Concurrency & Asynchrony in Declarative Workflows

Søren Debois[1], Thomas Hildebrandt[1], and Tijs Slaats[1,2] ⋆

[1] IT University of Copenhagen, {debois, hilde, tslaats}@itu.dk
[2] Exformatics A/S, 2100 Copenhagen, Denmark

Abstract. *Declarative* or *constraint-based* business process and workflow notations, in particular DECLARE and Dynamic Condition Response (DCR) graphs, have received increasing interest in the last decade as possible means of addressing the challenge of supporting at the same time flexibility in execution, adaptability and compliance. However, the definition of *concurrent* semantics, which is a necessary foundation for asynchronously executing distributed processes, is not obvious for formalisms such as DECLARE and DCR Graphs. This is in stark contrast to the very successful Petri-net–based process languages, which have an inherent notion of concurrency. In this paper, we propose a notion of concurrency for declarative process models, formulated in the context of DCR graphs, and exploiting the so-called "true concurrency" semantics of Labelled Asynchronous Transition Systems. We demonstrate how this semantic underpinning of concurrency in DCR Graphs admits asynchronous execution of declarative workflows both conceptually and by reporting on a prototype implementation of a distributed declarative workflow engine. Both the theoretical development and the implementation is supported by an extended example; moreover, the theoretical development has been verified correct in the Isabelle-HOL interactive theorem prover.

1 Introduction

The last decade has witnessed a massive revival of business process and workflow management systems driven by the need to provide more efficient processes and at the same time guarantee compliance with regulations and equal treatment of customers. Starting from relatively simple and repetitive business processes, e.g. for handling invoices, the next step is to digitalise more *flexible work processes*, e.g. of knowledge workers [22] that are *distributed* across different departments.

In many business process management solutions, notably solutions employing Business Process Model and Notation (BPMN), a distributed process will be described as a set of pools, where each pool contains a flow graph that explicitly describes the flow of control between actions at that particular location. However, the explicit design time specification of both distribution and control flow sometimes lead to overly rigid processes; and changes to the distribution and control flow at run time, i.e. delegation of activities to different locations, repetition or skipping of activities, is non-trivial to support. Moreover, flow diagrams describe constraints on the ordering of activities only

⋆ Authors listed alphabetically. This research is supported by ITU, Exformatics A/S and the Velux Foundation through the Computational Artefacts (CompArt) project (http://www.compart.ku.dk).

implicitly. For instance, a simple business rule stating that a bank customer must provide a budget before getting approved for a loan can be checked only by verifying that on every path from the request for a loan to an approval, there is a "receive budget" event. Depending on the exact process language, the complexity of verifying this simple rule ranges from challenging to undecidable.

Towards the challenge of accommodating flexibility and compliance, there has been a renewed and increasing interest in *declarative* or *constraint-based* process notations such as DECLARE [23,24] and Dynamic Condition Response (DCR) graphs [7,13,16]. In a declarative process notation, a process is described by the constraints it must fulfill, while the control flow is left implicit. This means that activities can be carried out in any order and at any location that fulfills the constraints. It also means that compliance rules and constraints are captured explicitly in the model. However, so far constraint-based process notations have only been equipped with sequential semantics allowing only one event to happen at a time. This is in stark contrast to successful Petri Net-based workflow specifications, which have an inherent notion of concurrency.

In the present paper we make the following contributions:

1. We provide an overview of the challenges a notion of concurrency must overcome for an event-based declarative workflow notation.
2. We give a "true concurrency" semantics for DCR graphs by enriching DCR graphs with a notion of *independent events*, and prove that the semantics of a DCR graph in this case gives rise to a labelled asynchronous transition system [25,27]. The development, which is quite technical, has been verified to be correct in the Isabelle-HOL interactive theorem prover [19]; the formalised development is available online [4].
3. We show how this semantic underpinning of concurrency admits practical asynchronous execution of declarative workflows. Essentially, this is achieved by assigning events to location. Thus, we capture asynchronous semantics for the entire spectrum of distributions, spanning from the fully centralized workflow where every event is happening at the same location, to the fully decentralized workflow, where every event is managed at its own location.
4. We demonstrate the practical feasibility of the developed theory by reporting on a prototype implementation of a distributed declarative workflow engine. The prototype is accessible online [3].

Related Work. Concurrency and distribution of workflows defined as flow graphs are well-studied. Declarative modelling and concurrency has been studied in the context of the Guard Stage Milestone (GSM) model [14] and declaratively specified (Business) Protocols [8–10,26]. In the GSM model [14], declarative rules govern the state of Guards, which in turn admits Stages to open and execute. The declarative rules reference a global state, which executing a Stage might change non-atomically. Stages may run concurrently; to prevent errors of atomicity, a transactional concistency dicipline based on locks is followed. That is, stages can be said to be concurrent if they do not have interferring reads and writes to the global state. Neither (core) DCR graphs nor (core) DECLARE has explicit notions of data, global state or state update. Writes and reads of data must be modelled in DCR graphs as events, and interferrence between such events by relations, i.e. any write event to a data location should explicitly exclude and include every other event representing access to the same location. Thus, dependencies between activities are not expressed implicitly as predicates on a global state, but instead explicitly through relations between activities.

In [8–10,26] protocols are given declaratively as rules governing which actions must and must not be available in a given state. Like GSM, the steps in the protocol entail modifying the global state, and the availability of actions in a particular state is directly expressed as predicates on this state. Unlike GSM, race conditions are resolved by either ordering the types of updates [8], or by projecting the global specification onto subsets of its rules in a way that avoids the problems of non-local state and blindness [10].

