A Formal Diagrammatic Approach to Timed Workflow Modelling

Hao Wang, Adrian Rutle and Wendy MacCaull*
Centre for Logic and Information
St. Francis Xavier University, Canada
Email: {hwang,arutle,wmaccaul}@stfx.ca

***Abstract*—A workflow model is an abstract representation of a real life workflow and consists of interconnected tasks depicting the desired executions of real life activities. Time information is an important aspect of many safety-critical workflows. This paper presents a new formal diagrammatic approach to timed workflow modelling involving principles from model-driven engineering. The approach extends the Diagram Predicate Framework, which is based on category theory and graph transformations, for the specification of workflow modelling formalisms. We develop a transition system to represent the dynamic semantics involving time in which transitions are described by specification transformations between instances. To model time, we use predicates for time *delay* and *duration* with transition rules for time advancement.**

***Index Terms*—Model-driven engineering; workflow modelling; real-time modelling; diagrammatic modelling; Diagram Predicate Framework;**

I. Introduction

Workflows coordinate activities performed by various participants, e.g., persons, robots or software components, in order to achieve a business goal. Workflows can serve various purposes ranging from documentation to reorganisation of businesses. By capturing the activities and the participants as well as the interdependencies in which activities are performed, one can obtain an overview of an organisation's business processes and rules. This overview, or workflow description, can be used for guiding, controlling, resource managing, automating, documenting or optimising business processes. Usually software systems are implemented for real life workflows in order to obtain some degree of control of the activities and to assist decision making. That is, the workflow software system must be consulted and will determine which activities should be performed – the so-called workflow *enactment*.

Both qualitative and quantitative time information play an important role in the specification of many safety-critical of workflows like health care. For example, "the patient receives a certain treatment after registration" is a statement with a *qualitative* notion of time. However, it is often necessary to include a *quantitative* notion of time in a workflow model, such as "the patient receives a treatment within an hour of registration". The quantitative time aspect will add to the complexity of a workflow model. A rigorous formalisation is required for the design, verification, and enactment of timed workflow models.

Ideally, there is a traceable correspondence between the real life workflow activities and the execution paths which the workflow software system allows. In this paper, we propose a model-driven engineering (MDE) approach to timed workflow modelling to support this correspondence. In MDE, workflow models are used as abstract representations of real life workflows. These models are used in refinement processes using model transformations and code-generation techniques to obtain executable workflow software systems. A workflow model consists of connected steps (called *tasks*) depicting real life activities. These models are used to drive the development of workflow software systems. Like many other development approaches, MDE employs workflow models to specify and understand real life workflows. In addition, MDE employs these models as the primary artefacts of the software development process. In this way, the relation between the result of the development process – the executable software – and the workflow models remains intact during software development, deployment, maintenance, etc. Following the MDE methodology, each instance of the workflow model represents one state of the workflow software system; each sequence of instance transitions corresponds to an execution path allowed by the workflow software system. In this way, workflow model designers can not only use verification, analysis and simulation techniques on workflow models, they also can conclude that the verification results are valid for the software system.

Our approach is based on the Diagram Predicate Framework (DPF) [1], [2], [3] which provides a formalisation of (meta)modelling and model transformations based on category theory and graph transformations [4]. DPF is an extension of the generalised sketches framework [5], [6]. Since workflow models are behavioural models, we extend the formal foundation of DPF to define semantics for behavioural models. In particular, we define the dynamic semantics of these models using a transition system where the states are instances and the transitions are described by specification transformations between instances. To model time, we introduce predicates for time *delay* and *duration* with transition rules for time advancement [7]. In this paper, we consider both continuous (real) and discrete time domains.

The remainder of the paper is structured as follows. Section II outlines DPF as the formal underpinning of our

*Hao Wang mainly contributed with formalisation of time aspects of workflow modelling; Adrian Rutle mainly contributed with the DPF-based formalisation of workflow modelling; Wendy MacCaull mainly contributed with expertise in workflow modelling in health services delivery.

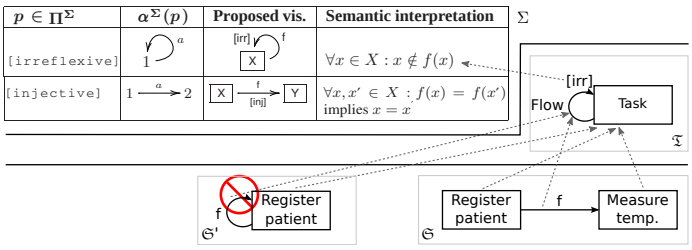


Figure 1: A signature Σ and specifications $\mathfrak{T}$, $\mathfrak{G}$ and $\mathfrak{G}'$, where only $\mathfrak{G}$ conforms to $\mathfrak{T}$

approach. Section III introduces our formal approach to timed workflow modelling. In Section IV, some related work is discussed. Finally, in Section V, some concluding remarks and ideas for future work are presented.

II. Diagram Predicate Framework

This section briefly reviews the basic concepts of DPF which we use for the formalisation of workflow modelling. In DPF, a model is represented by a *specification* $\mathfrak{G}$. A specification $\mathfrak{G} = (S, C^{\mathfrak{G}}{:}\Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{G}}$ which are specified by means of a *predicate signature* Σ. A predicate signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of *predicates* $p \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(p)$. An atomic constraint (p, δ) consists of a predicate $p \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(p) \to S$ from the arity of the predicate to the underlying graph S of a specification. A specification morphism $\phi : \mathfrak{G} \to \mathfrak{G}'$ is a graph homomorphism $\phi : S \to S'$ preserving constraints; i.e., if $(p, \delta) \in C^{\mathfrak{G}}$ then $(p, \delta; \phi) \in C^{\mathfrak{G}'}$. Fig. 1 shows a specification $\mathfrak{T}$ which is compliant with the requirements, "a task may be connected to other tasks by flows" and "a task can not be connected to itself". In $\mathfrak{T}$, these requirements are enforced by the underlying graph T consisting of the node $Task$ and the arrow $Flow$, and the atomic constraint $([\texttt{irreflexive}], \delta : (\ 1 \overset{a}{\circlearrowright}\ ) \to (\ \mathsf{Task} \overset{\mathsf{Flow}}{\circlearrowright}\ ))$.

