composition, this basic dependency also holds in the petri net representation and in the DVE code of the alternative forwarding composition.

Descriptions of Internal Choice, Speculative Choice, Backward handling, Forward handling and Programmable Compensations can be found from our website [16].

Construction Principle 1. *Construction principles for the graphical representation of tasks are as follows:*

- *The operators* $[\bullet, ;, \trianglerighteq, \triangleright, \divideontimes]$ *are used to connect the operand tasks sequentially. Atomic uncompensable tasks are connected by a single forward flow. Atomic*

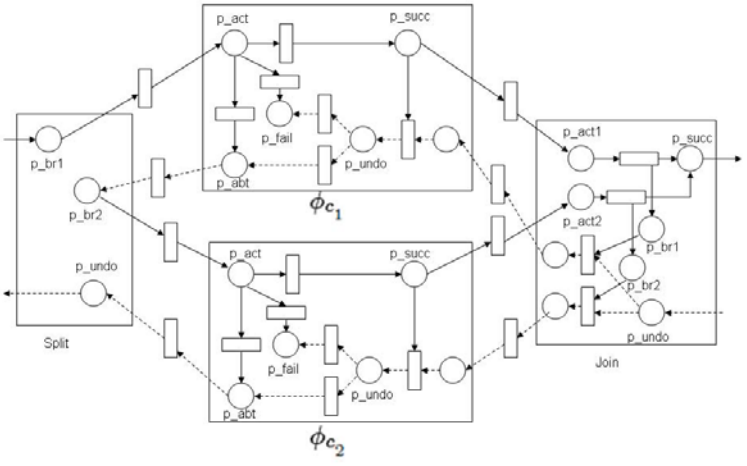

Fig. 6. Petrinet representation of alternative forwarding composition

compensable tasks are connected by two flows- one forward and one backward if they are connected by the sequential operator. Atomic uncompensable tasks and atomic compensable tasks are connected by a single forward flow;

- *The convention of ADEPT2 is followed in order to enable 'Poka-Yoke Workflows' [17], which supports "correctness by construction". A pair of split and join routing tasks are used for tasks composed by* $\{\wedge, \vee, \times, ||, \sqcap, \otimes, \rightsquigarrow\}$. *Atomic uncompensable tasks are connected with split and join tasks by a single forward flow. Atomic compensable tasks are connected with split and join tasks by two flows (forward and backward). The operators and its corresponding split and join tasks are shown in Table 2;*
- *Two split and join tasks for the same operator can be merged to a single split task (or join task) combining the branches. By this arrangement tasks can be composed in more than two branches of a split/join pair.*
- *Every compensable task is covered by an exception handler (forward or backward).*

If these principles are followed, the resulting graph is said to be "correct by construction".

Table 2. Operators and their associated split and join tasks

Operator	Split task	Join task
$\wedge$	AND-split	AND-join
$\vee$	OR-split	OR-join
$\times$	XOR-split	XOR-join
$\|\|$	PAR-split	PAR-join
$\sqcap$	INT-split	INT-join
$\otimes$	SPEC-split	SPEC-join
$\rightsquigarrow$	ALT-split	ALT-join

3.3 Analysis

We adapt the definition of soundness for CWF-net from [13]. Informally, the soundness of CWF-net require that for any case, the underlying coloured petri net will terminate eventually, and at the moment it terminates, there is a token in the output condition and all other places are empty. Formally, the soundness of CWF-net is defined as follows:

Definition 8. *A CWF-net* $C_N = (i, o, T, T_c, F)$ *is sound (or structurally sound) iff, considering the underlying petri net:*

1. *For every state (marking) M reachable from the initial state* M_i*, there exists a firing sequence leading from M to the final state* M_f*, where* M_i *indicates that there is a token in the input condition and all other places are empty and* M_f *indicates that there is a token in the output condition and all other places are empty. Formally:* $\forall M(M_i \rightarrow^* M) \Rightarrow (M \rightarrow^* M_f)$*;*
2. M_f *is the only state reachable from* M_i *with at least one token in the output condition. Formally:* $\forall_M(M_i \rightarrow^* M \wedge M \geq M_f) \Rightarrow (M = M_f)$*;*
3. *There are no dead transitions in* C_N*. Formally:* $\forall_{t \in T}, \exists_{M,M'} M_i \rightarrow^* M \rightarrow^t M'$*.*

We have the following theorem:

Theorem 3. *If a CWF-net is correct by construction, it is sound.*

Proof: Let C_N be a CWF-net which consists of some uncompensable and compensable tasks.

