Hierarchical Declarative Modelling with Refinement and Sub-processes

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

[1] IT University of Copenhagen, Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
{debois,hilde,tslaats}@itu.dk
[2] Exformatics A/S, Lautrupsgade 13, 2100 Copenhagen, Denmark

Abstract. We present a new declarative model with composition and hierarchical definition of processes, featuring (a) incremental refinement, (b) adaptation of processes, and (c) dynamic creation of sub-processes. The approach is motivated and exemplified by a recent case management solution delivered by our industry partner Exformatics A/S. The approach is achieved by extending the Dynamic Condition Response (DCR) graph model with *interfaces* and composition along those interfaces. Both refinement and sub-processes are then constructed in terms of that composition. Sub-processes take the form of hierarchical (complex) events, which dynamically instantiate sub-processes. The extensions are realised and supported by a prototype simulation tool.

1 Introduction

Business process design technologies today are predominantly based on *flow-oriented* process notations such as the Business Process Model and Notation (BPMN) standard [18], which *imperatively* describes how a process should proceed from start to end. Often, business processes are required to be compliant with regulations and constraints given by busines polices, standards and laws. E.g., a customer must be informed about alternatives and risks before getting a loan in a bank, or a decision on a grant application cannot be made before the deadline for submissions of applications has been reached.

Since the flow-oriented notations only captures *how* to fulfill the compliance rules, the description and verification of compliance rules require other notations and techniques. This leaves the process designers with three modelling tasks: To describe the compliance rules, to describe the process, and to verify that the process is compliant to the rules. Typically, compliance rules are described *declaratively* using a variant of temporal logic such as Linear-time Temporal Logic (LTL) [21]. Compliance can then be verified during execution using run-time verification techniques [12] and, if the flow-diagrams are based on a formal model, also at design time [6]. In most industrial design tools, the flow-diagrams are however *not* based on a formal model, and consequently, design time verification is not supported. This means that the process designers have to figure out manually how to interpret the constraints, and compliance is then subsequently

S. Sadiq, P. Soffer, and H. Völzer (Eds.): BPM 2014, LNCS 8659, pp. 18–33, 2014.
© Springer International Publishing Switzerland 2014

verified informally and approved by, e.g., a lawyer. At best, a formal run-time or post-execution verification is performed against the execution log.

In these situations there is a high risk that processes become either non-compliant or over-constrained by design, to faciliate manual verification. Over-constrained processes, however, rarely fits reality or are simply not suitable for knowledge-intensive processes. A way to avoid these problems is to use the declarative approach (also) for the process design. Several declarative process modelling notations and techniques have been proposed in the last decade, including DECLARE [2,1], CLIMB [13], GSM [11] and Dynamic Condition Response (DCR) graphs [8,14]. However, sometimes the declarative approach makes it less clear from the end-user, how a process will proceed from start to end. Even with a graphical notation (as in DECLARE, GSM and DCR graphs), it may be difficult to comprehend the interactions between different constraints.

The DCR graph process modelling notation stands out by supporting a simple and efficient run-time execution, which mitigates the complexity of comprehending the constraints and allows for run-time adaptation [15], while still being more expressive than (propositional) LTL (and thus DECLARE), in that it allows to describe every union of a regular and an ω-regular language [3,16,14].

DCR Graphs were conceived as both a generalization of event structures [22] and a formalization and generalization of the Process Matrix [17] invented by Danish company Resultmaker. Since its inception, the DCR Graph notation and theory have been developed further in collaboration with Exformatics A/S, a Danish provider of case, document and knowledge management systems. A version of DCR graphs with a simple notion of nesting [9], an additional milestone relation, and support for data now forms the core their workflow engine [20,7]. However, DCR graph models as currently implemented become difficult to comprehend and present at a certain size. They seem to lack encapsulation, modularity and hierarchy; the key techniques to make large models comprehensible in both imperative [19] and declarative settings [23]. Also, practical modelling efforts by Exformatics A/S has revealed that DCR graphs emphatically needs a notion of "dynamically created" or "instantiated" sub-process.

In the present paper, we seek to remedy these shortcomings of DCR graphs. Our contributions are as follows.

1. We introduce refinement-by-composition for DCR graphs.
2. We add to DCR graphs a notion of *dynamically spawned sub-process*, defining Hi-DCR graphs.
3. We demonstrate the use of both incremental process design using an example exctracted from a recent case management solution delivered by Exformatics A/S to a Danish funding agency.
4. We provide a publicly available Hi-DCR graph tool.

The tool allows simulation, model-checking of the finite fragment, automatic visualisation and more. The compositions and refinements presented in examples were not made by hand, they were executed by the tool; all DCR diagrams in this paper has been generated by it, and all examples are fully executable by it.

Hi-DCR graphs are fully formalised; we prove both soundness of refinement—that refinement cannot accidentally remove constraints of the extant model—as well as Hi-DCR being strictly more expressive than ω-regular languages.

1.1 Related Work

Hierarchy for declarative languages was studied in [23], where the authors add *complex activities* to DECLARE [2,1]. The authors make a compelling case that hierarchy is a necessity for constructing understandable declarative models. Our industry partner's experiences fully supports this thesis; this is in part what has led to our investigation of sub-processes.

A complex activity is one which contains a nested DECLARE model governing when that activity may complete. The nested model starts when the complex activity opens, and the complex activity conversely may only close once the nested model completes. Otherwise, there is no interaction between the nested model and the parent model. In the present paper, a sub-process may interact with its parent process: there can be multiple ways to start the sub-process, it can have different observable outcomes, and it is allowed to interact with other activities in the parent process.

