5

The DCR Workbench: Declarative Choreographies for Collaborative Processes

Søren Debois and Thomas T. Hildebrandt

Department of Computer Science, IT University of Copenhagen,
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

Abstract

The *DCR Workbench* is an online tool for simulation and analysis of collaborative distributed processes specified as *DCR graphs*. The Workbench is a robust and comprehensive implementation of DCR graphs, providing concrete syntax, specification by refinement, visualisation, simulation, static analysis, time analysis, enforcement, declarative subprocesses, data dependencies, translation to other declarative models, and more. This chapter introduces the Workbench and, through the features of the Workbench, surveys the DCR formalism. The Workbench is available on-line at `http://dcr.tools`.

5.1 Introduction

Citizens, businesses and public organisations increasingly rely on distributed business processes. Many such processes involve at the same time information systems, humans and mechanical artefacts, and are thus highly unpredictable. Moreover, such processes are constantly evolving due to advances in technology, improvement in business practices, and changes in legislation.

In this climate of distribution and continuous change, the traditional vision of verifying a system once and for all against a final formal description has little hope of realisation. Instead, we need tools and techniques for describing, building, and analysing systems of continuously changing distributed collaborative processes. Dynamic Condition Response graphs, DCR graphs, is a formal model developed in response to this need.

Developed through a series of research projects, DCR graphs today stands on three pillars: a substantial body of academic publications on both case studies [8, 10, 15, 17, 20, 38] and formal aspects [2, 6, 7, 12–14, 16, 18, 21–25, 29–31, 34, 37]; the DCR Workbench, implementing most major advances of the formalism (the subject of this chapter); and a commercial adaptive case-management system developed by independent vendor Exformatics A/S [9, 11, 15, 19, 23, 28].

Declarative process notations such as DCR graphs, DECLARE [33, 40] and GSM [26] generally support *specification and analysis* of requirements, whereas imperative notations such as Workflow Nets [1] and BPMN [32] generally support *implementation* of requirements. DCR graphs have the advantage of serving as *both* the specification of requirements *and* the run-time representation of a process instance, which can be adapted dynamically if the requirements change.

The DCR Workbench is a comprehensive tool for modelling with DCR graphs and analysing DCR models. The Workbench serves the dual purposes of being a communication and teaching tool, used both in classroom settings and in discussions with industry, as well as a test-bed for experimentation with new analysis and variants.

This chapter gives an introduction to DCR graphs in general and the Workbench in particular. As we shall see, the Workbench implements a majority of published DCR graph variants and analysis methods, as well as some work-in-progress experimental additions and algorithms that have yet to be published. Through the features of the Workbench, the chapter also provides a survey of the state-of-the-art of DCR graphs variants, their technical properties, and their published analysis methods and algorithms.

5.1.1 History of the DCR Workbench

DCR graphs were introduced in 2010 [18,29] by Thomas Hildebrandt and his group at the ITU. Soon after, Danish vendor of adaptive case-management solutions, Exformatics A/S, entered into a long-term collaboration with the ITU group; a collaboration which continues to this day. The continued financial support and interest of Exformatics A/S has been instrumental in the development of the formalism.

DCR graphs were implemented repeatedly as the formalism evolved. Notably, an early implementation created by industrial PhD Tijs Slaats in collaboration with Exformatics A/S [37] eventually grew into that company's current commercial DCR tool [9, 15, 28], available at `dcrgraphs.net`. In

2013 this tool was solidifying into a commercial offering. While the backing of commercial vendor was *extremely* helpful to DCR graph, the Exformatics tool was becoming too heavyweight for quick academic experiments. Accordingly, the ITU group in 2013 commenced development of a nimbler implementation. This effort was spearheaded by Søren Debois and became the DCR Workbench of the present chapter.

The two tools have different goals: Exformatics' offering is aimed at non-expert commercial users and emphasises stability and usability. Conversely, the DCR Workbench is aimed at academics and prioritises ease-of-experimentation overall. This division has so far been productive: sufficiently good ideas implemented in the Workbench has later been re-implemented by Exformatics in their commercial offering [8–10, 28].

The Workbench made its first appearance in a research paper in 2014 [12], and its first appearance in industry collaborations in 2015 [9]. Subsequently, the Workbench has provided implementation and examples for most major developments of the formalism [2, 6, 12–14, 16].

The DCR Workbench is implemented in F# [39], using the WebSharper library [5] to derive server- and client-side components from the same F# code base. The choice of implementation language and platform is no accident: On the one hand, F# is very well-suited to manipulating formal models; on the other, the web-based platform makes the Workbench *immediately* available to interested researchers: all it takes is a browser.

5.1.2 The DCR Workbench

The DCR Workbench is available at

```
http://dcr.tools/2017chapter
```

This URL leads to a special page supporting this chapter with the collection of examples used on the following pages. We encourage the reader to visit this page and actively try out the examples presented in the remainder of this chapter as he progresses through the it.

Overview In Section 5.2, we introduce a running example, and in Section 5.3, we recall DCR graphs. In Section 5.4 we introduce basic modelling, simulation and analysis of DCR graphs in the Workbench. In Section 5.5 we construct models by *refinement*; in Section 5.6 we discuss timed models; in Section 5.7 we talk about subprocesses; and in Section 5.8 data. In Section 5.9, we mention briefly other tools in the Workbench, before concluding in Section 5.10.

5.2 Running Example

As a running example we consider a stylised mortgage loan application process distilled from real-life cases [9,13]. 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 case worker should not spend time assessing the application if the documentation has not been collected or the budget is underspecified. The caseworker decides if the appraisal can be entirely statistical, i.e., carried out without physical inspection, or if it requires an on-site appraisal. For reasons of cost efficiency, only one appraisal should be carried out.

5.3 Dynamic Condition-Response Graphs

In this section, we recall DCR graphs [6,9,14,15,18,29,37]. We begin by an informal walkthrough, followed by a formal development in Section 5.3.4.

DCR graphs constitute a declarative modelling notation describing at the same time a process and its run-time state. The core notation comprises *labelled events*, *event states*, and five possible *relations* between events. The relations govern: (a) how executability of one event depend on the state of another, and (b) how execution of one event updates the states of another.

5.3.1 Event States

The *event state* consists of three booleans: The *executed, included,* and *pending* states of the event.

- The *executed* state simply registers whether the event has been previously executed (an event may execute more than once). It is updated to true whenever the event executes. It is never updated to false.
- The *included* state indicates whether the event is included, i.e. relevant for the process. Being included is a prerequisite for an event to execute.

- The *pending* state indicates whether the event is required to eventually execute (or become not included).