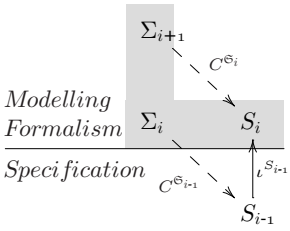
The semantics of graph nodes and arrows has to be chosen in a way which is suitable for the corresponding modelling environment. In structural data modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes, e.g., X, Y, as sets X, Y, and arrows, e.g., $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$, as partial multi-valued functions $f : X \to \wp(Y)$. For workflow modelling, we interpret nodes as singleton sets and arrows as functions (see Section III).

The semantics of predicates of the signature Σ (see Fig. 1) is described using the mathematical language of set theory. In an implementation, the semantics of a predicate is typically given by the code of a corresponding validator. A semantic interpretation $[\![..]\!]^{\Sigma}$ of Σ consists of a mapping that assigns to each predicate symbol $p \in \Pi^{\Sigma}$ a set $[\![p]\!]^{\Sigma}$ of graph homomorphisms $\iota : O \to \alpha^{\Sigma}(p)$, called valid instances of p, where O varies over all graphs. $[\![p]\!]^{\Sigma}$ is assumed to be closed under isomorphisms.

The semantics of a specification $\mathfrak{G} = (S, C^{\mathfrak{G}}{:}\Sigma)$ is given by the set of its instances. An instance of a specification $\mathfrak{G}$ is a pair (I, ι), where $\iota : I \to S$ is a graph homomorphism, such that for each atomic constraint $(p, \delta) \in C^{\mathfrak{G}}$ we have $\iota^* \in [\![p]\!]^{\Sigma}$, where the graph homomorphism $\iota^* : O^* \to \alpha^{\Sigma}(p)$ is given by the pullback $\alpha^{\Sigma}(p) \xleftarrow{\iota^*} O^* \xrightarrow{\delta^*} I$ of $\alpha^{\Sigma}(p) \xrightarrow{\delta} S \xleftarrow{\iota} I$.

In DPF, two kinds of conformance relations are distinguished: *typed by* and *conforms to*. A specification $\mathfrak{G}$ is typed by a graph T if there exists a graph homomorphism $\iota : S \to T$, called the *typing morphism*, between the underlying graph of the specification $\mathfrak{G}$ and T. A specification $\mathfrak{G}$ conforms to a specification $\mathfrak{T}$ if $\mathfrak{G}$ is typed by T and (S, ι) is an instance of $\mathfrak{T}$. Fig. 1 shows two specifications $\mathfrak{G}$ and $\mathfrak{G}'$, both typed by $\mathfrak{T}$, but only $\mathfrak{G}$ conforms to $\mathfrak{T}$, since in $\mathfrak{G}'$ the arrow f violates the atomic constraint $([\texttt{irreflexive}], \delta)$ in $C^{\mathfrak{T}}$.

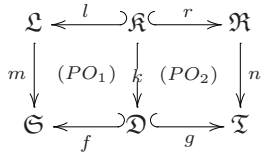
In this paper, we develop a language for timed workflow modelling. In MDE, each workflow model must conform to a metamodel defining the abstract syntax of the workflow's modelling language. A metamodel is actually just a model that has the role of being a metamodel w.r.t. models specified by the modelling language. In fact this extends to a metamodelling hierarchy to as many levels as necessary [8], [1].



In DPF, a modelling language is described as a modelling formalism $\mathcal{S}_i = (\Sigma_i, \mathfrak{S}_i, \Sigma_{i+1})$. The corresponding metamodel of the modelling language is represented by the specification $\mathfrak{S}_i$ which has its constraints formulated by predicates from the signature Σ_{i+1}. These constraints should be satisfied by all specifications, $\mathfrak{S}_{i-1} = (S_{i-1}, C^{\mathfrak{S}_{i-1}} : \Sigma_i)$, which are specified by $\mathcal{S}_i$. The specification $\mathfrak{S}_i$ defines the types and relations between types which are available in the language. The atomic constraining constructs which are available for the users of the modelling language are located in the signature Σ_i. These constructs may be used to add constraints to specifications specified by the modelling formalism. These constraints should be satisfied by instances of these specifications. The subscripts $i-1, i, i+1$ reflect that a modelling formalism may represent a modelling language at any level of a metamodelling hierarchy.

Later in this paper we will describe the dynamic semantics of workflow models by a transition system, which will be described by specification transformations. These *specification transformations* are based on *transformation rules* [4], [1]. A transformation rule $t = \mathfrak{L} \xleftarrow{l} \mathfrak{K} \xhookrightarrow{r} \mathfrak{R}$ consists of three specifications $\mathfrak{L}$, $\mathfrak{K}$ and $\mathfrak{R}$, and two inclusion specification morphisms l, r. $\mathfrak{L}$ and $\mathfrak{R}$ are the *left-hand side* (LHS) and *right-hand side* (RHS) of the transformation rule, respectively, while $\mathfrak{K}$ is their interface. $\mathfrak{L} \setminus l(\mathfrak{K})$ describes the part of a specification which is to be deleted, $\mathfrak{R} \setminus \mathfrak{K}$ describes the part to be added, and $\mathfrak{K}$ describes the overlap between $\mathfrak{L}$ and $\mathfrak{R}$.

A *direct application of a transformation rule*, $\mathfrak{S} \overset{<t,m>}{\Longrightarrow} \mathfrak{T}$, means finding a match m for $\mathfrak{L}$ in a source specification $\mathfrak{S}$ (i.e., a specification morphism $m : \mathfrak{L} \to \mathfrak{S}$) and replacing $\mathfrak{L}$ with $\mathfrak{R}$, leading to a target specification $\mathfrak{T}$. We use the double pushout approach (PO_1 and PO_2) to realise direct applications of transformation rules [4], [2].



We define a *specification transformation* $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$ as a sequence of applications of transformation rules starting with $\mathfrak{S}$. A *transition system* consists of all possible sequences of rule applications to a source specification $\mathfrak{S}$; i.e., it represents all specification transformations $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}'$, $\mathfrak{S} \overset{*}{\Longrightarrow} \mathfrak{S}''$, ... To achieve termination; i.e., the situation in which no more transformation rules are applicable, control structures such as *negative application conditions* (NACs), priorities, layering, etc. may be used. We control the application of transformation rules through (i) the use of *negative application conditions* (NACs) and (ii) requiring that each rule application results in a specification which conforms to a certain meta-specification.

