Motivation and Guaranteed Completion in Workflow

Ashley McNeile[1] and Ella Roubtsova[2]

[1] Metamaxim Ltd.,
48 Brunswick Gardens, London W8 4AN, UK
ashley.mcneile@metamaxim.com
[2] Open University of The Netherlands,
Postbus 2960,6401DL Heerlen, The Netherlands
ella.roubtsova@ou.nl

Abstract. This paper presents an approach to designing interactive workflow to achieve guaranteed completion. The approach is based on the idea of modeling *motivation*, representing how the workflow solicits actions from its environment. The concept of motivation is used to differentiate between actions that are solicited by the workflow and actions that are not solicited, and happen spontaneously. We describe how to analyze progress in workflows that contain both solicited and spontaneous actions and establish guaranteed completion. We describe how to reason about time, and determine the maximum time a workflow requires to reach completion.

Key words: Workflow, Model Checking, Process Algebra, Modal Logic

1 Introduction

It is now universally the case that an office worker has a desktop computer providing access to both local productivity applications, such as word processing and spreadsheets, and access via a network to the organization's central business systems. It is now routine for workers to interact with core business applications, and consequently for these systems to play an integral part in the execution and management of business processes. As a result, support for business processes is now a key component of enterprise systems, and the analysis and design of business process forms a central part of the software development and implementation lifecycle.

1.1 Workflow and Motivation

This paper is concerned with a particular class of process support systems, generally termed "workflow", where the software plays an active role in soliciting actions from the environment. To introduce the idea, we use the example of a car fuel gauge, as shown in Figure 1. The gauge shows the amount of fuel left in the tank and has a *low-fuel light*, shown by the "E" on the left hand side, that comes on when the tank is almost empty: say just one litre of fuel left. The purpose of the light is to alert the driver of the need to refuel, and in this sense the light has a deontic[1] significance in the sense that when the light is on, refueling *needs to* take place.

When the low-fuel light is lit we talk of refueling being *solicited*. Of course, the driver is quite free to refuel the car when the light is not lit, and then we talk of refueling being *spontaneous*. Similarly a business workflow is able to determine what actions should happen next and then interact with the environment to advance the process. Typically the operator or agent[2] responsible for performing the action is a human in an office environment, and the means used to solicit action is an "electronic to-do list" displayed on the agent's screen. This kind of mechanism is discussed by many authors on workflow, for instance by van der Aalst [21]. Different means of alerting users to the need for action are used in other types of environment, such as a visual or auditory signals in a factory or control room, but the underlying concept is the same. We call this mechanism *motivation* in the workflow.

Not all actions in a workflow are solicited. For instance, in an order processing workflow operated by a supplier, a customer who has placed an order that is progressing through the workflow process may choose to amend or cancel it. Such an action would obviously not be solicited by the workflow and would therefore, in our terminology, be modeled as spontaneous. If processing an order were at an advanced stage, a late amendment from the customer might require reprocessing earlier steps and so delay completion. It is quite often the case that spontaneous actions are, in this sense, regressive.

Note that spontaneous actions in a workflow are not initiated by an agent responding to a to-do list but by some other means. In the order processing example above, the customer might be able to enter an amendment or cancelation of their order to the workflow remotely, via a web based interface. It is also possible that the workflow is updated to reflect spontaneous actions as part of a batch update process; or, in a distributed workflow, for a spontaneous action to arrive as a message from another participant.

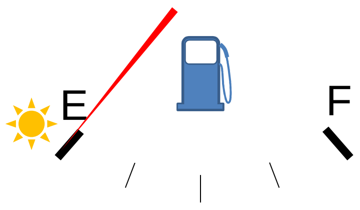


Fig. 1. Car Fuel Gauge

[1] Deontic means *concerned with obligation, permission, and related concepts.*
[2] We use *agent* and *operator* interchangeably.

1.2 Operator Commitments

In order for the low-fuel light to be effective, the operator (the driver of the car) has to understand its meaning. Specifically, we assume that the driver agrees to observe an *Operator Commitment* (hereafter abbreviated **OC**) in this case as follows: *As soon as the light comes on, the driver undertakes that within 20 km the car is refueled sufficiently for the light to go off again.* This OC gives a behavioral meaning to the light and, as along as it is obeyed, the car will never run out of fuel (provided that a litre of fuel is at least enough for 20 km of travel). As in the case of the low-fuel light, the operators from whom action is solicited by a workflow are bound by agreed OCs to react in a timely manner. In the business world such an agreement would normally be part of a formal (in the sense of agreed and documented) service agreement. The usual business terms, for instance used by the ITIL service management reference [18], are *Service level Agreement* (SLA) for an agreement that relates to end-to-end service time; and *Operation-Level Agreement* (OLA) for one that relates to an operational step or sub-processes forming part of a service. We have chosen to use the generic term *Operator Commitment* instead of SLA and OLA as these encompass aspects of service management and quality (such as reliability, availability and cost) that we do not consider.

We will assume that, when an action is solicited from an agent by a workflow, the agent is subject to an OC under which the agent undertakes to complete the action within a defined time. We will use this to reason about the time required for a workflow to complete.

1.3 Contribution

The use of workflow to help manage and drive business processes has been around for over twenty years. In the early days such systems were seen as electronic replacements for paper-based workflow systems but in recent times, driven by a combination of increasing service specialization and ubiquitous network connectivity both inside and across organizations, they are enabling new forms of business organization and process based around service orientation and business process outsourcing. This is now manifesting itself as business process integration across organization boundaries in such areas as supply chain, order processing, claims handling and financial trading. Ensuring service levels requires that the models that drive workflow are guaranteed to progress to completion successfully within defined target timescales. Such targets are increasingly being formalized in legal Service Level Agreements (SLAs) carrying financial penalties for non-compliance and it is becoming a commercial imperative to ensure that workflow automation always *works*, in the sense that a process once initiated will complete in the prescribed time under all possible event scenarios.

The approach described in this paper addresses the challenge of modeling and analyzing workflows where:

- The workflow propels itself forward by soliciting actions from its environment.
- Not all the actions that take place in the workflow are solicited, and some take place spontaneously at the whim of the environment, without solicitation.

The distinction between spontaneous and solicited actions is not supported by workflow modeling and model analysis techniques that have been previously described, but is required in order to reasoning correctly about progress and completion.

Specifically, the contributions made in this paper are:

- A new modeling technique that supports the distinction between **solicited** and **spontaneous** actions in workflow.
- An analysis method that establishes whether or not a workflow model employing both solicited and spontaneous actions is **bound to complete**; and, where completion is guaranteed, a method for determining **maximum completion time**.

This paper is organized as follows:

Section 2 explains how a workflow is modeled and illustrates this with a small example.
Sections 3 and **4** show how a workflow is analyzed to show that it progresses, first under certain simplifying assumptions and then in general.
Section 5 defines the conditions that a workflow is bound to complete and describes how maximum completion time can be determined.
Section 6 relates our work to other research into workflow and modal modeling.
Section 7 describes future directions of this research and **Section 8** provides conclusions.

2 Modeling Workflow

In this section, we describe how we model workflow. We start by describing the key concepts of *can-model* and *want-model*, which is the way we differentiate between solicited and spontaneous actions. We then describe how the can- and want-models of a workflow are represented.

2.1 Can Model and Want Model

We take the following to be axiomatic in our approach to modeling workflow:

1. A workflow progresses by soliciting actions from agents in the environment.
2. The agents responsible for supplying solicited actions enter into Operational Commitments (OCs) whereby they undertake to respond within a set time.
3. Actions that are not solicited (spontaneous actions) may also happen.
4. The determination of when an action needs to occur, and should therefore be solicited by the workflow, is driven by the state of the system.

To model the distinction between solicited and spontaneous actions we create two separate models, which we call the *can-model* and the *want-model*. At any given state of the workflow, the can-model states what actions **can happen**, whether solicited or not. Similarly, at any given state of the workflow, the want-model states what actions **need to happen** to achieve progress. The want-model defines which actions the workflow solicits from external agents by causing appropriate entries to appear on their to-do lists. We say that the want-model provides the *motivation* in the workflow.

We use a state-transition based modeling technique called *Protocol Modeling* to represent both the can- and the want-models of a workflow. The following sections provide an informal description of the technique; a full description is given by McNeile et al. in [2].