Questions about concurrency are left open by [23]: the authors do not report a formal semantics, and the paper has no examples of interleavings of complex activities. In the present approach, sub-process executions are naturally interleaved with other events and even other instances of the same sub-process; we shall see this in examples.

We believe it is straightforward to formalize complex activities of [23] in Hi-DCR Graphs: Use Hi-DCR relations to allow only a single start and end event for each sub-process, and cut off interaction between sub- and super-process by choosing only empty interfaces.

The Guard-Stage-Milestone (GSM) approach [11] to business modelling provides a data-centric notation with declarative flavour. The notation consists of *stages*, which in turn have *guards*, controlling when and how the stage may start, and *milestones*, controlling when and how a stage may close. Stages can contain sub-stages, giving GSM an inherent hierarchy. Where GSM is data-centric, the present formalism is event-based. Nonetheless, the sub-processes of Hi-DCR graphs are strongly reminiscent of GSM stages, with Hi-DCR interface events assuming the rôle of guards and milestones. In future work, we plan to further investigate the similarities between GSM and Hi-DCR Graphs, in the hope of providing a formal connection between them.

2 DCR Graphs

In this section, we recall DCR graphs as introduced in [8,14] and introduce our running example. The example is based on a workflow of a Danish funding agency; our industry partner, Exformatics A/S, has implemented system support for this workflow using the basic DCR graphs of this Section [20]. While

vindicating DCR graphs as a flexible and practical modelling tool, that work also highlighted the potential need for refinement and sub-processes, which we introduce in the following Sections. One key idea will be the development of models by *refinement*: start from a very abstract model, then successively refine it until it becomes suitably concrete. In this section, we introduce DCR graphs alongside such a very abstract model.

As the name suggests, DCR graphs are *graphs*, and we tend to represent them visually, as in Fig. 1. This figure depicts a highly abstracted model of the funding agency workflow. Events (boxes) with labels (the text inside them) are related to other events by various arrows. In this model there are only four events: the beginning of the application round Start round; receiving an application (Receive application); the deadline for application submission occurring (Application deadline); and finally the board meeting (Board meeting), at which the board of the funding institution decide which applications warrant grants and which do not.

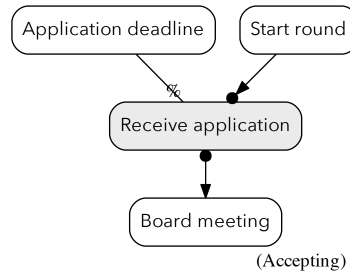


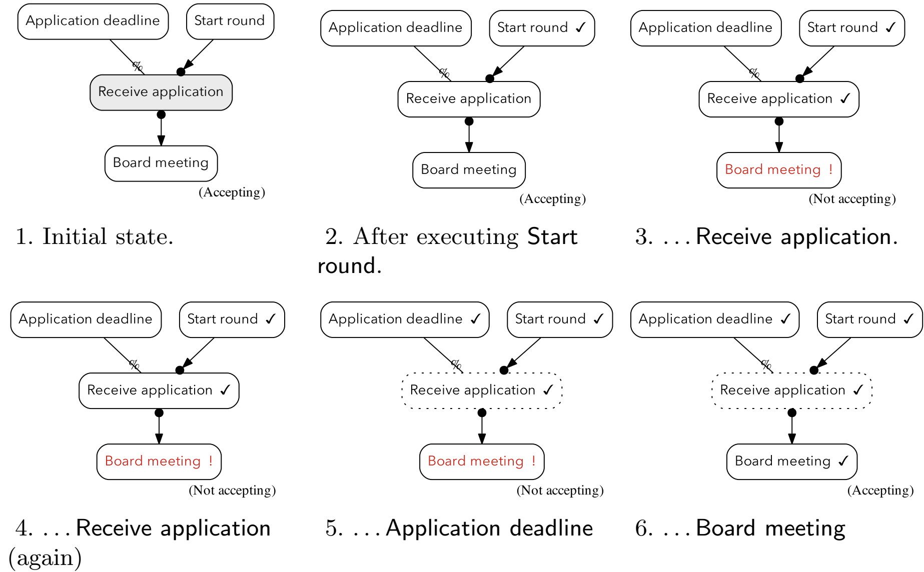
Fig. 1. A basic DCR graph

Relations between these events govern their relative order of occurrence. When not constrained by any relations, events can happen in any order and any number of times.

The *condition* relation, $e \rightarrow\!\bullet\; e'$, seen between Start round and Receive application indicates that the former must occur before the latter: we do not receive applications before the round has started. In the initial state of the DCR graph, Start round have yet to happen, and so Receive application cannot execute; hence it has been greyed out in the visual representation. The notation and semantics of this relation is similar to the precedence constraint in DECLARE [2,1].

Between Receive application and Board meeting we have a *response* relation, $e \bullet\!\!\rightarrow e'$. This indicates that if Receive application happens, then Board meeting must subsequently happen. This does not necessarily mean that each occurrence of the former is followed by a unique occurrence of the latter; it's quite all right to receive seven applications, have a board meeting, receive five more applications, then have a final board meeting. If Receive application has been executed without a following Board meeting, we say that Board meeting is *pending*.

Finally, between the Application deadline and Receive application events we have an *exclusion* relation, $e \rightarrow\!\% \; e'$. Once the Application deadline event occurs, the Receive application event becomes excluded, which means that it is from then on considered irrelevant for the rest of the workflow. While excluded, it cannot execute; any response obligations on it are considered void; and if it is a condition for some other event, that condition is disregarded. Dual to the exclusion relation is the *inclusion* relation. It is not exemplified in this DCR graph, but its meaning is straightforward: it re-includes an event in the workflow. DCR graph have also a fifth and final relation, the *milestone* relation $e \rightarrow\!\diamond\; e'$; we will postpone explaining that until we use it in the next Section.