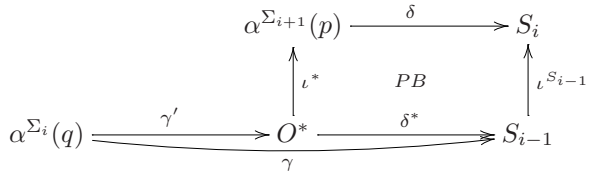
III. Workflow Modelling

In this section, we extend the modelling formalism introduced in Section II to deal with timed workflow models. This extension is necessary because structural data modelling differs from workflow modelling: while in the former we consider only two modelling levels, in the latter we need to consider three levels because the definition of a semantics for constructs of a workflow modelling formalism requires us to consider the instances of workflow models defined by the formalism in addition to the workflow models. To explain the need for three metamodelling levels, consider a metamodel where we have the general concept Task, a workflow model which consists of specific tasks, e.g., Measure temp., and an instance of the workflow model where we have different execution states for each specific task, e.g., one may be running, another may have finished its execution, etc. In this metamodelling hierarchy, in order to define the semantics of the concept Task, we need to consider its execution states; i.e., the instances of its instances. For example, in order to define meaningful behaviours for tasks, we need to express the fact that each task instance has to be running before it can be finished, etc. Using only two metalevels would force us to define type-instance relations in the metamodel. This leads to a mixture of domain concepts with language concepts in the same modelling level [9], [10]. To address this, we introduce the notion of one signature being *semantically dependent* on another signature, and require what we call *strong conformance* between workflow models and their instances. Examples of semantic dependency and strong conformance are shown in the sequel.

Definition 1 (Semantic Dependence): Given two signatures $\Sigma_{i+1} = (\Pi^{\Sigma_{i+1}}, \alpha^{\Sigma_{i+1}})$ and $\Sigma_i = (\Pi^{\Sigma_i}, \alpha^{\Sigma_i})$, we say the semantic interpretation of Σ_{i+1} is dependent on Σ_i, and write $[\![..]\!]^{\Sigma_{i+1}}_{\Sigma_i}$, iff for each predicate $p \in \Pi^{\Sigma_{i+1}}$, $[\![..]\!]^{\Sigma_{i+1}}_{\Sigma_i}$ assigns to p a set $[\![p]\!]^{\Sigma_{i+1}}_{\Sigma_i}$ of specifications $\{\mathfrak{O} = (O, C^{\mathfrak{O}} : \Sigma_i)\}$ where $\iota : O \to \alpha^{\Sigma_{i+1}}(p)$ is in $[\![p]\!]^{\Sigma_{i+1}}$.

Definition 2 (Strong Conformance): Given a modelling formalism $\mathcal{S}_i = (\Sigma_i, \mathfrak{S}_i, \Sigma_{i+1})$ where Σ_{i+1} is semantically dependent on Σ_i, a specification strongly conformant to $\mathfrak{S}_i = (S_i, C^{\mathfrak{S}_i} : \Sigma_{i+1})$ is a specification $\mathfrak{S}_{i-1} = (S_{i-1}, C^{\mathfrak{S}_{i-1}} : \Sigma_i)$ together with a bijective graph homomorphism $\iota^{S_{i-1}} : S_{i-1} \to S_i$ such that for each atomic constraint $(p, \delta) \in C^{\mathfrak{S}_i}$, with $p \in \Pi^{\Sigma_{i+1}}$, we have $\mathfrak{O} = (O^*, C^{\mathfrak{O}} : \Sigma_i) \in [\![p]\!]^{\Sigma_{i+1}}_{\Sigma_i}$ where

- $\iota^* : O^* \to \alpha^{\Sigma_{i+1}}(p)$ and $\delta^* : O^* \to S_{i-1}$ are given by the pullback in the following:



- $C^{\mathfrak{O}} = \{(q, \gamma') | \gamma'; \delta^* = \gamma \ \forall (q, \delta) \in C^{\mathfrak{S}_{i-1}}\}$, $q \in \Pi^{\Sigma_i}$.

From the above definitions we see that the set of valid instances of the predicates of a signature Σ_{i+1} dependent on a signature Σ_i consists of specifications $(O, C^{\mathfrak{O}} : \Sigma_i)$ rather than graphs, and for a specification $\mathfrak{S}_{i-1}$ to be strongly conformant to a specification $\mathfrak{S}_i$, we require both that the underlying graph S_{i-1} is an instance of $\mathfrak{S}_i$, and that $\mathfrak{S}_{i-1}$ satisfies $C^{\mathfrak{S}_i}$.

Time is represented using clocks: a global clock which keeps track of time for the whole system and a local clock for each task to keep track of time for that task's executions. In our formalisms, time advances at the same pace in all clocks. For discrete (resp. continuous) time modelling, time advances by a non-negative integer (resp. non-negative real number). It is a standard practice [11], [12] to define time bounds for actions, which in our case are workflow tasks. We include for each task a fixed time constraint, called *delay*, to indicate how long the task should wait before it starts running and a time constraint, called *duration*, to indicate an upperbound for which the task must be finished.

A. Modelling Formalisms for Timed Workflow Modelling

Workflow modelling languages provide constructs to define tasks and their splitting and merging flows. In this section we define a modelling formalism $\mathcal{S}_2 = (\Sigma_2, \mathfrak{S}_2, \Sigma_3)$ used for the specification of workflow specifications, with a metamodel defining the types Task and Flow, and a signature containing predicates for dependency, splitting and merging (see Fig. 2). The specification $\mathfrak{S}_2$ and the signature Σ_3 are the same as $\mathfrak{T}$ and Σ in Fig. 1, respectively. The signature Σ_2 semantically depends on the signature Σ_1. Note that Fig. 2 shows only parts of Σ_2, Σ_1, $\mathfrak{S}_1$ and the sample specification $\mathfrak{S}_0$; the details are shown in Tables I and II and Fig. 3, respectively.

