NOVA Workflow Components

1 Compensable Transaction

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.

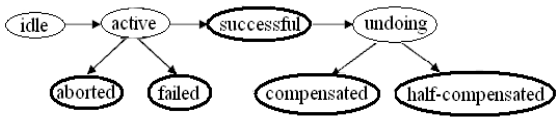

Fig. 1. State transition diagram of compensable transaction

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. There are five binary relations 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.

These relations can be mathematically expressed as the following formulae, where s is a sequence of actions and $s[i]$ denotes the ith element in the sequence:
($R1$) s satisfies a $<$ b *iff* $\exists i, j.\ (i<j \wedge s[i] = a \wedge s[j] = b\ ) \vee \forall i.(s[i] \neq a \wedge s[i] \neq b)$

($R2$) s satisfies a $\prec$ b *iff* $\forall i.\ (s[i] = a \Rightarrow \exists j.\ (j > i \wedge s[j] = b))$
($R3$) s satisfies a $\ll$ b *iff* $\forall i.\ (s[i] = b \Rightarrow \exists j.\ (j<i \wedge s[j] = a))$
($R4$) s satisfies a $\leftrightarrow$ b *iff* $\exists i, j.\ (s[i] = a \wedge s[j] = b) \vee \forall i.(s[i] \neq a \wedge s[i] \neq b)$
($R5$) s satisfies a $\nleftrightarrow$ b *iff* $\exists i.\ (s[i] = a \Rightarrow \forall j.s[j] \neq b)$

The transactional composition language, t-calculus, 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 I 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 [1,5-7] and described in section 3.

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	$S \divideontimes T$

Table I: t-calculus syntax

2 Compensable Workflow nets

An atomic task is an indivisible unit of work. Atomic tasks can be either compensable or uncompensable.

Definition 1 *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;*

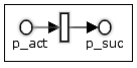

Fig. 2. Petri net representation of an uncompensable task

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 2 *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;*

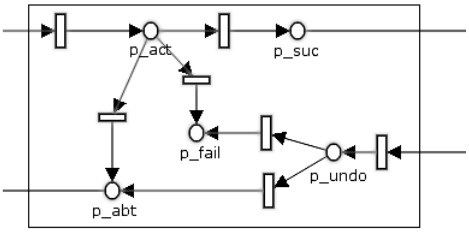

Fig. 3. Petri net representation of a compensable task

The 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. It is important to note that as the petri nets of atomic tasks (compensable and uncompensable) start and end with places, an arc (solid or doted) connecting two tasks actually stands for a transition and two arcs in the petri net format.

Definition 3 *A compensable task (ϕ_c) is recursively defined by the following well-formed formula:*

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

where t_c is an atomic compensable task, and $\odot \in \{;, ||, \sqcap, \otimes, \rightsquigarrow, \trianglerighteq, \triangleright, \divideontimes\}$ 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},*
- *ϕ_{c_1} || ϕ_{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 4 *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 2.1. 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 5 *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,*
- $\mathtt{o}$ *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),*

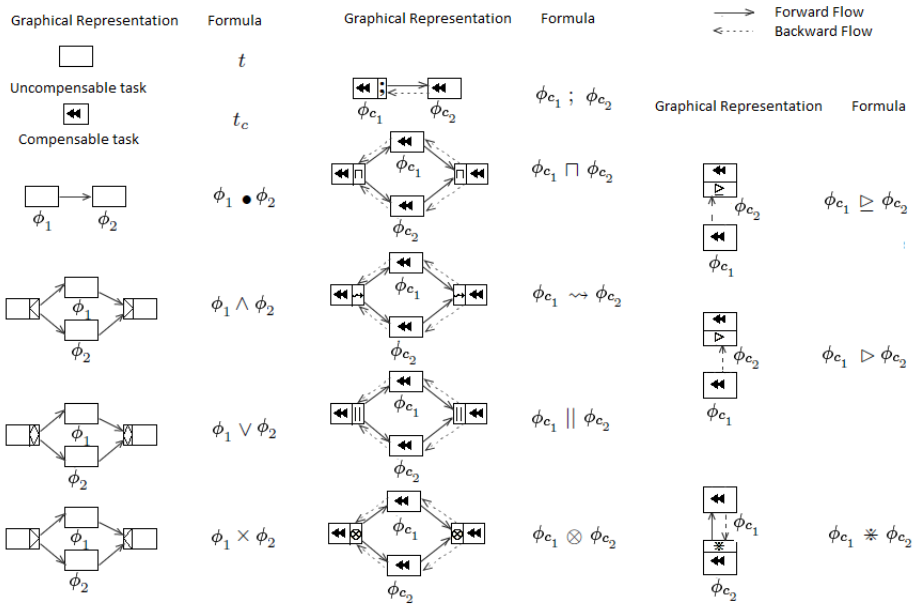

Fig. 4. Graphical Representation of Tasks

- *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* `i` *to* `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).