The Agent-based approach of [15], while philosophically similar to the present approach, sidesteps the issue of concurrency. Agents manage or invoke services, comprised of tasks; tasks are explicitly declared as being in sequence, in parallel, etc. Before invoking a service, the invoking agent must negotiate the particulars of its usage with the managing agent; this negotiation is specified in part declaratively. It is left to the implementation of agents and services to ensure that concurrency issues do not arise.

Concurrency is less well-studied in the setting of pure declarative formalisms without explicit data and global state, like DECLARE and DCR graphs. We took tentative steps for DCR graphs in [1]. For DECLARE, [11,12] provide pattern based translations of a subset of DECLARE LTL constraints to Petri Nets by giving a net for each constraint. These works do not cover the full expressive power of LTL (in particular, they only cover finitary semantics). In contrast, DCR Graphs are known to be equivalent to Büchi-automata [5,16,18], and thus express infinitary liveness conditions and are more expressive than LTL. [20] offers a fully automatic mapping from Declare to finite state automata to Petri Nets, but disregard the independence relation in their translation. Finally, [21] considers declarative, event-based workflow specifications. Local constraints for each event are derived from a global specification provided in an LTL-like temporal logic. However, the use of the temporal logic makes the setting dependent on an initial calculation of the local constraints, which only provide the independence relation implicitly.

2 Concurrency & declarative workflows

In this section, we explain through examples the issues surrounding concurrency in declarative workflow specifications, and give the main gist of our proposed solution. Along the way, we will recall the declarative model of DCR graphs.

2.1 A mortgage credit application workflow

As our main example, we will use a declarative specification of a workflow from the financial services industry, specifically the mortgage application process of a mortgage credit institution. The example is based on an ongoing project with the Danish mortgage credit institution BRFKredit. For confidentiality reasons, we are unable to present an actual process of BRFKredit; instead, we have distilled down the major challenges discovered in that project into the following wholly fictitious application process.

Mortgage application processes are in practice *extremely* varied, depending on the type of mortgage, the neighbourhood, the applicant, and the credit institution in question. The purpose of the process is to arrive at a point where the activity *Assess loan application* can be carried out. This requires in turn:

1. Collecting appropriate documentation,
2. collecting a budget from the applicant, and
3. appraising the property.

In practice, applicants' budgets tend to be underspecified, so an intern will screen the budget and request a new one if the submitted one happens to be so.

The caseworker decides if the appraisal can be entirely statistical, i.e., carried out without physical inspection, but rather based on a statistical model taking into account location, tax valuation, trade history etc.; or if it requires an on-site appraisal. On-site appraisals are cursory in nature, and do not require actually entering the property. For reasons of cost efficiency, one may not do both on-site and statistical appraisals, not even in the case of an audit. However, if the neighbourhood is insufficiently uniform, a thorough on-site appraisal is required. This thorough appraisal requires physical access to the property, so the mobile consultant performing the appraisal will in this case need to book a time with the applicant.

Appraisals are occasionally audited as a matter of internal controls; an audit may entail an on-site appraisal, which may or may not coincide with an ordinary on-site appraisal. It is customary, however, to consider a statistical appraisal an acceptable substitute for an on-site appraisal during an audit.

2.2 A DCR formalisation

This textual description of the application process is inherently *declarative*: we have described constraints on the ordering of activities in the process rather than positing a particular sequencing. Thus, this process is naturally described by a declarative process model such as DECLARE or DCR graphs. Presently, we give a DCR graph-based declarative model in Figure 1 on page 5, produced with the tool available at [3].

DCR models are graphical; activities, also known as "events" are represented by boxes, labelled by the name of the activity and the role or participant executing that activity. E.g., the top-right box represents an activity *Collect documents* which is carried out by a caseworker. Activities are colored according to their state: grey is not currently executable, red text is required, and greyed out is excluded. Arrows between boxes represent constraints between activities. DCR graphs define in their most basic form only 4 such constraints: Conditions and Responses, Inclusions and Exclusions.

Conditions. A condition, drawn as an arrow with a dot at the head, represents the requirement that the source activity must be executed at least once before the target activity can be executed. In our model, we see, e.g., that *Collect documents* must be executed before *Assess loan application* can be.

Responses. An activity can be required for the workflow to be considered complete, usually called *accepting*. Incomplete or "pending" activities are labelled in red and

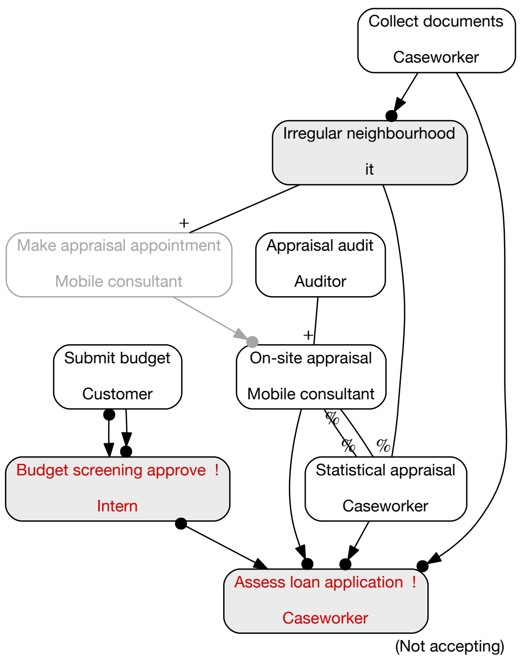

Fig. 1. Declarative DCR model of a mortgage application process

have an exclamation mark next to them. In the model, the activities *Budget screening approve* and *Assess loan application* are initially pending. A response, represented by an arrow with a dot at the tail, indicates that executing the source activity imposes the requirement to later do the target activity, that is, executing the former makes the latter pending. In the model, when an applicant does *Submit budget*, this imposes the requirement of a subsequent screening, and so there is a response from *Submit budget* to *Budget screening approve*.