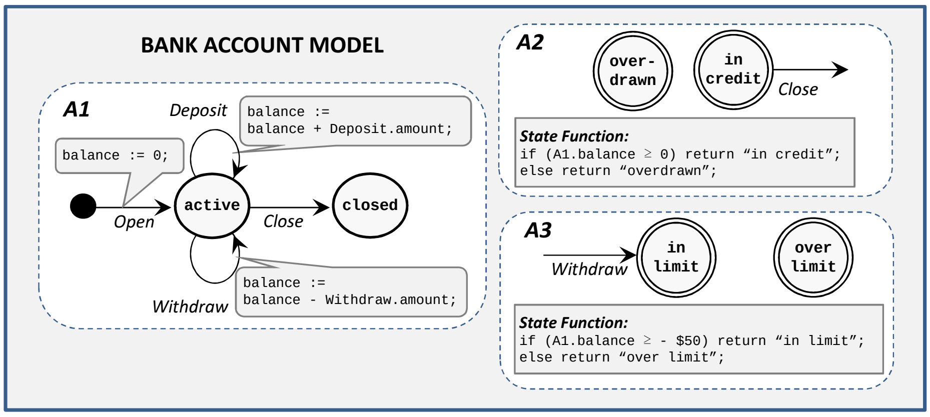

BANK ACCOUNT MODEL
A1
Deposit
balance := balance + Deposit.amount;
balance := 0;
Open
active
Close
closed
Withdraw
balance := balance - Withdraw.amount;
A2
over-drawn
in credit
Close
State Function:
if (A1.balance ≥ 0) return "in credit";
else return "overdrawn";
A3
Withdraw
in limit
over limit
State Function:
if (A1.balance ≥ - $50) return "in limit";
else return "over limit";

Fig. 2. Protocol Model of a Bank Account

2.2 Basics of Protocol Modeling

As an introduction to Protocol Modeling (hereafter abbreviated **PM**) we describe a simple bank account model. In Figure 2 a simple bank account is described using three state machines, called *protocol machines*, labeled $A1$, $A2$ and $A3$. These three machines are in parallel composition with composition semantics following that of the parallel operator $\|$ of Hoare's CSP [5]. So the behavior of the account is described as $A1 \| A2 \| A3$. We now explain each of the three machines in turn, and then explain how they combine to give the account behavior.

The first machine, $A1$, provides the basic account behavior, showing that the *Open* must happen first, followed by any number of *Deposit* and *Withdraw* actions, followed by the *Close*. In some states more than one action is possible:

so in the state `active` $A1$ can respond to *Deposit*, *Withdraw* or *Close* but will refuse *Open*. This machine maintains the balance of the account using the updates indicated in the bubbles attached to the transitions. The `balance` is a local variable owned by $A1$ and only $A1$ may alter its value, although other composed machines may access it. This machine takes the form of a traditional state machine in which each transition takes the machine to a new state (or, in the case of reflexive transitions like those for the *Deposit* and *Withdraw* events, back to its old state). The current state should be thought of as another local variable of the machine with an enumerated type, allowing values {`dot`, `active`, `closed`}. The set of actions that a machine describes, in this case: {*Open*, *Close*, *Deposit*, *Withdraw*}, is called the *alphabet* of the machine.

Any machine in PM may have local storage variables, although in this example only $A1$ does so. Only the machine owning a variable can change its value, and only when moving to its next state in response to an event.

The machine $A2$, whose alphabet is just {*Close*}, expresses the fact that the account may only be closed if it is in credit. Instead of having its state stored as part of the local storage, this machine has its state derived by a *state function*, as shown in Figure 2, that returns an enumerated type allowing values {`in credit`, `overdrawn`}. The arrow for *Close* represents the fact that *Close* is only possible for this machine when the state is `in credit`. Because their states are derived rather than driven by transitions, protocol machines that use derived states can be topologically disconnected; so as $A2$ does not use transitions as a basis for state determination, the arrow for *Close* does not need to lead to a state. For graphical emphasis, the state icons in derived-state machines ($A2$ and $A3$) are shown with a double outline.

The final machine $A3$, whose alphabet is just {*Withdraw*}, expresses the fact that the account has a limit of -\$50 below which the balance is not allowed to go. The machine does this by constraining a *Withdraw* action on the basis that the state resulting from the withdrawal must be `in limit`. In PM this is called a *post-state constraint*, as it constrains the possibility of an event on the basis of the state that would result if it were to take place.[3] Like $A2$, $A3$ is topologically disconnected because its state is derived rather than driven by transitions.

As the three machines are in parallel composition with composition semantics following that of the parallel operator || of CSP, whereby an event can take place unless refused (not allowed) by any machine that has the event in its alphabet. Thus the *Close* can only take place if $A1$ is in state `active` and $A2$ is in state `in credit`, with $A3$ uninvolved as *Close* is not in its alphabet. A post-state constraint violation is taken as a refusal, on equal footing with a pre-state constraint (the lack of a transition starting at the current state), so a *Withdraw*

[3] The discussion of workflow in the remainder of this paper does not use post-state constraints. However, they can be used without change to any of the ideas or analysis techniques.

may only take place if $A1$ is in the state `active` and the result of the withdrawal is that $A3$ is in the state `in limit`. Throughout this paper we use the graphical convention that multiple machines in the same figure, such as those in Figure 2, are in parallel composition.

Finally note that protocol models describe the possible orderings of actions (the possible traces) of a system but have no role in selecting between traces in the context of a particular execution. If the PM of Account is in a state where any of *Deposit*, *Withdraw* or *Close* is possible next, it is the **environment** that decides which actually happens and not the **model**.

As this example illustrates, protocol machines come in two types: *stored-state* ($A1$) and *derived-state* ($A2$ and $A3$), reflecting whether the state attribute of a machine (the attribute which indicates its current state) is a *stored* or a *derived* attribute. Conceptually, a derived state attribute is identical to the UML concept of *Derived Element*, see [15]; although UML does not envisage this concept being utilized to model state.

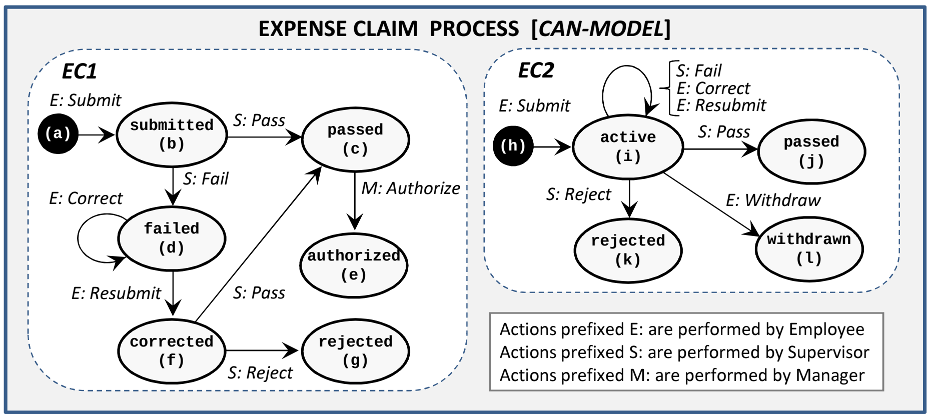


Fig. 3. Expense Claim Process: Can-Model

2.3 Example Workflow

Now we illustrate how PM is used to model the can-model and the want-model of a workflow. Figure 3 shows a simple expense claim approval process. The process involves actions by an **Employee**, who may *Submit* and *Withdraw* a claim; a **Supervisor**, who reviews a claim and can *Pass* or *Fail* the claim; and a **Manager** who may *Authorize* payment against claim once it has been passed by the Supervisor. The Manager has discretion to authorize less than the full claim amount. If the Supervisor does not pass the claim on review, the Employee may *Correct* and *Resubmit* it after correction, but may only resubmit once.

Figure 3 is the *can-model* of the workflow, and is expressed as the composition of two stored-state machines: $EC1 \parallel EC2$. The first machine, $EC1$, shows the main path of the process. The second, $EC2$, represents the fact that the Employee may *Withdraw* the claim at any point up to when it is passed or rejected.