- *Case 1,* C_N consists of only one atomic task (t): as t is connected to the input condition and the output condition. t will be activated by the input condition and will continue the forward flow to the output condition, the flow will terminate. Therefore C_N is sound.
- *Case 2,* C_N consists of only atomic uncompensable tasks composed by operators $\{\wedge, \vee, \times, \bullet\}$: according to the construction principle, every split task must have a same type of join task. This pair of split and join tasks provides a safe routing of the forward flow; all the tasks of the workflow are on a path from the input condition to the output condition, which ensures that there is no dead task in the workflow and the flow always terminates. Therefore C_N is sound.
- *Case 3,* C_N includes some atomic uncompensable tasks and atomic compensable tasks: First let us consider that C_N has one atomic compensable task (t_c). t_c is activated by some atomic task or the input condition. If t_c is successful during the execution, it will activate the next task (or the output condition) by continuing the forward flow. If t_c is aborted, it will start the compensation flow. As this is the only compensable task (it is the initial task itself), the compensation flow is connected to the output condition. It is easy to see that if t_c is aborted, the flow also terminates. If t_c fails during execution, the error handler will be in effect and will either terminate the flow to the output condition (backward handler) or continue the forward

flow (forward handler). Therefore C_N is sound. Now let us consider there is more than one atomic compensable task in C_N. For every compensable task there is an initial subtask and the compensation flow of the initial subtask is connected to the output condition. If the compensable tasks do not fail, they will continue the forward flow until the output condition is reached. If the composition of compensable tasks is aborted, the compensation flow will reach the initial subtask, which will direct the compensation flow to the output condition. Therefore C_N is sound.

4 Verification

Once a workflow is designed with compensable tasks, its properties can be verified by model checkers such as SPIN, SMV or DiVinE. Modeling a workflow with the input language of a model checker is tedious and error-prone. Leyla et al. [19] translated a number of established workflow patterns into DVE, the input language of DiVinE model checker. Based on this translation, we proposed an automatic translator which translates a graphical model constructed using the YAWL editor to DVE, greatly reducing the overall effort for model checking (see [20] and the extension involving time in [23]).

DiVinE is a distributed explicit-state *Linear Temporal Logic* (LTL) model checker based on the automata-theoretic approach. In this approach, the system properties are specified as LTL formulas. LTL is a type of temporal logic which, in addition to classical logical operators, uses the temporal operators such as: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), and next time ($\bigcirc$) [15]. In the automata-theoretic approach, the system model and the property formula are each associated to an automaton. The model checking problem is reduced to detecting an accepting cycle in the product of the system model automaton and the negation of the property automaton. DiVinE provides several model checking algorithms which efficiently employ the computational power of distributed clusters. DVE is sufficiently expressive to model general problems.

In [1] two types of verification were proposed for the verification of compensable transactions: i) Acceptable Termination States (ATS), ii) Temporal Verification. However they cannot verify the compensable transactions with the whole workflow, and cannot handle the state space explosion problem. Our approach can verify the workflow with the compensable tasks. As we are using a distributed model checker, we can handle the large state space of a large model. Since LTL specification formulas can be used to verify the temporal specification of the workflow it is possible to check the Acceptable Termination States (ATS). Behavioural dependencies and requirement specifications both can be checked using this approach.

In the DVE translation, workflow processes, subprocesses and activities are mapped to DVE processes and control flows are managed using DVE variables, guards and effects. We give one example below.