Inclusions and exclusions. An activity is always in one of two states: it is either included or excluded. In diagrams, excluded activities are drawn with a thin gray; regularly drawn activities are included. An excluded activity cannot execute; it cannot prevent the workflow from being accepting, even if it is pending; and it cannot prevent other activities from executing, even if they have conditions on it. For ease of reading, conditions from excluded activities are also drawn with a thin gray, to indicate that they currently do not have effect.

An activity may cause other activities to be included or excluded when it is itself executed. This is indicated diagrammatically with arrows that has "+" and "%" as heads. In the model, the *Irregular neighbourhood* activity—which is an automated activity executed by IT-systems—includes the *Make appraisal appointment*, which is initially excluded; this in turn makes *On-site appraisal* non-executable until the appointment has been made, by virtue of the condition from *Make appraisal appointment* to *On-site appraisal*. Conversely, the *On-site appraisal* and *Statistical appraisal* activities exclude each other: after doing one, one may no longer do the other.

The semantics of a DCR model is the set of (finite and infinite) sequences of activities in which every pending activity is eventually executed. We call such sequences "traces". For finite traces, this means that no activity is pending at the end.

Example 2.1. The model in Figure 1 admits (among infinitely many others), the following three traces. The first is the "happy path", the usual and simplest case. The second is the "happy path" for the less frequent case of an irregular neighborhood. The third is a convoluted special case, with audit and re-submission of a pre-screened budgets.

1. *Collect documents, Submit budget, Statistical appraisal, Budget screening approve, Assess loan application.*
2. *Submit budget, Collect documents, Irregular neighbourhood, Budget screening approve, Make appraisal appointment, On-site appraisal, Assess loan application.*
3. *Collect documents, Submit budget, Statistical appraisal, Irregular neighbourhood, Budget screening approve, Appraisal audit, Make appraisal appointment, Submit budget, On-site appraisal, Budget screening approve, Assess loan application.*

2.3 Concurrency in the example workflow

It would appear that certain activities in this workflow could happen concurrently, whereas others are somehow in conflict. It is clear from the textual specification that, e.g., the process of submitting and screening the budget is independent from the appraisal model, and we would expect to be able to execute them concurrently in practice.

Our DCR model of Figure 1 appears to bear out this observation: there are no arrows—and so it would seem no constraints—between *Submit budget* and *Budget screening approve* on the one hand; and *Appraisal audit*, *On-site appraisal*, and *Statistical appraisal* on the other. This insight begets the question: Exactly when are two activities concurrent? Exactly when will it always be admissible to swap two activities? These questions have practical relevance: E.g., the mobile consultant might be without internet connectivity when he executes the activity *On-site appraisal*; but this is admissible *only* if it is somehow guaranteed that only concurrent activities happen simultaneously.

We proceed to examine what is a reasonable notion of concurrency of activities through a series of examples. We will attempt to obtain a set of principles to help us later judge what is and is not a good definition of "concurrency".

Example 2.2. The traces indicated above gives an indication that there is indeed some form of independence, in that, e.g., in the first trace, the activities *Submit budget* on the one hand and *Statistical appraisal* on the other can be swapped and we still have an admissible trace. In fact, it is not terribly difficult to prove that in any admissible trace, we can always swap adjacent activities when one is among the budget activities and the other is among the appraisal activities, and the trace we then get is still admissible.

The principle we observe here is that adjacent concurrent activities should be able to happen in either order.

Example 2.3. A very easy example of activities that cannot be considered concurrent are ones related by a condition. If one requires the other to have happened previously, clearly they cannot in general happen at the same time. This is the case for, e.g., *Collect documents* and *Irregular neighbourhood.*

The principle we observe here is that concurrent activities cannot enable each other.

Example 2.4. However, clearly not every two activities can be reasonably swapped. For instance, the activies *On-site appraisal* and *Statistical appraisal* are specified to be mutually exclusive (in most cases) in the textual specification, and in the DCR model each excludes the other. If one happens, the other cannot, and so they cannot reasonably be considered concurrent: When they cannot happen one after the other, surely they should not be allowed to happen simultaneously.

The principle we observe here is that concurrent activities cannot disable each other.

Example 2.5. A different way activities can be in conflict is if their executions have mutually incompatible effects on the state of the DCR graph. For instance, the *Appraisal audit* includes *On-site appraisal*, whereas *Statistical appraisal* excludes it. Clearly, *Appraisal audit* and *Statistical appraisal* cannot be executed concurrently: if they were to happen at the same time, what would be the resulting state of *On-site appraisal*—included or excluded?

The principle we observe here is that concurrent activities cannot have incompatible effects on the state of other activities.

Example 2.6. The examples we have seen so far have one thing in common: activities that could not be considered concurrent were related by arrows in the model. Could it be that events not directly related are necessarily concurrent?