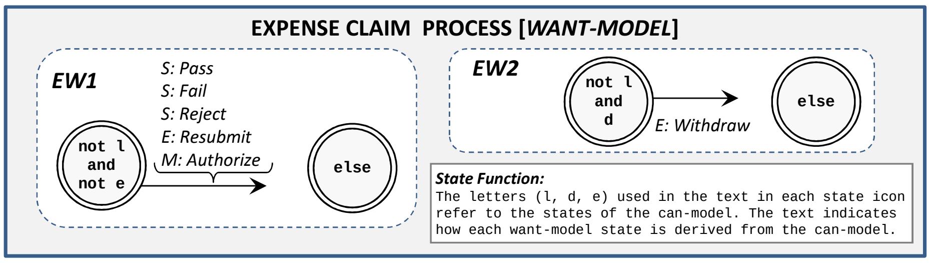

EXPENSE CLAIM PROCESS [*WANT-MODEL*]
EW1
not l and not e
S: Pass
S: Fail
S: Reject
E: Resubmit
M: Authorize
else
EW2
not l and d
E: Withdraw
else
State Function:
The letters (l, d, e) used in the text in each state icon refer to the states of the can-model. The text indicates how each want-model state is derived from the can-model.

Fig. 4. Expense Claim Process: Want-Model

However, this can-model by itself is like the fuel gauge without the low-fuel light: there is no model of motivation. Figure 4 shows the *want-model* for the expense claim workflow and models the motivation, by showing when actions are solicited. The want-model consists of two composed derived-state machines: $EW1 \parallel EW2$ (although, as their alphabets are disjoint, these machines are effectively independent). Both machines use a state derived from the state of the can-model so that, for instance, when the $EC1$ is in state `submitted` (which has label `(b)`) and $EC2$ is **not** in state `withdrawn` (label `(l)`) then the want-model indicates that *Pass* and *Fail* are both "wanted actions". However, although both of these actions are wanted, the can-model ensures that *only one of them* can actually take place.

2.4 Discussion

The can-model and the want-model work together to define the behavior of the workflow. The following scenario illustrates this:

1. Employee Fred *Submits* claim *1234*.
2. As this claim is now in the state (`submitted` and `active`) in the can-model, the actions *Pass 1234* and *Fail 1234* are possible. Moreover, by virtue of $EW1$, these actions are both also wanted and, as these are Supervisor actions, they are solicited from Supervisor Sue by appearing in her electronic to-do list.
3. Supervisor Sue chooses to *Fail 1234* and submits this decision to the system. As the claim is now in state (`failed` and `active`) in the can-model, the actions *Pass 1234* and *Fail 1234* are no longer possible and so these are removed from Sue's to-do list. In this new state the actions *Resubmit 1234* and *Withdraw 1234* are now both possible. The first of these is wanted by

virtue of $EW1$ and the second is wanted by virtue of $EW2$ and, as they are both Employee actions, both now appear in Fred's to-do list.
4. Fred decides to *Withdraw* his claim. As the can-model is now in state `withdrawn`, no further action is either possible or wanted, so the workflow has finished.

As this scenario shows, the synchronization between the workflow and the to-do lists are based on the states of both the can-model and the want-model, so an action appears on a to-do list only when it is both possible (in the can-model) and wanted (in the want-model). For instance, after step 1 in the above scenario when Fred has submitted a claim and Sue's to-do list asks her to either *Pass* or *Fail* it, Fred could *Withdraw* it. If this happened the claim would be in the state `withdrawn`, so Sue's actions would no longer be wanted by $EW1$ and would be removed from her to-do list, even though she has not acted on them.

Note that it is possible for an action to be wanted in one state and not in another, even though it is possible in both. The *Withdraw* action illustrates this: after *Submit* it is possible but not wanted, but after *Fail* it is both possible and wanted. So after a *Fail*, *Withdraw* will appear in the employee's to-do list. This is like refueling the car, which is possible whether or not the low fuel light is lit.

The job of the expense claim system is to drive a process whereby actions/decisions are solicited from the agents competent to make them. Thus Sue is asked to *Pass* or *Fail* a claim. The workflow does not replace or automate the function of these agents.

2.5 Form of the Want-Model

The machines in the want-model ($EW1$ and $EW2$) use derived states and own no attributes. This means that their transitions perform no update to the state of the workflow, and thus are side-effect free. We always require that the want-model is side-effect free for the following reason. Suppose that machine P can engage in an action x, and consider two occasions on which x is possible and takes place:

– On the first occasion, x is wanted by P, as modeled by P's want-model. The occurrence of x is therefore accompanied by a transition firing in both P's want-model and P's can-model.
– On the second occasion, x is not wanted by P. The action takes place and is accompanied by a transition firing in P's can-model, but no transition fires in P's want-model.

If the transition firing in P's want-model had side-effects, causing updates to P's local storage, then these updates would happen on the first occasion but not on the second. However, the updates that result in P from engagement in x should reflect **only the semantics of the x action** and should not be affected by whether the action is wanted or not. See [3] for further discussion of this.

3 Progress Analysis

Our aim is to construct workflows that are bound to complete. The strategy for doing this is in two parts:

- Defining conditions for *progress*, which ensure that a workflow will not rest indefinitely in one state.
- Defining further conditions for *guaranteed completion*, which ensure that the workflow does not cycle and so progress will always arrive at a Completion State.

This section and the next (Section 4) are concerned with the first part, namely that workflow will always progress. The second part, namely that workflow will always complete, is addressed in Section 5.

Progress analysis is based on examination of the topology of the can- and want-models. The key idea is that progress relies on the occurrence of actions that are solicited by the want-model, but must not be jeopardized by occurrence of spontaneous actions.

3.1 Simplifying Assumptions

In the initial treatment of progress we will make the following assumptions:

1. The can-model consists of a composition of **stored-state** machines.
2. The states of the want-model are expressed as **Boolean combinations of the states of the can-model**.

We relax these assumptions in Section 4.

3.2 Topological Composition

In order to analyze the topology of a can-model represented as a composition we need to be able to combine the multiple models into a *single machine*. We describe a technique for doing this below. This technique only applies to stored-state machines; we describe how to handle derived-state machines later, in Section 4.2.

As a basis for combining machines, we define a *representation* of a stored-state machine as a tuple $P = \langle \Lambda_P, \Sigma_P, \Gamma_P, \Delta_P \rangle$ where:

Λ_P is the alphabet of P, the set of events which P will either accept or refuse. Elements range over x, y, ...

Σ_P is a finite set of states of P. Elements range over σ_i, σ_j, ...

$\Gamma_P \subseteq (\Lambda_P \times \Sigma_P)$ is a binary relation. $\langle x, \sigma_i \rangle \in \Gamma_P$ means that x can be accepted by P when P is in state σ_i. $\langle x, \sigma_j \rangle \notin \Gamma_P$ means that x is refused by P when P is in state σ_j.

Δ_P is a total mapping $\Gamma_P \to \Sigma_P$ that defines for each member of Γ_P the next state that P adopts as a result of accepting an event. So $\Delta_P(\langle x, \sigma_k \rangle) = \sigma_l$ means that if P accepts x when in state σ_k it will then adopt state σ_l.

Note that this kind of representation is defined for *stored-state* machines, where the next state that a machine adopts as the result of an action is driven by topological rules (embodied in the function Δ). It is not defined for *derived-state* machines as their behavior cannot be depicted in pure topological form.

Suppose we have two stored-state machines, P and Q. Using the semantics of CSP $\|$ composition we can create a representation of $P \| Q$ as follows:

$$\Lambda_{P\|Q} = \Lambda_P \cup \Lambda_Q$$
$$\Sigma_{P\|Q} = \Sigma_P \times \Sigma_Q \quad \text{(the Cartesian product of } \Sigma_P \text{ and } \Sigma_Q\text{)}$$

Given $a \in \Lambda_{P\|Q}$ and a state $\langle \sigma_p, \sigma_q \rangle \in \Sigma_{P\|Q}$, we determine whether $\langle x, \langle \sigma_p, \sigma_q \rangle \rangle \in \Gamma_{P\|Q}$ as follows:

$$\begin{array}{ll} (x \in \Lambda_P \wedge \langle x, \sigma_p \rangle \notin \Gamma_P) \Rightarrow & \langle x, \langle \sigma_p, \sigma_q \rangle \rangle \notin \Gamma_{P\|Q} \\ (x \in \Lambda_Q \wedge \langle x, \sigma_q \rangle \notin \Gamma_Q) \Rightarrow & \langle x, \langle \sigma_p, \sigma_q \rangle \rangle \notin \Gamma_{P\|Q} \\ \text{Otherwise:} & \langle x, \langle \sigma_p, \sigma_q \rangle \rangle \in \Gamma_{P\|Q} \end{array}$$

And we construct $\Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle \rangle)$ as follows:

$$\begin{array}{ll} (x \in \Lambda_P) \wedge (x \notin \Lambda_Q) \Rightarrow & \Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle \rangle) = \langle \Delta_P(\langle x, \sigma_p \rangle), \sigma_q \rangle \\ (x \notin \Lambda_P) \wedge (x \in \Lambda_Q) \Rightarrow & \Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle \rangle) = \langle \sigma_p, \Delta_Q(\langle x, \sigma_q \rangle) \rangle \\ \text{Otherwise:} & \Delta_{P\|Q}(\langle x, \langle \sigma_p, \sigma_q \rangle \rangle) = \langle \Delta_P(\langle x, \sigma_p \rangle), \Delta_Q(\langle x, \sigma_q \rangle) \rangle \end{array}$$

Because of the simplifying assumption from Section 3.1, that the can-model comprises only *stored-state* machines, we can use this procedure repeatedly to render any can-model as a single topological representation.

