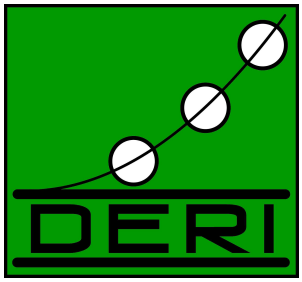

Formal frameworks for workflow modelling

Eyal Oren Armin Haller

DERI Technical Report 2005-04-07
April 2005

DERI Galway
University Road
Galway
IRELAND
www.deri.ie

DERI Innsbruck
Technikerstrasse 13
A-6020 Innsbruck
AUSTRIA
www.deri.ie

DERI Technical Report
DERI Technical Report 2005-04-07, April 2005

Formal frameworks for workflow modelling

Eyal Oren[1] Armin Haller[1]

Abstract. We survey formal frameworks for workflow modelling. We summarise important aspects of workflow management and approaches to evaluate current workflow management systems. We discuss a number of formalisms for workflow modelling, namely Petri nets, Temporal Logic, and Transaction Logic. We decribe how these formalisms are used specifically for workflow modelling, their possibilities and their disadvantages.

Keywords: workflow management, workflow modelling, formal methods, workflow verification.

[1]Digital Enterprise Research Institute, National University of Ireland, Galway.

Acknowledgements: This material is based upon works supported by the Science Foundation Ireland under Grant No. 02/CE1/I131.

Contents

1 Introduction

This paper surveys formal modelling frameworks for workflow management. Such frameworks describe approaches in which a formalism is specifically applied to workflow modelling. We survey existing opinions on these frameworks; we try to give a balanced overview and refrain from introducing our own opinion.

Workflow management deals with supporting business processes in organisations, it involves managing the flow of work through an organisation [3]. Workflows are a collection of coordinated tasks designed to carry out a well-defined complex process [29].

A workflow management system is a generic information system that supports modelling, execution, management and monitoring of workflows. Such a system operates on a workflow specification, a description of the business processes in the organisation that should be supported. A workflow management system can be compared to a database management system: it is a generic system that operates on a schema definition of the (processes in the) organisation.

Workflow modelling is the task of creating workflow specifications. Such specifications will usually be used as input to a workflow management system. The use of formal workflow specifications has often been advocated [8, 23]; formal methods reduce ambiguity and open possibilities for verification and analysis. Different formalisms have been suggested for modelling workflows; those suggestions are the subject of this paper.

We will first summarise some important aspects of workflow management and discuss a recent approach to systematically evaluate some workflow requirements. In section 2 we discuss Petri nets, and advantages and critiques of using Petri nets for workflow modelling. In section 2.4 we discuss YAWL, a recent approach to overcome the limitations of Petri net-based workflow modelling. In section 3 we describe various approaches based on Temporal Logic, a powerful logic to reason about dynamic situations. Finally, in section 4 we describe an approach based on Concurrent Transaction Logic, a versatile logic for database updates.

Aspects of workflow management Jablonski and Bussler [24] distinguish five key aspects of workflow specifications and workflow management: the functional, behavioural, informational, organisational, and operational aspect. These aspects can be seen as dimensions on which a workflow can be defined.

The *functional* aspect describes what should be done in the workflow: it gives a functional decomposition of activities in the workflow. The *behavioural* aspect describes when something should be done: it gives the execution order and dependencies (control flow) of activities in the workflow.

The *informational* aspect describes the data used in the workflow and the data dependencies between activities. In a workflow management system we can distinguish internal data, which is managed by the workflow management system, and external data, which is managed by the environment and exists independently from the workflow management system. Activities can depend on data from other activities; the workflow management system can usually transport such data from one activity to another, which is described in the data-flow between activities.

The *organisational* aspect describes who should do the work in the workflow: it describes constraints on the resource allocation to activities. Activities usually require resources to execute; such resources can be human employees, but also for example computing power or a meeting room. The organisational aspect describes the resources in the organisation, the hierarchy that exists

between these resources, and the policies for assigning resources to activities.

The *operational* aspect describes how the workflow management system should interact with its environment: it describes the methods for accessing or invoking external applications (e.g. interaction mode, invocation mode, parameters) and how to communicate with human users.

Van der Aalst and van Hee [3, 4] define a workflow on three dimensions: the *case* dimension, the *process* dimension, and the *resource* dimension. The case dimension denotes that a workflow consists of different cases that are handled individually: they do not directly influence each other, and are dealt with independently by the workflow management system. The process dimension denotes that the workflow consists of tasks that are related to each other in some routing order. The resource dimension denotes that the tasks in the workflow are carried out by resources, that can be grouped in some roles or organisational units. These dimensions are related to the aspects of Jablonski and Bussler: the process dimension corresponds to the behavioural aspect, the resource dimension corresponds with the organisational aspect.

Workflow Patterns Recently an initiative started to systematically evaluate features of workflow management systems. First a set of *control flow patterns* was compiled by van der Aalst *et al.* [6], based on the various features available in existing systems and common recurring business requirements; these patterns address the behavioural perspective of Jablonski and Bussler. A set of *data patterns* was compiled later by Russell *et al.* [36], covering the information perspective. These patterns identify a comprehensive workflow functionality[1] and provide the basis for comparing workflow management systems and evaluating the suitability of workflow languages.

Control flow patterns The control flow patterns are divided in several groups: basic patterns, advanced branching patterns, structural patterns, multiple instance patterns, state-based patterns, and cancellation patterns.

The basic patterns capture elementary control flow: a *sequence* of activities in which two activities execute after each other (meaning that the second one becomes enabled when the first one completes); a *parallel split (and-split)* and *parallel synchronisation (and-join)* of activities in which several activities execute in parallel (or in any order) and are later synchronised (the synchronise activity waits until all parallel branches are completed, and then continues); and an *exclusive choice (xor-split)* and *exclusive synchronisation (xor-join)* in which one out of several branches is chosen based on some condition and later synchronised (the synchronise activity waits until the active branch completes, since only one branch is assumed to be active).[2]

The advanced branching patterns capture more complicated branching scenarios and are often not directly supported in existing systems: the *multi-choice (or-split)* in which one or more out of several branches is chosen based on some condition; such a split can be joined in three ways: the *synchronising merge* waits for all active branches of the split, and proceeds only after they have completed; the *discriminator* is similar, but the first active branch that completes already

[1] note that these patterns do not describe any minimal business requirements, they are a systematic summary of available features in existing systems. An open question is which of these patterns are essential and necessary for a usable workflow management system.

[2] although the synchronisation patterns are described independently from their split counterparts, it is assumed that they are used together, i.e. that a parallel split is followed by a parallel synchronisation.

triggers the activity following the discriminator, all other active branches that complete later are ignored; the *multi-merge* joins the active branches without synchronisation, the activity following the multi-merge is started once for each active branch.