2.1 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.

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 ϕ_{c_1} ; ϕ_{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 behaviour dependencies. The following two basic dependencies describe behaviour of sequential composition: **i)** $(\phi_{c_1}, suc) \prec (\phi_{c_2}, act)$; **ii)** $(\phi_{c_2}, abt) \prec (\phi_{c_1}, und)$;.

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. 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*, which forcefully aborts a transaction and undos its partial effects. To sum up, when compensating, tasks which are composed in parallel are required to be compensate in parallel.

In Fig. 4, we can see the two tasks ϕ_{c_1} and ϕ_{c_2} which are composed in parallel. It represents the formula $\phi_{c_1} \parallel \phi_{c_2}$. Compensable tasks ϕ_{c_1} and ϕ_{c_2} will run in parallel but if any of the tasks aborts or fails, the other task will be aborted forcefully. The petrinet representation of the parallel composition is shown in Fig. 5.

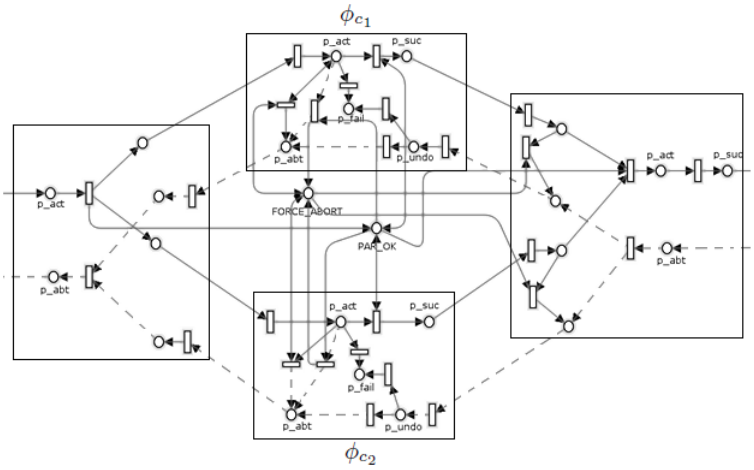

Fig. 5. Petrinet representation of parallel composition

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)** $(\phi_{c_1}, suc) \leftrightarrow (\phi_{c_2}, suc)$;

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

Internal Choice Tasks composed using internal choice will be selected and activated depending on some internal decisions. During execution only one branch will be activated and upon abort or failure the compensable flow will be executed. In Fig. 4, we can see the two tasks ϕ_{c_1} and ϕ_{c_2} which are composed with internal choice composition. It represents the formula $\phi_{c_1} \sqcap \phi_{c_2}$. Note that there can be more than two branches composed with the internal choice. The basic behavioural dependency indicates that only one of the tasks, ϕ_{c_1} or ϕ_{c_2}, will activate: $(\phi_{c_1}, act) \nleftrightarrow (\phi_{c_2}, act)$;. The petrinet representation of the internal choice split and join tasks are shown in Fig. 6.

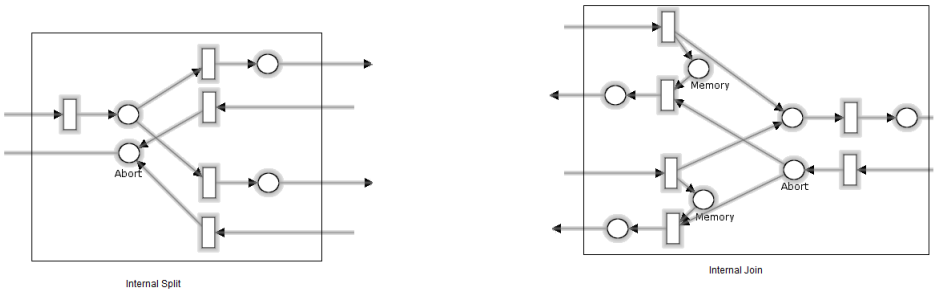

Fig. 6. Petrinet representation of internal choice composition

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 aborts.

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 basic dependency is described by: $(\phi_{c_1}, abt) < (\phi_{c_2}, act)$.

The petrinet representation of the alternative split and join tasks are shown in Fig. 7.