We give events and initial states for the running example in Figure 5.1. Except Request new budget, which becomes relevant only when a budget has been submitted, all events are included. The Assess loan application and Submit budget events are pending: they are required to complete the process.

5.3.2 Relations

Each pair of events may be related by one of five different relations. Relations regulate (a) which events may execute in a given graph (condition, milestone) and (b) the effect of executing an event (inclusion, exclusion, response).

We give a full DCR model of the running example[1] in Figure 5.2.

Conditions. A condition $e \rightarrow\!\bullet\, f$ causes the target activity f to be not executable whenever the source activity e is included (its “included” state is true) and has not been previously executed (its “executed” state is false). E.g., in Figure 5.2, we must execute Collect documents before Assess loan application can be executed.

Milestones. A milestone $e \rightarrow\!\diamond\, f$ causes the target activity f to be not executable whenever the source activity e is included and pending (its “included” and “pending” states are true). In Figure 5.2, whenever Submit budget is pending, Assess loan application is prevented from executing.

Event	Role	Initial state
Collect documents	Caseworker	
Budget screening approve	Intern	
Request new budget	Intern	Excluded
Submit budget	Customer	Pending
On-site appraisal	Mobile consultant	
Statistical appraisal	Caseworker	
Assess loan application	Caseworker	Pending

Figure 5.1 Events and initial states (marking) for the mortgage application process. Where nothing else is indicated, the initial state of an event is not executed, included, and not pending.

[1]This graph is in fact the output of the DCR Workbench visualiser; we describe in Section 5.4.2 exactly how the visualiser represents event state.

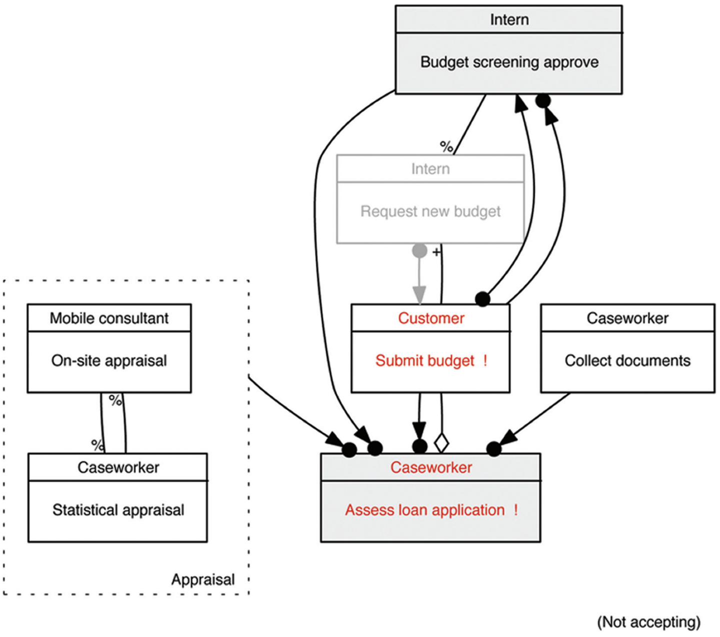


Figure 5.2 DCR graph modelling the mortgage application process.

Responses. A response e •→ f causes the target activity f to be pending (its "pending" state is true) whenever the source activity e is executed. In Figure 5.2, when an applicant executes Submit budget, we require a subsequent screening: there is a response from Submit budget to Budget screening approve.

Inclusions and exclusions. An inclusion e →+ f resp. an exclusion e →% f causes the "included" state of the target activity f to be true resp. false when the source activity e is executed. In Figure 5.2, the activity Submit budget includes the activity Request new budget.

5.3.3 Executing Events

Enabled events are the ones which have their "included" state true, and which do not have their execution prohibited by a condition or milestone as indicated above. Conditions or milestones from excluded events *do not* disable their target events. E.g., in Figure 5.2, both On-site appraisal and Statistical appraisal are conditions for Assess loan application, but once one executes, the other is excluded and thus no longer required for Assess loan application to execute.

Executing an enabled event in a DCR graph updates event states as indicated by inclusion, exclusion, and response relations. Conceptually, we can think of the execution as producing a new DCR graph with the same relations but with updated event states.

The *denotation* of a DCR model is the set of (finite and infinite) sequences of event labels, corresponding to a sequence of such event executions, where at each step, every pending event is eventually executed or excluded at some later step. We refer to such sequences as *accepting traces*. It follows that for finite accepting traces, in the final state, no event is pending and included.

We consider potential traces for Figure 5.2.

- $s_0 = \langle \mathsf{Collect\ documents}, \mathsf{Assess\ loan\ application} \rangle$. This sequence intuitively corresponds to assessing the loan application *without* getting a budget and appraising the property. After Collect documents, the event Assess loan application is not enabled, so s_0 is not a trace.
- $s_1 = \langle \mathsf{Collect\ documents}, \mathsf{Submit\ budget} \rangle$. This sequence is a trace, but not an accepting one, since Assess loan application is pending and included in the final state. It follows that s_1 sequence is not part of the denotation of Figure 5.2.
- $s_2 = \langle \mathsf{Collect\ documents}, \mathsf{Submit\ budget}, \mathsf{Budget\ screening\ approve}, \mathsf{Statistical\ appraisal}, \mathsf{Assess\ loan\ application} \rangle$ is an accepting trace.

Notice that between the notions of enabledness and accepting trace, DCR graphs express both permissions and obligations. We return to expressiveness of DCR graphs in Section 5.7 below.

5.3.4 Formal Development

Definition 1 (DCR Graph [18])**.** A *DCR graph*, ranged over by G, is a tuple $(\mathsf{E}, \mathsf{R}, \mathsf{M}, \ell)$ where

- E is a finite set of (labelled) *events*, the nodes of the graph.
- R is the edges of the graph. Edges are partitioned 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.
- ℓ is a labelling function assigning to each $e \in \mathsf{E}$ a label comprising an activity name and a set of roles.

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

Notation. Let $R \subseteq X \times Y$ be a relation. For $y \in Y$ we take $Ry = \{x \in X \mid (x,y) \in R\}$; dually for $x \in X$ we take $xR = \{y \in Y \mid (x,y) \in R\}$. We use this notation for relations, e.g.,, $(\rightarrow\!\bullet\ e)$ is the set of events that are conditions for e.

Definition 2 (Enabled events)**.** Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M}, \ell)$ be a DCR graph, with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. An event $e \in \mathsf{E}$ is *enabled*, written $e \in \mathsf{enabled}(G)$, 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) have their included conditions executed, and (c) have no included milestone with an unfulfilled responses.