3.3 Analysis Model

Suppose that, using the procedure above, we have a can-model expressed as a single stored-state machine. We can distinguish between the transitions of this machine as follows:

- Where a transition is possible (allowed by the can-model) **but not** wanted (by the want-model) it is termed a *c-transition.*
- Where a transition is possible (allowed by the can-model) **and** wanted (by the want model) it is termed a *w-transition.*

Because we assume (assumption 2 in Section 3.1) that the states of want-model are Boolean expressions of the states of the can-model, the state function of any want-model is single valued for a given state of the can-model (once expressed

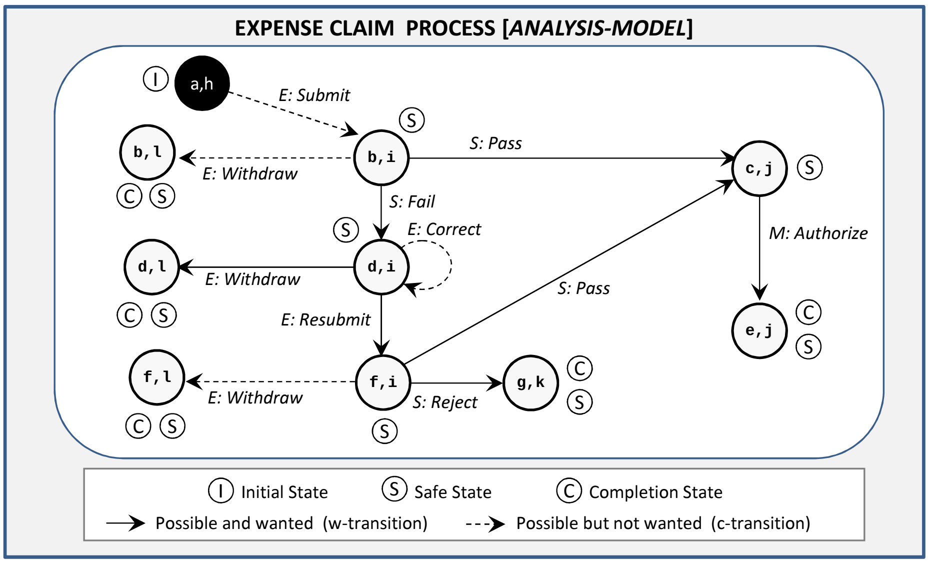


Fig. 5. Expense Claim Process: Analysis-Model

as a single machine as described above). This means that it is possible to make an unambiguous designation of every transition in the analysis-model as either a c-transition or a w-transition.

The resultant model is called the *analysis-model*. Figure 5 shows the analysis-model of the expense claim workflow, and shows the w-transitions as solid arrows and the c-transitions as dashed arrows. The analysis of progress and completion of a workflow is performed using this model. It is important to understand that the role of the analysis-model is **only** to support reasoning about progress and it does **not** usurp the role of being the design of the workflow; this role remains with the individual can- and want-models (such as those shown in Figures 3 and 4).

3.4 Types of States

As a basis for progress analysis we classify the states in the analysis-model. Suppose that the set of states of the analysis-model is denoted by Σ. We distinguish different kinds of state in Σ as follows.

Initial State. A state $\sigma \in \Sigma$ is an *Initial State* if and only if it is not the end state of any transition (either a c-transition or a w-transition).

Completion State. A state $\sigma \in \Sigma$ is a *Completion State* if and only if no further action is wanted, whether it is possible (allowed by the can-model) or not.

Safe State. A state $\sigma \in \Sigma$ is a *Safe State* if and only if it is either a Completion State, or it is the start state of at least one w-transition and every w-transition from σ leads to a Safe State.

Note that this is not a partitioning of the states of Σ: there may be states that do not qualify as any of these, and states that qualify as more than one type (see Figure 5). Note also that there is no requirement that a Completion State has no outgoing transitions, only that *no transition is wanted*, whether any transition is possible or not.

3.5 Well-Formedness

The analysis-model of a properly engineered workflow must have the following properties:

1. It has at least one Initial State and at least one Completion State.
2. Every Initial State has at least one outgoing c-transition and no outgoing w-transition.
3. Two successive w-transitions (one entering a state and the other leaving it) cannot be for the same action.
4. Every Completion State should be a *business completion* of the workflow, in the sense that no party involved expects anything further to happen to complete the business transaction managed by the workflow.

Properties 2 and 3 merit explanation. Property 2, requiring that an Initial State has no outgoing w-transition, ensures that the workflow cannot solicit instances of itself, otherwise the system would flood with instances. Property 3, requiring that we never have two successive w-transitions for the same action, means that once an agent performs a solicited action it disappears from her to-do list, otherwise it is not possible to give a clear definition of an Operator Commitment (OC).

We will assume that every workflow has these four properties.

3.6 Operational Commitments

As stated in Section 2.1, we take it that solicited actions are subject to an OC and now define exactly what this means in the model. We take it that:

– For every action, x, that can be solicited, the associated OC defines a time, $OC(x)$, within which the agent undertakes to perform the action.
– An OC **only** applies to actions solicited by the workflow. Our assumption is that an action that is possible but not wanted will **not** appear on any agent's to-do list (although it must still be possible to do this action).
– The clock against which performance of the OC for x is measured starts ticking at the point where workflow first enters a state with an outgoing w-transition for x, at which time x appears on the agent's to-do list. The clock remains ticking until the workflow enters a state with no outgoing w-transition for x, at which time x disappears from the agent's to-do list.

- The OC means that the agent undertakes not to let x remain on his/her to-do list once the clock reaches $OC(x)$.

The normal reason that x disappears from the agent's to-do list is that the agent performs x (supplies an x message to the workflow). However there is no implication that this is the only reason. Before the clock reaches $OC(x)$ a different action, either solicited or spontaneous, may move the workflow to a state that has no outgoing w-transition for x and so it is removed from the agent's to-do list. In our expense claim workflow this happens if the employee withdraws the claim when it is in state `f,i`, in which case the entries on the supervisor's to-do list to either pass or reject the claim would disappear without the supervisor doing either. Note also that the well-formedness property 3 in Section 3.5 means that performing a possible and wanted action always causes it to cease to be possible and wanted, otherwise an agent could perform an action but find that it obstinately persists on the to-do list.

Suppose that σ is a Safe State of a workflow but not a Completion State, and that $WT(\sigma)$ is the set of actions with w-transitions leaving σ. By the definition of Safe State we are guaranteed that $WT(\sigma)$ is not empty. The time that the workflow can rest in this state is bounded above by:

$$\min_{x \in WT(\sigma)} (OC(x))$$

The workflow cannot remain in σ longer than this without an OC being violated.

3.7 Sufficient Conditions for Progress

Based on the above, we can state a sufficient condition for a workflow that starts at an Initial State and is "kicked-off" by a c-transition to maintain progress: *Every c-transition in the analysis-model of the workflow ends at a Safe State.* A short argument shows that this guarantees that the workflow will not rest indefinitely in a given state.