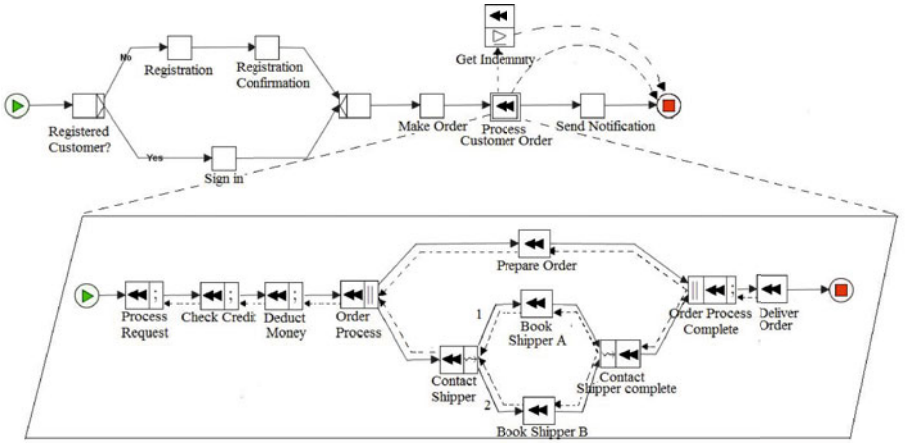

Fig. 7. Order processing workflow

Sequential Composition. Let A and B be two compensable tasks composed by the sequential operator (;). Each task corresponds to a process in DVE. When process A is activated, it can transit to `successful`, `aborted` or `failed`. B is activated only if A transits to `successful`. Once activated, B can also transit to `successful`, `aborted` or `failed`. If B is `aborted`, it sets positive value to the variable named B_abt (represents a token in its place p_abt).

```
int A_SUC = 0;
int A_ABT = 0;
. . . .
process A{
. . . .
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_SUC=A_SUC +1;},
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_ABT=A_ABT +1;},
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_FAIL= A_FAIL +1;},
tr -> tr { guard B_ABT > 0 ; effect B_ABT = B_ABT - 1  , A_ABT = A_ABT +1;},
tr -> tr { guard B_ABT > 0 ; effect B_ABT = B_ABT - 1 , A_FAIL = A_FAIL +1;};}
process B{
. . . .
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_SUC = B_SUC + 1;},
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_ABT = B_ABT + 1;},
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_FAIL = B_FAIL + 1;};}
```

5 Case Study

In this section we provide a case study adapted from [1]. A customer's order processing workflow is described here with compensable transactions. Only a registered customer can make an order to the company. If the customer is not registered, s/he performs the registration and then signs in to the system. Once the order is made, the system starts processing the order. The workflow is presented in Fig. 7. In the workflow *Process Customer Order* the process is a nested

compensable task which has been decomposed into a CWF_t-net . The workflow provides lots of flexibility and exception handling through which the system can compensate in exceptional scenario. The *Prepare Order* and *Contact Shipper* are processed in parallel as these are two time consuming tasks. However if either *Prepare Order* or *Contact Shipper* aborts, the other will be aborted immediately. In this example, the seller has two shippers (A and B). *Shipper A* is cheaper but hard to book whereas *Shipper B* is more expensive but is always available. To save money, *Shipper A* is preferred; *Shipper B* is booked only when *Shipper A* is unavailable. The selected shipper is responsible for delivering these items. Note that if the customer cancels the order during processing, the compensation for completed parts will be activated. When the compensation cannot properly undo the partial effects, the seller would ask for extra indemnities from the customer which is transacted by backward handling. Here the backward handler is composed with a nested task which will be activated if any of the compensable tasks fails inside the CWF_t-net.

5.1 Verification Results

We have developed an editor for compensable transaction which includes a translator that can translate a CWF-net model to DiVinE model checker. The process is fully automated and the tool is available in our website [16].

Reachability: The reachability result shows that there is no deadlock state and no error state in the system. There are a total of 27620 states and 56089 transitions in experiment. We verified five properties, due to space limits, we only give details for *Prop1: Customer cannot make order without sign in or registration.*

To verify this property we define a global integer variable in DVE named `customer_sign_in`. The initial value of the variable is set to 0. The value is changed to 1 when the customer is signed in, means when the process *Sign_in* is in successful state. We define three atomic propositions in the following way:

```
#define order_made ( Make_Order_SUC == 1 )
#define sign_in ( customer_sign_in == 1 )
#define registration ( customer_registration == 1 )
```

The LTL specification using these propositions for the property is:

```
G F !( !( sign_in || registration) && order_made )
```

Other properties and the corresponding LTL formulas:

- *Prop2: Shipper A and B will not be contacted at the same time.*
  `G F !( shipper_a_is_successful && shipper_b_is_successful )`
- *Prop3: If Prepare Order is aborted, the Order Process task is compensated.*
  `G( prepare_order_aborted → F( order_process_compensated ) )`
- *Prop4: Get Indemnity is activated if Book Shipper A fails to compensate.*
  `G ( shipper_a_failed → F( get_indemnity_for_deduct_money ) )`
- *Prop5: Any order is eventually delivered or compensated if no task fails.*
  `G ( order_made → F ( order_delivered || order_compensated ) )`

Table 3. Verification Results for the DiVinE model checker

Property	Acc Cycle	States	Memory (MB)	Time (s)
Prop1	No	27623	166.891	0.388079
Prop2	No	27620	166.891	0.377525
Prop3	No	28176	167.012	0.370857
Prop4	No	27626	166.895	0.35127
Prop5	No	42108	170.023	0.623221

6 Conclusion and Future Work

Over the last decade, there has been increasing recognition that modeling languages should be more expressive and provide comprehensive support for the control-flow, data, resource and exception handling perspectives. In this paper we have shown how a workflow can be better represented by the composition of compensable tasks. The idea of workflow with compensable task will add a new dimension of flexibility and exception handling.