Definition 3 (Execution)**.** Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M}, \ell)$ 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 DCR graph $G' = (\mathsf{E}, \mathsf{R}, \mathsf{M}', \ell)$ 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; and (3) remove from In those events excluded by e, then adding those included by e.

Technically, the operational semantics of a DCR graph is the labelled transition system where states are graphs and transitions are executions.

Definition 4 (Transitions)**.** 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.

The denotation of a DCR graph is the set of *accepting* finite and infinite traces allowed by its operational semantics.

Definition 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 $m > n$ 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 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 conclude this Section by noting that by Definitions 2 and 3, because the set of events is finite, both the set of enabled events and the result of executing an event are computable in polynomial time.

5.4 Modelling with the Workbench

A typical configuration of the Workbench can be seen in Figure 5.3. The Workbench is divided into panels. In the configuration in Figure 5.3, we see the Visualiser, Parser and Activity Log panels.

The Workbench maintains at all times a current DCR graph and a current trace. Each panel allow the user to interact with this current graph and current trace. A few panels also maintain a DCR graph of their own.

Panels are dynamic: The user is free to remove panels by clicking “close” in the lower-left corner of a panel; or to add panels by selecting a new panel in the “Add a new panel” section of the Workbench panel. At the time of writing, the Workbench implements 22 different panels.

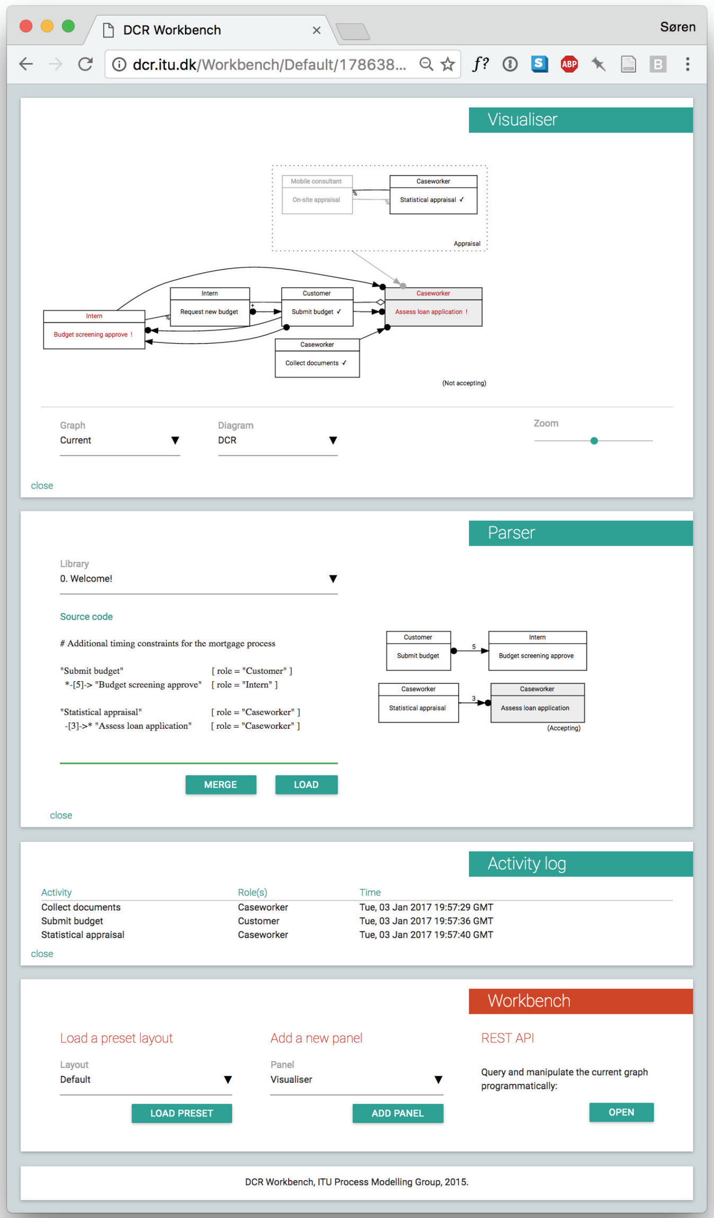
When working with the Workbench, it is customary to have several panels open; e.g., a visualiser and one or more analysis panels. The Workbench panel contains a selection of seven pre-made such panel configurations called “presets”. These presets are accessible through the left-hand “Load a preset” section of the Workbench panel.

Finally, the Workbench can function as a process engine, making some of its functionality available programmatically as a REST interface; see the right-hand “REST API” section.

5.4.1 Inputting a Model: The Parser Panel

The parser panel allows input of DCR graphs as plain text. The parser accepts programs written according to the grammar of Figure 5.4.

As an example program, consider the abridged variant of our running example given in Figure 5.6; the corresponding input program is listed in Figure 5.5. The Workbench accepts such source programs as input, producing visualisations automatically. (Visualisations in this chapter was so produced.)



Graph: Current

Diagram: DCR

Library: 0. Welcome!

```
# Additional timing constraints for the mortgage process

"Submit budget"                     [ role = "Customer" ]
  *-[5]-> "Budget screening approve"  [ role = "Intern" ]

"Statistical appraisal"             [ role = "Caseworker" ]
  -[3]->* "Assess loan application"   [ role = "Caseworker" ]
```

Activity	Role(s)	Time
Collect documents	Caseworker	Tue, 03 Jan 2017 19:57:29 GMT
Submit budget	Customer	Tue, 03 Jan 2017 19:57:36 GMT
Statistical appraisal	Caseworker	Tue, 03 Jan 2017 19:57:40 GMT

Layout: Default

Panel: Visualiser

Query and manipulate the current graph programmatically:

DCR Workbench, ITU Process Modelling Group, 2015.