The structural patterns capture modelling situations that are usually forbidden in workflow management systems: *arbitrary cycles* are unstructured loops[3] without predefined entry- and exit-points; *implicit termination* captures that a process should terminate implicitly when there are no activities left to perform.

The patterns involving multiple instances capture situations on which one case in a workflow has several child cases that are being instantiated in parallel. Each of these child cases needs to complete before the parent case can continue. The problems relate to being able to instantiate child cases from a parent case, and to being able to synchronise these instances and continue with the parent case after all child cases complete. For synchronisation the number of child cases is relevant, this number can be determined at design time or at runtime. If determined at run time it can be fixed before the instantiations start, or it can dynamically change during execution of the child cases. These parameters lead to different patterns: *multiple instances without synchronisation*, *multiple instances with design time knowledge*, *multiple instances with a-priori runtime knowledge*, and *multiple instances without a priori runtime knowledge*.

State-based patterns involve scenarios where an explicit notion of the state of the workflow process is relevant: the *deferred choice* is a split in which one out of several branches is chosen not by the workflow management system but by the environment; all branches are offered to the environment and the first one to be chosen invalidates the others; the *interleaved parallel routing* is an advanced sequential pattern, in which the order of the activities is arbitrary but the activities are not executed in parallel; the *milestone* pattern captures that an activity becomes enabled after completion of some activity but that it becomes disabled again if some other activity starts.

Cancellation patterns involve retracting running cases or executing activities: the *cancel activity* captures that some running activity can be cancelled by another activity or some external event; the *cancel case* captures that the whole running case can be cancelled by the execution of some activity or some external event.

Data patterns Continuing the research describing control flow patterns, Russell *et al.* [36] have studied the use of data patterns for the modelling of the data aspect of workflow management systems. They conclude that the data representation in different workflow tools and business process modelling paradigms has a number of common characteristics. Russell *et al.* identify 39 data patterns recurring in workflow systems divided into four distinct groups.

Data visibility patterns define the scope in which data elements are visible and capable of being utilised, which is primarily determined by the construct to which the data element is bound. Eight patterns are defined that relate to data visibility. They range from the simple pattern where data elements are bound to a task (pattern 1), to data accessible in multiple instances of a single task (pattern 5), to environment data defined in the workflow system that can be accessed within a workflow instance (pattern 8).

[3] "arbitrary cycles relate to structured loops like goto statements relate to while loops" [6].

Data interaction patterns examine the way data elements can be passed between components within a workflow process and are classified in two categories: internal and external data interaction. Six patterns describe the internal data interaction covering from the simple scenario where data flows between task instances (pattern 9), to more complex scenarios where data flows to or from a multiple instance task to a subsequent instance task (patterns 12-13). External data interaction patterns identify how external data is transferred between a component of the workflow and an external application. There are twelve external data interaction patterns, characterised by their dimensions: The type of workflow component that is interacting with the environment (task, case, workflow), whether the interaction is push or pull and what component constitutes the initiation role (the workflow component or the environment).

Data transfer patterns focus on the transfer of data elements between workflow components and on the mechanisms by which data elements can be passed between workflow components. There are seven distinct patterns in this category, ranging from transfer of the the value only (pattern 26 and 27) to patterns dealing with the possible interaction when applying a common data store (pattern 28, 29, 30) to patterns describing scenarios where data transformation functions are applied to a data element (pattern 31, 32).

The final set of patterns, called data-based routing patterns, capture the various ways in which data elements can interact with other workflow aspects. There are seven patterns in this category, ranging from scenarios where data-based preconditions (pattern 33 and 34) or postconditions (pattern 35 and 36) have to be checked prior to execution of a task to data-based routing (pattern 39) describing the ability to alter the control flow within a workflow case to trigger one or several subsequent task instances depending on an expression evaluating the values of data elements.

2 Petri nets

2.1 Classical Petri nets

Petri nets are a formalism for modelling dynamic systems. They are graphical, mathematically formalised, and well analysable. The original nets were developed by Petri [32]; later, various extensions were developed providing new modelling constructs. Some of these extensions provide easier modelling keeping the same expressiveness as classical Petri nets, some provide more expressiveness [31]. Petri nets have been applied to a large number of areas, including communication protocols, performance evaluation, and distributed systems [30], because they are very general and were the first to model concurrency [10].

A classical Petri net consists of places and transitions. Places contain zero or more tokens, transitions connect places to each other with directed arcs: each transition has a number of input places and a number of output places. A Petri net has an initial marking, stating the number of tokens in each place. The marking changes discretely by *firing* of transitions: when a transition fires, it consumes one token from all its input places, and produces one token in all its output places. A transition can only fire when it is *enabled*: if in each of its input places there is at least one token. When more than one transition is enabled any one of them can fire but they cannot fire simultaneously.

input places	transition	output places
preconditions	event	postconditions
input data	computation step	output data
resources needed	task or job	resources released
conditions	logical clause	conclusions

Table 1: Typical interpretations of places and transitions in Petri nets

Definition Formally, a Petri net is a tuple $PN = (P, T, F, M_0)$, where P is a set of places, T is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation from places to transitions and vice versa, and $M_0 : P \to \mathbb{N}^+$ is the initial marking. The sets of places and transitions are disjoint and a Petri net should contain at least one place or one transition: $P \cap T = \emptyset$, $P \cup T \neq \emptyset$.

The evolution of a Petri net is described by a sequence of markings representing subsequent global states of the Petri net: $(M_0, M_1, \cdots, M_n)$. A successive pair of markings (M_i, M_j) denotes the firing of one transition. Such a pair is legal if and only if there is one transition that is enabled in M_i, and the marking of the input places of that transition is decreased by one between M_i and M_j, and the marking of the output places of that transition is increased by one between M_i and M_j.

Alternative definitions A Petri net can alternatively be described using an incidence matrix that denotes how a transition changes the marking of each place [30, 34]. For a Petri net with n transitions and m places, the incidence matrix $A = [a_{ij}]$ is an $n \times m$-matrix. Each entry $a_{ij} = a_{ij}^+ + a_{ij}^-$ in this matrix denotes the change of tokens in place j induced by the firing of transition i. The number of tokens produced in j is denoted by a_{ij}^+, the number of tokens consumed in j is denoted by a_{ij}^-. A transition i is enabled if and only if for all places j the number of tokens to be consumed is smaller than its marking: $a_{ij}^- \leq M(a_j)$.

A Petri net can also be depicted graphically, using boxes for transitions and circles for places. The flow relation is depicted with arcs connecting places and transitions, and the marking is depicted with black dots in each place.