1. Initial state. 2. After executing Start round. 3. . . . Receive application. 4. . . . Receive application (again) 5. . . . Application deadline 6. . . . Board meeting

Fig. 2. Execution of the DCR graph of Fig. 1

A key advantage of DCR graphs is that the graph *directly* represents the state of execution. There is no distinction between design-time and run-time. We will illustrate this by example: in Fig. 2 we have a finite execution of Fig. 1. In the upper-left corner, (1) is the initial state, the DCR graph presented in Fig. 1. The Start round event executes, taking us to (2). We can observe events having been executed in the state of the graph: executed events have little check-marks next to them, so in (2), Start round has such a check-mark. Also, with Start round executed, the condition for Receive application is fulfilled; it is now executable and thus no longer greyed out. We execute it to get to (3). Because there is a response from Receive application to Board meeting, that execution puts a pending response on Board meeting. This is indicated in (3) by the red text and the exclamation mark.

We execute Receive application to get to (4). This execution brings no change to the graph, which already had Receive application marked as previously executed, and already had a response on Board meeting. So we execute Application deadline, getting to (5). Because of the exclusion relation from that to Receive application, the latter becomes excluded, indicated by its box being dotted in (5). Even though excluded events cannot be executed, we do not grey them out; the dotted box is enough. Finally, we execute Board meeting to get to (6). This of course fulfils the pending response, which disappears: the text of Board meeting goes back to black, and the exclamation mark disappears.

A DCR graph is *accepting* if it has no included pending responses. (An infinite run is accepting if every incurred response is eventually executed or excluded.) The acceptance state of the graph is indicated in the lower-right corner of each graph. That indication is technically superfluous: the graph will be accepting

exactly if it has no red labels/labels with exclamation marks. For large graphs, it can be convenient to have the single indicator anyway.

3 Hierarchy and Refinement

We now come to the core contributions of this paper. We present a notion of "refinement" of DCR graphs, defined in terms of a more primitive notion of "composition" of DCR graphs. Refinement is always achieved by composing an *abstract DCR Graph* with a *refinement DCR Graph*, which introduces new events and/or add additional constraints.

3.1 Refinement

We wish to refine our model to express in greater detail the decision mechanics of the board. We will model board meetings by the DCR graph in Fig. 3. The results of an application round must be gathered in a report. This report is updated and approved repeatedly during the application round. This gives rise to two new events: Update report and Approve report.

Applications are discussed over the course of several board meetings and the results of the board meetings must be worked into the report. To allow the secretary to work efficiently she is *not* required to formally update the report after every single board meeting, but she may combine the outcomes of several of them in a single update. This constraint is represented by the response relation from Board meeting to Update report.

While there are such pending changes to the report, it can of course not be approved. This is modelled using a *milestone* relation Update report $\rightarrow\!\diamond$ Approve report. This relation means that *while* Update report is pending, Approve report can not execute.

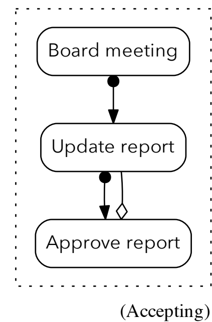


Fig. 3. Expanded model of the Board meeting

Note that this model does not preclude the board from re-approving a report that has not been updated. While not a particularly sensible thing to do, it is not against the rules, and as such *should* be permitted by the model.

Now, we wish to add these new details about board meetings to our original abstract model of Fig. 1; that is, we wish to *refine* Fig. 1 by Fig. 3. We do so by *composing* them: we fuse together events that are the same in both graphs. In this case only Board meeting. The result can be seen in Fig. 4. (The dashed box in that figure has no semantic ramifications; it is there are simply to make the graph easier to understand. See also [23].)

It is of course important that such a refinement does not accidentally *remove* constraints of the original model. Because of the inclusion and exclusion relations, that might happen, e.g., inclusions in the refining model might cause events excluded in the abstract one to be suddenly allowed. We shall prove in Theorem 4.10 that, roughly, when the two models agree on when fused events

are included or excluded, the composition will not admit new behaviour; in this case we call it a refinement. In the present case, the only fused event is Board meeting, which has no inclusions or exclusions going into it in either model, so this composition is really a refinement.

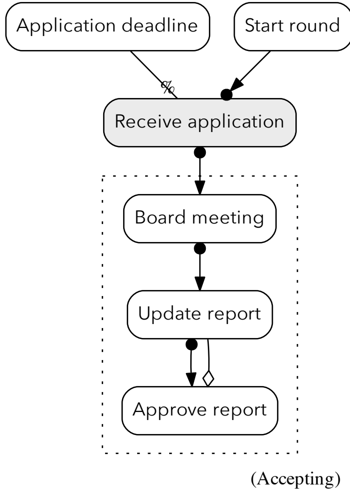


Fig. 4. Refinement of Fig. 1 by Fig.3

Refinement-as-composition in conjunction with DCR graphs having no distinction between design-time and run-time means that we can *refine a running model*. Suppose, for instance, that we have deployed our initial abstract model of Fig. 1, and have reached state (5) in Fig. 2 when it is decided that compliance with the board meeting report procedure of Fig. 3 must be enforced. We may add in these new constraints by refining the running model (Fig. 2, part 5) with the new constraints (Fig. 3). Doing so yields the new DCR graph seen in Fig. 5. Note how the pending state of the fused Board meeting event is preserved. And again, by virtue of the refinement Theorem 4.10, we can be assured that all constraints on execution of the original model is still preserved in this new refined one.