Speculative Choice Speculative choice composition is used to decide between two or more equivalent tasks which have the same or similar goals. Speculative choice will execute two independent tasks in parallel and will select the task which completes first. It is designed to reduce the time complexity of a system by executing two tasks simultaneously which could satisfy a requirement, but there is no preference between either tasks. The process of buying air tickets can be modeled with speculative choice

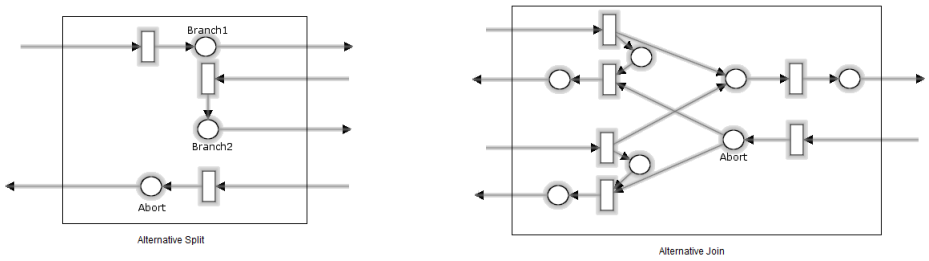

Fig. 7. Petrinet representation of alternative forwarding composition

tasks. The system orders tickets from two different airlines in parallel, then takes the one that is confirmed first and cancels the other booking.

In Fig. 4, we can see the two tasks ϕ_{c_1} and ϕ_{c_2} which are composed by speculative Choice. It represents the formula $\phi_{c_1} \otimes \phi_{c_2}$. Fig. 8 shows the petrinet representation of speculative choice composition.

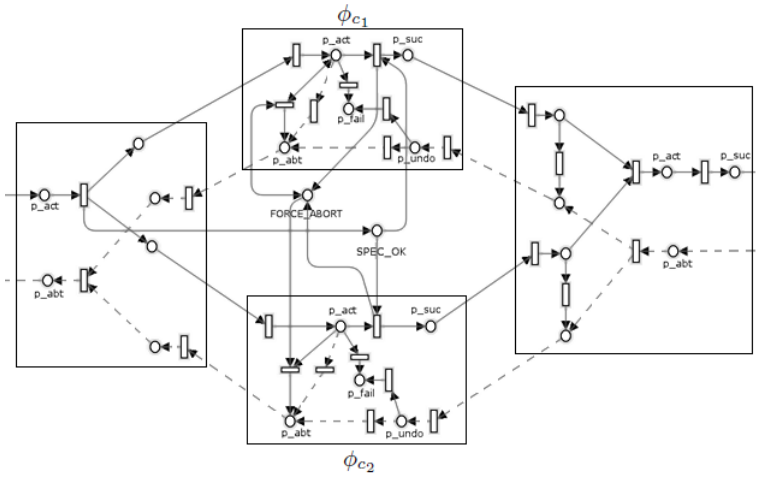

Fig. 8. Petrinet representation of speculative choice composition

Note that, if one task entered the *aborted* state before any task has completed then the other task will continue to operate. The basic behavioural dependencies are formalized as follows: **i)** $(\phi_{c_1}, act) \leftrightarrow (\phi_{c_2}, act)$; **ii)** $(\phi_{c_1}, suc) \nleftrightarrow (\phi_{c_2}, suc)$; **iii)** $(\phi_{c_1}, suc) \nleftrightarrow (\phi_{c_2}, fal)$; **iv)** $(\phi_{c_1}, fal) \nleftrightarrow (\phi_{c_2}, suc)$. These dependencies hold in the petri net representation and in the DVE code of the speculative choice composition.

It is important to note that the speculative choice is an unique operator w.r.t. the structural soundness of the whole petri net. Let $\phi_c = \phi_{c_1} \otimes \phi_{c_2} ... \otimes \phi_{c_n}$ ($n \geq 2$ is a finite integer), ϕ_c can be deemed as successful if ϕ_{c_i} $(1 \leq i \leq n)$

succeeds and all other tasks must be compensated. However, only when ϕ_c is aborted can the compensation flow proceed to the task immediate preceding ϕ_c. Therefore, the tokens in all of the compensated subtasks will remain in their `p_abt` places. As this situation will not affect the success of the overall workflow, we consider these tokens as *invisible* and will ignore them in the discussion of structural soundness (see section **??**).

Backward handling If an error occurs then the composed backward handling task will attempt to remove the partial effects. If this is completed successfully the flow terminates to the output condition.

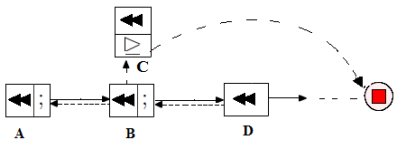

Fig. 9. Example of Backward handling

Fig. 9 shows a backward handling composition and its transitions, representing the formula $A\ ;\ (\ B \trianglerighteq C\ )\ ;\ D$. A and B are two compensable tasks composed with sequential composition. Task C is a backward handling task composed with task B. There are two kinds of errors that may happen during the execution of task B, aborted (compensation is possible) or failed (compensation is not possible). If task B enters into the `failed` state, the failure of B triggers the execution of C, which tries to undo all the partial effects. Once the partial effects are removed from the system, C terminates the flow.