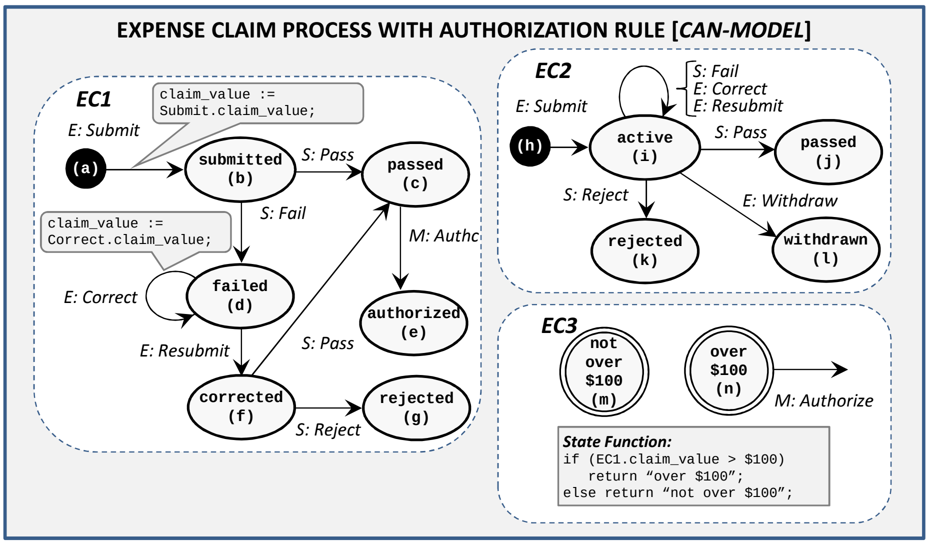
Suppose that the workflow is in a Safe State that is not a Completion State. If an action takes place, it must either be a c-transition or a w-transition. By virtue of the condition given above, a c-transition will take the workflow to a Safe State. By virtue of the definition of Safe State, a w-transition will also take the workflow to a Safe State. Moreover, by well-formedness condition 2 in Section 3.5, the first ("kick-off") transition must be a c-transition and so take the workflow to a Safe State. This means that once the workflow has started it can **never leave** the subgraph of Safe States. Because, as shown above, the time that a workflow can remain in a Safe State that is not a Completion State is bounded, a workflow that is not in Completion State must progress.

Progress is not enough to guarantee that the workflow will **complete**. If it contains a cycle of Safe States none of which is a Completion State, a workflow could cycle indefinitely. Section 5 discusses the further conditions required to ensure completion.

3.8 Discussion

By virtue of the want-model we are able to model the motivation in a workflow. Modeling motivation allows us to define “Safe States”, states where the progress of workflow is guaranteed. To ensure progression, a workflow must be designed so that it moves through Safe States from start to finish. The want-model combined with the OCs can be thought of as providing “liveness” to the workflow, meaning that it can never get stuck in a non-completion state.

Figure 5 shows progress analysis for the expense claim workflow. The two machines, $EC1$ and $EC2$, of the can-model in Figure 3 have been composed to form a single topological representation using the procedure given in Section 3.2. Then the want-model, $EW1$ and $EW2$, in Figure 4 has been used to classify each transition as either a c-transition or a w-transition. Finally, the states have been classified according to the rules in Section 3.4. As each c-transition in the resultant graph ends at a Safe State, the progress condition is established.



```
claim_value :=
Submit.claim_value;
```

```
claim_value :=
Correct.claim_value;
```

State Function:

```
if (EC1.claim_value > $100)
   return “over $100”;
else return “not over $100”;
```

Fig. 6. Expense Claim Process with Authorization Rule: Can-Model

4 Relaxation of Assumptions

In this section we relax the two simplifying assumptions made in Section 3.1.

4.1 Workflow with Authorization Rule

The first simplifying assumption is that the can-model consists of a composition of **stored-state** machines, and we now consider a workflow rule based on a data value that requires a derived-state machine in the can-model. Consider the following change to the expense claim workflow: *The Manager is only required to authorize claims that are over $100.*

To design the change to the workflow for this new rule we need to answer the question: should it be *possible* for the Manager to authorize claims which are not over $100, even though such authorization is not wanted? Initially we will assume that the answer to this is *No* (with the answer *Yes* considered later in Section 4.3).

In the light of this decision, the modified can-model is shown in Figure 6. A new machine *EC*3 has been added that only allows the *Authorize* action to take place if the value of the claim is over $100, and the other processes are unchanged. The new machine has a derived state, based on the attribute `claim_value` owned by *EC*1. The maintenance of this attribute by *EC*1 has been shown explicitly in Figure 6: it is set when the claim is submitted, and updated if the claim is corrected.

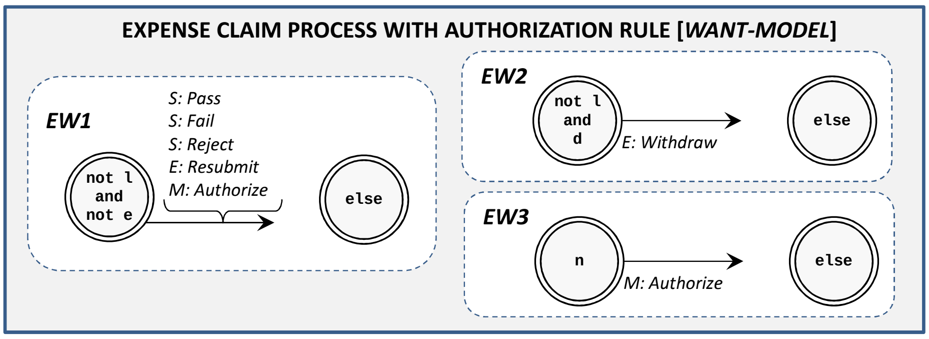


Fig. 7. Expense Claim Process with Authorization Rule: Want-Model

The modified want-model is shown in Figure 7. The change required here is the addition of *EW*3 that states that *Authorize* is only wanted if *EC*3 is in state `over $100` (label `(n)`). Note that *Authorize* is now in the alphabets of both *EW*1 and *EW*3 and, by the semantics of CSP composition, this means that it is wanted if and only if both of these machines say so.

4.2 Analysis of Revised Workflow

We can now proceed to analyze the revised version for progress. In order to perform the analysis, we need to represent the can-model as a single stored machine. The new machine, *EC*3, of the can-model is a *derived-state machine*, but the topological composition procedure in Section 3.2 is only defined for *stored-state machines*. However it is possible to create a stored-state approximation of a

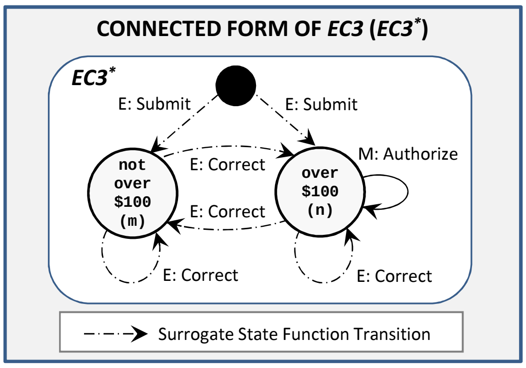


Fig. 8. Connected Form of $EC3$

derived-state machine called the *connected form* of the machine for the purpose of analysis. This is done in three steps:

1. Form the state space as that defined by the machine's state function.
2. Add transitions between the states to create a topological surrogate for the state function.
3. Create transitions representing the constraints of the original machine, but in topologically connected form.

Figure 8 shows the connected form representation, $EC3^*$, for the machine $EC3$ from Figure 6. Step 1 identifies the two states: {`over $100`, `not over $100`} and to these a starting state (black dot) is added. In step 2, transitions (shown as dashed for graphical emphasis) are added as a surrogate for the state function. In step 3, the transition for *Authorize* is shown as both starting and ending at the state `over $100` as the action can only happen from this state and leaves the state unchanged.

Step 2 merits some elaboration. The transitions that need to be added in this step (those shown as dashed arrows in Figure 8) are identified as follows:

– The state function is examined to determine the set of attributes referenced in the calculation. In this case, it is just `EC1.claim_value`.
– The machine(s) that own these attributes are examined to determine which transitions cause their values to change. In this case it is *Submit* and *Correct* in $EC1$ as can be seen from the update bubbles attached to these transitions in Figure 6.
– Corresponding transitions are added to each state of the connected form machine being constructed, according to how they can alter the state. In this case *Submit* can create the claim in either state; and *Correct* may or may not change the state from `over $100` to `not over $100` or vice-versa.