Table I shows signature Σ_2 with some predicates which are useful for workflow modelling. The first predicate `[time(`de, du`)]` is used to indicate time delay and duration; i.e., an (enabled) task has to wait de time units before it starts running, and runs for at most du time units. The default values of de and du are 0. The second predicate `[condition,`c`]`is used to define conditions on flows. A

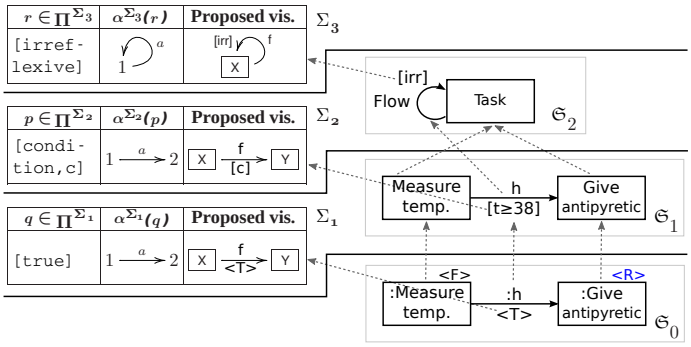


Figure 2: Modelling formalisms $\mathcal{S}_2 = (\Sigma_2, \mathfrak{S}_2, \Sigma_3)$ and $\mathcal{S}_1 = (\Sigma_1, \mathfrak{S}_1, \Sigma_2)$, a specification $\mathfrak{S}_0$ strongly conformant to $\mathfrak{S}_1$

condition c is a proposition which may evaluate to true or false, or it may be unevaluated. Note, the empty condition; i.e., a flow without `[condition,`c`]`on it, is a tautology.

The other predicates have spans or sinks of two arrows as arity, and are used to define relations between different flows. We may define these predicates with arities being spans or sinks of any finite number of arrows, however two arrows will serve the purpose of explaining the modelling formalism. The predicate `[exactly-one]` is used to indicate that exactly one of the flows must be followed, the predicate `[all-premises]` is used to indicate that both flows `f` and `g` must be followed. To save space, we omit from Table I the other usual splitting and merging predicates used in workflow modelling, such as `[all-together]`, `[at-least-one]`, `[exactly-one-premise]` and `[at-least-one-premise]`.

The semantic interpretation of the predicates $p \in \Pi^{\Sigma_2}$ in Table I is formulated by enumerating the valid instances in $[\![p]\!]^{\Sigma_2}_{\Sigma_1}$. We use x:X<D\|E\|R> to denote situations where we have: x:X<D>, x:X<E> or x:X<R>, and we use x:X<LC≥0> to denote situations where we have <LC=t> for a non-negative number t. The abbreviations <D>, <E>, <R>, <F>, <T>, <⊥>, <?>, [de>, <du], <LC=t>, <GC=t>, and <FT=t> stand for the predicates `[disabled]`, `[enabled]`,`[running]`,`[finished]`,`[true]`,`[false]`, `[not-evaluated]`,`[delay,`de`]`,`[duration,`du`]`,`[localclock,`t`]`,`[globalclock,`t`]` and `[finished-time,`t`]`, respectively. These predicates come from the signature Σ_1 in Table II, and give some intuition regarding the semantic dependence of Σ_2 on Σ_1.

Fig. 3 shows details of the specification $\mathfrak{S}_1$ which is used as the metamodel of $\mathcal{S}_1$ in Fig. 2. The specification $\mathfrak{S}_1$ is compliant with the following requirements:

- R1: Patient has to be registered first before any other activities are performed; this task should start running immediately and be finished in 20 time units.
- R2: After registration, measure temperature and assess pain; both these tasks should start running after being enabled; the former should be finished in 5 time units and the latter in 7 time units.
- R3: If temperature is higher than 38 degrees give antipyretic medication; this task should start running after being enabled and be finished in 5 time units.
- R4: If pain level is between 2 and 3, give a weak opioid; this task should be delayed for 5 time units and be finished in 5 time units once started.
- R5: If pain level is higher than 3, give a strong opioid; this task should be delayed for 5 time units and be finished in 5 time units once started.

We describe how some of the above requirements are specified in $\mathfrak{S}_1$. R1 is specified by the task Register patient as a task with no incoming flows, and the constraint $(\texttt{[time(}0,20\texttt{)]}, \delta_0)$. R2 is specified by the tasks Measure temp. and Assess pain, the two flows f and g, and the constraints $(\texttt{[time(}0,5\texttt{)]}, \delta_1)$, $(\texttt{[time(}5,10\texttt{)]}, \delta_2)$ and $(\texttt{[all-together]}, \delta_3)$. R3 is specified by the task Give antipyretic, the flow h, and the constraints $(\texttt{[time(}0,5\texttt{)]}, \delta_4)$ and $(\texttt{[condition,}t \geq 38\texttt{]}, \delta_5)$.

B. Semantics of Timed Workflow Models

The *static semantics* of a workflow specification $\mathfrak{S}_1$ is given by the set of its instances; i.e., the set of all specifications $\mathfrak{S}_0$ which strongly conform to $\mathfrak{S}_1$. These instances represent states of the workflow software system developed from the workflow specification. In this section we define a modelling formalism $\mathcal{S}_1 = (\Sigma_1, \mathfrak{S}_1, \Sigma_2)$ used for the specification of instances of the workflow specification $\mathfrak{S}_1$ (see Figs. 2 and 3). We use $\mathfrak{S}_1$ as the metamodel of $\mathcal{S}_1$, however, the explanation can be generalised to any workflow specification defined by $\mathcal{S}_2$. While specifications defined by $\mathcal{S}_2$ only need to be conformant to $\mathfrak{S}_2$, specifications defined by $\mathcal{S}_1$ must be *strongly* conformant to $\mathfrak{S}_1$ since $\mathcal{S}_1$ is used to specify states $\mathfrak{S}_0$. Checking whether a specification $\mathfrak{S}_0$ represents a state of $\mathfrak{S}_1$ only requires checking whether it is strongly conformant to $\mathfrak{S}_1$; it is not necessary to consider the dynamic semantics, e.g., which other instances come before and after $\mathfrak{S}_0$.

At any state of the workflow software system, task instances, e.g., :Register patient in $\mathfrak{S}_0$ in Fig. 3, are either *disabled*, *enabled*, *running* or *finished*, according to the constraints specified in the workflow specification $\mathfrak{S}_1$. We use the signature Σ_1 shown in Table II to annotate task instances in $\mathfrak{S}_0$ accordingly. Moreover, the condition on a flow may be waiting for evaluation, or be evaluated to either true or false. The signature Σ_1 also includes predicates to annotate flow instances in $\mathfrak{S}_0$ accordingly; i.e., as not-evaluated, true or false. Furthermore, a task instance may have values for its local clock and the global clock, the value of the global clock when the task is finished, and two other values for its delay and duration which are inherited from the values in `[time(`de, du`)]`. The signature Σ_1 also includes predicates `[localclock,`t`]`, `[globalclock,`t`]`,`[finished-time,`t`]`,`[delay,`de`]` and `[duration,`du`]` to annotate task instances accordingly. The purpose of repeating the values de, du will be clear later when we use them in the transition system. The signature Σ_1 has the same structure as Σ_2, but has no semantic counterpart since for this modelling environment instances of $\mathfrak{S}_0$ do not have any practical meaning.

