# Dynamic Condition Response Graphs as Foundation for Event-based Languages and Systems

Position Paper

Søren Debois
IT University of Copenhagen
Rued Langgaardsvej 7
2300 Copenhagen S
Denmark
debois@itu.dk

Thomas Hildebrandt
IT University of Copenhagen
Rued Langgaardsvej 7
2300 Copenhagen S
Denmark
hilde@itu.dk

## ABSTRACT

We discuss the use of Dynamic Condition Response (DCR) Graphs as a formal foundation and run-time for event-based systems. DCR Graphs constitute an event-based process notation derived from event-structures and developed in collaboration with our industrial partner Exformatics. Several extensions to the core theory, including dynamic creation of sub processes, data, I/O, analysis and distribution techniques have been developed by the authors and co-workers, and implemented by the first author in the DCR workbench available for experimentation at tiger.itu.dk, a web-based interactive research prototype. Some of the features have been transferred to the industrial cloud-based DCR Graphs design and simulation tool developed by Exformatics and available at DCRGraphs.net

## Keywords

Declarative, Event-driven programming, DCR, I/O, Data

## 1. INTRODUCTION TO DCR GRAPHS

Event-driven programming is gaining increasing interest and find natural applications in the programming of systems driven by user input or events from sensors, such as user-interfaces, embedded systems, robots, web applications and case management systems.

Typically, event-based programming languages are derived from other programming paradigms by adding a way of handling events: Functional reactive programming [3] is derived from functional programming (and the data flow paradigm) essentially by considering functions on streams. For imperative programming languages, event-driven programming is performed using event-handlers and binders.

In the present paper, we discuss the use of Dynamic Condition Response (DCR) Graphs as a formal foundation and run-time for event-driven programming. DCR Graphs are derived from *event structures* [14], a formal model for concurrency, causality and conflict based on a partial ordered set of events and a conflict relation. A core DCR Graph is a directed graph of labelled events connected by edges expressing temporal and causal relations between events, and a marking of each event providing its state as three booleans $(h, i, r)$. The first boolean indicates whether the event has (h)appened (at least once), the second whether it is currently (i)ncluded in the process and the last whether it is (r)equired to happen in the future (or be excluded) in order for the execution to be accepting. We will identify the event and its label, unless we need several events with the same label.

One finds a similar kind of state in forms: Some fields in a form may already have been filled out, either by the user or by a default value. Some fields may be required to be filled out (or changed, i.e. because there is something wrong with the data filled in) in order for the form to be in a completed state. And some fields may be dynamically included or excluded depending on the actions taken.

### 1.1 Grant application process example

Core DCR Graphs have only five different kinds of edges: Condition (`-->*`), Response (`*-->`), Exclusion (`-->%`), Inclusion (`-->+`) and Milestone (`--<>`) . To define a simple grant application process, we may declare five events, `Apply`, `Accept`, `Reject`, `Payout` and `Close`. We can then declare that `Apply` must happen before `Accept` and `Reject` by *condition* relations `Apply-->*Accept` and `Apply-->*Reject`. We can declare that `Accept` and `Reject` are mutually exclusive by *exclude* relations `Reject-->%Accept`, `Accept-->%Reject`. We may declare by `%Payout` that the event `Payout` is initially excluded, and by the *include* relation `Accept-->+Payout` and *response* relation `Accept*-->Payout` that `Payout` will be respectively included and required as response if `Accept` happens. Finally, we can declare that `Payout` excludes itself (so we only payout once) and `Close` excludes every event (by having exclude relations to all events, including itself, and thereby closing the case). By the milestone relation `Payout--<>Close` we declare that `Close` is not enabled if `Payout` is presently required to be executed. The full DCR Workbench process (with notational shorthands) thus looks as follows:

```
Apply -->*(Accept Reject)
Accept -->% Reject
Reject -->% Accept
```