A Petri net is an abstract model, it can be used to represent different situations. Table 1 shows some examples of possible interpretations, from [30]. For example, input and output places can be used to describe pre- and postconditions for events, the firing of a transition denotes the occurrence of an event. Or, input and output places can represent resources that are reserved and released (or used and produced) by an activity. One can also use a Petri net with an interpretation that combines the examples given; usually the labels given to transitions and places, together with the accompanying text, explain the interpretation that is given to the Petri net. Note that the semantics of a Petri net is independent of the interpretation given to it: a Petri net behaves exactly the same whether its places represent input data, preconditions, or rule antecedents.

Petri net variants Many extensions have been developed that ease the modelling of situations using Petri nets or extend the expressivity of classical Petri nets; we will shortly discuss a number of relevant extensions for workflow modelling.

Weighted Petri nets allow modelling classical Petri nets in a more succinct way: weights are added to arcs to denote that multiple tokens are consumed and produced during the firing of a

transition. A k-weighted arc corresponds to k parallel arcs in classical Petri nets.

Coloured Petri nets [25, 41] allow modelling the identity of individual tokens by attaching values (or colour) to tokens. This enables more detailed modelling of the objects that the tokens represent in the Petri net; in a Petri net of a workflow model tokens represent for example cases that are handled during the workflow, or resources that are required for activities. In coloured Petri nets one can model attributes of the cases, for instance a complaint number or the salary of an employee. Transitions can change the values of tokens; they operate on the objects that are represented by the tokens. The colour of tokens can also be a mare elaborate data structure.

Timed Petri nets allow modelling the temporal behaviour of a system. There are a number of ways to introduce time into the classical Petri net [1]. A common approach is to associate a timestamp with each token (denoting the moment in time that the token is available for consumption) and to associate a time delay with each transition (denoting the change to the timestamp of produced tokens). When modelling time delays in Petri nets it is possible to analyse quantitative performance indicators such as response times, occupation times, and throughput times.

Hierarchical Petri nets allow compositional modelling: a transition can be not only atomic but also complex, in which case there is some subnet that defines this complex task. The net can be unfolded by replacing all complex transitions with their defining subnets. Hierarchical (or modular) modelling makes it easier to create, maintain, and understand models.

A special class of high-level Petri nets suitable for workflow modelling is identified by van der Aalst [3]; these *workflow nets* have two special properties: they have exactly one input place and one output place, and no dangling transitions or places exist (each transition and each place are on a path from the input place of the net to the output place of the net).

2.2 Workflow modelling using Petri nets

Recall that according to van der Aalst [4], a workflow can be viewed on three dimensions: the case dimension, the process dimension, and the resource dimension. The case dimension denotes that a workflow consists of different cases that are treated individually, the process dimension denotes that a workflow consists of several tasks in some routing order, and the resource dimension denotes that the tasks in a workflow are carried out by some organisational resources.

Such a workflow can be modelled in a workflow net as follows [3]: cases are modelled by high-level tokens, that have an identity, are distinguishable from each other, and can contain structured data; the process dimensions is constituted by places and transitions: tasks are modelled by transitions, conditions on tasks are modelled as places. A workflow model can thus be constructed specifying how cases are routed through tasks, and how the execution of tasks changes the conditions for executing tasks. The Petri net can model routing constructs like sequences, parallel splits and joins, exclusive splits and joins, and iterations; see [6, 24] for demonstrations on modelling advanced routing constructs using Petri nets. Since a workflow management system is not in complete control of a workflow but is only supporting it, the difference between the enabling of a task and the execution of a task is important. Each task needs to be enabled before it can be executed, but an enabled task does not have to execute. The execution of a task is determined by the environment and not by the workflow management system. This is modelled by triggers, which are external events that leads to the execution of an enabled task. A trigger can be a user action, a clock event, an external message, or automatic (which means that if the task is enabled, it is directly executed).

Advantages Many researchers have advocated using Petri nets for workflow modelling, e.g. [4, 18, 24, 37, 42]. Van der Aalst [2] mentions three reasons for using Petri nets: their formal semantics, the fact that they are state-based instead of event-based, and the abundance of analysis techniques available:

First, having formal semantics has the advantages that a workflow specification is unambiguous, that the interpretation of a workflow specification is defined mathematically and does not depend on the implementation decision of particular tools, and that it is possible to reason formally about properties of a workflow specification. (These advantages apply equally to all formal frameworks, they are not specific to Petri nets.)

Second, being state-based instead of event-based means that states are modelled explicitly. In contrast, event-based frameworks only model the transitions between states explicitly. Modelling the states of a workflow explicitly is useful in workflow modelling since (a) it allows a distinction between enabling of a task and execution of a task (a task must be enabled before it is executed, but it does not *have to* execute when it is enabled), an enabled task can wait for a trigger: a user action, an external message, or a time event; (b) it allows modelling competitive tasks (where two tasks are enabled for the same case and the first task to execute 'steals' the case from the other one); (c) it allows cancellation of cases by removal of tokens from places; (d) in distributed workflows, it allows transfer of cases from one system to another by transferring the relevant tokens; (e) it models workflow systems as reactive to the environment by allowing external events to change the state of the system.

Finally, many analysis techniques exist for verifying certain qualitative and quantitative properties of a Petri net workflow model. Interesting qualitative properties include checking possible occurrences of deadlocks, checking reachability of all tasks, checking boundedness of the tokens in all places. It is also possible to verify performance measures such as response times, waiting times and resources occupation rates.

2.3 Problems of Petri nets

Workflow management systems are reactive According to Eshuis and Dehnert [20] the standard token-game semantics of Petri nets is not completely suitable for modelling workflow management systems. The token-game semantics of Petri nets models closed active systems, where workflow management systems are actually open, reactive systems. The behaviour of some workflow model under the token-game semantics can thus be different than the run-time behaviour of a workflow engine using that same model.

A workflow management system does not execute activities but merely coordinates the execution of these activities by human actors or software systems. A workflow management system does not control the environment but reacts to events in the environment (start of cases, termination of scheduled tasks) by creating certain effects in the environment (initialisation of activity instances, scheduling of follow-up tasks). A reactive system is usually modelled using event-condition-action rules, stating the actions with which the system should respond to events. A reactive system must respond to events in the environment with the actions specified in its rules.

Eshuis *et al.* [19, 20, 21] describe several problems when using Petri nets for modelling workflow management systems. The underlying problem is that the distinction between the environment and the workflow management system is not captured:

Events In a Petri net events can be modelled in various ways:

Events can be modelled as tokens: a place is created for each input event, in which the environment is supposed to place a token when an event occurs. The problem is that event broadcasting is not possible since transitions consume these event tokens, this makes for example cancellation events problematic to model.

Events can be modelled as transitions: a transition is created for each event. Synchronisation is needed between transitions that model activities and transitions that model events, to model that a transition is triggered by the occurrence of an event.

Activities In a Petri net activities can be modelled in various ways:

Activities can be modelled as transitions: a transition is created for each activity, denoting the state change by executing the activity. The first problem is that transitions in a Petri net are instantaneous while activities in a workflow are not (this can be addressed by modelling both a begin-activity and an end-activity transition).