3.2 Subprocesses

Refinement gives us a disciplined method for extending models with new components; thus it gives us a hierarchical notion of process design. However, it does not fully capture the notion of sub-processes in traditional business modelling notations. Here, a sub-processes is a complex activity in the model that has underlying behaviour which is *instantiated* when the sub-process is started and closed when the sub-process ends. Such sub-processes can both be single-instance, meaning that only one instance of the sub-process will be active at any time, or multi-instance, meaning that multiple instances of the sub-processes can execute concurrently.

To enable modelling such sub-processes we extend DCR graphs to Hi-DCR graphs. In these, we may associate with an event an entire *other* Hi-DCR graph which, when the event fires, is composed onto the current graph. We exemplify Hi-DCR Graphs by adding to our funding agency model a more detailed description of the process for an individual application. As many applications may be received and evaluated at the same time, we need a notion of sub-processes (and in particular multi-instance sub-processes) to fully capture this behaviour.

An application must receive some number of reviews, with at least one from a lawyer. The reviews are collected in a review report. Based on the review report,

the application is accepted or rejected and the round report is updated with this decision. It is not uncommon that the decision on an application is reverted, changing an "accept" to a "reject" or vice versa, and this may even happen several times as discussions progress. Of course, each change in the decision requires an update to the round report. Finally, each applications cannot remain in limbo and must always eventually be either accepted or rejected.

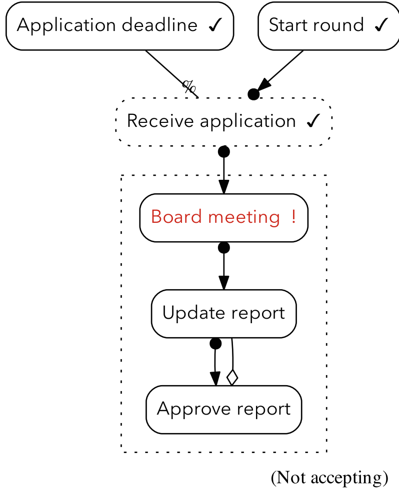


Fig. 5. Refinement of Fig. 2 part 5 by Fig. 3

The DCR graph in Fig. 6 models this process. At the top is Lawyer review and Other review. Of these two only Lawyer review is a condition for Review report, with the effect that we cannot write the review report unless we have at least a review from a lawyer.

After the Review report is completed, the reviewers may Accept or Reject the application; as mentioned, there is no restriction that these events happen only once. However, each new verdict requires Update report because of the response relation from Accept and Reject to Update report.

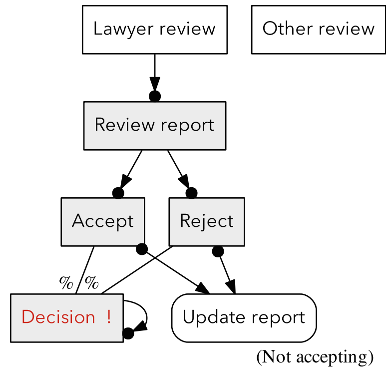


Fig. 6. The per-application sub-process

Finally we need to model the fact that either Accept or Reject needs to occur at least once, similar to the *choice* construct in DECLARE. Hi-DCR graphs contain no construct directly analogue to choice; but fortunately, there is a straightforward way—a DCR graph idiom, if you will—to achieve the intended semantics. We explicitly model the fact that a decision is needed as an event Decision. We make this event a condition of itself, meaning that it cannot possibly be executed. We also make it initially included and pending, so that once the application is started, a decision needs to eventually be made. Finally we let both Accept and Reject exclude Decision, indicating that these two both represents a valid decision. Once Decision becomes excluded, it no longer prevents the larger graph from achieving an accepting state.

Now, we wish the entire sub process of Fig. 6 to be instantiated *once per application*. In Hi-DCR graphs this is achieved by associating with the event Receive application in Fig. 5 the entire application processing DCR graph of Fig. 6. After executing Receive application a new copy of the application DCR Graph is composed with the main DCR Graph, and we get the DCR graph of Fig. 7.

Observe that once again, the common event Update report has been fused between the two DCR graphs. So far, this should be unsurprising: it is a straightforward application of the composition mechanism of DCR graphs. The key difference is that a Hi-DCR graphs is equipped also with a partitioning of its events into *interface events* (indicated by boxes with rounded corners in Fig. 6) and *local events* (indicated by boxes with non-rounded corners). This partitioning has the effect that under composition, only interface events are fused, whereas local events are not, even if their labelling overlap. The effect of these interfaces and local events will be apparent if we consider what happens when Receive application executes a *second* time; refer to Fig. 8. Here, we see that the second application process has fused its interface event Update report, but has duplicated its remaining events, which are all local. This has the following two important consequences:

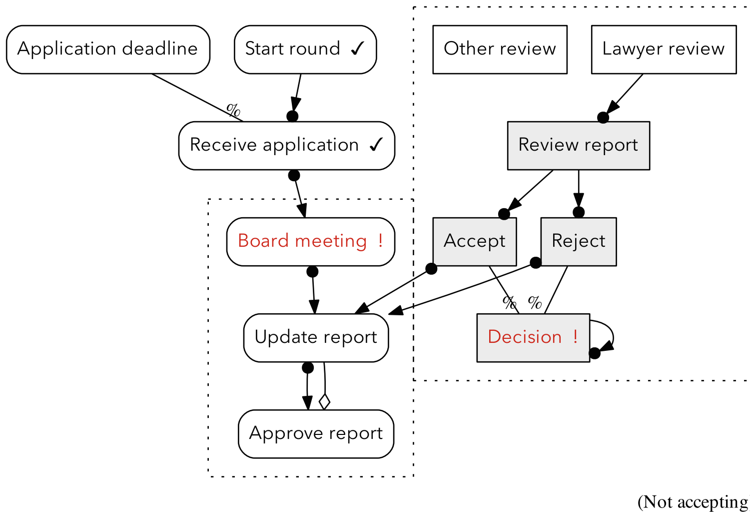


Fig. 7. Updated model with one spawned subprocess

1. Each application process is represented separately.
2. Approve report effectively synchronises decisions: whenever the decision on any application is changed, the report needs to be updated.