No! Consider the events *Irregular neighbourhood* and *On-site appraisal.* These are not directly related: there are no arrows from one to the other. However, *Irregular neighbourhood* includes *Make appraisal appointment*, which is a condition for *On-site appraisal*. Thus executing *Irregular neighbourhood* prevents the execution of *On-site appraisal*. Thus we might observe the ordering first *On-site appraisal* followed by *Irregular neighbourhood*, but never the opposite order. In the abstract, like for conditions, one of these activities precludes the execution of the other, and so they cannot be considered concurrent—even though there is no arrow between them.

The principle we observe here we saw already before: concurrent events cannot disable each other.

In subsequent sections, we formalise concurrency of DCR activities in terms of Labelled Asynchronous Transition Systems. We shall see that within the notion of concurrency embodied in those, the handful of examples we have given above in fact embody *all* the ways activities of a DCR graph can be non-concurrent.

3 DCR Graphs

In this Section we define DCR graphs formally. This is a necessary prerequisite for defining concurrency of DCR graph events (activities) in the next Section.

The formalisation here mirrors a mechanised but somewhat less readable formalisation in the proof-assistant Isabelle-HOL [19]; results of the next section are verified to be correct by Isabelle-HOL. The formalisation is available online [4].

We will need the following notation. For a set E we write $\mathcal{P}(E)$ for the power set of E (i.e. set of all subsets of E) and $\mathcal{P}_{ne}(E)$ for the set of all non-empty subsets of E. For a binary relation $\rightarrow \subseteq E \times E$ and a subset $\xi \subseteq E$ of E we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{e \in E \mid (\exists e' \in \xi \mid e \rightarrow e')\}$ and the set $\{e \in E \mid (\exists e' \in \xi \mid e' \rightarrow e)\}$ respectively. For convenience, we write $\rightarrow e$ and $e \rightarrow$ instead of the tiresome $\rightarrow \{e\}$ and $\{e\} \rightarrow$.

In Def. 3.1 below we formally define DCR Graphs.

Definition 3.1 (DCR Graph). *A Dynamic Condition Response Graph (DCR Graph) G is a tuple* $(\mathsf{E}, \mathsf{M}, \mathsf{R}, \mathsf{L}, l)$, *where*

(i) E *is a set of* events *(or activities),*
(ii) $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \in \mathcal{M}(G)$ *is the* marking, *for* $\mathcal{M}(G) =_{def} \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E})$ *(mnemonics: Executed, Response-required, and Included),*
(iii) $\mathsf{R} = (\rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%)$ *are the* condition, response, include *and* exclude *relation respectively, with each relation* $\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$.
(iv) L *is the set of* labels *and* $l : \mathsf{E} \rightarrow \mathsf{L}$ *is a labeling function mapping events to labels.*

For the remainder of this paper, when a DCR graph G is clear from the context, we will assume it has sub-components named as in the above definition; i.e., we will write simply $\bullet\rightarrow$ and understand it to be the response relation of G.

An event of a DCR graph is enabled if it is included and every one of its conditions were previously executed:

Definition 3.2. *For an event* e *of a DCR graph G, we say that* e *is* enabled, *written* $G \vdash e$, *iff* $e \in \mathsf{In} \wedge (\mathsf{In} \cap \rightarrow\bullet e) \subseteq \mathsf{Ex}$.

In the following definitions we then define the result of executing an event of a DCR Graph. Firstly, in Def. *3.3* we define the *effect* of the execution of the event, i.e. which event was executed(Δe), which events are being included(ΔI), which events are being excluded(ΔX) and which events are being made pending(ΔR). We then in Def. *3.4* define how the effect is applied to the (marking of the) DCR Graph to yield a new (marking of the) DCR Graph: Δe is added to the set of executed events, first ΔX are removed from the set of included events and afterwards ΔI are added to the set of included events (meaning that events that are both included and excluded in a single step will remain included), finally Δe is removed from the set of pending responses before ΔR is added to the set of pending responses (meaning that if an event is a response to itself it will remain pending after execution). Finally in Def. *3.5* we define how these two operations are used together to execute an event on a DCR Graph, yielding a new DCR Graph.

Definition 3.3. *The* effect *of the execution of an event* e *on a DCR Graph G is given by* $\textsc{effect}(G, e) = (\Delta e, \Delta I, \Delta X, \Delta R)$ *where:*

(i) $\Delta e = \{e\}$ *the singleton set containing the event being executed,*
(ii) $\Delta I = e \rightarrow+$ *the events being included by e,*
(iii) $\Delta X = e \rightarrow\%$ *the events being excluded by e,*
(iv) $\Delta R = e \bullet\rightarrow$ *the events being made pending by e.*

When the DCR Graph G is given from the context we will below write δ_e for $\text{EFFECT}(G, e)$.

Definition 3.4. *The* action *effect* $\delta_e = (\Delta e, \Delta I, \Delta X, \Delta R)$ *on marking* $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *is:*

$$\delta_e \cdot (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) = \big(\mathsf{Ex} \cup \Delta e, (\mathsf{Re} \setminus \Delta e) \cup \Delta R, (\mathsf{In} \setminus \Delta X) \cup \Delta I\big)$$

The action of effect δ_e on a DCR Graph $G = (\mathsf{E}, \mathsf{M}, \mathsf{R}, \mathsf{L}, l)$ is then defined as:

$$\delta_e \cdot (\mathsf{E}, \mathsf{M}, \mathsf{R}, \mathsf{L}, l) = (\mathsf{E}, \delta_e \cdot \mathsf{M}, \mathsf{R}, \mathsf{L}, l)$$

Definition 3.5. *For a Dynamic Condition Response Graph G and event $G \vdash e$, we define the result of executing e as $G \oplus e =_{def}$ $\text{EFFECT}(G, e) \cdot G$.*

Towards defining accepting executions of DCR graphs, we first define the *obligations* of a DCR graphs to be its set of included, pending events.