Figure 5.3 The DCR Workbench (`http://dcr.tools`).

```
⟨expressions⟩ ::= ⟨expressions⟩ ⟨relation⟩
                | ⟨expressions⟩ ⟨relation⟩ WHEN ⟨condition⟩
                | ⟨expressions⟩ ⟨event⟩
                | ⟨expressions⟩ GROUP { ⟨expressions⟩ }
                | ⟨empty⟩

⟨relation⟩    ::= ⟨event⟩ ⟨arrow⟩ ⟨event⟩
                | ⟨event⟩ ⟨arrow⟩ ⟨relation⟩

⟨arrow⟩       ::= -->* | --<> | -->+ | -->% | *-->
                | -[ ⟨num⟩ ]->*                          (Timed condition)
                | *-[ ⟨num⟩ ]->                          (Timed response)

⟨event⟩       ::= % ⟨event⟩
                | / ⟨event⟩
                | ! [⟨num⟩] ⟨event⟩
                | : [⟨num⟩] ⟨event⟩
                | ( ⟨event⟩+ )
                | ⟨identifier⟩ [⟨meta⟩] [⟨sub⟩]

⟨meta⟩        ::= [ [⟨identifier⟩] [ ⟨string⟩ = ⟨string⟩ ]* ]

⟨sub⟩         ::= [ ? ] { ⟨expressions⟩ }

⟨condition⟩   ::= (* ... *)
```

Figure 5.4 EBNF definition of the language recognised by the Parser panel.

```
( "Collect documents"          [ role = Caseworker ]      3
  "Submit budget"              [ role = Customer   ] )    4
  -->*                                                    5
  !"Assess loan application" [ role = Caseworker ]        6
```

Figure 5.5 Source code for the core process.

The concrete syntax specifies events and relations such as "condition from A to B" with expressions such as "`A -->* B`". An event state is specified by prefixing an event with modifiers such as `!` or `%`. We see this on line 6 in Figure 5.5. If the event occurs more than once in the program, it is sufficient to prefix the modifier only once. We specify roles by adding a role tag to the event. We see this on line 3 in Figure 5.5. More than one role may be added; in general, the same tag may be added multiple times.

It is occasionally convenient to relate more than one event at the same time. In the present case, Assess loan application needs conditions on both

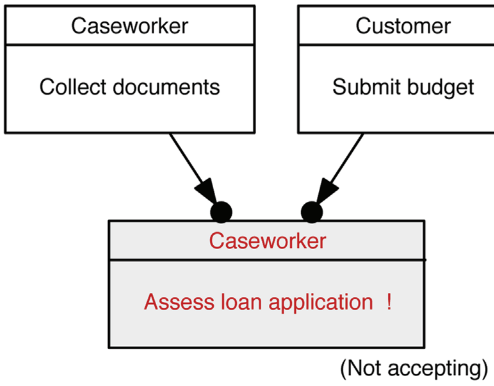


Figure 5.6 Visualisation of core process of Figure 5.5.

Collect documents and Submit budget. We specify these conditions concisely by enclosing the latter two in parenthesis, as seen on line 3–4.

As the user types in the parser, a preview of the graph being input is presented on the right. The input graph substitutes the current graph and resets the current trace when the user clicks “Load”.

The parser also understands the XML format output by Exformatics A/S commercial `http://dcrgraphs.net` tool [15, 38].

5.4.2 Visualisation and Simulation: The Visualiser and Activity Log Panels

The Visualiser, top-most in Figure 5.3, displays a visualisation of the current DCR graph. In Figure 5.3, the chosen visualisation is simply the graph layout; alternatively, the underlying transition system may be shown (see below).

The visualisation of the core application process (Figure 5.5) is reproduced in Figure 5.6. The visualiser represent events as boxes, labelled by the activity of the event (centre) and the role or participant executing that activity (top). E.g., the top-left box represents an activity Collect documents which is carried out by a Caseworker.

Activities are coloured according to their state: grey background is not currently executable (Assess loan application in Figure 5.6), red label! with an exclamation mark is pending (ditto); “greyed out” boxes are excluded events (Request new budget in the original Figure 5.2); and finally, executed events have a tick mark after their action label, (Submit budget in Figure 5.3).

Simulation The visualiser allows executing events by clicking. E.g., to execute Submit budget, simply click it. This will extend the current trace with that execution, and replace the current graph with the one obtained by applying the updates to event state resulting from the execution of Submit budget (in this case, setting "executed" of that event true). Use the browser's back buttons to revert to a previous state.

The Activity Log panel, third from the top in Figure 5.3, displays the current trace, analogous to the way the visualiser displays the current graph.

State-space enumeration As mentioned in Definition 4, a DCR graph gives rise to a labelled transition system (LTS), where states are markings and transitions are labelled event executions. The visualiser can be configured to render a visualisation of the state space of the DCR graph rather than the DCR graph itself, through the drop-down button on the left of the panel. The visualiser highlights the current run in that LTS. The visualisation of the full LTS of the full mortgage application process of Figure 5.2.

The visualiser was originally reported in [12], with the transition system generator following in [9].

5.5 Refinement

We proceed to construct step-wise the full mortgage process application by *refinement* [6, 14]. We begin with the core process of Figures 5.5 and 5.6. We first add the process fragments for budget submission and screening given in Figures 5.8 and 5.9.

The Workbench supports step-wise refinement: by using in the parser the "Merge" button rather than the "Load" button. Whereas "Load" replaces the current graph and sets the current trace to empty, the "Merge" button preserves both, replacing the global current graph with (graph) union $G \oplus H$ of the current graph G and the parser's current local graph H.

To refine the core process by the budget fragment, we make sure that the core process is the current graph, then enter the fragment (Figure 5.9) in the parser, and click "Merge". The result is the graph in Figure 5.10. As can be seen, the resulting process is close to the full running example in Figure 5.2, except the process fragment for appraising the property is missing.