This research is part of an ambitious research and development project, Building Decision-support through Dynamic Workflow Systems for Health Care [21] in a collaboration with Guysborough Antigonish Strait Health Authority (GASHA) and technology industrial partner. Real world workflow processes can be highly dynamic and complex in a health care setting. To manage the ad hoc activities efficiently, a flexible workflow system with better exception handling mechanism must be designed. Hao and MacCaull [22] developed several new Explicit-time Description Methods (EDM), which enable general model checkers like DiVinE to verify real-time models. Based on them, we are extending CWML to describe real time information in workflow models. More importantly, our group is developing an innovative workflow modeling framework named *NOVA Workflow* that is compeNsable, Ontology-driven, Verifiable and Adaptive.

Acknowledgment

This research is sponsored by Natural Sciences and Engineering Research Council of Canada, by an Atlantic Computational Excellence Network (ACEnet) Post Doctoral Research Fellowship and by the Atlantic Canada Opportunities Agency. The computational facilities are provided by ACEnet.

References

1. Li, J., Zhu, H., He, J.: Specifying and verifying web transactions. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 149–168. Springer, Heidelberg (2008)
2. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. 16(3), 249–259 (1987)
3. Butler, M., Hoare, T., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)

4. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing Two Approaches to Compensable Flow Composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
5. He, J.: Compensable programs. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 349–363. Springer, Heidelberg (2007)
6. He, J.: Modelling coordination and compensation. In: Leveraging Applications of Formal Methods, Verification and Validation, vol. 17, pp. 15–36 (2009)
7. Li, J., Zhu, H., He, J.: Algebraic semantics for compensable transactions. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 306–321. Springer, Heidelberg (2007)
8. Li, J., Zhu, H., Pu, G., He, J.: Looking into compensable transactions. In: The 31st IEEE Software Engineering Workshop, pp. 154–166. IEEE CS press, Los Alamitos (2007)
9. Murata, T.: Petri nets: properties, analysis, and applications. Proc. IEEE 77(4), 541–580 (1989)
10. Narahari, Y., Viswanadham, N.: On the Invariants of Coloured Petri Nets. In: Rozenberg, G. (ed.) APN 1985. LNCS, vol. 222, pp. 330–345. Springer, Heidelberg (1986)
11. van der Aalst, W.M.P., ter Hofstede, A.: YAWL: Yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
12. Dadam, P., Reichert, M., Rinderle, S., et al.: ADEPT2 - Next Generation Process Management Technology. Heidelberger Innovationsforum, Heidelberg (April 2007)
13. Van der Aalst, W.M.P., Van Hee, K.: Workflow Management: Models, Methods and Systems. The MIT Press, Cambridge (2002)
14. DiVinE project, `http://divine.fi.muni.cz/` (last accessed on August 2010)
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
16. Center for Logic and Information, St. Francis Xavier University, `http://logic.stfx.ca/` (last accessed on August 2010)
17. Reichert, M., Dadam, P., Rinderle-Ma, S., et al.: Enabling Poka-Yoke Workflows with the AristaFlow BPM Suite. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Business Process Management. LNCS, vol. 5701. Springer, Heidelberg (2009)
18. Barkaoui, K., Ben Ayed, R., Sbai, Z.: Workflow Soundness Verification based on Structure Theory of Petri Nets. International Journal of Computing and Information Sciences 5(1), 51–61 (2007)
19. Leyla, N., Mashiyat, A., Wang, H., MacCaull, W.: Workflow Verification with DiVinE. In: The 8th International Workshop on Parallel and Distributed Methods in verification, PDMC 2009 (2009) (work in progress report)
20. Rabbi, F., Wang, H., MacCaull, W.: YAWL2DVE: An automated translator for workflow verification. In: The 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), pp. 53–59. IEEE CS press, Los Alamitos (2010)
21. Miller, K., MacCaull, W.: Toward Web-based Careflow Management Systems. Journal of Emerging Technologies in Web Intelligence (JETWI) Special Issue E-health Interoperability 1(2009), 137–145 (2009)
22. Wang, H., MacCaull, W.: An Efficient Explicit-time Description Method for Timed Model Checking. In: The 8th International Workshop on Parallel and Distributed Methods in verifiCation 2009 (PDMC 2009). EPTCS, vol. 14, pp. 77–91 (2009)
23. Mashiyat, A., Rabbi, F., Wang, H., MacCaull, W.: An Automated Translator for Model Checking Time Constrained Workflow Systems. In: FMICS 2010. LNCS, vol. 6371, pp. 99–114. Springer, Heidelberg (2010)