Connecting local and interface event is a highly expressive mechanism. For instance, if we want to have a review report ready for every application before the

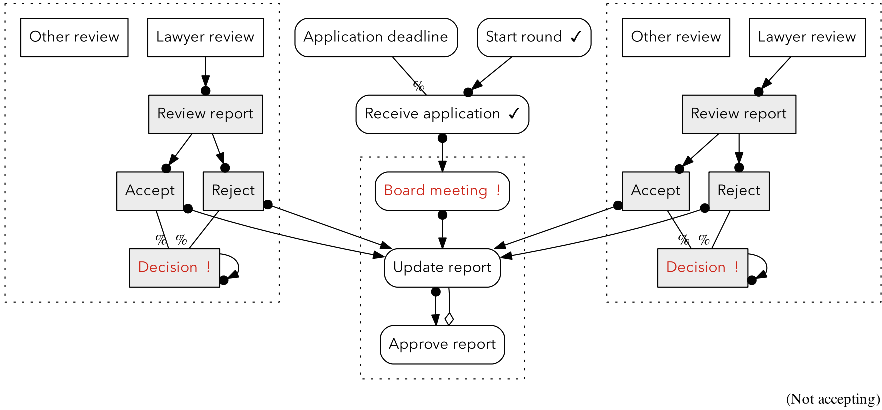


Fig. 8. After spawning a *second* subprocess in Fig. 7

board meeting commences, it is enough to have, in the sub-process definition in Fig. 6 a condition from the local event Review report to a new interface event Board meeting.

4 Foundations

In this section, we review the formal theory of DCR graphs, then formally introduce their refinement and their generalisation to Hi-DCR graphs.

We distinguish between events and labels. In a single workflow, the same label may occur multiple times. For instance, the label "Review report" occurs twice in Fig. 8. We accommodate such multiplicity by considering *events* (the boxes), as distinct from their *label* (the text in the boxes). When the distinction between events and labels does not matter, we use the words interchangeably. For instance, in Fig. 1 and 3 we speak of "the event Board meeting", since the label Board meeting uniquely identifies an event. To simplify the presentation we will let the labelling of events remain implicit in the formal definitions.

Definition 4.1 (DCR Graph [8]). *A* DCR graph *is a tuple* $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ *where*

- E *is a finite set of (labelled)* events*, the nodes of the graph.*
- R *is the edges of the graph. Edges are partioned into five kinds, named and drawn as follows: The* conditions $(\rightarrow\!\bullet)$*,* responses $(\bullet\!\rightarrow)$*,* milestones $(\rightarrow\!\diamond)$*,* inclusions $(\rightarrow\!+)$*, and* exclusions $(\rightarrow\!\%)$*.*
- M *is the* marking *of the graph. This is a triple* $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *of sets of events, respectively the previously executed* (Ex)*, the currently pending* (Re)*, and the currently included* (In) *events.*

When G is a DCR graph, we write, e.g., $\mathsf{E}(G)$ *for the set of events of G, as well as, e.g.,* $\mathsf{Ex}(G)$ *for the executed events in the marking of G.*

Notation. For a binary relation $\rightarrow \subseteq X \times Y$ we write "$\rightarrow Z$" for the set $\{x \in X \mid \exists z \in Z.\ x \rightarrow z\}$, and similarly for "$Z \rightarrow$". For singletons we usually omit the curly braces, writing $\rightarrow e$ rather than $\rightarrow \{e\}$.

With the definition of DCR graphs and notation in place, we define the dynamic semantics of a DCR graph. First, the notion of an event being *enabled*, ready to execute.

Definition 4.2 (Enabled events). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *be a DCR graph, with marking* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$*. We say that an event* $e \in \mathsf{E}$ *is* enabled *and write* $e \in \mathsf{enabled}(G)$ *iff (a)* $e \in \mathsf{In}$*, (b)* $\mathsf{In} \cap (\rightarrow\!\bullet\, e) \subseteq \mathsf{Ex}$*, and (c)* $\mathsf{In} \cap (\rightarrow\!\diamond\, e) \subseteq \mathsf{E} \backslash \mathsf{Re}$.

That is, enabled events (a) are included, (b) their included conditions already executed, and (c) have no included milestones with an unfulfilled responses.

Definition 4.3 (Execution). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *be a DCR graph with marking* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$*. Suppose* $e \in \mathsf{enabled}(G)$*. We may* execute *e obtaining the* resulting *DCR graph* $(\mathsf{E}, \mathsf{R}, \mathsf{M}')$ *with* $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ *defined as follows.*

1. $\mathsf{Ex}' = \mathsf{Ex} \cup e$
2. $\mathsf{Re}' = (\mathsf{Re} \setminus e) \cup (e \bullet\!\!\rightarrow)$
3. $\mathsf{In}' = (\mathsf{In} \setminus (e \rightarrow\!\%)) \cup (e \rightarrow\!+)$

That is, to execute an event e one must: (1) add e to the set Ex of executed events. (2) Update the currently required responses Re by first removing e, then adding any responses required by e. (3) Update the currently included events by first removing all those excluded by e, then adding all those included by e.