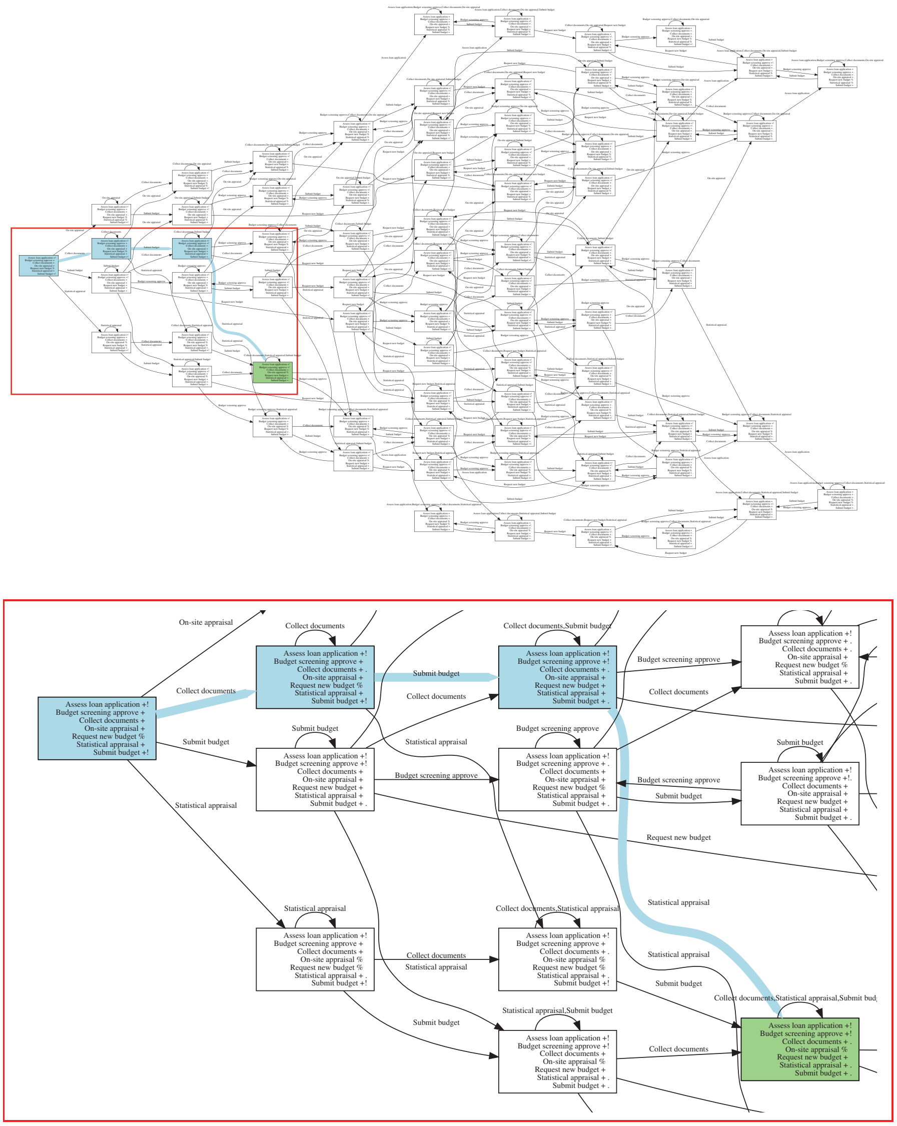


Figure 5.7 Transition system of the full mortgage application process (top), with the red box expanded for readability (bottom).

Repeating the Merge procedure with the process fragment in Figure 5.11 adds the missing bits and leaves us with exactly Figure 5.2—this is how the examples for the present chapter has been constructed.

```
!"Submit budget"
  -->* "Budget screening approve"   [ role = Intern ]
  -->* "Assess loan application"    [ role = Caseworker ]

"Submit budget"
  --<> "Assess loan application"

"Submit budget"
  -->+ %"Request new budget"        [ role = "Intern" ]
  *--> "Submit budget"              [ role = "Customer" ]
  *--> "Budget screening approve"
  -->% "Request new budget"
```

Figure 5.8 Budget process fragment.

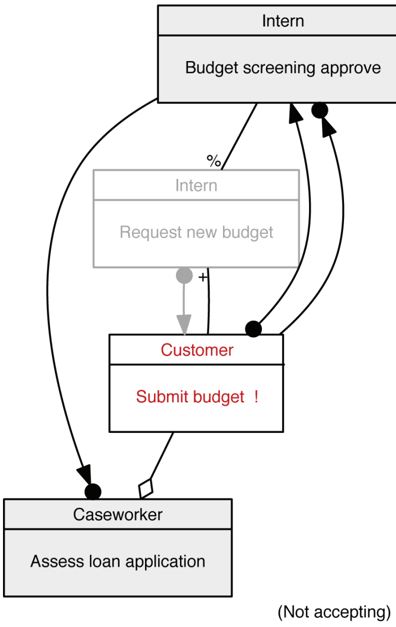


Figure 5.9 Visualisation of the budget process fragment of Figure 5.8.

Not every such merge preserves the language of the original graph. Exclusions may void conditions, giving the merged graph behaviour not present in the original graph, even when restricting attention to only the events of that graph. As a very simple example, consider the two graphs $G = \mathsf{a} \rightarrow\!\bullet\ \mathsf{b}$ and $H = \mathsf{c} \rightarrow\!\%\ \mathsf{a}$. The union $G \oplus H = \mathsf{a} \rightarrow\!\bullet\ \mathsf{b}, \mathsf{c} \rightarrow\!\%\ \mathsf{a}$ has the trace $\langle \mathsf{c}, \mathsf{b} \rangle$; even if we dismiss the new event c, G could not by itself exhibit the trace $\langle \mathsf{b} \rangle$.

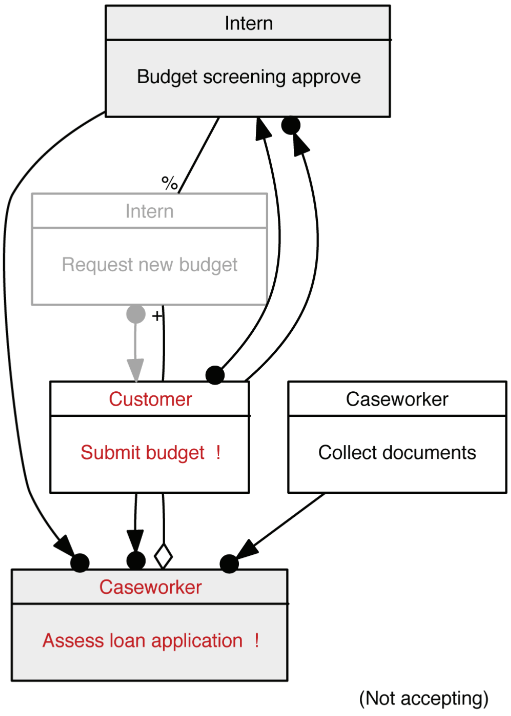


Figure 5.10 Visualisation of the core process (Figures 5.5 and 5.6) refined by the budget fragment (Figures 5.8 and 5.9).

```
Group "Appraisal" {
    "On-site appraisal"           [ role = "Mobile consultant" ]
    "Statistical appraisal"       [ role = "Caseworker" ]
}

"Statistical appraisal" -->% "On-site appraisal"
"On-site appraisal" -->% "Statistical appraisal"

"Appraisal"
 -->* "Assess loan application" [ role = "Caseworker" ]
```

Figure 5.11 Appraisal process fragment.

This situation was investigated in detail in previous work [6, 14], where a sufficient condition for a refinement to be language preserving in the above sense was established in a much richer setting. For the present notion of DCR graphs, it is sufficient to require that the refining graph does not exclude or include any events of the original graph.

Notation. Given a sequence s and an alphabet Σ, write $s \downharpoonright_\Sigma$ for the largest sub-sequence s' of s s.t. $s'_i \in \Sigma$; e.g, if $s = AABC$ then $s \downharpoonright_{A,C} = AAC$.