Whereas the state function tells us **exactly** the effect of actions on the state, the surrogate transitions show, and can only show, the **possibilities**. Generally, the transitions added in step 2 create a machine that is non-deterministic. While the original machine $EC3$ in Figure 6 is deterministic the connected form in Figure 8 is not. However, the connected form machine is purely for analysis and does not replace the derived-state version in the definition of the workflow. Further examples and discussion about this procedure can be found in [1].

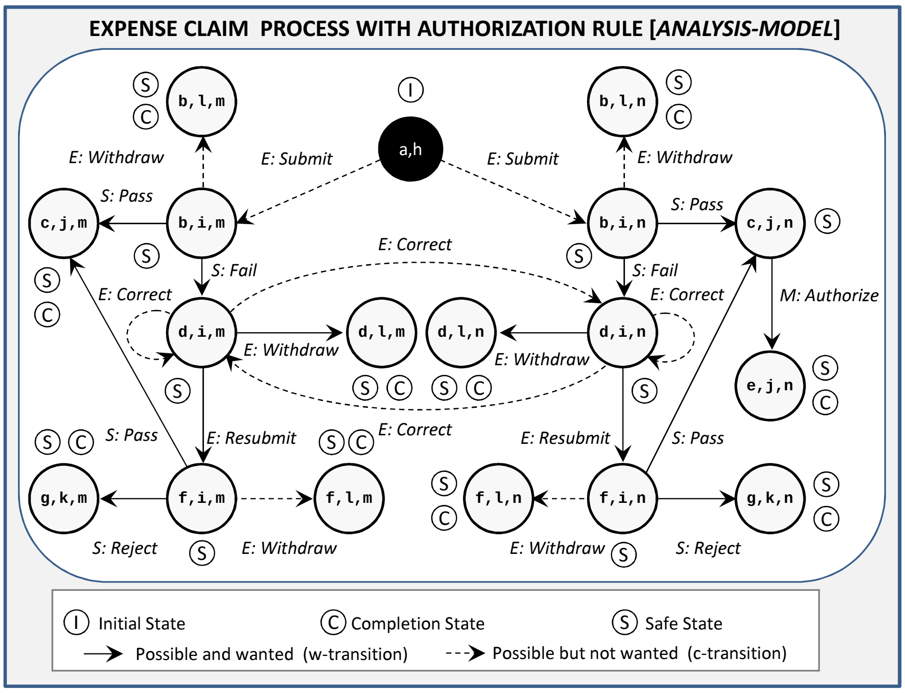


Fig. 9. Expense Claim Process with Authorization Rule: Analysis-Model

The three machines $EC1$, $EC2$ and $EC3^*$ can now be combined using the procedure defined in Section 3.2 and progress analysis carried out as before. The result is shown in Figure 9 which essentially represents two copies of Figure 5, one for each value of the two states of $EC3$. It is clear from this diagram that the modified workflow meets the conditions for progress.

4.3 Topology and Data

The second simplifying assumption in Section 3.1 is that the states of the want-model are expressed as **Boolean combinations of the states of the can-model**. We illustrate relaxation of this assumption by supposing that the Manager can authorize *any* claim, but is only *solicited to do so by the workflow* when the claim is over \$100.

The change needed is to remove $EC3$ from the can-model, so that possibility of the *Authorize* action is not contingent on the value of the claim. However, in the want-model we need to retain the rule that *Authorize* is only wanted if the claim is over $100, and this means that we need to retain the calculated state `(c and n)` shown in Figure 7. As `(n)` is a state of $EC3$, rather than remove $EC3$ we just remove the outgoing transition for *Authorize* (as we have shown in $EC3'$ in Figure 10), so that the machine models states used in the want-model (`over $100` and `not over $100`) but does not constrain any actions. The result of this is that the new analysis-model looks exactly as shown in Figure 9, except that there is a c-transition for *Authorize* from the state labeled `(c,j,m)` (on the left hand side of the diagram) to a new state labeled `(e,j,m)`, representing the fact that this is still possible even if the claim is for less that $100. The state labeled `(c,j,m)` is still a Completion State of the workflow as there are no outstanding wanted actions, and so the progress analysis of the workflow is unchanged as we would expect.

This example illustrates that we can use derived-state machines to represent distinctions in data, in this case whether the value of the claim is over $100 or not, whether these affect can-model behavior or not, and this provides a general approach to converting data into topology for analysis purposes. If a given topological state, σ, in the can-model represents multiple execution states of the workflow that differ in the value of some attribute a, then the introduction of a new machine with derived states based on different values of a can be used to split σ into multiple states in the topological composition procedure. Thus it is always possible to arrange that the states of the can-model are sufficiently granular to be used as the basis for the states of the want-model, as required by the second assumption in Section 3.1.

5 Guaranteed Completion

So far we have focused on the conditions for *progress*: ensuring that a workflow will not rest indefinitely in one state. As pointed out in Section 3, the presence of cycles in the model may mean that a workflow may cycle through Safe States without reaching a Completion State. In this section we discuss further conditions on the workflow that ensure it does not cycle in a way that prevents completion. We will see that cycling can be prevented by either *data* or *time*, and address these in turn.

If a well-formed workflow obeys the progress rules set out in Section 3 and has no cycle that is not limited either by data or time, it is described as being GC (for Guaranteed Completion). A workflow that is GC is bound to complete in finite time.

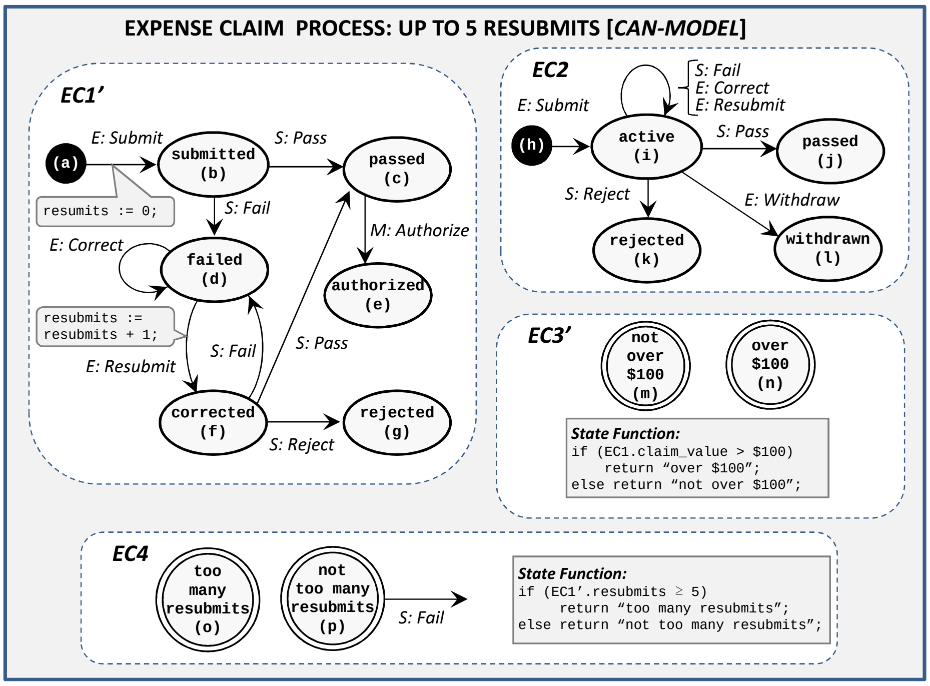


Fig. 10. Expense Claim with up to 5 Resubmits: Can-Model

5.1 Cycles

In all cases, the consideration of cycles is in the **analysis-model** of the workflow (see Section 3.3).