The second problem is that using transitions to model activities gives the false impression that the workflow management system executes activities; this is not the case, activities are executed by the environment. Other transitions in the Petri net however do model some action of the workflow management system (e.g. routing and decision transitions). But the distinction between transitions that indicate actions by the environment and transitions that indicate actions by the workflow management system is not captured in the Petri net. This poses problems if we would like these transitions to behave differently.

The third problem is that in the token-game semantics of Petri nets transitions are not directly suitable to model a reactive system: enabled transitions *may* fire (but do not have to do so), but a reactive system *must* react to occurring events; in the worst case firing in a Petri net can be postponed forever which contradicts the notion of reactivity. To model a reactive system correctly one needs to distinguish transitions that are used to model activities from transitions that are used to model events: the semantics of these transition types should differ.

Activities can also be modelled as places: a place is created for each activity denoting that the activity is in progress. The problem is that changes in a Petri net can only occur through the firing of transitions. Since activities should be able to update case attributes they can not be captured modelling them as places.

Data In a Petri net several issues exist concerning data modelling:

Who updates the data? A workflow management system does not execute any activity itself, it merely routes cases to workflow participants. However, activities are usually modelled as transitions, and the case data (usually modelled using coloured Petri nets) are changed in the transition. This does not model the situation faithfully. Again, it is necessary to distinguish between the environment and the workflow management system to capture that data is updated by workflow participants.

How to ensure data integrity? When activities are modelled as transitions that update case data (token data) one needs mechanisms to model and ensure transactional properties of data access without unnecessarily restricting concurrent data access. This is cumbersome to model in Petri nets.

Eshuis *et al.* propose two solutions for these issues: the first solution is based on UML activity diagrams, for which a reactive formal semantics is provided. Workflows specified in activity diagrams can be verified using temporal logic; we describe this in section 3.2.

The second approach consists of an alternative, reactive, semantics for Petri nets instead of the classical token-game semantics that deals with the above issues [20]. The approach takes three steps to address the problems noted.

First, the representation of workflow tasks is refined such that a task is not modelled by a single transition, but by transitions *announce_task, begin_task, end_task,* and *record_task_completion.* Explicitly modelling these phases in a workflow task allows one to distinguish more clearly between the things that are performed by the workflow management system and those that are performed by the environment of the workflow management system: announcing and recording completion of tasks is done by the workflow management system, but beginning and ending tasks (actually performing the task) is done by the environment (an employee of the organisation or an external software system).

Secondly, the division of power between the workflow management system is explicitly modelled by distinguishing internal and external transitions. Internal transitions are transitions that are executed by the workflow management system, external transitions are executed by the environment. Announcing tasks and recording their completion is represented by an internal transition, and also decision and routing tasks are represented by internal transitions (these are tasks in which the workflow management system decides which activity should be performed next based on some business rules). The beginning and ending of tasks are modelled as external transitions, as are events (those are not things that the workflow management system does, but it merely observes them).

The final step is to adjust the semantics of classical Petri nets to use the separation between internal and external transitions to model reactive systems. To transform a Petri net into a reactive Petri net the standard firing rule should change, because it states that an enabled transition *may* fire. That is appropriate for modelling the environment since it is not controlled by the workflow management system. But for the workflow management system itself a *must* firing rule is more appropriate since otherwise the system might fail to respond to an external event, which would contradict the definition of a reactive system. Also, internal transitions have priority over external transitions, which ensures that the system actually reacts to all external events in an appropriate way.

These changes lead to a model where the environment leads: an external transition triggers all relevant internal transitions, which will fire to react to the external transition; then the environment can change again after which the system can react again.

Support for workflow patters According to van der Aalst and ter Hofstede [5], Petri nets have three limitations when modelling workflows: they are ill-suited for the workflow patterns involving (1) multiple instances, (2) advanced synchronisation, and (3) cancellation.

Patterns involving multiple instances occur in workflows where some child case is instantiated a number of times while dealing with some case. The number of instantiations of these child cases can be defined during design-time or during run-time. For example, during a conference review process, for each submitted paper a number of reviewers is selected and asked for their opinions; these reviews are child-cases of the submitted paper, the number of reviewers can vary on the content of the submitted paper, and it necessary to synchronise the outcome of these child-cases.

These patterns are problematic since the workflow execution needs to keep track and synchronise these multiple instantiations, in order to keep the workflow consistent. It is possible to model such patterns using Petri nets, but it is quite involving. First, one needs to synchronise all child-cases of one case but not confuse them with child-cases of another case. Second, one needs keep track of the number of active instances of child cases (the child cases can only be synchronised when there are no active child cases; the number of child cases can differ per execution of the workflow).

Advanced synchronisation patterns occur in workflows where branching and merging in splits and joins are defined optionally: sometimes all branches are taken, sometimes only some branches are taken. When synchronising such branches the number of branches to synchronise (and wait for) obviously depends on the number of branches that were activated in the split. The second issue is that there are a number of patterns for the activity that follow such a synchronisation of optional branches: sometimes it is desired that this activity is executed only once (first all branches are synchronised, and then the activity is executed), sometimes it is desired that this activity is executed once for each active branch. It is theoretically possible to model these patterns in a Petri net. One can explicitly add synchronisation information into the model (information can be passed from the split node to the join node), or one can change the firing rule of transitions (and change the semantics of Petri nets) to consider whether more tokens are able to arrive at a synchronising transition. Again the burden is on the workflow designer to explicitly model these patterns.

Cancellation patterns occur in workflow where some activity should lead to the withdrawal (cancellation) of cases from the workflow. Cancellations can occur at random moments and should have the desired effect independent of the state of the workflow execution. The problem is that in Petri nets all rules are local to a single transition or state, whereas cancellation is a pattern that has global effects: it does not matter in which state the workflow is, the entire case should be cancelled. One could model this with a cancellation transition that tries to remove a token from any place where it could appear, but the resulting net would be very complicated (it would contain spaghetti-like arcs to remove tokens from all combinations of all places).

Global Constraints The problem of modelling cancellation patterns in Petri nets is actually only a specific example of the more general limitation of Petri nets: according to Davulcu *et al.* [17] it is not possible to specify global constraints that relate multiple states, resources, or activities to each other. Local constraints range over a single state, global constraints range over a sequence of states, or over an execution trace of the workflow. Typical global constraints as formulated by Klein [27] are occurrence (if a is ever executed, then b must also be executed at some point) and order (if a is ever executed, then b must be executed before a).

Davulcu [16] describes three common frameworks for specifying workflow are: control flow graphs, triggers (or event-condition-action rules), and temporal constraints. Control flow graphs, for example Petri nets, specify the control flow dependencies between activities: they typically specify the initial and final activities, the successor relation between activities, and the branching relation between successor activities (e.g. parallel or exclusive). One can add more advanced controls as discussed in for example [6]. The control flow dependencies between activities can be enriched with conditions, such that the successor activity may only start if the condition is satisfied. Such conditions range over the *current* state of the workflow (which can include both workflow internal data and external application data).