Table I: A sample signature $\Sigma_2 = (\Pi^{\Sigma_2}, \alpha^{\Sigma_2})$ used for workflow modelling

p	$\alpha^{\Sigma_2}(p)$	Proposed vis.	Semantic interpretation
[time-(de, du)]	1	$[de,du]$ X	<D> <LC=-1> x:X; <E> <LC≥0> x:X; <R> <LC≤du> x:X <GC=t>; <F> <LC=0> x:X <GC=t><FT=t>
[condition, c]	1 —f→ 2	X —f [c]→ Y	<D|E|R> x:X —:f <?>→ y:Y <D>; <F> x:X —:f <T>→ y:Y <E|R|F>; <F> x:X —:f <⊥>→ y:Y <D>
[exactly-one]	1 —f→ 2, 1 —g→ 3	X —f [xor]→ Y, X —g→ Z	<D|E|R> x:X —:f <?>→ y:Y <E|R|F>, —:g <?>→ z:Z <D>; <F> x:X —:f <⊥>→ y:Y <D>, —:g <⊥>→ z:Z <D>; <F> x:X —:f <T>→ y:Y <E|R|F>, —:g <⊥>→ z:Z <D>; <F> x:X —:f <T>→ y:Y <D>, —:g <T>→ z:Z <D>
[all-premises]	1 —f→ 2, 3 —g→ 2	X —f [and']→ Y, Z —g→ Y	<D|E|R> x:X —:f <?>→ y:Y <D>, <D|E|R> z:Z —:g <?>→ y:Y; <F> x:X —:f <⊥>→ y:Y <D>, <F> z:Z —:g <⊥>→ y:Y; <F> x:X —:f <T>→ y:Y <D>, <F> z:Z —:g <⊥>→ y:Y; <F> x:X —:f <T>→ y:Y <E|R|F>, <F> z:Z —:g <T>→ y:Y

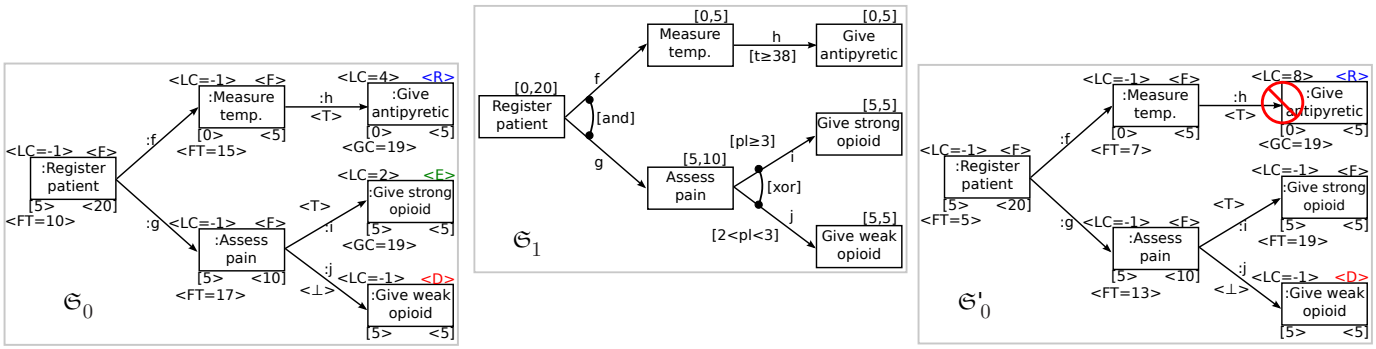


Figure 3: The specification $\mathfrak{S}_1$ and two sample specifications $\mathfrak{S}_0$ and $\mathfrak{S}'_0$, where only $\mathfrak{S}_0$ is strongly conformant to $\mathfrak{S}_1$

Fig. 3 shows a specification $\mathfrak{S}_0$ which is specified by $\mathcal{S}_1$ and is strongly conformant to $\mathfrak{S}_1$. For the sake of illustration, the figure also shows a specification $\mathfrak{S}'_0$ which is not strongly conformant to $\mathfrak{S}_1$. In $\mathfrak{S}'_0$, the constraint $(\texttt{[time(}0,5\texttt{)]}, \delta_4)$ on the task Give antipyretic is violated since the local clock of the task instance :Give antipyretic has the value 8, indicated by `[localclock,8]`, while the duration time has the value 5, indicated by `[duration,5]`. In a real life scenario this would correspond to not prescribing medication within the defined time limit. Note that both $\mathfrak{S}_0$ and $\mathfrak{S}'_0$ are conformant to $\mathfrak{S}_1$, however, only $\mathfrak{S}_0$ is strongly conformant to $\mathfrak{S}_1$.

The *dynamic semantics* of a workflow specification $\mathfrak{S}_1$ is defined by a transition system, which is described by states and transitions between states. In a workflow software system, as the workflow progresses, the execution states of the tasks are changed according to certain rules. For example, a task in the disabled state may either remain disabled or become enabled; a task in the enabled state may either remain enabled or change to running, etc. Correspondingly, for the specification $\mathfrak{S}_1$, a task instance (in a specification which strongly conforms to $\mathfrak{S}_1$) annotated with the predicate `[disabled]` may either remain annotated with `[disabled]` or become annotated with `[enabled]`; a task instance which is annotated with the predicate `[enabled]` may either remain annotated with `[enabled]` or become annotated with `[running]`, etc.