Definition 7. Given DCR graphs G and H, we say that H *is a refinement of* G iff $\mathsf{lang}(H) \downharpoonright_{l(\mathsf{E}(G))} \subseteq \mathsf{lang}(G)$.

That is, the language of H restricted to the labels used by events in G must be a subset of the language of G. We can now state the following Proposition [6, Theorem 43]:

Proposition 1. *Let G and G' be DCR processes such that for every $e \in \mathsf{E}(G)$, there is no relation $x \rightarrow\% \, e$ or $x \rightarrow+ \, e$ in G'. Then the graph union $G \oplus G'$ is a refinement of G.*

If the Workbench has current graph G and the parser has graph G' *not* satisfying (a published [6, 14] generalisation of) the conditions of Proposition 1, the Parser issues a warning and requires confirmation before merging.

DCR refinement was originally suggested in [17, 24] and worked out comprehensively in [6, 14]. The Workbench implements this latter mechanism.

5.6 Time

The Workbench supports the extension of DCR graphs with time [2, 23]. Time is modelled discretely by a special action tick modelling the passage of time; conditions are augmented with an optional delay, $\mathsf{e} \xrightarrow{k}\!\bullet \mathsf{f}$, and responses with an optional deadline $\mathsf{e} \bullet\!\xrightarrow{k} \mathsf{f}$. Intuitively, the delay in the timed condition requires that at least k ticks have passed after the last execution of e before f may execute; dually, the deadline in the timed response requires that at most k ticks pass after the last execution of e before f must execute.

The former requirement makes it possible to have *timelocks*, i.e., situations where, say, f must execute, but is not allowed to. As a very simple example, consider the DCR graph $G = \mathsf{e} \xrightarrow{3}\!\bullet \mathsf{f}, \mathsf{e} \bullet\!\xrightarrow{2} \mathsf{f}$. In this graph, once e executes, *at least* 3 ticks must pass before f *can* execute because of the condition delay, but *at most* 2 ticks may pass before f *must* execute because of the response deadline. After the sequence $\langle \mathsf{e}, \mathsf{tick}, \mathsf{tick} \rangle$, the graph is said to be *time-locked*: Time cannot advance without a constraint being violated.

For our running example, suppose that (a) the initial screening of the customer's budget must be completed within 5 days, and (b) that the final assessment of the loan application must wait a 3-day "grace period" after a statistical appraisal (in order to prevent caseworkers from doing overly optimistic statistical appraisals). We model these constraints as a timed DCR graph directly using a timed response and condition in Figures 5.12 and 5.13.

Timed DCR graphs are still in finite state [23], but deciding time-lock freedom naively by exploring in the state space is infeasible. Recent research [2] established a sufficient condition for a graph to be time-lock–free and gave a generic "enforcement mechanism" for time-lock–free graphs, that is, a device which monitors the progression of time and a DCR graph and proactively causes events to execute to avoid missing deadlines.

The Workbench implements time as defined in [23], and time-lock–analysis and enforcement as defined in [2].

```
"Submit budget"                         [ role = "Customer" ]
  *-[5]-> "Budget screening approve"    [ role = "Intern" ]

"Statistical appraisal"                 [ role = "Caseworker" ]
  -[3]->* "Assess loan application"     [ role = "Caseworker" ]
```

Figure 5.12 Additional timing constraints for the mortgage application process in Figure 5.2.

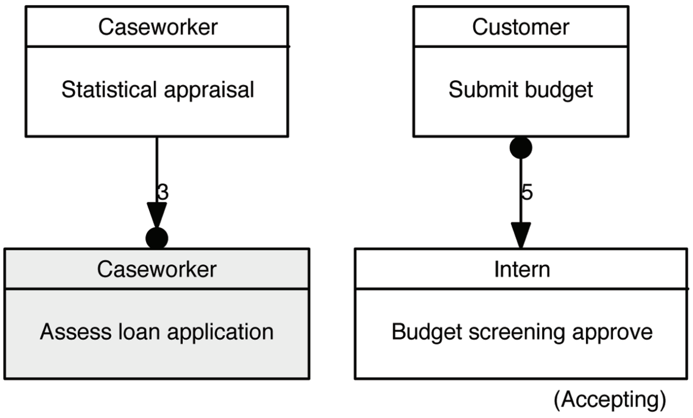


Figure 5.13 Visualisation of additional timing constraints for the mortgage application process in Figure 5.2.

5.7 Subprocesses

It may happen that a customer *during the application process* applies for pre-approval of an expected increase in property value due to, e.g., on-going kitchen remodellings. In this case, the caseworker must assess the limit extension before deciding on the mortgage application itself. At the caseworker's discretion, an intern may or may not collect bank statements from the customer for the limit extension assessment; however, collecting that statement requires the customer's explicit consent.

Such limit extensions in practice may happen several times during a mortgage application due to, e.g., expanded scope of a kitchen remodelling project. Thus, the limit extension fragment is a *subprocess*: A process that may be added to the main process when necessary, and possibly repeatedly.

Note that since subprocesses may be added repeatedly, each such addition must duplicate the events of the subprocess. This situation is akin to bound names under replication being duplicated in in the π-calculus [35]. The subprocess may contain both events local to the subprocess, *bound* events, and references to events of the containing graph. The former are indicated syntactically with a / prefix as seen in lines 6–8.

The Workbench supports subprocesses; we add the above limit extension process in Figures 5.14 and 5.15. Note that in visualisation, the subprocess is not visible until it has been expanded once.

The visualisation shows the triggering event Apply for limit extension—singled out as spawning a subprocess by the ⊞ following contemporary business process notations, e.g., BPMN [32]. The bound events in a subprocess are shown with round corners and inside a dashed box[2].

```
"Assess loan application"                                      3
                                                               4
"Apply for limit extension"          [ role = Customer ]       5
    { /"Assess limit extension"      [ role = Caseworker ]     6
      /"Collect consent"             [ role = Intern ]         7
       -->* /"Collect bank statement"  [ role = Intern ]       8
      "Submit budget"                [ role = Customer ]       9
       --<> !"Assess limit extension"  [ role = Caseworker ]   10
       -->* "Assess loan application"  [ role = Caseworker ]   11
    }                                                          12
    *--> "Submit budget"                                       13
```

Figure 5.14 Additional subprocess constraints (credit limit extension) for the full mortgage application process of Figure 5.2.

[2]If a subprocess adds new global events—as opposed to bound ones—these would appear with square corners inside the box.