Definition 4.4 (Transitions, runs, traces). *Let G be a DCR graph. If* $e \in \mathsf{enabled}(G)$ *and executing e in G yields H, we say that G has* transition on e to *H and write* $G \longrightarrow_e H$*. A* run *of G is a (finite or infinite) sequence of DCR graphs* G_i *and events* e_i *such that:* $G = G_0 \longrightarrow_{e_0} G_1 \longrightarrow_{e_1} \ldots$
A trace *of G is a sequence of labels of events* e_i *associated with a run of G. We write* $\mathsf{runs}(G)$ *and* $\mathsf{traces}(G)$ *for the set of runs and traces of G, respectively*

Not every run or trace represents an acceptable execution of the graph: We need also that every response requested is eventually fulfilled or excluded.

Definition 4.5 (Acceptance). *A run* $G_0 \longrightarrow_{e_0} G_1 \longrightarrow_{e_1} \ldots$ *is* accepting *iff for all n with* $e \in \mathsf{In}(G_n) \cap \mathsf{Re}(G_n)$ *there exists* $m \geq n$ *s.t. either* $e_m = e$*, or* $e \notin \mathsf{In}(G_m)$*. A* trace is accepting *iff it has an underlying run which is.*

Acceptance tells us which workflows a DCR graph accepts, its *language.*

Definition 4.6 (Language). *The* language *of a DCR graph G is the set of its accepting traces. We write* $\mathsf{lang}(G)$ *for the language of G.*

We now know enough to formalise the first DCR graph we saw.

Example 4.7. The DCR graph of Fig. 1 and 2 has events a, s, r, b labelled Application deadline (a), Start round (s), Receive application (r), and Board meeting (b). It has relation R given by $\rightarrow\!\% = \{(a, r)\}$, $\rightarrow\!+ = \emptyset$, $\rightarrow\!\bullet = \{s, r\}$, $\bullet\!\!\rightarrow = \{r, b\}$ and $\rightarrow\!\diamond = \emptyset$. We can find a run of this DCR graph in Fig. 2: $B_1 \longrightarrow_s B_2 \longrightarrow_r B_3 \longrightarrow_r B_4 \longrightarrow_a B_5 \longrightarrow_b B_6$
Here, B_1, B_2 are accepting, whereas B_3–B_5 have b pending and so are not.

4.1 Composition and Interfaces

Composition of DCR graphs was originally introduced in [10].

Definition 4.8 (Composition of DCR graphs). *The composition $G \mid H$ of DCR graphs G and H is defined by taking the union of all components. Formally: $G \mid H = (\mathsf{E} \cup \mathsf{E}', \mathsf{R} \cup \mathsf{R}', (\mathsf{Ex} \cup \mathsf{Ex}', \mathsf{Re} \cup \mathsf{Re}', \mathsf{In} \cup \mathsf{In}'))$*
The empty *or* zero *DCR graph, $\mathbf{0}$, is the unique DCR graph with no events.*

Composition does not in itself give "refinement" in the classical sense: in DCR graphs, even if G, H share events and labels, the language of $G \mid H$ might actually be *larger* than either G or H. The following definition helps narrow down what are "good" compositions.

Notation. The *projection* of a sequence σ to a set E is obtained by removing every element of σ not in E. For instance, the projection of $\sigma = AABCABC$ to $E = \{A, C\}$ is $\sigma|_E = AACAC$. We lift projection to sets of sequences pointwise.

Definition 4.9 (Refinement). H refines G *iff* $\big(\mathsf{lang}(G \mid H)\big)|_{\mathsf{E}(G)} \subseteq \mathsf{lang}(G)$.

To help establish refinements, we have the following theorem, which states that a DCR graph G is refined by a DCR-graph H if they have no shared event which may be included or excluded unilaterally by H.

Theorem 4.10. *H refines G if in H shared labels are associated only with shared events, and for all $f \in \mathsf{E}(G) \cap \mathsf{E}(H)$ and $e \in \mathsf{E}(H)$ we have that:*

1. *If $e \rightarrow\!\%_H f$ then also $e \rightarrow\!\%_G f$,*
2. *if $e \rightarrow\!+_H f$ then also $e \rightarrow\!+_G f$,*
3. $\mathsf{Ex}(H) \cap \mathsf{E}(G) \subseteq \mathsf{Ex}(G)$,
4. $\mathsf{In}(H) \cap \mathsf{E}(G) \subseteq \mathsf{In}(G)$.

Conditions (1) and (2) mean that H cannot unilaterally include or exclude shared events; conditions (3) and (4) that the marking of H does not change the inclusion- or execution-state of shared events.

Example 4.11. As an example, taking G to be the DCR graphs of Fig. 1 and H to be the one of Fig. 3, then both G, H fulfil the criteria of Theorem 4.10. Thus, we can be sure that when we compose them to obtain $G \mid H$ in Fig. 4, their local behaviour is preserved: the valid execution orders of Application deadline, Start round, Receive application, and Board meeting in Fig. 4 are all also valid according to Fig. 1.

4.2 Hi-DCR graphs

Towards DCR-graphs with sub-processes, we need first DCR graphs with interfaces, i-DCR graphs.