Because conditions in control flow graphs only range over one (viz. the current) state of the workflow, it is not possible to express global constraints which range over a sequence of states. One

can construct a control flow graph that satisfies certain global constraints, but one cannot specify these constraints in the control flow graph. This may seem similar, but it is not in case a set of constraints can be satisfied by different control flow graphs. Given such a set of constraints, one can give many control flow graphs that satisfy it; giving just one of those graphs only conveys part of the constraint information: it gives one solution to the (constraint satisfaction) problem, but does not describe the problem itself anymore.

Adam *et al.* [7] show how one can model various dependencies in Petri nets, including control flow dependencies, and then verify and analyse the resulting Petri nets. They describe how one can model the solutions to common constraints, including Klein's constraints, using Petri nets. The constructed Petri nets contain one solution to the constraint problem, but do not completely contain the constraints themselves.

What is the difference between modelling the constraints in a declarative framework or constructing a Petri net in which the constraints are guaranteed to hold? According to Mukherjee *et al.* [29] a workflow specification serves two purposes: verification and scheduling. An important verification issue concerning workflow specifications is whether some constraint is ensured by some set of other constraints (so that the constraint does not need to be explicitly enforced). In Petri nets this means verifying that one Petri net (that contains the explicit constraint) is equivalent with another Petri net (that does not explicitly contain this constraint), but such verification is in general undecidable [31].

Scheduling is to automatically construct, from a set of constraints, a Petri net that satisfies these constraints. This could be done by programming a scheduler that follows the manual approach from Adam *et al.* [7]. But such a scheduler does currently not exist [16, p. 34].

2.4 YAWL

To overcome the disadvantages of Petri nets in modelling workflow patterns van der Aalst and ter Hofstede [5] propose a new workflow language. YAWL is based on Petri nets (workflow nets) but has additional mechanisms for direct and intuitive modelling of all workflow patterns. YAWL can be mapped to Petri nets, but has an independent semantics.

A YAWL specification is a hierarchical extended workflow net, consisting of tasks (transitions) and conditions (places). Tasks are either atomic or composite, referring to a lower level extended workflow net. Tasks can be directly connected to each other, a hidden condition is then added implicitly between those tasks.

Basic and advanced branching patterns are directly supported: special tasks exist for modelling and/xor/or splits and joins. Patterns involving multiple instances are also directly supported: tasks can have multiple instances, one can specify a lower and upper bound for the number of instances created, and whether the number of instances for a task can be changed during execution of the task. One can also specify whether such a multiple-instance task completes when a threshold number of its instances complete, or whether it should wait for all its instances to complete.

Cancellation patterns are directly supported using a notation that indicates token removal: one can connect (with rounded rectangles) a task to a number of places, denoting that upon execution of the task any number of tokens in those places will be removed; the task is not dependent on these tokens, if none are present the task will still execute.

An operational semantics for YAWL is given by a transition system that specifies (for an execution of a YAWL specification) the state space and allowed transitions. A definition of soundness,

comparable to soundness of workflow nets, is given as a minimal notion of correctness; a specification is sound if and only if it can properly complete (it completes exactly once) without dangling tasks (tasks that are still executing).

Since the semantics of YAWL is not given in terms of Petri nets one can not directly reuse verification and analysis results developed for Petri nets. The authors include a proof of compositionality of YAWL specifications, comparable to compositionality of workflow nets, to indicate that many results from Petri nets can be carried over to YAWL. However, to the best of our knowledge such work does not exist yet. Therefore it is difficult to judge the value of YAWL in its achievements over Petri nets: although it allows far more suitable modelling constructs it is unclear what analysis can be achieved.[4]

3 Temporal Logic

Temporal logic is a generic logic for representing and reasoning about temporal information. Different variants of temporal logic have been developed, an overview is given by Chomicki and Toman [15]. Temporal logic was introduced to computer science by Pnueli [33], it is widely used for formal specification and verification of concurrent and reactive programs [28].

In general, temporal logic extends classical logic with modal operators that denote a temporal modifier, such as the necessity operator $\Box$ and the possibility operator $\Diamond$. Formulas are evaluated over a path (sequence) of states, $\Box e$ is true if e holds in all states in a sequence; $\Diamond e$ is true if e holds in some state in a sequence.

3.1 Workflow modelling using temporal logic

Attie *et al.* [9] propose to model intertask dependencies in workflows using computational tree logic, a temporal logic variant. They extend the work of Klein [27] to a more expressive, more general, and more formal framework. The approach deals with specifying intertask dependencies in a multi-database environment, in which certain transactional properties of task executions should be ensured.

A workflow consists of a number of different tasks that need to be performed. The ordering or control flow of these tasks is not predefined, instead tasks are (at run time) requested for execution by several independent databases. A central scheduler is responsible for managing the (scheduled) execution of these tasks such that they satisfy some global dependencies. The scheduler does not execute tasks but schedules their execution through an event dispatcher that communicates with the requesting databases.

The exact nature of the tasks in this framework is not relevant, all tasks are characterised by events. A task can be in one of the following states: not executing, executing, done, aborted, and committed. The transitions between these states constitute the significant events of a task: start, pre-commit, commit, and abort. Intertask dependencies are constraints over these significant task events.

Events can be forcible (the system can initiate them), rejectable (the system can prevent them), or delayable (the system can delay them). Table 2 [9] shows the attributes of significant events

[4] given the current state of theoretical developments YAWL is more interesting as a workflow system than as a theoretical framework for workflow modelling. As a system it is already quite advanced; we will investigate it during a system survey in the m3pe project.

event	forcible	rejectable	delayable
start	yes	yes	yes
pre-commit	no	no	no
commit	no	yes	yes
abort	yes	no	no

Table 2: attributes of significant events

commonly found in database systems. Requested events are submitted to a scheduler, and correspond to a requested phase transition. The scheduler decides whether a submitted event violates some dependency; if not, the event is allowed to be executed, otherwise the event is delayed (if delayable) and re-attempted later.

The different attributes of events lead to different strategies to achieve certain dependencies. For example, Klein's occurrence dependency $e_1 \rightarrow e_2$ is enforceable if $rejectable(e_1)$ and $delayable(e_1)$, since e_1 can either be delayed until e_2 is submitted, or e_1 can be rejected if e_2 is never submitted. The latter can be derived when abnormal termination of e_2 has been submitted.

The dependencies between events are specified using computational tree logic. Each dependency is represented as a finite state automaton. Such a automaton represents the possible orders of events on which the dependency is satisfied. The automaton can be automatically constructed from the specification of the dependency.