Definition 3.6. *Given a DCR graph $G = (\mathsf{E}, \mathsf{M}, \mathsf{R}, \mathsf{L}, l)$ with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, we define the* obligations *of G to be* $\text{OBL}(G) = \mathsf{Re} \cap \mathsf{In}$.

Having defined when events are enabled for execution, the effect of executing an event and a notion of obligations for DCR Graphs we define in Def. 3.7 the notion of finite and infinite executions and when they are accepting. Intuitively, an execution is accepting if any obligation in any intermediate marking is eventually executed or excluded.

Definition 3.7 (DCR Semantics). *For a DCR graph G an* execution *of G is a (finite or infinite) sequence of tuples $\{(G_i, e_i, G'_i)\}_{i \leq k}$ (for $k \in \mathbb{N} \cup \omega$) each comprising a DCR Graph, an event and another DCR Graph such that $G = G_0$ and for $i < k$ we have $G'_i = G_{i+1}$; moreover for $i \leq k$ we have $G_i \vdash e_i$ and $G'_i = G_i \oplus e_i$. We say the execution is* accepting *if for $i \leq k$ we have for all $e \in \text{OBL}(G_i)$ there is a $j \geq i$ with either $e_j = e$ or $e \notin \text{OBL}(G'_j)$. We denote by* $\mathsf{exe}(G)$ *respectively* $\mathsf{acc}(G)$ *the sets of all executions respectively all accepting executions of G. Finally, we say that a DCR graph G' is reachable from G iff there exists a finite execution of G ending in G'.*

4 Asynchronous Transition Systems & DCR Graphs

With the DCR graphs in place, we proceed to imbue DCR graphs with a notion of concurrency. For this, we use the classical model of asynchronous transition systems [27], here extended with labels as in [25]. As mentioned, the development has been verified in Isabelle-HOL [19]; the formalisation source is available online [4].

Once we embed DCR graphs in labelled asynchronous transition systems, we shall find that the examples of concurrent and non-concurrent activities from Section 2 actually exemplify *independent* and non-independent events. Moreover, the examples will turn out to be exhaustive, in the sense that each example exemplifies one of the properties necessary for events to be (or not to be) independent.

We apply the results of the present section in Section 5, when we present a prototype implementation of a distributed declarative workflow engine. The correctness of this engine hinges on the notion of independence presented here.

First, we recall the definition of labelled asynchronous transition systems [25].

Definition 4.1 (LATS). *A* Labelled Asynchronous Transition System *is a tuple* $A = (S, s_0, \mathsf{Ev}, \mathsf{Act}, l, \rightarrow, I)$ *comprising states* S*, an initial state* $s_0 \in S$*, events* Ev*, a labelling function* $l : \mathsf{Ev} \rightarrow \mathsf{Act}$ *assigning labels (actions) to events, a transition relation* $\rightarrow \subseteq S \times \mathsf{Ev} \times S$*, and an irreflexive, symmetric independence relation* I *satisfying*

1. $s \xrightarrow{e} s'$ *and* $s \xrightarrow{e} s''$ *implies* $s' = s''$
2. $s \xrightarrow{e} s'$ *and* $s' \xrightarrow{e'} s''$ *and* eIe' *implies* $\exists s'''$ *such* $s \xrightarrow{e'} s'''$ *and* $s''' \xrightarrow{e} s''$
3. $s \xrightarrow{e} s'$ *and* $s \xrightarrow{e'} s''$ *and* eIe' *implies* $\exists s'''$ *such* $s' \xrightarrow{e'} s'''$ *and* $s'' \xrightarrow{e} s'''$

In words, the first property says simply that the LATS is *event-determinate*: an event will take you to one and only one new state. The second says that independent events do not enable each other. The third that independent events can be re-ordered. In the context of DCR graphs, the first property is trivially true, and we have seen an example of the second property holding in Example 2.3, and of the third in Example 2.2.

For the remainder of this section, we establish that a DCR graph G gives rise to a LATS $\mathcal{A}(G)$. Along the way, we shall see how the various definitions we set up to eventually arrive at independence arise from the examples of "obviously concurrent" and "obviously non-concurrent" behaviours we saw in Section 2.

Towards finding a suitable notion of independence, we first define a notion of effect-orthogonality for events of a DCR graph. As we shall see, this orthogonality characterises the situation where the effects of events commute on markings.

Definition 4.2. *We say that events* $e \neq f$ *of a DCR graph G are* effect-orthogonal *iff*

1. *no event included by* e *is excluded by* f *and vice versa, and*
2. e *requires a response from some* g *iff* f *does.*

We lift this notion to effects themselves, saying δ_e, δ_f *of G are orthogonal iff* e, f *are.*

Here, the first condition says that effect-orthogonal events cannot have conflicting effects. We saw an example of such conflicts in Example 2.5: the *Appraisal audit* includes *On-site appraisal*, whereas *Statistical appraisal* excludes it. The second condition is perhaps less intuitive, saying that if one event makes the other pending, the other event hides this effect by making itself pending. A more intuitive, but also more restrictive alternative, would be to require that neither event has a response on the other.

Proposition 4.3. *Let* δ_e, δ_f *be effects of a DCR graph G, and let* M *be a marking for G. If* e, f *are orthogonal then* $\delta_e \cdot (\delta_f \cdot M) = \delta_f \cdot (\delta_e \cdot M)$*.*

Proof (in Isabelle). See [4], Lemma "orthogonal-effect-commute".