*REBLS* '15

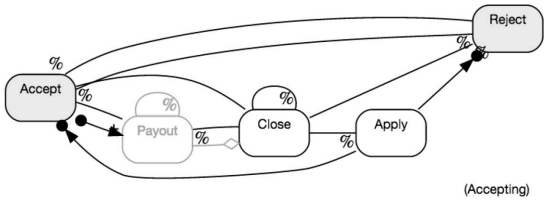


**Figure 1: DCR Graph for Grant application**

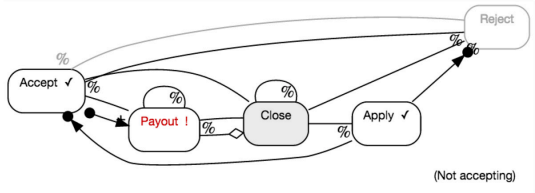


**Figure 2: DCR Graph after executing `Apply` and `Accept`**

```
%Payout
Accept -->+Payout -->%Payout
Accept *-->Payout --<>Close
Close -->%(Apply Reject Accept Payout Close)
```

It can be tested in the DCR Workbench at tiger.itu.dk by pasting the code into the text box in the Parser panel. Pressing load will display the graph in the Visualiser panel as shown in Fig. 1 below, with disabled events being greyed out, excluded events being dashed, required events being marked with an exclamation mark and executed events being indicated by a checkmark.

Events can then be executed by clicking the activity in the panel. After executing `Apply` and `Accept`, the graph looks as shown in Fig. 2. One can use the back/forward button at the browser to step backwards and forwards in the execution.

## 1.2 Key features of DCR Graphs

A key feature of DCR Graphs is the absence of explicit sequencing: if nothing else is declared, events can happen in any order and any number of times. This promotes flexibility and distributed concurrent execution. Indeed, we have shown in [2] that it is possible to statically determine which events can be executed concurrently, which can be used for distributing events on different processors. It also makes it easy to merge in (or remove) events dynamically, which can be used for run-time adaptation [1, 11].

Another key feature of DCR Graphs is the ability to distinguish between possible and required execution of events, in that only events that are initially required or become required because of a response relation must be executed (or be excluded) before the execution ends. Both features are important in the support of flexible case management processes for knowledge workers such as physicians at hospitals. We believe that these characteristics can be relevant for concurrent event-based and reactive programming in general, in particular for distributed processes. In [2] we


**Figure 3: `Apply` shown as text field.**

describe briefly how events (or groups of events) can be implemented as distributed REST services, that only need to interact with (and possibly lock) their immediately related events during execution. This feature can also be tested in the DCR Workbench.

In its use for supporting and partially automating workflows, events are also assigned a number of roles. In the commercial run-time engine of Exformatics, roles can be assigned to both users and the execution engine. In the latter case, the event will be automatically executed by the engine if it is both enabled and required.

## 1.3 Data, I/O and dynamic sub processes

A real funding application process of course needs data, e.g. the actual grant proposal and the amount applied, and some I/O (for receiving and sending data to other processes). To date, DCR Graphs have been applied in the Exformatics process engine to control the order of activities in workflows, while data has been treated as side-effects of the activities and only been visible in the model by allowing the relations being conditioned on guards. That is, the meaning of a guarded relation is that it is only present if the guard evaluates to true. Also, an event can be bound to a database trigger, and thereby making changes to data detectable in the process as events. However, if DCR Graphs should be extended to an event-based programming language, data and I/O should be included as native elements in the notation.

In the DCR workbench we have prototyped data and I/O, allowing one to declare `Apply` as an *input* event taking two values stored in variables title and amount by adding the declaration `Apply?(title,amount){}`. The Activity panel will now show input events as text fields as shown in Fig. 3.

Note that title and amount will be implicitly declared as global variables that can be accessed by any output event (and viewed in the workbench by adding a Data panel). In particular, `Accept`, `Reject` and `Payout` can be declared as *output* events by

```
Reject<<"We are sorry to reject "++title>>
Accept<<"We are happy to accept "++title>>
Payout<<"Transfer "++amount>>
```

One can see the values of the input and output events in the History panel of the DCR workbench. We do not have space for the formal semantics here, but it is worth noting that the expressions in the output events are evaluated every time the event is executed. This means, that if the `Apply` action is repeated with new values, the data will change. Also, the use of the value creates an implicit condition from `Apply` to the output events.