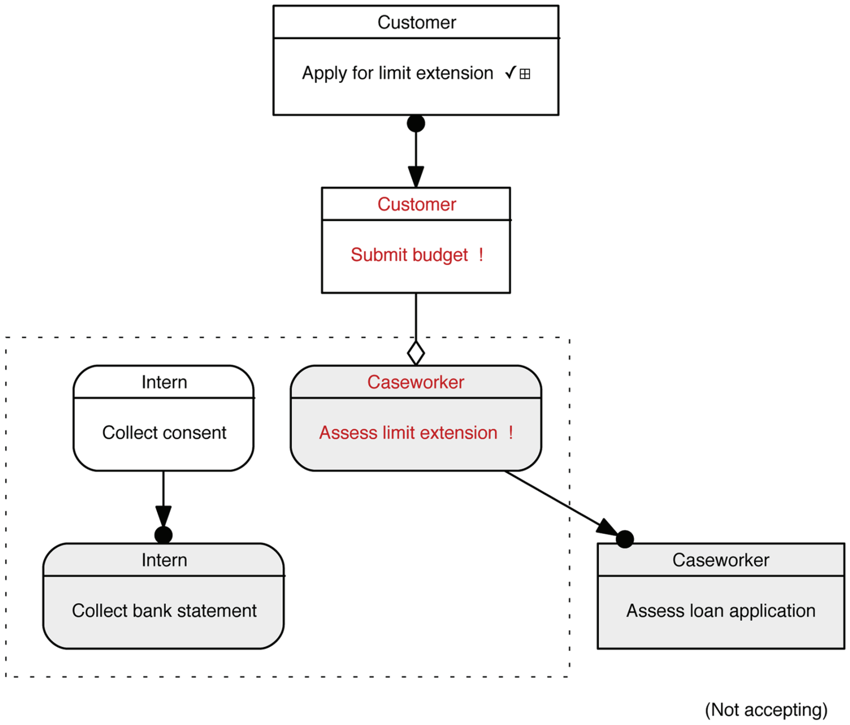


Figure 5.15 Visualisation of additional subprocess constraints (credit limit extension) for the full mortgage application process of Figure 5.2; after one execution of Apply for limit extension.

Subprocess semantics is based on graph union; if $G = \dots, \mathsf{e}\{H\}$ contains a subprocess-spawning event e, then executing e will form the graph $G \oplus H$, then apply the effects of e. Note the importance of bound names here: If events in H were not bound, then repeated instantiation of the subprocess would not change the graph, i.e., $G \oplus H \oplus H = G \oplus H$. This equation emphatically does not hold under the current semantics, where events in H may be bound, and thus replicated. In the running example, executing Apply for limit extension *twice* would result in all rounded-box event to be replicated *twice*—we invite the reader to try this out in the Workbench.

Adding subprocesses and bound events significantly increase the expressive power of DCR graphs [6, Theorem 9]:

Theorem 1. *DCR graphs express the union of regular and ω-regular languages. Graphs with subprocesses and bound events are Turing complete.*

In particular, while event-reachability and refinement is decidable for plain and timed DCR graphs, they are undecidable for DCR graphs with subprocesses and bound events.

DCR graphs was extended with a notion of subprocesses and bound events in [12], followed by an investigation of expressive power in [6, 14]. The Workbench implements subprocesses in the sense of [12].

5.8 Data

The Workbench augments DCR graphs with a notion of "input events" and relations conditional on data. Suppose for our running example that if the amount applied for in a credit limit extension exceeds EUR 10.000, then having a bank statement becomes a condition for evaluating the loan application.

Technically, this is accomplished by: (a) adding the option of inputting a value when a subprocess is spawned; and (b) adding data-guards on select relations. When the subprocess is instantiated, condition on the input value dictates whether, e.g., the relation e $\rightarrow\bullet$ f takes effect or not.

We extend the running example with such a conditional condition in Figure 5.16. Note that since the "variable" associated with an event is simply the name of the event, it becomes convenient to specify separately the name and label of the event. This is done in line 3, where it is specified that the event limit has label Apply for a limit extension and role Customer. (Without an explicit specification, the Workbench identifies event and label.)

The visualiser does not show data-guarded relations, and the formal semantics of DCR graphs with data have yet to be published.

```
limit["Apply for limit extension" role = Customer ] ?          3
    { /"Assess limit extension"                                4
      /"Collect consent" -->* /"Collect bank statement"        5
        "Submit budget"                                        6
            --<> !"Assess limit extension"                     7
            -->* "Assess loan application"                     8
      "Collect bank statement"                                 9
        -->* "Assess loan application"                         10
        when "$limit > 10000"                                  11
    }                                                          12
    *--> "Submit budget"                                       13
```

Figure 5.16 Alternative subprocess-with-data constraints (credit limit extension) for the full mortgage application process of Figure 5.2.

An interesting application of data is that of specifying user-input "forms" (think Web forms) via DCR graph, associating with each event in a graph an input field in such a form. This idea was implemented in the Actions panel and later realised [27] in collaboration with Exformatics A/S.

5.9 Other Panels

We mention here briefly a few panels of the Workbench not discussed so far.

1. An encoding from DCR to the GSM model [26] was defined in recent research [16]; the Workbench implements this encoding an outputs CMMN [4] XML.
2. Notions of concurrency and independence of DCR events, following the standard notion of labelled asynchronous transition systems [3, 36], was recently investigated [13]. The Workbench' Concurrency panel implements these notions, automatically identifying concurrent events.
3. Work on applications of DCR in practical settings suggested a need for simplifying process models when presented to end users [9]. The Workbench contains a number of such simplifying views, most notably a "swimlane" view of the current trace in the panel of the same name, and a mechanism for projecting a graph in various ways to sub-graphs of interest.

5.10 Conclusion

We have given an introduction to DCR graphs, and an overview of the DCR Workbench. Since the Workbench implements most major variations of DCR graphs, this Chapter has also served as a survey of the state-of-the-art of DCR graphs as modelling and analysis tool for continuously changing distributed collaborative processes.

The Workbench has been instrumental for scientific research, providing a test-bed for quick experiments with new ideas; for teaching, providing students the opportunities for hands-on learning of abstract concepts; and for collaborations with industry and knowledge transfer to industry. In all these instances, providing a practical platform on which to demonstrate sometimes difficult-to-communicate abstract concepts helps to cement the reality and applicability of DCR as a modelling methodology. In particular, the Workbench has paved the way for academic results [9, 12, 18, 21, 27] to find their way to implementation in commercial tools [8, 28].

We invite the reader to use the Workbench for research and teaching. It is available at `http://dcr.tools`.

References

[1] Wil M. P. van der Aalst. Verification of Workflow Nets. In *Proc. of the 18th Int. Conf. on Application and Theory of Petri Nets, ICATPN*, pages 407–426, 1997.