To check whether a requested event should be permitted or rejected, the scheduler queries all automata (representing all dependencies). If all automata can accept the event, the event is given to the dispatcher, and all automata change their local state to stay synchronised to the global execution state. If all automata can reject the event, i.e. if they can accept a reject transition for the event, the event is rejected. Otherwise, the event is put in a pending set, and a decision on it will be reattempted later.

3.2 Temporal logic for workflow verification

Eshuis *et al.* [19, 21, 22] use temporal logic to specify requirements (constraints) on a workflow and a temporal logic model checker to verify whether these requirements are met by a workflow specification. The goal is to hide the use of temporal logic from the workflow modeller: workflow are specified using UML activity diagrams, requirements on the workflow are (planned to be) specified using an abstract requirements language. The activity diagram representing a workflow is automatically transformed into a transition system; the requirements are translated to temporal logic; a model checker verifies whether a trace of the transition system can be constructed such that the constraints are satisfied.

3.3 Problems of temporal logic

Rusinkiewicz and Sheth [35] discuss a number of approaches to specification and execution of transactional workflows. They conclude that although the approach of Attie *et al.* [9] is correct and safe, the computational costs are too high. Given that the architecture is centralised they find the approach not appropriate for managing many dependencies, at least not without additional optimisations.

To address these issues, Singh [39, 40] proposes a decentralised process algebra for events, that can specify acceptable executions or traces of a workflow. Again the language abstracts from specifics of tasks: event symbols are the atoms of the language. According to Mukherjee *et al.* [29], it is still unclear how this algebra can be used for modelling hierarchical workflows, or for verifying constraint redundancy or whether some constraint is implied by a set of constraints.

A different approach for optimising the computational cost is described by Davulcu *et al.* [17]: instead of using general model checking for temporal logic, they develop a verification algorithm that deals with the specific kind of constraints that occur in workflow management; we describe this in the next section. Their work addresses an issue in temporal logic: because is general and powerful, the logic is computationally expensive; using a more specific logic for the problem at hand could lead to optimised algorithms.

4 Transaction Logic

Several publications propose using Transaction Logic ($\mathcal{TR}$) for modelling, executing and reasoning about workflows [12, 17, 26, 29, 38]. $\mathcal{TR}$ is an extension of classical predicate logic which provides a logical foundation for state changes in databases and logic programs. The logic programming fragment of $\mathcal{TR}$ provides a logical environment for programming evolutions of databases. The full logic can be used to reason about such programs [11].

Programming in $\mathcal{TR}$ means specifying a set of transactions that operate on a database; given an initial state of the database, executing these transactions will bring the database to a certain state. Transactions are either atomic operations (queries and updates on the database) or complex actions (composed out of atomic operations in certain ways). $\mathcal{TR}$ focuses on operators for composing complex transactions.

Atomic operations: data and transition oracles The atomic operations that are available to the $\mathcal{TR}$ programmer, and the semantics of these actions, are not predefined in the logic. Rather, one can parametrise the logic with a data oracle and a transition oracle: the data oracle specifies a set of primitive database queries (the static semantics of the database), the transition oracle specifies a set of primitive database updates (the dynamic semantics of the database).

A data oracle maps a state identifier to a set of formulas that are true in that state; the logic does not need to know the real nature of the database: to answer a query on a state it asks the data oracle which formulas are true in that state, and compares those formulas to the query posed. A transition oracle maps ordered pairs of state identifiers to sets of ground atomic formulas. Those ground atoms are elementary updates, the transition oracle defines the set of atoms that produce a certain state change.

A relational database would for example be represented as follows: the state of a relational database is a set of ground atomic formulas (relations are predicates, tuples are ground atoms) thus the data oracle simply returns this set of ground formulas for a given state. The transition oracle would consist of, for each predicate symbol p in a state, an operation $p.ins(x)$ and $p.del(x)$. The definition of these insert and delete operations depends on the update semantics of the relational database: one could define strict updates, such that $p.ins(x) \in (D_1, D_2)$ if and only if $x \notin D_1 \wedge D_2 = D_1 \cup p(x)$.

One could also define transition oracles that behave differently (that is indeed the point of

separating the semantics of atomic operations from the rest of the logic); one could for example define a delete operation that succeeds even if the tuple to be deleted is not in the database.

$\mathcal{TR}$ is thus a logic for combining elementary database operations. By separating the definition of elementary operations from combining them, the logic does not commit itself to a particular theory of updates and can accommodate a wide variety of database semantics.

Logical programming: executing a database evolution Transactions are executed by asking the logical proof system to proof a certain formula (a *goal*). The system tries to construct a proof of that goal; that proof is built on the axioms of $\mathcal{TR}$ and the data and transition oracles. In constructing a proof the system poses queries and updates to the database.

This is achieved through the notion of *executional entailment*: formulas in $\mathcal{TR}$ have both a truth value and a side effect on the database. Truth values of formulas are evaluated over paths (sequences of states). The truth value denotes whether the formula can execute along a path, the side effect denotes how the database is changed by executing the formula. Formally, this is written as $\mathbf{P}, \mathbf{D_1}, \mathbf{D_2} \models u$, meaning that, given a transaction base $\mathbf{P}$, the elementary update u changes the database from state $\mathbf{D_1}$ to $\mathbf{D_2}$. Which entailments hold depends on the data and transition oracles.

4.1 Flavours of Transaction Logic

There are a number of different flavours of $\mathcal{TR}$; all these flavours come with a model theory for the full logic, and a sound-and-complete proof theory for the logic programming fragment. The proof theory for the logic programming fragment can be used to execute programs. The flavours differ in the operators for composing atomic operations into complex operations. In all these flavours, atomic formulas (predicates) represent elementary actions or queries on a database.

(Sequential) Transaction Logic [14] For composing atomic formulas into complex ones, (sequential) $\mathcal{TR}$ uses the usual classical operators $(\neg, \wedge, \vee, \leftarrow)$ and two additional operators: serial conjunction, denoted $\otimes$, and a modal operator for executional possibility, denoted $\diamond$. These operators mean the following:

- *serial conjunction*: $\alpha \otimes \beta$ means first execute α and then execute β;
  $\mathbf{M}, \pi \models \alpha \otimes \beta$ if and only if $\mathbf{M}, \pi_1 \models \alpha$ and $\mathbf{M}, \pi_2 \models \beta$ and $\pi = \pi_1 \bullet \pi_2$. Here $\mathbf{M}$ is a path structure which determines the truth value of each formula on a path (indicating whether the formula can execute along a multipath), π is a path in $\mathbf{M}$, and $\bullet$ denotes the concatenation of two paths.

- *classical conjunction*: $\alpha \wedge \beta$ means execute α such that it is also a valid execution of β; $\mathbf{M}, \pi \models \alpha \wedge \beta$ if and only if $\mathbf{M}, \pi \models \alpha$ and $\mathbf{M}, \pi \models \beta$.