Next, we define that two events are cause-orthogonal. The intention is that for such event pairs, executing one cannot change the executability of the other.

Definition 4.4. *Events* e, f *of a DCR-graph G are* cause-orthogonal *iff*

1. *neither event is a condition for the other,*
2. *neither event includes or excludes the other, and*
3. *neither event includes or excludes a condition of the other.*

We saw examples of all three conditions previously. Specifically, for (1), we saw in Example 2.3 that *Collect documents* is a condition for *Irregular neighbourhood*, and so these activities cannot be considered non-causal. For (2), we saw in Example 2.4 how *On-site appraisal* and *Statistical appraisal* exclude each other and thus cannot be cause-orthogonal. For (3), we saw in Example 2.6 how *Irregular neighbourhood* included a condition of *On-site appraisal*, and thus those two events cannot be cause-orthogonal.

From effect- and cause-orthogonality, we obtain the requisite notion of independence. This explains the contents of the examples we have seen so far: activities that could be considered "concurrent" are independent; those that could not are not.

Definition 4.5. *Given a DCR graph G, we say that events e, f are* independent *if they are both effect- and cause-orthogonal. We write I_G for the independence relation induced by a DCR-graph G.*

We must of course prove that our proposed independence relation I_G satisfies the conditions for an independence relation of Definition 4.1.

Theorem 4.6. *Let G be a DCR graph. If e, f are independent events of G then any marking in G satisfies the concurrency properties (1–3) of Definition 4.1.*

Proof (in Isabelle). See [4], Theorem "causation-and-orthogonality-entails-independence".

And with that, we arrive at a formal definition of concurrency for the declarative workflow model of DCR graphs: Each DCR graph has an associated independence relation, and thus an associated LATS, which tells us which activities (events) can be considered concurrent and which cannot.

Corollary 4.7. *Let G be a DCR graph. Then $\mathcal{L}(\mathcal{G})$ is a Labelled Asynchronous Transition System when equipped with independence relation I_G. We call this LATS $\mathcal{A}(\mathcal{G})$.*

Proof (in Isabelle). See [4], Theorem "DCR-LATS".

We shall see in Section 5 how Corollary 4.7 and Theorem 4.6 enables a practical distributed implementation of declarative workflows in general, and in particular of our mortgage application example. We conclude this section by noting in Table 1 which events of our running example are in fact independent.

5 A Process Engine for Distributed Declarative Workflows

The previous sections supply an understanding of DCR graphs as labelled asynchronous transition systems and in particular of independence of DCR graph events. With that, the door opens to a distributed implementation of a declarative workflow language. We have implemented such a prototype engine; in this Section, we describe by example the workings of that engine.

The central idea is to exploit the extremely local nature of DCR events in conjunction with the notion of independence. Because of the locality of DCR events, we can partition the set of events of a DCR graph into *components*, assigning each component

	Appraisal audit	*Assess loan application*	*Budget screening approve*	*Collect documents*	*Irregular neighbourhood*	*Make appraisal appointment*	*On-site appraisal*	*Statistical appraisal*	*Submit budget*
Appraisal audit			x	x	x	x			x
Assess loan application						x			x
Budget screening approve	x			x	x	x	x	x	
Collect documents	x		x			x	x	x	x
Irregular neighbourhood	x		x						x
Make appraisal appointment	x	x	x	x				x	x
On-site appraisal			x	x					x
Statistical appraisal			x	x		x			x
Submit budget	x	x		x	x	x	x	x	

Table 1. Independence relation for activites of the model of Figure 1.

to a distinct node in a distributed system. The node is responsible for executing the particular event, and for notifying other components of executions, when such executions requires them to update their state.

However, a node cannot freely execute its events; that would leave us open to all the mistakes of non-concurrency exemplified in Section 2. We therefore employ a locking mechanism to ensure that *only concurrent events can be executed simultaneously*.

We exemplify this by forming a distributed version of our running example. For ease of presentation, we distribute the workflow over only two nodes: one for the "Mobile consultant" (presumably his mobile device), and one for the rest. However, the principles of distribution employed here apply to arbitrarily fine sub-divisions of DCR graphs, right down to each node hosting only a single event.

Presently, we obtain the two components in Figure 2. The diagram for each component represents remote events as dashed boxes. Moreover each component retains only remote events with which some local event is not independent. For the "Mobile consultant" component (Figure 2), that means that all events related to budgets are gone, as is the initial *Collect documents*. The "other" component (Figure 2) retains all the "Mobile consultant" events, because every event of the Mobile consultant is in fact in conflict with some event local to "other".

The procedure for executing an event, in detail, is as follows. A component wishing to execute an event e must first request[3] and receive locks on all (local and remote) events that are in conflict (i.e., not independent) with e (thus, in particular, on itself). It then queries the state of remote events to determine if e is currently executable. If it is, it instructs remote events affected by firing e to change state accordingly. Finally, it releases all locks.

For example, if the "other" component wishes to execute the *Assess loan application* event in the DCR graphs of Figures 2 and 2, it will first request and receive a lock on

[3] All components request locks in the same fixed order to prevent deadlocks.