[2] David A. Basin, Søren Debois, and Thomas T. Hildebrandt. In the nick of time: Proactive prevention of obligation violations. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016*, pages 120–134. IEEE Computer Society, 2016.

[3] Marek Bednarczyk. *Categories of asynchronous systems*. PhD thesis, U. Sussex, 1988.

[4] BizAgi and others. Case Management Model and Notation (CMMN), v1, May 2014. OMG Document Number formal/2014-05-05, Object Management Group.

[5] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. *Composing Reactive GUIs in F# Using WebSharper*, pages 203–216. Springer, 2011.

[6] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Replication, refinement & reachability: Complexity in Dynamic Condition-Response graphs. *Acta Informatica*, 2017. Accepted for publication.

[7] Søren Debois, Thomas T. Hildebrandt, Paw Høsgaard Larsen, and Kenneth Ry Ulrik. Declarative process mining for DCR graphs. In *SAC '17*, 2017. Accepted for publication.

[8] Søren Debois, Thomas T. Hildebrandt, Morten Marquard, and Tijs Slaats. Bridging the valley of death: A success story on danish funding schemes paving a path from technology readiness level 1 to 9. In *SER&IP 2015*, pages 54–57. IEEE, 2015.

[9] Søren Debois, Thomas T. Hildebrandt, Morten Marquard, and Tijs Slaats. Hybrid process technologies in the financial sector. In *BPM 2015, Industry track*, volume 1439 of *CEUR Workshop Proceedings*, pages 107–119. CEUR-WS.org, 2015.

[10] Søren Debois, Thomas T. Hildebrandt, Morten Marquard, and Tijs Slaats. The DCR graphs process portal. In *BPM 2016*, volume 1789 of *CEUR Workshop Proceedings*, pages 7–11. CEUR-WS.org, 2016.

[11] Søren Debois, Thomas T. Hildebrandt, and Lene Sandberg. Experience report: Constraint-based modelling and simulation of railway emergency response plans. In *ANT 2016 / SEIT-2016*, volume 83 of *Procedia Computer Science*, pages 1295–1300. Elsevier, 2016.

[12] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Hierarchical declarative modelling with refinement and sub-processes. In *BPM 2014*, volume 8659 of *LNCS*, pages 18–33. Springer, 2014.

[13] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Concurrency and asynchrony in declarative workflows. In *BPM 2015*, volume 9253 of LNCS, pages 72–89. Springer, 2015.

[14] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM 2015*, pages 143–160, 2015.

[15] Søren Debois, Thomas T. 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 Computer Society, 2014.

[16] Rik Eshuis, Søren Debois, Tijs Slaats, and Thomas T. Hildebrandt. Deriving consistent GSM schemas from DCR graphs. In *ICSOC 2016*, volume 9936 of *Lecture Notes in Computer Science*, pages 467–482. Springer, 2016.

[17] Thomas T. Hildebrandt, Morten Marquard, Raghava Rao Mukkamala, and Tijs Slaats. Dynamic condition response graphs for trustworthy adaptive case management. In *OTM 2013 Workshops*, volume 8186 of *LNCS*, pages 166–171. Springer, 2013.

[18] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In *PLACES 2010*, volume 69 of *EPTCS*, pages 59–73, 2010.

[19] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Declarative modelling and safe distribution of healthcare workflows. In *FHIES 2011*, volume 7151 of *LNCS*, pages 39–56. Springer, 2011.

[20] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Designing a cross-organizational case management system using dynamic condition response graphs. In *EDOC 2011*, pages 161–170. IEEE Computer Society, 2011.

[21] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Nested dynamic condition response graphs. In *FSEN 2011, Revised Selected Papers*, volume 7141 of *Lecture Notes in Computer Science*, pages 343–350. Springer, 2011.

[22] Thomas T. Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Safe distribution of declarative processes. In *SEFM 2011*, volume 7041 of *LNCS*, pages 237–252. Springer, 2011.

[23] Thomas T. Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats, and Francesco Zanitti. Contracts for cross-organizational workflows as timed dynamic condition response graphs. *J. Log. Algebr. Program.*, 82(5–7):164–185, 2013.

[24] Thomas T. Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats, and Francesco Zanitti. Modular context-sensitive and aspect-oriented processes with dynamic condition response graphs. In *FOAL 2013*, pages 19–24. ACM, 2013.

[25] Thomas T. Hildebrandt and Francesco Zanitti. A process-oriented event-based programming language. In *DEBS 2012*, pages 377–378. ACM, 2012.

[26] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. Heath III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculín. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *WS-FM 2010*, LNCS. Springer, 2010.

[27] Morten Marquard, Søren Debois, Tijs Slaats, and Thomas T. Hildebrandt. Forms are declarative processes! In *BPM 2016 Industry Track (to appear)*, 2016.

[28] Morten Marquard, Muhammad Shahzad, and Tijs Slaats. Web-based modelling and collaborative simulation of declarative processes. In *BPM 2015*, volume 9253 of LNCS, pages 209–225. Springer, 2015.

[29] Raghava Rao Mukkamala. *A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs*. PhD thesis, IT University of Copenhagen, 2012.

[30] Raghava Rao Mukkamala and Thomas T. Hildebrandt. From dynamic condition response structures to büchi automata. In *TASE 2010*, pages 187–190. IEEE, 2010.

[31] Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Tijs Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *EDOC 2013*, pages 127–136. IEEE Computer Society, 2013.

[32] Object Management Group BPMN Technical Committee. Business Process Model and Notation, version 2.0, 2013. `http://www.omg.org/spec/BPMN/2.0.2/PDF`

[33] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: full support for loosely-structured processes. In *EDOC 2007*, pages 287–300, 2007.

[34] Søren Debois and Tijs Slaats. The analysis of a real life declarative process. In *CIDM 2015*. IEEE, 2015. Accepted for publication.

[35] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[36] M. W. Shields. Concurrent machines. *Computer Journal*, 28(5):449–465, 1985.

[37] Tijs Slaats. *Flexible Process Notations for Cross-organizational Case Management Systems*. PhD thesis, IT University of Copenhagen, January 2015.

[38] Tijs Slaats, Raghava Rao Mukkamala, Thomas T. Hildebrandt, and Morten Marquard. Exformatics declarative case management workflows as DCR graphs. In *BPM '13*, volume 8094 of *LNCS*, pages 339–354. Springer, 2013.

[39] Don Syme, Jack Hu, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, et al. The F# 4.0 language specification. Technical report, 2005.

[40] Wil M. P. van der Aalst and Maja Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM 2006*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.