Definition 4.12 (i-DCR graph). *An* i-DCR graph *is a tuple $G = (\mathsf{E}, \mathsf{R}, \mathsf{M}, \mathsf{I})$ such that $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ is a DCR graph and $\mathsf{I} \subseteq \mathsf{E}$. Events $\mathsf{L} = \mathsf{E} \setminus \mathsf{I}$ are* local *events. An i-DCR graph inherits notions of enabled events, execution, and zero from its underlying DCR-graph.*

We note that once we can speak of *execution*, we have using Definitions 4.4, 4.5, and 4.6 also definitions of transistions, runs, traces, acceptance, and language.

The point of the interface I is to allow us to choose which events should fuse with similar events in composition, and which should be considered private. To avoid fusing of private events, we must sometimes employ renamings.

Definition 4.13 (Freshness, compatibility). *If G, H are i-DCR graphs we say an that G is* fresh *for H iff* $\mathsf{L}(G) \cap \mathsf{E}(H) = \emptyset$. *We say that they are* compatible *iff they are both fresh for the other. We say that G, H are equivalent if they are structurally identical up to the choice of local events.*

The composition of i-DCR graphs guarantees that local events of compatible graphs do not overlap.

Definition 4.14 (i-DCR composition). *The composition $G \mid H$ of i-DCR graphs G, H is defined as for DCR graphs, taking interfaces of the combined graph to be* $\mathsf{I}(G) \setminus \mathsf{L}(H) \cup \mathsf{I}(H) \setminus \mathsf{L}(G)$.

For compatible i-DCR graphs, this definition is equivalent to taking simply $\mathsf{I} \cup \mathsf{I}'$.

Definition 4.15 (Hi-DCR). *A Hi-DCR graph is a tuple $G = (\mathsf{E}, \mathsf{R}, \mathsf{M}, \mathsf{I}, \mathsf{S})$ where S is a map taking events to Hi-DCR graphs and $(\mathsf{E}, \mathsf{R}, \mathsf{M}, \mathsf{I})$ is the* underlying *i-DCR graph $G|_\iota$ of G. An event e of G is* enabled *in G iff it is in $G|_\iota$.*

Note that if one wants an event e to *not* spawn any sub-process, one simply maps it to sub-process definition to zero, i.e., takes $\mathsf{S}(e) = \mathbf{0}$.

Definition 4.16 (Hi-DCR execution). *Suppose e is an event of the Hi-DCR graph G, that $e \in \mathsf{enabled}(G)$ and that $\mathsf{S}(e) = H$. Then to execute e in G:*

1. *Pick some H' equivalent to H but fresh for G.*
2. *Execute e in $G \mid H'$ (considered a DCR graph) to obtain H.*

That is, if $G \mid H' = (\mathsf{E}, \mathsf{R}, \mathsf{M}, \mathsf{I}, \mathsf{S})$, we execute e in $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ obtaining $(\mathsf{E}, \mathsf{R}, \mathsf{M}')$, then declare the execution of e in $G \mid H'$ to be $(\mathsf{E}, \mathsf{R}, \mathsf{M}', \mathsf{I}, \mathsf{S})$.

Example 4.17. The notion of i-DCR graph and the definition of Hi-DCR graph execution explains formally why the event Approve decisions is *not* duplicated when a subprocess is spawned between Fig. 7 and 8: it is an interface event, and so is fused in the composition that happens when new sub-processes are spawned. The event Review report in the sub-process, on the other hand, *is* duplicated: It is local, and because execution of Hi-DCR graphs choose fresh names for local events during spawning, it is duplicated.

Theorem 4.18. *Hi-DCR graphs are strictly more expressive then DCR Graphs and therefore also strictly more expressive then ω-regular languages.*

Proof. Hi-DCR graphs conservatively extend DCR graphs, which are known to express exactly ω-regular languages [14]. But in Fig. 8, every time we execute Receive application we are will execute at least one Accept or Reject. This requires *counting* Receive application which is impossible for ω-regular languages.

5 Implementation

For experimentation, we have implemented a prototype tool for working with Hi-DCR graphs. This tool features a simulation engine capable of executing transitions, and of dynamic re-configuration using both unconstrained composition (Definition 4.8) and refinement (Definition 4.9). For finite-state graphs, the tool can do also basic model-checking tasks, such as finding a path to dead-lock, termination, acceptance, or some event being enabled. Whereas in the other sections of this paper, we represented DCR graphs graphically, as figures produced by the tool, the tool inputs a textual representation. As an example of that representation, consider again Fig. 6. Its equivalent textual representation is below. All events are interface events by default; local events are specified by prefixing them with a slash '/' (line 11-13). Events can also be prefixed '+', '%', and '!', (line 5) indicating that they are initially included, excluded, respectively pending. For convenience, the language allows both chaining of events and relations (line 2–5) as well as relating multiple things (line 7).

```
"Other review"                            1
"Lawyer review"                           2
-->* "Review report"                      3
-->* ("Accept" "Reject")                  4
-->% !"Decision"                          5
"Decision" -->* "Decision"                6
("Accept" "Reject") -->% "Decision"       7
("Accept" "Reject")                       8
 *--> "Update report"                     9
                                          10
/("Other review" "Lawyer review"          11
  "Review report"                         12
  "Accept" "Reject" "Decision")           13
```

The tool uses Graphviz [5] to automatically produce diagrams. The diagrams in the present paper were all so generated. The tool is implemented in F# and runs on the major platforms. The executable and source code can be found at [4].