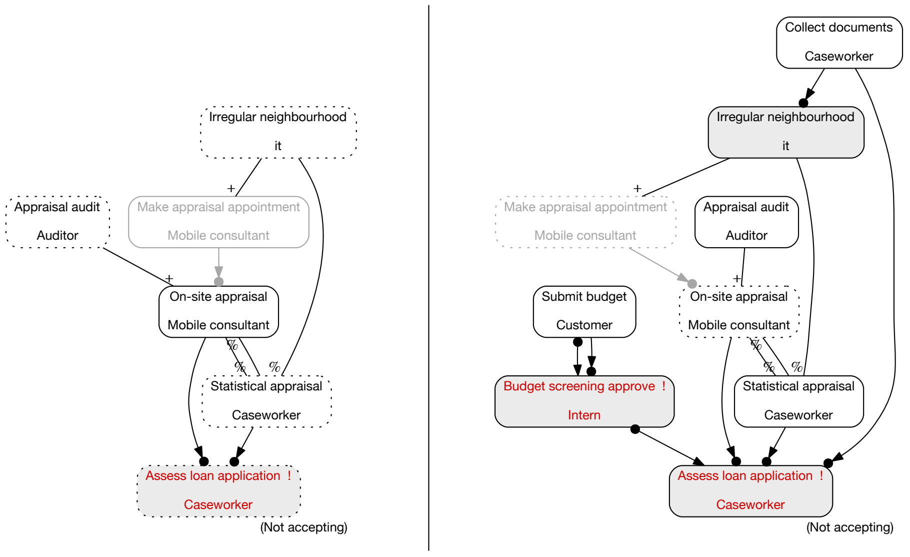

Fig. 2. Component models for the Mobile consultant (left) and for other roles (right).

On-site appraisal; then query the state of *On-site appraisal*; then find out that that event is not previously executed; and will then release the lock on *On-site appraisal*.

Notice that since this procedure is based on independence, it allows concurrency in the very concrete sense that the "other" component is free to execute any of the events *Collect documents*, *Submit budget*, and *Budget screening approve* **without** communication with the "Mobile consultant" component, because these three events are all independent with all the events of the "Mobile consultant" component. Conversely, any other event requires communication, since these other events are all in conflict with the some event of the "Mobile consultant" component.

Implementation. We have implemented the technique described here in the DCR Workbench, an existing web-based tool for experimenting with DCR graphs; see, e.g., [7]. The diagrams in this paper are all output from this prototype.
The prototype allows specifying components by accepting for each activity an optional indication of a URL at which the event is located. E.g, in the component model for other roles (Fig 2), the remote activity "On-site appraisal" is given as:

```
"On-site appraisal"
  [ role = "Mobile consultant"
    url  = "http://localhost:8090/events/On-site%20appraisal" ]
```

The DCR Workbench then enables starting separate REST services for each such component model. Each service accesses information about state of remote events by issuing a GET to URLs derived from the specified one. E.g., in the other roles component model, "On-site appraisal" is a condition for "Assess loan application"; accordingly, to execute "Assess loan application", the REST service for that model will query the executed state of "On-site appraisal" by issuing a GET request to:

```
http://localhost:8090/events/On-site%20appraisal/executed
```

Similarly, PUT requests are used to update the state of remote activities; e.g., “Irregular neighbourhood” will, when executed, make “Make appraisal appointment” excluded by issuing an appropriate PUT.The implementation ensures that before state of remote events is queried or updated, all independent activities are locked. Please refer to the prototype [3] to experiment with declarative concurrency first-hand!

6 Conclusion

We have studied concurrency of pure declarative workflow models. This problem is important, since its solution is a prerequisite for implementing distributed engines for declaratively specified workflows. Concretely, we investigated reasonable examples and non-examples of concurrency for the declarative DCR model by example; we formally added a notion of concurrency between events of DCR graphs, enriching the standard semantics to a semantics of the classical true concurrency model of labelled asynchronous transition systems. We backed this foundational contribution by (a) a formal verification in Isabelle-HOL of the development [4], and (b) a proof-of-concept implementation of a distributed declarative workflow engine, available at [3].

6.1 Discussion and Future Work

The present work considers only core DCR Graphs, which can represent only finite state processes and have no (practical) representation of data, as events can not be parametrized by data. This consitutes of course a noteworthy gap between the theory and practice.

The practical commercial use of DCR graphs by Exformatics has succesfully employed DCR graphs as a control-flow layer on top of an underlying database, using database triggers as events signalling changes to data values [6]. Processes dynamically handling multiple instances of business artifacts (e.g. multiple instances of the budget in our running example) with separate life cycles were realised by different DCR graphs, one for each data object being processed, interacting via the underlying database. In this case, the present work would apply to the individual models for each artifact, but not accross the models.

In [5, 7], DCR Graphs have been extended to DCR Graphs with sub-processes, allowing dynamically created multiple instances of sub processes and thus enabling analysis of processes as described above. We believe that the present work on concurrency can be lifted to DCR Graphs with sub-processes. The increased expressiveness however comes at the cost of making the model Turing complete [5].

Regarding data, we are presently working on extending the work on sub-processes for DCR-graphs [7] to *parametric* sub-processes: Events which take data values as input can spawn a new sub-process as a continuation, whose shape depends on the data inputs (and in particular allows to declaratively ”store” the revieved data in the continuation, as in functional programming languages). This should be compared to declarative models facilitating data and state as side-effects on a global state such as [14].

On a different note, given the similarity of DCR graphs and DECLARE, it is natural to ask whether the presently introduced notion of concurrency and subsequent distribution of executable models can be transferred to DECLARE. To this end, it’s important to realise that the present work relies crucially on the notion of "event state" inherent in DCR graphs. Concurrency and independence can be framed in terms of which events may or may not update the states of other events by firing. DECLARE does not come with a similar notion of state, and so it would appear that the present approach does not apply directly. However, there is still hope: Looking at the standard relations of DECLARE instead of LTL in general, it seems plausible that one might define an alternate semantics either by encoding of DECLARE into DCR Graphs or in terms of some similar notion of "activity state"; and then apply the approach of the present paper.