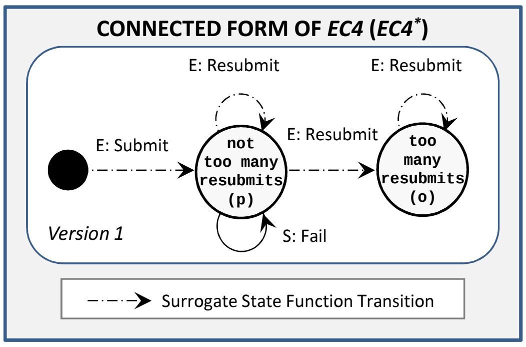

CONNECTED FORM OF EC4 (EC4*)
E: Resubmit
E: Resubmit
E: Submit
E: Resubmit
not too many resubmits (p)
too many resubmits (o)
Version 1
S: Fail
Surrogate State Function Transition

Fig. 11. Connected Form of *EC*4 (*EC*4*)

Data Limited Cycling Consider the following change to the expense claim process: *Instead of being able to resubmit a claim only once after it has been failed* *by the Supervisor, an Employee can correct and resubmit a claim up to 5 times.* The changes required to the can-model are shown in Figure 10. This modification adds a new derived-state machine, $EC4$, to the can-model which prevents the Supervisor from choosing *Fail* (which would allow the Employee another round of *Correct* and *Resubmit*) if the Employee has already resubmitted at least 5 times. The want-model does not need any change. Note that the Supervisor is presented with up to three possibilities, depending on what the can-model allows, in her to-do list for a resubmitted claim: *Pass*, *Fail* or *Reject*.

In order to create the analysis-model we need to create a connected form version of the new derived-state can-model machine $EC4$, shown in Figure 11. The resultant analysis-model contain a cycle involving *Resubmit* and *Fail* between the two states `(d,i,m/n,p)` and `(f,i,m/n,p)`, and in principle this cycle could continue indefinitely and prevent completion. However, it is easy to reason that the cycle will always exit by noting that the *State Function* for state `p` in $EC4$ in Figure 10 means that:

$$\texttt{resubmits} < 5$$

is an invariant of state `p`. As the *Submit* action sets the `resubmits` count to zero and the *Resubmit* action increments it by 1 in $EC1'$, *Resubmit* can only terminate finitely often at state `p` in $EC4^*$. So eventually a *Resubmit* in must terminate at state `o` which exits the cycle. Reasoning of this kind can generally be used to show where a cycle is data limited.

Time Limited Cycling The idea of a time limited cycle is that exit of the cycle is forced by an Operational Commitment (OC). Suppose that we have a workflow with action alphabet Λ whose analysis-model exhibits a cycle involving a set of states $\mathcal{C}$. Suppose however that:

$$\exists x \in \Lambda \text{ such that } \forall \sigma \in \mathcal{C}, x \in WT(\sigma)$$

We show that the OC on action x will force exit. First note that the action x cannot form part of the cycle (have starting and ending states both in $\mathcal{C}$) for, if this were the case, the ending state would have incoming and outgoing w-transitions for x in violation of well-formedness condition 3 in Section 3.5. So x must leave the cycle. Secondly, as x is both possible and wanted for all states of $\mathcal{C}$, x will remain on some agent's to-do list for as long as the workflow remains in the cycle. As this cannot be longer than $OC(x)$ the workflow must leave the cycle within this time.

An example of a time limited cycle is that formed by the action *Correct* on the state `d,i` in Figure 5. In this case, the cycle has a single state and this state is the start of two w-transitions, *Withdraw* and *Resubmit.*

In theory, if *Correct* is done infinitely fast, it could be done infinitely often before the cycle is exited by *Withdraw* or *Resubmit.* However we will discount this possibility and take it that traces are finite in length.

5.2 Workflow Response Time

Suppose that we have a trace, $\mathcal{T}$, of the workflow which has GC expressed as a sequence $\sigma_i, 0 \leq i \leq n$ of states of the analysis-model, where σ_0 is an Initial State and σ_n is the first encounter of a Completion State. We will now construct the longest time that the workflow can take to follow this trace.

First we form the function $t(i, x)$ for $1 < i < n$ and $x \in \Lambda$ as follows:

$$maxrest(i) = \min_{y \in WT(\sigma_i)} (t(i, y)) \tag{1}$$

$$x \in WT(\sigma_i) \text{ and } x \notin WT(\sigma_{i-1}) \Rightarrow t(i, x) = OC(x) \tag{2}$$

$$x \in WT(\sigma_i) \text{ and } x \in WT(\sigma_{i-1}) \Rightarrow t(i, x) = t(i-1, x) - maxrest(i-1) \tag{3}$$

$$x \notin WT(\sigma_i) \Rightarrow t(i, x) \text{ is undefined.} \tag{4}$$

Here, $t(i, x)$ represents the time that an action x that is possible and wanted in state σ_i has left before breach of its OC occurs; and $maxrest(i)$ is the maximum time that the workflow can remain in state σ_i without violation of any OC. Equation (1) says that $maxrest(i)$ is the minimum of the OC times remaining for any possible and wanted action in state σ_i. Equation (2) says that the time remaining for an action that becomes possible and wanted at the i^{th} step is the OC time of that action. Equation (3) says that the time remaining for an action in that is possible and wanted at the i^{th} step and was also possible and wanted at the $(i-1)^{th}$ step, is reduced by the time that the workflow spends in the $(i-1)^{th}$ step.

Note that $t(i, x)$ can be computed because:

- The GC condition requires that all states for $0 < i < n$ are Safe States with $WT(\sigma_i) \neq \emptyset$.
- By the definition of Initial State, $WT(\sigma_0) = \emptyset$.

The maximum time for the trace $\mathcal{T}$ is then:

$$\sum_{0<i<n} maxrest(i)$$

This can be used to determine whether the end-to-end response time of a workflow conforms to limits that are specified in an SLA (Service Level Agreement) based on the OC limits specified in the various OLAs (Operation-Level Agreement) for the actions involved. In theory, all traces of the workflow would need to be analyzed to determine the maximum response time, which is possible as the absence of cycles means that the number of traces is finite. In practice, a few key paths are important and the analysis does not need to be exhaustive.

5.3 Agent Reliability

The approach to "guaranteed completion" described in this paper is essentially dependent on the reliability of the agents involved to honor their OCs. Clearly if OCs are not obeyed there is no guarantee of either progress or completion. In practice, two techniques are used to assure adherence to OC timings.

Where a human agent reacts to items presented in a workflow to-do list, items that are most urgent (nearest to risking violation of an OC) can be moved to the top of the list, high-lighted, or transferred to an "expedite!" list. Where this is not sufficient, or other factors are involved, a *time-out* can be used. This can be viewed as an action supplied by a clock, which is taken to be a fully reliable agent. Using time-outs can ensure that no Safe State of the workflow lacks a w-transition supplied by a reliable agent. However the time-out will probably put the workflow into some kind of exception state where special action is needed to get it back on track.

6 Related Work

The introduction of the concept of motivation, and construction of a notion of progress based around motivation, is a central theme of this paper. As far as we know our work is the first to explore this idea. The focus of this section is therefore to explain the differences between our work and the other main approaches to the description and analysis of workflow in the published literature.

6.1 Reachability and Soundness

The most common notion used in the analysis of workflow properties is that of *reachability* and the related one of *soundness*, in particular as summarized by van der Aalst et al. [22]. This kind of analysis determines whether or not a workflow **can** reach a specified terminal state or states, and thus whether it **can** complete. No distinction between solicited and non-solicited actions is needed in the context of reachability analysis, as it is performed on the basis of the can-model alone. This is a weaker result than the one to which we aspire, as we aim to address service guarantee and this requires that completion is *certain*, not just *possible.*

6.2 Liveness and Fairness

Concepts of *liveness* and *fairness* are stronger than reachability, being concerned with whether a system is **bound** to reach a desired state; and such questions clearly bear resemblance to the question of whether a workflow is bound to complete. Liveness and fairness have been studied widely and these studies have spawned a variety of definitions and analytical approaches. The main focus of the work has been the study of models that are **non-deterministic** as the result of concurrency, as discussed by Kindler [7]. Here the challenge is to establish sufficient constraints on the non-deterministic choices made in the system to ensure liveness. As papers in this area (such as those by Owicki and Lamport [17] and by Giannakopoulou et al. [6]) explain, this requires that the non-deterministic choices obey *fairness constraints* that place conditions on sequences of choices without determining individual choices. The non-determinacy in our workflow models is of a limited kind, being associated with which action, of all those possible, takes place next. This is sometimes called *external non-determinism.* However, we are **not** concerned to determine constraints on this non-determinism to achieve liveness (completion). Rather, we are concerned to show that under **all** possible choices of next action (provided OCs are obeyed) the workflow will complete.

Note that there is a more limited meaning of the term *fair* that is associated not with non-determinism, but with *loops.* This is the sense in which, for instance, Kindler uses the term in his discussion of workflow [8]. This, different, notion of fairness only concerns that cyclic behavior is limited; and this issue is relevant to our work as discussed in Section 5. Our treatment is the same as Kindler's, except that Kindler does not consider *time limited* cycles.