We call a task instance *active* if it is annotated with `[enabled]` or `[running]`, and *inactive* if annotated with `[disabled]` or `[finished]`. We need the local clock to keep track of time only when the task instance is active, otherwise we deactivate the local clock using the value -1; i.e., the predicate `[localclock,`-1`]` is assumed on each task instance which is annotated with `[disabled]` or `[finished]`. We increment the global clock as follows: at the moment a disabled task instance changes its annotation to `[enabled]`, the current value of the global clock will be obtained from its preceding task instance. The value on an

Table II: A signature Σ_1 used for annotation of instances

$q \in \Pi^{\Sigma_1}$	$\alpha^{\Sigma_1}(q)$	Proposed visual.
[enabled]	1	X <E>
[disabled]	1	X <D>
[running]	1	X <R>
[finished]	1	X <F>
[localclock,t]	1	<LC=t> X
[globalclock,t]	1	<GC=t> X
[finished-time,t]	1	<FT=t> X
[delay,de]	1	[de> X
[duration,du]	1	X <du]
[true]	1 —f→ 2	X —f <T>→ Y
[false]	1 —f→ 2	X —f <⊥>→ Y
[not-evaluated]	1 —f→ 2	X —f <?>→ Y

inactive task will not increment. The time t at which a task instance is finished is recorded using [finished-time,t] predicate. The predicate [finished-time, -1] is assumed on each task instance which is not finished yet. The value of the global clock is obtained by checking the [globalclock,t] annotation on any of the active tasks. In Fig. 3, we only show the global clock of the active tasks.

We distinguish between *time advancing transitions* and *task transitions*. A time advancing transition will increase all local clocks and the global clock in active task instances by the same number. The other transitions are task transitions. We comment that all task transitions are deemed as instantaneous w.r.t. time. To avoid *Zeno* behaviours where time does not advance [7], we assume that at any time instant, i.e., between any immediate pair of time advancing transitions, only a finite number of task transitions occur. For continuous time modelling, we assume a lowerbound for any time increase: i.e., we disallow time advances by arbitrarily small amounts.

Annotation changes in transitions must respect the constraints in $\mathfrak{S}_1$ in order to continue being a specification which is strongly conformant to $\mathfrak{S}_1$. Further, a transition from one state to another must include at least one annotation change of one task instance, and cannot include two consecutive annotation changes of the same task instance. For a given workflow specification $\mathfrak{S}_1$ and two specifications $\mathfrak{S}_0, \mathfrak{S}'_0$ both strongly conformant to $\mathfrak{S}_1$, a transition is defined as a direct application $\mathfrak{S}_0 \overset{<t,m>}{\Longrightarrow} \mathfrak{S}'_0$ of a transformation rule t.

By defining the notions of states and the transitions between states we obtain a transition system which we can use to provide the dynamic semantics for workflow specifications $\mathfrak{S}_1$ defined by $\mathcal{S}_2$. The transition system is described by a transformation system consisting of the transformation rules shown in Table III. We will briefly explain the general pattern of, and relations between, these rules. Rules t_1 and t_2 are used to change the annotation from [not-evaluated] to [true] or [false] on flow instances a. The rule t_1 also changes the annotation of task instance y to [enabled], resets the local clock of y to 0 by changing the value t of the annotation from [localclock,-1] to [localclock,0] and "copies" the value of the global clock from x. This resetting of the local clock and copying of the global clock occur in all other rules (e.g. t_7 and t_{17}) which change the annotation of a task from [disabled] to [enabled]. Since the predicate [localclock,-1] is assumed on each task instance which is annotated with [disabled] or [finished] the LHS of t_1 and t_2 are equal, leading to the fact that at any situation when a match of these rules is found, the system will proceed nondeterministically to either of the two states indicated by the RHSs of the rules. That is, the annotation [not-evaluated] on the flow instance a will be changed to either [true] or [false] and the task instance y may remain annotated with [disabled] or become annotated with [enabled].

Rules t_3 and t_4 are for changing annotations of task instances from [enabled] to [running] and from [running] to [finished], respectively, respecting the delay and duration times of x. These delay and duration times are specified by the predicate [time(de, du)] on tasks in $\mathfrak{S}_1$, and they are repeated in specifications $\mathfrak{S}_0$ strongly conformant to $\mathfrak{S}_1$ by the predicates [delay,de] and [duration,du]. The rule t_3 is applicable only if the local clock of x is equal to the delay value; the rule t_4 is applicable only if the local clock of x is smaller than or equal to the duration value; e.g., if $du = 4$, then t_4 is applicable if x is annotated with [localclock,3.1], [localclock,4], etc. We remark that the delay time of a task is realised by the predicate [localclock,$t = de$] on x – i.e., the local clock's value must be equal to the delay value in order to change the annotation of the task instance x from [enabled] to [running]. Moreover, the duration time of a task is realised by the predicate [localclock,$t \leq du$] on x – i.e., the local clock's value must be less than or equal to the duration value in order to change the annotation of the task instance x from [running] to [finished]. Note that a time advancing transition may not increment the local clock of x to a value greater than the duration value, making the rule t_4 not applicable, because such a transition would result in a specification which violates the constraint specified by the predicate [time(de, du)].

The rule t_5 is the only time advancing rule we have in the transition system. t_5 is a transformation rule with *variable patterns* indicated by the dotted rectangle [13]; i.e., the rule will apply to all task instances annotated with [enabled] or [running] simultaneously. In this way, all local clocks and the global clock in active task instances will increment by the same number. We comment that we do not require that the transition system alternates the application of task and time advancing transitions.

The rules t_6 to t_{10} are used to describe the transitions of spans of flow instances, and the rest of the rules are used to describe the transitions of sinks of flow instances. Similar to t_1 and t_2, the LHS of t_6 to t_{10} are equal, hence each of the annotations [not-evaluated] on the flow instances a and b may be changed to [true] or [false] and each of the task instances y and z may remain annotated with [disabled] or become annotated with [enabled]. Note that not all combinations are considered in these rules since some of the combinations would not make sense – i.e., would not yield a specification which is strongly conformant to $\mathfrak{S}_1$ – in our transition system. To save space we omit some of the rules in Table III. The rest of the rules are defined and explained similarly; however, a difference can be seen in the LHS of t_{16} and t_{17} compared to the other rules. In these two rules, one may proceed and evaluate the condition on one of the flow instances a although the task instance z is not yet finished and thus the condition on the flow instance b is not yet evaluated. This behaviour is necessary for flows which are constrained by the [at-least-one-premise] predicate.

In Table III the LHS of all rules from t_6 to t_{17} are NACs for t_1 and t_2. Informally, this forbids changing the annotation on a flow instance or a task instance if they are part of a

bigger structure. This is necessary since in a bigger structure there may be dependencies (imposed by some predicates from Table I on $\mathfrak{S}_1$) between flow instances.