Our work with industry suggests that the flexibility of DCR Graphs is sought for, but the difficulty of presenting and understanding declarative models is a major obstacle to wider adaptation of declarative methodologies. This often stems from fairly small models defining sometimes quite complex behaviour. We believe that the ability to distribute DCR Graphs and understand the independence between events is likely to help presenting the models. For instance, defining independence for DCR graph events as labelled asynchronous transition systems (lats) opens the door to an encoding of DCR graphs into Petri nets using the mapping from lats to Petri nets in [25]. In addition to opening up for the application of the many tools and techniques developed for Petri Nets, it would give a way of deriving flow diagrams from DCR graphs in a concurrency-preserving way, which should be compared to the work in [11].

Finally, the concurrent semantics opens up for possible use of partial-order reduction model checking techniques [2] towards more efficient static analysis of DCR graphs than the current implementations based on verification on Büchi-automata [16–18].

Acknowledgments The authors gratefully acknowledges helpful comments from Younes Nielsen, Morten Marquard, Nicholas Guenot, and anonymous reviewers.

References

1. Distributed declarative control of networked cyber-physical systems. In *Proceedings of UIC 2010*, volume 6406 of *LNCS*, pages 397–413, 2010.
2. Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*. MIT Press, 2008.
3. Søren Debois. DCR Workbench. `http://tiger.itu.dk:8021/static/bpm2015.html`, 2015.
4. Søren Debois. Isabelle-hol formalisation of present paper. `http://www.itu.dk/people/debois/dcr-isabelle`, 2015.
5. Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *Proceedings of FM 2015*. LNCS, 2015.
6. Søren Debois, Thomas Hildebrandt, Tijs Slaats, and Morten Marquard. A Case for Declarative Process Modelling: Agile Development of a Grant Application System. In *EDOC Workshops '14*, pages 126–133. IEEE, September 2014.
7. Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Hierarchical declarative modelling with refinement and sub-processes. In *BPM '14*, volume 8659 of *LNCS*, pages 18–33. Springer, 2014.
8. Nirmit Desai, Amit K. Chopra, Matthew Arrott, Bill Specht, and Munindar P. Singh. Engineering Foreign Exchange Processes via Commitment Protocols. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 514–521. IEEE, 2007.

9. Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Trans. Softw. Eng. Methodol.*, 19(2):6:1–6:45, October 2009.
10. Nirmit Desai and Munindar P. Singh. On the enactability of business protocols. AAAI'08, pages 1126–1131. AAAI Press, 2008.
11. Dirk Fahland. Synthesizing Petri nets from LTL specifications - an engineering approach. In Stephan Philippi and Alexander Pinl, editors, *Proc. of Algorithmen und Werkzeuge fr Petrinetze (AWPN), Arbeitsbericht aus dem Fach Informatik, Nr. 25/2007*, pages 69–74, Universitt Koblenz-Landau, Germany, September 2007.
12. Dirk Fahland. Towards analyzing declarative workflows. In *Autonomous and Adaptive Web Services*, number 07061 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
13. Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, volume 69 of *EPTCS*, pages 59–73, 2010.
14. Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, III Heath, Fenno(Terry), Stacy Hobson, Mark Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculin. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *Web Services and Formal Methods*, volume 6551 of *LNCS*, pages 1–24. Springer, 2011.
15. N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based Business Process Management. *Int'l. J. of Cooperative Inf. Sys.*, 05(02n03):105–130, June 1996.
16. Raghava Rao Mukkamala. *A Formal Model For Declarative Workflows - Dynamic Condition Response Graphs*. PhD thesis, IT University of Copenhagen, March 2012.
17. Raghava Rao Mukkamala, Thomas Hildebrandt, and Tijs Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *EDOC*, pages 127–136. IEEE, 2013.
18. Raghava Rao Mukkamala and Thomas T. Hildebrandt. From dynamic condition response structures to büchi automata. In *TASE*, pages 187–190. IEEE Computer Society, 2010.
19. Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
20. Johannes Prescher, Claudio Di Ciccio, and Jan Mendling. From declarative processes to imperative models. In *SIMPDA '14*, pages 162–173, 2014.
21. M. P. Singh. Synthesizing distributed constrained events from transactional workflow specifications. In *Proc. of the Twelfth Int'l Conf. on Data Eng.*, pages 616–623. IEEE, 1996.
22. Keith D. Swenson. *Mastering the Unpredictable: How Adaptive Case Management Will Revolutionize the Way That Knowledge Workers Get Things Done*. Meghan-Kiffer, 2010.
23. Wil van der Aalst, Maja Pesic, Helen Schonenberg, Michael Westergaard, and Fabrizio M. Maggi. Declare. Webpage, 2010. `http://www.win.tue.nl/declare/`.
24. Wil M.P van der Aalst and Maja Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM '06*, volume 4184 of *LNCS*, pages 1–23. Springer Verlag, 2006.
25. Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of Logic and the Foundations of Computer Science*, volume 4, pages 1–148. OUP, 1995.
26. Pinar Yolum and Munindar P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. AAMAS '02, pages 527–534. ACM, 2002.
27. W. Zielonka. Notes on finite asynchronous automata. *Informatique Théorique et Applications*, 21(2):99–135, 1987.