6.3 Time in Workflow Models

Our discussion of timing, as given in Section 5.2, is very basic. Other authors have considered time and temporal constraints in workflow using more sophisticated approaches, in particular:

– Modeling workflow using variants of standard modeling formalisms in which the states of the model encode time as well as data. Execution traces then explicitly represent the progress of time, as well as the evolution of the data and state of the process, allowing standard model proving techniques to explore temporal properties of the workflow. This approach is used, for example, by Kazhamiakin et al. [16] who introduce a timed version of Labeled Transition Systems, and by van der Aalst [20] who embeds a representation of time into Petri Net modeling.
– Layering a *temporal constraint language* over the top of a process model to describe and reason about the ability of a process to meet timing requirements. This normally involves arguing about whether constraints in the small (at the level of activities) are compatible with constraints in the large (end-to-end). This approach is explored, for instance, by Bettini et. al [4] and by Marjanovic and Orlowska [14].

Both such approaches could be used with the modeling approach described in this paper to provide tools that allow a more sophisticated treatment of time. This would require extension or adaption of the techniques to handle the difference between c-transitions and w-transitions, but there seems to us no reason in principle why this should not be done.

6.4 Modal Specification

There is apparent similarity between our scheme and formalisms that allow a modal dimension to the specification of behavior: two examples are *Modal Transition Systems* (MTS) and *Live Sequence Charts* (LSC). However the semantics of both MTS and LCS differ significantly, but in different ways, from the semantics of our motivation modeling.

In MTS [11] transitions are classed as *required*, *possible* or *not possible* in a manner that appears similar to the scheme using can- and want-models described here. However the distinctions in MTS refer to the degree of *certainty in a specification*, rather an expression of motivation in the final system. As D'Ippolito et al. say when describing MTS: *In MTS, each transition can be either required or maybe. The latter means that it is not yet certain if the interaction modeled by the transition is required or prohibited in the final system.* [12]. The use of *maybe* in MTS therefore reflects looseness or incompleteness in a specification which is expected to be tightened by refinement. This is clearly quite different from the semantics we describe.

The LCS [19] formalism extends Message Sequence Charts in various ways, including the ability to specify whether behavior beyond a location in the chart is mandatory or "provisional". If a location on a LSC is designated *hot*, behavior depicted beyond that point is bound to happen; whereas if a location is designated *cold* behavior is possible but not guaranteed. The guarantee of occurrence of mandatory behavior is semantically similar to the fulfilment of a post-condition, where arrival at the hot location forms the pre-condition; so failure of mandatory behavior to take place represents a failure of the system to meet its specification. This contrasts with the semantics of a w-transition which is only bound to happen in so far as the external agent obeys her OC. Non-occurrence could be because some other transition happens first and does not impugn the model in any way.

Perhaps the key point is that motivation, as embodied in the want-model, concerns a relationship between the software system and the agents outside who interact with it. Modal specification formalisms (such as MTS and LCS) on the other hand are only concerned with the specification of the software itself.

6.5 Treatment of Data

Our aim is to provide a capability to model and analyze applications that have significant data. We do this using a modeling language (Protocol Modeling) that has the capability to model data and behavior and the relationship between the two. We face the well known problem that behavioral properties (such as liveness and soundness) are analyzed as topological properties. Our solution to this is to provide a means of translating behavioral rules based on data into a topological form as described in Section 4.3. The same technique is used by Sidorova et al. [13] working with Petri nets who characterize it as *Checking Correctness in the Presence of Data While Staying Conceptual.* As we do, they use this technique to extend the scope of formal model proving techniques by representing data as topology. This suggests that this technique may have general utility.

7 Future Work

The ideas presented in this paper can be extended to distributed workflows, using a *choreography* to describe the interaction (see Fu et al. [23], Honda et al. [10], Lanese et al. [9] and McNeile [1]). The idea is that Guaranteed Completion (GC) in the distributed workflow can be established by:

– GC analysis of each participant, to show that each participant will complete; and
– GC analysis of the choreography used between the participants, to show that the collaboration between the participants will complete.

Future work will address the techniques and proofs required to address guaranteed completion in distributed workflows using these ideas.

8 Conclusion

The increasing focus on business process management, service based collaborations and outsourcing means that the ability to ensure that workflows will complete and thus support Service Level Agreements is increasingly important. We have shown how to model *motivation* in workflows as an active agent in ensuring progress and completion. We have shown how this is done in local (non-distributed workflow), and in distributed workflow using choreography techniques.

The ideas presented here can contribute to the development of more powerful techniques for the design and verification of e-commerce applications and cross-enterprise workflow to ensure that they will always progress to completion. Such techniques are particularly needed where service level agreements require guaranteed response and non-compliance has financial penalty implications.

References

1. McNeile, A.: Protocol Contracts with Application to Choreographed Multiparty Collaborations. *Service Oriented Computing and Applications*, 4(2):109–136, June 2010.
2. McNeile, A., Simons, N.: Protocol Modelling: A Modelling Approach that supports Reusable Behavioural Abstractions. *Journal of Software and System Modeling*, 5(1):91–107, 2006.
3. McNeile, A., Simons, N.: A Typing Scheme for Behavioural Models. *Journal of Object Technology*, 6(10):81–94, November 2007.
4. Bettini, C., Sean Wang, X., Jajodia, S.: Temporal Reasoning in Workflow Systems. *Distrib. Parallel Databases*, 11:269–306, May 2002.
5. Hoare, C.: *Communicating Sequential Processes.* Prentice-Hall International, 1985.

6. Giannakopoulou, D., Magee, J., Kramer, J.: Checking Progress with Action Priority: Is it Fair? In O. Nierstrasz and M. Lemoine, editor, *Software Engineering ESEC/FSE 99*, volume 1687 of *Lecture Notes in Computer Science*, pages 511–527. Springer Berlin / Heidelberg, 1999.
7. Kindler, E.: Safety and Liveness Properties: A Survey. *EATCSBulletin*, 53(53):268–272, 1994.
8. Kindler, E., Martens, A., Reisig, W.: Inter-operability of Workflow Applications: Local Criteria for Global Soundness. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 235–253, London, UK, 2000. Springer-Verlag.
9. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *Proceedings of SEFM'08, 6th IEEE International Conferences on Software Engineering and Formal Methods*, pages 323–332, Washington, DC, USA, 2008. IEEE Computer Society.
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. *SIGPLAN Not.*, 43(1):273–284, 2008.
11. Larsen, K., Thomsen, B.: A Modal Process Logic. In *LICS*, pages 203–210, 1988.
12. D'Ippolito, N., Fishbein, D., Foster, H., Uchitel, S.: MTSA: Eclipse support for modal transition systems construction, analysis and elaboration. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 6–10, New York, NY, USA, 2007. ACM.
13. Sidorova, N., Stahl, C., Trčka, N.: Workflow Soundness Revisited: Checking Correctness in the Presence of Data While Staying Conceptual. In Pernici, Barbara, editor, *Advanced Information Systems Engineering*, volume 6051 of *Lecture Notes in Computer Science*, pages 530–544. Springer Berlin Heidelberg, 2010.
14. Marjanovic, O., Orlowska, M.: On Modeling and Verification of Temporal Constraints in Production Workflows. *Knowl. Inf. Syst.*, 1(2):157–192, 1999.
15. Object Management Group: UML 2.0 Superstructure Final Adopted Specification. *OMG Document reference ptc/03-08-02*, August 2003.
16. Kazhamiakin, R., Pandya, P., Pistore. M.: Timed Modelling and Analysis in Web Service Compositions. In *International Conference on Availability, Reliability and Security*, pages 840–846, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
17. Owicki, S., Lamport, L.: Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
18. UK Cabinet Office: *OCG Books ITIL - Service Design.* The UK Stationery Office (TSO) for The Office of Government Commerce (OGC), 2011.
19. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.
20. van der Aalst, W.: Interval Timed Coloured Petri Nets and their Analysis. In *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, pages 453–472, London, UK, 1993. Springer-Verlag.
21. van der Aalst, W.: The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
22. van der Aalst, W., van Hee, K., ter Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve M., Wynn, M.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. *Formal Aspects of Computing*, pages 1–31, 2010.
23. Fu, X., Bultan, T., Su, J.: Conversation protocols: a Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37, 2004.