If the task C results in an `aborted` or `failed` state after being activated, the partial effects of B will remain in the system. This results in an equivalent `failed` state for the composed transaction. Fig. 10 shows the petrinet representation of speculative choice composition.

Forward handling Forward handling is an error handling mechanism opposite to backward handling. Instead of compensation, forward handling tries to fulfil the business goal in the presence of failure.

In Fig. 11, task C is a forward handling task which is triggered when task B fails to remove the effects of its committed results. It represents the formula $A\ ;\ (\ B \triangleright C\ )\ ;\ D$. If C succeeds, it tries to remove the partial effects and directs the forward flow to activate D. If B's compensation flow is activated by D, then the compensation flow will be redirected to the forward handler C to remove its effects, as it was executed before. Its basic behavioural dependency is given as: $(B, fal) < (C, act)$

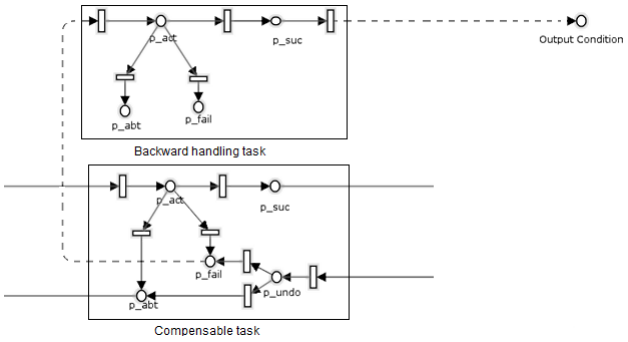

Fig. 10. Petrinet representation of backward handling task

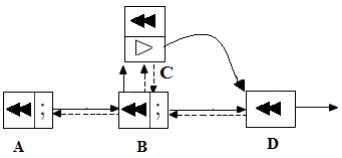

Fig. 11. Composition of Forward handling compensation

Programmable Compensation The construction of programmable compensation provides better readability and extra functionality. In this composition once a task has successfully completed, another task is stored as its compensation. Fig. 13 shows two tasks C and E, where the compensation of C is stored as task E. It represents the formula $A\ ;\ B\ ;\ (\ C \divideontimes E\ )\ ;\ D$. If task D is aborted, task E will be activated. After successful completion of task E, the compensation flow will continue from C.

In this model we allow a programmable compensation task to be a nested task, by this construction a compensable task can be compensated again. The behaviour of programmable compensation task can be expressed in the form of the following relation: $(C, suc) \ll (E, act)$.

Note that all these dependencies hold in the petri net representation and in the DVE code of the their corresponding operators.

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 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', which supports "correctness by construction". A pair of split and join routing tasks*

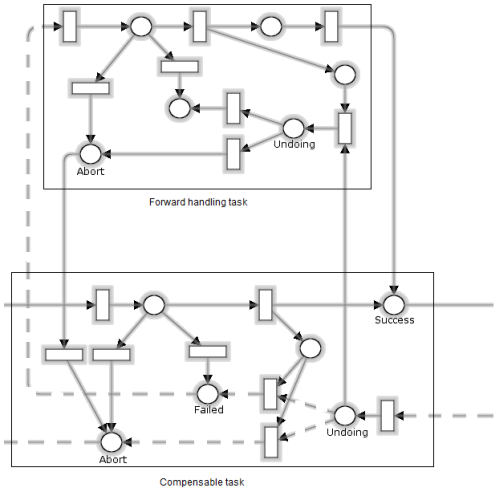

Fig. 12. Petrinet representation of forward handling task

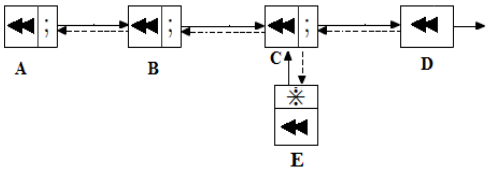

Fig. 13. Example of Programmable Compensation

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 II;*

- *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. An example of this shorthand modeling method is shown in Fig. 14;*
- *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".

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

Table II: Operators and their associated split and join tasks

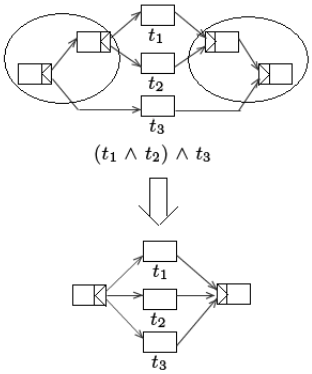

Fig. 14. Example of shorthand for modeling