6 Conclusion

In this paper we first demonstrated how DCR Graphs can be used for incremental, declarative design of processes, by introducing a notion of compositional refinement that guarantees language inclusion (with respect to the labels of events present in the original process) and thus preserves compliance for accepting executions. We then used these techniques to introduce Hi-DCR Graphs, a conservative extension of DCR graphs, which allows events to spawn sub-processes. The extensions have been presented and motivated using as an example an abstraction of a real-world case supplied by our industry partner, Exformatics A/S. We provided a formal semantics for Hi-DCR Graphs and demonstrated by example that they are more expressive then ω-regular languages. Finally we reported on a prototype implementation of Hi-DCR Graphs which supports a programming-like syntax, automatic visualisation, simulation, and rudimentary model-checking.

6.1 Future Work

While the notion of refinement introduced in the present paper preserves compliance for accepting executions, it may in fact introduce errors such as livelocks and deadlocks. The simplest example would be to refine a process with a DCR graph containing a single, included (local) event having itself as condition and being initiatlly required as response. In [15] it is shown how adaptations can be verified for safety and liveness, by relying on the map from DCR Graphs to Büchi-automata [16], which is further mapped to Promela and verified in the SPIN model-checker. However, as demonstrated in [15], this approach is not very efficient. We are therefore currently investigating techniques for more efficient verification of DCR Graphs in the presense of adaptations.

As was mentioned in the related work section the development of Hi-DCR Graphs brings us closer to the GSM notation, and we plan to use this work in the future as a basis for formal mappings between GSM and DCR Graphs models.

The question of the precise expressive power of Hi-DCR graph remains open. We conjecture that they are Turing-equivalent. This points to another relevant question, namely how to constrain Hi-DCR graphs to allow safety and liveness guarantees. An obvious possible constraint would be to bound the number times each sub-process can be spawned by a constant. Because of the formalization of spawning based on composition, it follows that this constrains the model to the expressiveness of standard DCR Graphs, that is, to Büchi-automata.

Formal expressive power aside, in practice many idiomatic constructs, like the “disjunctive responses” used in Fig. 8 could be formalised as derived constructs in their own right, potentially making them more accessible to end-users, much like DECLARE is formalised in terms of LTL. Similarly, other questions regarding the usability of the approach will be investigated in future studies through empirical investigations undertaken in cooperation with our industrial partners and end-users.

Acknowledgments. We gratefully acknowledge fruitful discussions with Rik Eshuis. The work is supported by grant VELUX 33295, 2014-2017 and the Danish Agency for Science, Technology and Innovation.

References

1. van der Aalst, W.M.P., Pesic, M., Schonenberg, H., Westergaard, M., Maggi, F.M.: Declare. Webpage (2010), http://www.win.tue.nl/declare/
2. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
3. Carbone, M., Hildebrandt, T., Perrone, G., Wasowski, A.: Refinement for transition systems with responses. In: FIT. EPTCS, vol. 87, pp. 48–55 (2012)
4. Debois, S.: DCR exploration tool v.6. IT University of Copenhagen (2014), http://www.itu.dk/research/models/wiki/index.php/DCR_Exploration_Tool
5. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz - open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–484. Springer, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45848-4_57

6. Groefsema, H., Bucur, D.: A survey of formal business process verification: From soundness to variability. In: Proceedings of the Third International Symposium on Business Modeling and Software Design, pp. 198–203 (2013), http://www.cs.rug.nl/ds/uploads/pubs/groefsema-bmsd.pdf
7. Hildebrandt, T., Marquard, M., Mukkamala, R.R., Slaats, T.: Dynamic condition response graphs for trustworthy adaptive case management. In: Demey, Y.T., Panetto, H. (eds.) OTM 2013 Workshops. LNCS, vol. 8186, pp. 166–171. Springer, Heidelberg (2013)
8. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES. EPTCS, vol. 69, pp. 59–73 (2010)
9. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: Arbab, F., Sirjani, M. (eds.) FSEN 2011. LNCS, vol. 7141, pp. 343–350. Springer, Heidelberg (2012)
10. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Safe distribution of declarative processes. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 237–252. Springer, Heidelberg (2011)
11. Hull, R., et al.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles (Invited talk). In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
12. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.P.: Runtime verification of LTL-based declarative process models. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 131–146. Springer, Heidelberg (2012)
13. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
14. Mukkamala, R.R.: A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs. Ph.D. thesis, IT University of Copenhagen (June 2012)
15. Mukkamala, R.R., Hildebrandt, T., Slaats, T.: Towards trustworthy adaptive case management with dynamic condition response graphs. In: EDOC, pp. 127–136. IEEE (2013)
16. Mukkamala, R.R., Hildebrandt, T.: From dynamic condition response structures to büchi automata. In: TASE, pp. 187–190. IEEE Computer Society (2010)
17. Mukkamala, R.R., Hildebrandt, T., Tøth, J.B.: The resultmaker online consultant: From declarative workflow management in practice to ltl. In: EDOCW, pp. 135–142. IEEE Computer Society (2008)
18. Object Management Group BPMN Technical Committee: Business Process Model and Notation, version 2.0, http://www.omg.org/spec/BPMN/2.0/PDF
19. Reijers, H., Mendling, J., Dijkman, R.: On the usefulness of subprocesses in business process models. BPM Reports 1003, Eindhoven (2010)
20. Slaats, T., Mukkamala, R.R., Hildebrandt, T., Marquard, M.: Exformatics declarative case management workflows as DCR graphs. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 339–354. Springer, Heidelberg (2013)
21. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
22. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)
23. Zugal, S., Soffer, P., Pinggera, J., Weber, B.: Expressiveness and understandability considerations of hierarchy in declarative business process models. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Wrycza, S. (eds.) BPMDS 2012 and EMMSAD 2012. LNBIP, vol. 113, pp. 167–181. Springer, Heidelberg (2012)