- *classical disjunction*: $\alpha \vee \beta$ means execute either α or β; $\mathbf{M}, \pi \models \alpha \vee \beta$ if and only if $\mathbf{M}, \pi \models \alpha$ or $\mathbf{M}, \pi \models \beta$.

- *negation*: $\neg\alpha$ means execute anything but α; $\mathbf{M}, \pi \models \neg\alpha$ if and only if $\mathbf{M}, \pi \not\models \alpha$.

- *subprocedure*: $\alpha \leftarrow \beta$ means that to execute α one can execute β; $\mathbf{M}, \pi \models \alpha$ if and only if $\alpha \leftarrow \beta$ is in $\mathbf{M}$ and $\mathbf{M}, \pi \models \beta$.

- *possibility*: $\diamond\alpha$ holds in a state if α will be true at some state in the execution path; $\mathbf{M}, \mathbf{D} \models \diamond\alpha$ if and only if $\mathbf{M}, \mathbf{D} \bullet \pi \models \alpha$ for some path π.

Concurrent Transaction Logic [13] In (sequential) $\mathcal{TR}$ one can only compose elementary database operations in a sequential manner. One can specify a linear order on a set of transactions, and build complex transactions out of simpler ones.

Concurrent Transaction Logic ($\mathcal{CTR}$) enables concurrent compositions of operations. It extends $\mathcal{TR}$ with two operators: concurrent composition, denoted $|$, and a modal operator of isolation, denoted $\odot$.

In $\mathcal{CTR}$ the truth value of transactions is evaluated on multipaths, which are sets of sequences of paths. A multipath records the execution history of a process and represents periods of continuous execution, separated by periods of

- *concurrent composition*: $\alpha \mid \beta$ means execute α and β concurrently in an interleaved fashion; $\mathbf{M}, \pi \models \alpha \mid \beta$ if and only if $\mathbf{M}, \pi_1 \models \alpha$ and $\mathbf{M}, \pi_2 \models \beta$ and the multi-paths π_1 and π_2 can be interleaved to π.

- *isolation*: $\odot\alpha$ means execute α in isolation without interleaving; $\mathbf{M}, \pi \models \odot\alpha$ if and only if $\mathbf{M}, \pi \models \alpha$ and π is a path but not a multipath.

Transaction Datalog [12] Transaction Datalog ($\mathcal{TD}$) is a reduction of the Horn-fragment of $\mathcal{CTR}$, just as Datalog is a reduction of the Horn-fragment of classical logic. $\mathcal{TD}$ is equivalent to Horn-$\mathcal{CTR}$ without negation, and where all the rules are safe, i.e. no variables appear in the head of a rule that do not appear in the body of a rule. Safeness guarantees that the database domain is not expanded during execution since no tuples can be introduced during execution.

Bonner [12] develops a layered family of $\mathcal{TD}$-languages, that each restrict a certain modelling feature. These restrictions gradually reduce the computational complexity of the language. Full $\mathcal{TD}$ is data complete[5] for RE[6]. Removing updates from the language lowers the complexity to PTIME (since $\mathcal{TD}$ without updates reduces to classical Datalog), while removing concurrency from the language lowers the complexity to EXPTIME. Since these modelling features are necessary for workflow modelling, a complex restriction called *full boundedness* is introduced, that provides a blend of modelling features, and is data complete for NP. Full boundedness means that a recursive transaction can only be executed a finite number of times, and that the data flow between transactions is acyclic.

4.2 Workflow modelling using $\mathcal{TR}$

Straightforward workflow scheduling Kifer [26] and Bonner [12] describe how variants of $\mathcal{TR}$ can be used for modelling and executing (simulating) workflows. A workflow is specified as a set of $\mathcal{TR}$ formulas that describe dependencies between tasks. Tasks are modelled using predicates, they can be either atomic or be defined as a subprocedure (this allows for compositional workflow modelling). A sound-and-complete proof theory exists for the concurrent-Horn subset of

[5] data completeness of a language for a complexity class means that the most complex transaction of the language is in that class; the data complexity of a transaction is the complexity of determining whether a given pair of database states is in that transaction.

[6] the complexity class denoting Turing completeness

$\mathcal{TR}$, which includes sequential composition, concurrent composition, and disjunction (it does not include classical conjunction, as we will see in the next paragraph). The proof theory can be used to search for proofs of workflow models; such proofs describe executions of workflows, it describes an execution path of a database. Scheduling a workflow execution consists of proving a goal from a workflow model.

Simple sequential and concurrent workflow can be defined using the basic $\mathcal{TR}$ operators for sequential and concurrent composition. Choice in a workflow (or preconditions for tasks) can be modelled by sequential composition of a query and a task: if the query succeeds the task is executed. Sharing of resources can be modelled using a database predicate that indicates usage and release of the resource: each task queries for the availability of the resource as precondition for its execution; before starting to execute, the task removes the availability of the resource from the database, and inserts it again after committing the task: concurrent processes thus lock shared resources before using them.

Synchronisation of tasks can also be achieved by communicating through the database: one task can have as precondition the commitment of another task; this other task indicates its commitment by inserting some predicate to the database, that is a precondition for the first task. Transactional properties of the workflow are guaranteed by the logic: if an operation fails inside a transaction, the transaction is rolled back. Kifer [26] demonstrates how one can easily model partial commits, if-then-else rules or while-loops on top of the standard $\mathcal{TR}$ operators.

Scheduling workflows under constraints Davulcu [16] extends the straightforward application of $\mathcal{TR}$ to allow efficient reasoning about workflow executions with constraints. Straightforward execution of $\mathcal{TR}$ programs that include conjunctive constraints (of the form $program \wedge constraint$) on the execution is not efficient because the constraints have to be verified at every step of the execution; when a constraint violation is detected a new execution path must be tried. Determining whether a $\mathcal{CTR}$ program that includes conjunctive constraints is executable (whether it has some execution path that satisfies all constraints), is NP-complete. On the other hand, Horn-$\mathcal{CTR}$ without classical conjunction is efficiently implementable.

Davulcu describes how to compile certain constraints into the workflow specification during design-time. The result of the compilation step is a workflow definition in which these constraints are guaranteed to be satisfied and do not need to be checked anymore during runtime. Although the compilation step is computationally expensive it can be performed before execution of the workflow, during design-time; the run-time scheduling of the workflow can then be done efficiently. After compiling the conjunctive constraints into the workflow, the time complexity of scheduling is linear in the size of the original control flow graph.

Senkul *et al.* [38] describe how to integrate $\mathcal{CTR}$ with constraint logic programming. A workflow is specified as a set of $\mathcal{CTR}$ formulas and a set of constraints. The workflow consists of activities and resources, these resources have some cost and can be assigned to activities. Two kinds of constraints can be expressed:*cost constraints* that limit the cost of an assignment of resources to activities, and *control constraints* that limit the order of assignments of resources to activities. A schedule for a workflow is constructed using a $\mathcal{CTR}$ reasoner and a standard constraint solver.