The observant reader will have noted the curly braces `{}` following the `Apply` input event above. The DCR workbench supports the extension of DCR Graphs allowing for dynamic creation of sub processes and fresh local events [1]

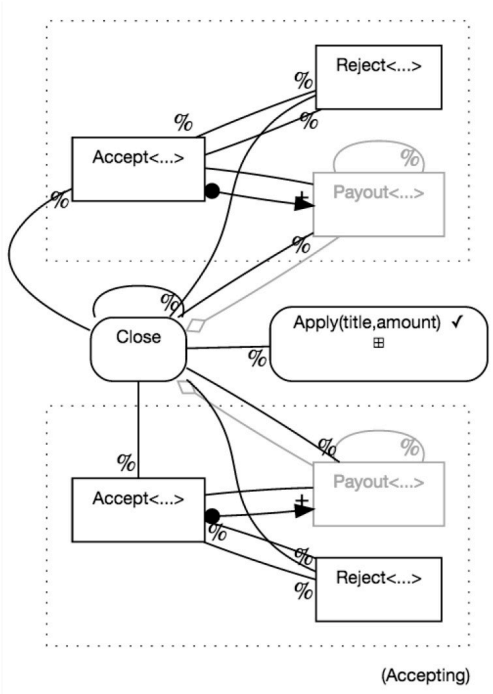


**Figure 4: Graph after two occurrences of `Apply`**

by inserting a graph declaration within the curly braces. So, by defining the process as

```
Close -->%Apply
Apply?(title,amount){
   /(Payout Accept Reject)
   Accept -->% Reject
   Reject -->% Accept
   %Payout
   Accept -->+Payout
   Accept *-->Payout -->% Payout --<> Close
   Close -->%(Reject Accept Payout Close)
   Reject<<"We are sorry to reject "++title>>
   Accept<<"We are happy to accept "++title>>
   Payout<<"Transfer "++amount>>
}
```

one gets a process where only the events `Close` and `Apply` are present initially. Now, every time `Apply` happens, three fresh local events are created because of the declaration `/(Payout Accept Reject)`. Fig. 4 shows the graph after `Apply` has happened twice. The local events are connected to each other by relations, but also to the global `Close` event. In this way, every application can get its own decision and payout, but still all events are excluded by the single close event, and the close event is disabled as long as there are any required payouts. Also, in the current prototype implementation, the expressions in the output events in the sub process are evaluated only once, namely when the process is generated.

## 2. BACKGROUND AND DISCUSSION

The research on DCR Graphs was initiated in [10, 8] as a result of the TrustCare research project jointly with Copenhagen University and the danish company Resultmaker. The theory and applications have subsequently been developed jointly with our new industrial partner Exformatics. On the practical side, the technology is developed to a point where it serves as the foundation for the adaptive case management engine used commercially by Exformatics [13, 7], who also have developed a collaborative cloud based design and simulation tool, DCRgraphs.net. On the academic side, a number of papers have investigated the formal properties of DCR Graphs: run-time adaptation and verification [11], refinement [1] and distributed execution [9] and possible extensions to the model. The theory is prototyped in the web-based DCR workbench at tiger.itu.dk, of which we have exemplified some above. The DCR workbench also include a syntactic representation of DCR Graphs, reminiscent of a early prototype event-based language based on DCR Graphs [6]. One expressiveness result [1, 10] is that the core model allows to express all $\omega$-regular languages, and thus allows to describe both safety and liveness properties. Conversely, any core DCR Graph can be mapped to a Büchi-automaton [12], preserving the set of accepted event traces, meaning that both safety and liveness properties of DCR Graphs can be determined using standard automata based model checking techniques. Concretely, we demonstrate in [11] how DCR Graphs can be mapped to the input language of the SPIN model-checker. A second expressiveness result [1] is, that the extension with dynamically created sub processes and fresh events exemplified above makes the model Turing complete, even without any use of data. Sub processes therefore make many properties, such as the presence of deadlocks and safe refinement as studied in [1] in general undecidable. However, it is possible to identify non-trivial syntactic restrictions that guarantee deadlock freedom and safe refinement [1] even in the presence of dynamic sub processes.