C. Analysis

With the dynamic semantics defined, we can outline some temporal properties that we require workflow specifications to satisfy. Alpern and Schinerder [14] proved that every property can be expressed as the conjunction of a safety property and a liveness one. Informally, a safety property is one which states that something (bad) will *not* happen and a liveness property is one that states that something (good) *must* happen [15]. However, Henzinger [16] stated that this standard classification of properties does not fit real time properties and proposed the notions of safety and liveness *relative* to the progress of time. For example, *response*, i.e., something (good) will happen eventually, is a typical liveness property, while *bounded response*, i.e., something will happen within a certain amount of time, is a safety property relative to the progress of time. In this paper, we are interested in the structural properties of *reachability* and *termination* of a workflow, and the *timing delay* and *bounded response time* of task instances.

First we define the *start* state. For our workflow specification $\mathfrak{S}_1$, the start state $\mathfrak{S}_0^s$ is a specification which is strongly conformant to $\mathfrak{S}_1$ such that (i) the transition system can generate all other specifications starting from $\mathfrak{S}_0^s$, which are strongly conformant to $\mathfrak{S}_1$, and, (ii) all its task instances with no incoming flows are annotated with `[enabled]` and all other task instances are annotated with `[disabled]`.

We define reachability property of a workflow as a safety property by requiring that it has no "dead" tasks; i.e., tasks which are disabled in all execution scenarios. We define the termination property as a liveness property by requiring that all states can always transit to a terminating state. For two tasks X and Y, timing delay $td(\mathsf{X},\mathsf{Y})$ and bounded response $br(\mathsf{X},\mathsf{Y})$ indicate the earliest time and the latest time at which Y can be finished after X is finished, respectively. In practice, users of the workflow can specify a timing delay and a bounded response time between any pair of tasks. Note that both properties are safety properties. In summary, these properties for our workflow specification $\mathfrak{S}_1$ are listed as follows:

1) Reachability: for each task X, the transition system results in at least one specification transformation $\mathfrak{S}_0^s \overset{*}{\Longrightarrow} \mathfrak{S}_0^{\mathsf{X}e}$, where $\mathfrak{S}_0^{\mathsf{X}e}$ denotes a state in which a task instance x:X is annotated with `[enabled]`.
2) Termination: for any state $\mathfrak{S}_0$ generated by the transition system starting from the start state $\mathfrak{S}_0^s$, there exist a specification transformation $\mathfrak{S}_0 \overset{*}{\Longrightarrow} \mathfrak{S}_0^e$ such that no more rules are applicable to $\mathfrak{S}_0^e$.
3) Timing delay: let $\mathsf{x}_{\langle\mathsf{FT}\rangle}$ and $\mathsf{y}_{\langle\mathsf{FT}\rangle}$ be the values of the finished time annotations of the task instances x:X and y:Y, respectively. The timing delay $td(\mathsf{X},\mathsf{Y})$ is satisfied iff after the task instance x:X is annotated with `[finished]` either (i) the task instance y:Y is annotated with `[finished]` and $\mathsf{y}_{\langle\mathsf{FT}\rangle} - \mathsf{x}_{\langle\mathsf{FT}\rangle} \geq td(\mathsf{X},\mathsf{Y})$; or (ii) the task instance y:Y is annotated with `[disabled]`, `[enabled]` or `[running]` and $globalclock \geq \mathsf{x}_{\langle\mathsf{FT}\rangle} + td(\mathsf{X},\mathsf{Y})$.
4) Bounded response: the property $br(\mathsf{X},\mathsf{Y})$ is satisfied iff after the task instance x:X is annotated with `[finished]`, y:Y is annotated with `[finished]` and $\mathsf{y}_{\langle\mathsf{FT}\rangle} - \mathsf{x}_{\langle\mathsf{FT}\rangle} \leq br(\mathsf{X},\mathsf{Y})$.

We could use model checking to prove that each workflow specification $\mathfrak{S}_1$ satisfies the above-mentioned properties by exploring the transition system.

IV. Related Work

Van der Aalst et al. [17] present a declarative modelling framework called *Declare*. The approach is based on constraints; i.e., anything is possible as long as it is not explicitly forbidden by some constraint. In this way, the models implicitly specify the execution procedure by means of constraints. Our approach is similar in the sense that we also use constraints to declaratively specify workflow models. Moreover, we define formal semantics for the constraints and follow a metamodelling approach which enables us to explain the conformance relation between models and metamodels.

Ghamarian et al. [18] employ a graph transformations-based framework, GROOVE, to provide semantics to behavioural models. Our approach extends graph transformations by using constraint-aware model transformations – i.e., considering diagrammatic constraints in transformation rule definitions and applications – which facilitate the definition of more fine grained rules and better control of their applications [**?**].

Brüning et al. [19] present a strict metamodelling approach to workflow modelling, which makes it possible to easily express semantics of sophisticated transition relationships between activities using UML class diagrams and OCL constraints. Due to shortcomings of UML and OCL w.r.t. constraint evaluation combined with multi-level metamodelling, this approach flattens the three levels – metamodel, model and instance – into two levels. OCL constraints are defined at the metamodel level and they are forced at the model/instance level. Our approach allows multi-level metamodelling with a unification of structural and OCL constraints in one formalism, and clearly distinguishes between the different levels of the metamodelling hierarchy.