5 Summary

We have discussed various formal frameworks for modelling workflows. Petri nets are widely advocated for modelling workflows, mainly because their state-based approach would allow natural modelling of certain (state-based) scenarios; other advantages would include their graphical nature, their formal definition, and the analysis techniques available. We have discussed some critiques in the literature on Petri net-based workflow modelling, mainly that they are not suitable for modelling reactive systems (which is what workflow management systems are), that they are not suitable for modelling certain scenarios (involving advanced synchronisation, multiple instances, and cancellation), and that they are not suitable for modelling and reasoning with global constraints (that range over a sequence of states).

Temporal logic has also been suggested for workflow modelling, being a general and powerful logic for modelling temporal situations. We have discussed several approaches that use temporal logic; the main critique on these approaches is that exactly because temporal logic is very general, verification techniques are not optimised for the specific situation of workflow modelling.

A number of approaches use Transaction Logic, we have discussed the logic and how one can model workflows with it. Several variants of Transaction Logic have been developed; these variants differ in expressive power and computational complexity of reasoning tasks. Transaction Logic comes with a reasoning system, in which one can reason about general statements and program and execute database evolutions. Workflows without conjunctive constraints are efficiently implementable in this logic programming environment. With conjunctive constraints the reasoning becomes hard to implement; some recent approaches show how one can reason efficiently even in the presence of these constraints, by taking the computational hurdle in a design-time preprocessing step.

We will use this survey in our future work, where we will try to develop a simple model that incorporates these frameworks. Such a model can then be used to integrate different modelling approaches.

References

[1] W. M. P. van der Aalst. *Timed coloured Petri nets and their application to logistics.* Ph.D. thesis, Eindhoven University of Technology, 1992.

[2] W. M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In S. Navathe and T. Wakayama, (eds.) *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96),*, pp. 179–201. 1996.

[3] W. M. P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[4] W. M. P. van der Aalst. Making work flow: On the application of petri nets to business process management. In J. Esparza and C. Lakos, (eds.) *23rd International Conference on Applications and Theory of Petri Nets*, vol. 2360 of *Lecture Notes in Computer Science*, pp. 1–22. Springer-Verlag, Berlin, 2002.

[5] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[6] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[7] N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10:131–158, 1998.

[8] G. Alonso, D. Agrawal, A. E. Abbadi, and C. Mohan. Functionality and limitations of current workflow management systems. *submitted to IEEE Expert*, 1997.

[9] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and enforcing intertask dependencies. In *Proceedings of the 19th VLDB Conference*. 1993.

[10] J. C. M. Baeten. A brief history of process algebra. Rapport CSR 04-02, TU Eindhoven, 2004.

[11] A. Bonner and M. Kifer. Results on reasoning about updates in transaction logic. In B. Freitag, H. Decker, M. Kifer, and A. Voronkov, (eds.) *Transactions and Change in Logic Databases*, vol. 1472 of *Lecture Notes in Computer Science*, pp. 166–196. Springer-Verlag, Berlin, 1998.

[12] A. J. Bonner. Workflow, transactions and datalog. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database System (PODS)*, pp. 294–305. 1999.

[13] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pp. 142–156. MIT Press, Bonn, Germany, 1996.

[14] A. J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, (eds.) *Logics for Databases and Information Systems*, chap. 5, pp. 117–166. Kluwer Academic Publishers, 1998.

[15] J. Chomicki and D. Toman. Temporal logic in information systems. In J. Chomicki and G. Saake, (eds.) *Logics for Databases and Information Systems*, chap. 3, pp. 31–70. Kluwer Academic Publishers, 1998.

[16] H. Davulcu. *A Game Logic for Workflows of Non-cooperative Services*. Ph.D. thesis, State University of New York at Stony Brook, 2002.

[17] H. Davulcu, M. Kifer, C. Ramakrishnan, and I. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pp. 25–33. 1998.

[18] C. A. Ellis and G. J. Nutt. Office information systems and computer science. *ACM Computing Surveys*, 12(1):27–60, 1980.

[19] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. Ph.D. thesis, University of Twente, 2002.

[20] R. Eshuis and J. Dehnert. Reactive petri nets for workflow modeling. In W. M. P. van der Aalst and E. Best, (eds.) *Application and Theory of Petri Nets 2003*, vol. 2679 of *Lecture Notes in Computer Science*, pp. 295–314. Springer-Verlag, Berlin, Berlin, 2003.

[21] R. Eshuis and R. Wieringa. Comparing petri nets and activity diagram variants for workflow modelling – a quest for reactive petri nets. In H. Ehrig, W. Reisig, and G. Rozenberg, (eds.) *Petri Net Technologies for Communication Based Systems*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2002.

[22] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proceedings of the 24rd International Conference on Software Engineering*, pp. 166–176. 2002.

[23] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.

[24] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. Int. Thomson Press, 1996.

[25] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

[26] M. Kifer. Transaction logic for the busy workflow professional, 1996. Available from ftp://ftp.cs.sunysb.edu/pub/techreports/kifer/tr-for-wf.ps.

[27] J. Klein. Advanced rule driven transaction management. In *Compcon Spring '91. Digest of Papers*, pp. 562–567. IEEE, 1991.

[28] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science Journal*, 83(1):97–130, 1991.

[29] S. Mukherjee, *et al.* Logic-based approaches to workflow modeling and verification. In J. Chomicki, R. van der Meyden, and G. Saake, (eds.) *Logics for Emerging Applications of Databases*, chap. 5, pp. 167–202. Springer-Verlag, Berlin, 2004.

[30] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[31] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[32] C. A. Petri. *Kommunikation mit Automaten*. Ph.D. thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

[33] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–67. 1977.

[34] W. Reisig and G. Rozenberg, (eds.) *Lectures on Petri Nets I: basic models*, vol. 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[35] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, (ed.) *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pp. 592 – 620. Addison Wesley Longman, 1995.

[36] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns. QUT Technical report FIT-TR-2004-01, Queensland University of Technology, 2004.

[37] K. Salimifard and M. Wright. Petri net-based modelling of workow systems: An overview. *European Journal of Operational Research*, 134(3):664–676, 2001.

[38] P. Senkul, M. Kifer, and I. H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 694–705. 2002.

[39] M. P. Singh. Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the International Workshop on Database Programming Languages*. 1995.

[40] M. P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proceedings of the 12th IEEE Intl. Conference on Data Engineering*, pp. 616–623. 1996.

[41] C. R. Zervos. *Coloured Petri Nets: their properties and applications*. Ph.D. thesis, University of Michigan, 1977.

[42] M. D. Zisman. *Representation, Specification and Automation of Office Procedures*. Ph.D. thesis, Wharton School, University of Pennsylvania, 1977.