Regarding applications, we have in [5] demonstrated the possibly application of DCR Graphs for describing context-dependent and aspect oriented features, making use of dynamic inclusion and exclusion of events based on context events. DCR Graphs have also been extended with time in [4].

## 3. CONCLUSION AND FUTURE WORK

We have above described the use of the DCR Graphs model and some of its extensions for declarative event-based programming which have been prototyped in the DCR workbench available at tiger.itu.dk. We believe that the DCR Graphs model provides an interesting semantic foundation for event-based programming, in itself as a declarative event-based language and possibly as semantical model for existing event-based programming languages. Indeed, DCR Graphs have already proven their worth in practice as run-time process engine for the Exformatics Adaptive Case Management system which has been in active use by a funding agency for more than a year. As future work we will continue the exploration of DCR Graphs as an event-based programming language, in particular investigating time, modularity, refinement [1] and process inheritance. We will also look into giving semantics to (fragments of) existing event-based programming languages using DCR Graphs. In addition to providing a formal semantics, it could allow for transferring techniques, i.e. for verification, run-time refinement, adaptation and distribution, developed for DCR Graphs to the other languages.

# 4. REFERENCES


[1] S. Debois, T. Hildebrandt, and T. Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM 2015*, number 9109 in LNCS, pages 143–160. Springer, 2015.

[2] S. Debois, T. T. Hildebrandt, and T. Slaats. Concurrency and asynchrony in declarative workflows. In H. R. Motahari-Nezhad, J. Recker, and M. Weidlich, editors, *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2015.

[3] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[4] T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti. Contracts for cross-organizational workflows as timed dynamic condition response graphs. *The Journal of Logic and Algebraic Programming*, 82(5–7):164–185, 2013. Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11).

[5] T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti. Modular context-sensitive and aspect-oriented processes with dynamic condition response graphs. In *Proceedings of the 12th Workshop on Foundations of Aspect-oriented Languages*, FOAL '13, pages 19–24, New York, NY, USA, 2013. ACM.

[6] T. Hildebrandt and F. Zanitti. A process-oriented event-based programming language. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 377–378, New York, NY, USA, 2012. ACM.

[7] T. T. Hildebrandt, M. Marquard, R. R. Mukkamala, and T. Slaats. Dynamic condition response graphs for trustworthy adaptive case management. In *OTM Workshops*, volume 8186 of *LNCS*, pages 166–171. Springer, 2013.

[8] T. T. Hildebrandt and R. R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, volume 69 of *EPTCS*, pages 59–73, 2010.

[9] T. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Safe distribution of declarative processes. In G. Barthe, A. Pardo, and G. Schneider, editors, *SEFM*, volume 7041 of *LNCS*, pages 237–252. Springer, 2011.

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

[11] R. R. Mukkamala, T. Hildebrandt, and T. Slaats. Towards trustworthy adaptive case management with dynamic condition response graphs. In *EDOC*, pages 127–136. IEEE, 2013.

[12] R. R. Mukkamala and T. T. Hildebrandt. From dynamic condition response structures to Büchi automata. In *TASE*, pages 187–190. IEEE Computer Society, 2010.

[13] T. Slaats, R. R. Mukkamala, T. T. Hildebrandt, and M. Marquard. Exformatics declarative case management workflows as DCR graphs. In *BPM*, volume 8094 of *LNCS*, pages 339–354. Springer, 2013.

[14] G. Winskel. *Events in Computation.* PhD thesis, University of Edinburgh, 1980.