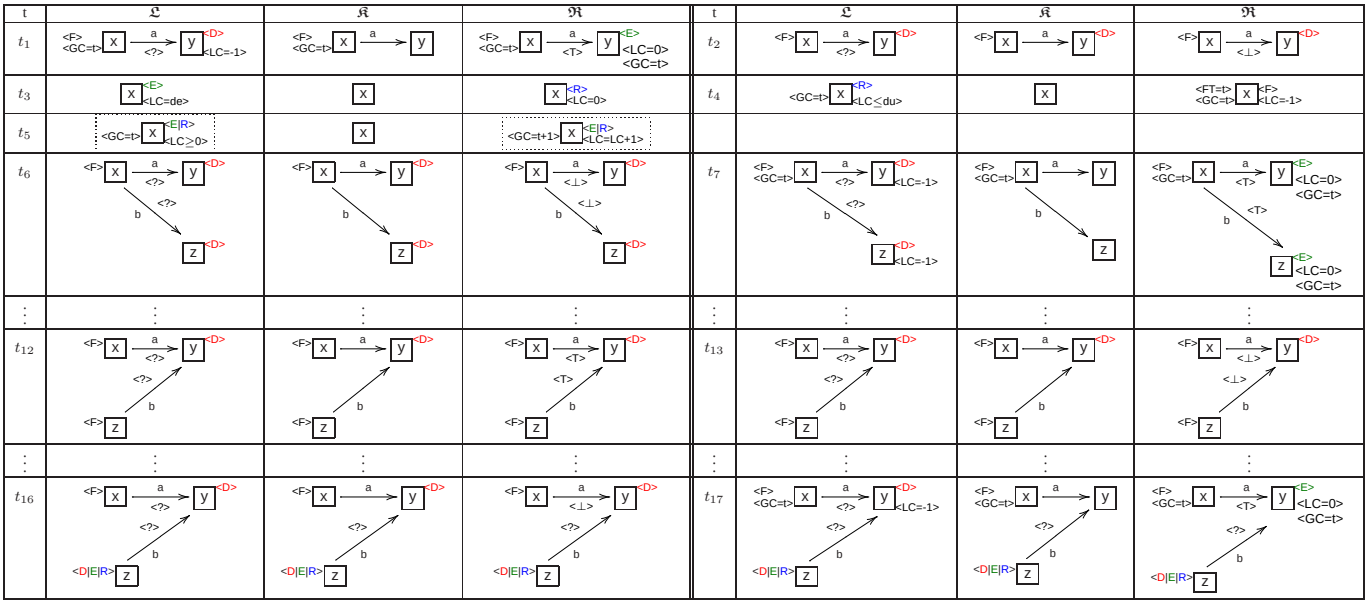
In the Real-time Tool Suite of *Fujaba* [20], *Real-time Statechart* is proposed based on the UML state machine and timed automata. Properties are specified in a real-time OCL variant, RT-OCL. The models and properties are then translated to Uppaal [21] for verification.

Mashiyat et al. [22] present the graphical language Timed Compensable Workflow Modelling Language (CWML_T), which is founded on time Petri nets, to specify workflow models with time delays and durations. The approach does not follow MDE, which may lead to impaired links between the workflow model and the workflow software system generated.

V. Conclusion and Future Work

This paper outlines a formal approach to timed workflow modelling based on the DPF formalism. The approach com-

Table III: The transformation rules $t = \mathfrak{L} \hookleftarrow \mathfrak{K} \hookrightarrow \mathfrak{R}$ of our transition system



bines MDE-methodologies and formal modelling principles for timed workflow modelling. In particular, time representation is included in the two modelling formalisms we define in this paper, one for the specification of workflow models and the other for the specification of their instances. In addition, we have extended DPF with notions to define the dynamic semantics of models, considering continuous/discrete time, described by specification transformations between instances of workflow models.

In future work, we will investigate analysis techniques for the transitions between instances of workflow models, such as model checking transformation rules in order to ensure that a set of transformation rules will not produce deadlocks or livelocks. We will use model checking also to prove that a workflow specification satisfies the above mentioned properties by exploring the transition system.

Acknowledgments. This research is sponsored by Natural Sciences and Engineering Research Council of Canada and by the Atlantic Canada Opportunities Agency.

References

[1] A. Rutle, "Diagram Predicate Framework: A Formal Approach to MDE," Ph.D. dissertation, Department of Informatics, University of Bergen, Norway, 2010.

[2] A. Rossini, "Diagram Predicate Framework meets Model Versioning and Deep Metamodelling," Ph.D. dissertation, Department of Informatics, University of Bergen, Norway, 2011.

[3] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter, "A formal approach to the specification and transformation of constraints in MDE," *JLAP*, vol. 81/4, pp. 422–457, 2012.

[4] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.

[5] Z. Diskin, B. Kadish, F. Piessens, and M. Johnson, "Universal Arrow Foundations for Visual Modeling," in *Diagrams 2000*, ser. LNCS, M. Anderson, P. Cheng, and V. Haarslev, Eds., vol. 1889. Springer, 2000, pp. 345–360.

[6] Z. Diskin, *Encyclopedia of Database Technologies and Applications*. Information Science Reference, 2005, ch. Mathematics of Generic Specifications for Model Management I and II, pp. 351–366.

[7] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, "Modeling time in computing: A taxonomy and a comparative survey," *ACM Computing Surveys*, vol. 42, pp. 6:1–6:59, March 2010.

[8] T. Kühne, "Matters of (meta-)modeling," *SoSyM*, vol. 5, no. 4, pp. 369–385, 2006.

[9] C. Gonzalez-Perez and B. Henderson-Sellers, *Metamodelling for Software Engineering*. Wiley, 2008.

[10] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," *ACM Trans. on Modeling and Computer Simulation*, vol. 12, no. 4, pp. 290–321, 2002.

[11] L. Lamport, "Real-Time Model Checking Is Really Simple," in *CHARME 2005: 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, ser. LNCS, vol. 3725. Springer, 2005, pp. 162–175.

[12] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[13] R. Grønmo, "Using Concrete Syntax in Graph-based Model Transformations," Ph.D. dissertation, Department of Informatics, University of Oslo, Norway, February 2010.

[14] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, 1985.

[15] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[16] T. A. Henzinger, "Sooner is safer than later," *Inf. Process. Lett.*, vol. 43, no. 3, pp. 135–141, 1992.

[17] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D*, vol. 23, no. 2, pp. 99–113, 2009.

[18] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova, "Modelling and analysis using GROOVE," *STTT*, pp. 1–26, 2011.

[19] J. Brüning, M. Gogolla, and P. Forbrig, "Modeling and Formally Checking Workflow Properties Using UML and OCL," in *BIR 2010*, ser. LNBIP, vol. 64. Springer, 2010, pp. 130–145.

[20] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer, "Fujaba4eclipse real-time tool suite," in *MBEERTS 2010: Model-Based Engineering of Embedded Real-Time Systems*, ser. LNCS. Springer, 2010, vol. 6100, pp. 309–315.

[21] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL a Tool Suite for Automatic Verification of Real Time Systems," in *Workshop on Verification and Control of Hybrid Systems III*, ser. LNCS, vol. 1066. Springer, 1995, pp. 232–243.

[22] A. S. Mashiyat, F. Rabbi, and W. MacCaull, "Modeling and Verifying Timed Compensable Workflows and an Application to Health Care," in *FMICS 2011*, ser. LNCS, vol. 6959. Springer, 2011, pp. 244–259.