Declarative Model Event-reachability via Genetic Algorithms

Tróndur Høgnason and Søren Debois

IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S
Denmark
{thgn,debois}@itu.dk

Abstract. In declarative process models, a process is described as a set of rules as opposed to a set of permitted flows. Oftentimes, such rule-based notations are more concise than their flow-based cousins; however, that conciseness comes at a cost: It requires computation to work out which flows are in fact allowed by the rules of the process. In this paper, we present an algorithm to solve the Activity Reachability problems for the declarative Condition Response (DCR) graphs notation: the problem given a DCR graph and a task, say “Payout reimbursement”, to find a flow allowed by the graph that ends with the execution of that task. Existing brute-force solutions to this problem are generally unhelpful already at medium-sized graphs. We present here a genetic algorithm solving Activity Reachability. We evaluate this algorithm on a selection of DCR graphs, both artificial and from industry, and find that on the real-world examples, the genetic algorithm *invariably* outperforms the brute-force solutions by two orders of magnitude.

Keywords: Genetic algorithm, DCR, Event reachability, Activity reachability

1 Introduction

In the field of business process modelling, there are currently two major modelling paradigms. In declarative notations such as such as DECLARE [2,20], DCR graphs [6,13], GSM [15] and CMMN [18] a model comprises the rules governing process execution. In imperative notations such as Petri- and Workflow nets nets[1,3] and BPMN[19], a model comprises a more explicit representation of the set of possible executions.

The choice between the two paradigms is in essence a trade-off of conciseness for computational requirements. A declarative model may, via its rules, concisely represent a very large number of possible process executions: A DCR graph or DECLARE model may represent a space of process executions exponentially larger than the graph itself. Many interesting problems of DCR graphs are consequently computationally hard [8].

In this paper, we explore the the problem of determining given a model M and an activity a whether there is a sequence of activity executions admitted by M which ends with a. We will study this problem in the setting of DCR graphs, where activities are typically conflated with events. Here, the problem is called "Event reachability" [8,5].

In practice solving this problem is an important model-checking tool for model development. E.g., suppose we are constructing a DCR model of an insurance process involving an event "Payout reimbursement", and we are interested in knowing whether from a given state of the model it is still possible to reach this "Payout reimbursement" event.

Definition 1. *Given a DCR graph G and an event e of G, we say that e* is reachable *from G if there exists a path $e_1, \ldots, e_n$ of event executions admitted by G such that $e_n = e$. We call in this case the path $e_1, \ldots, e_n$ a* witness *for reachability of e.*

Existing solutions to this problem are either brute-force approaches that are impractical on all but the smallest models [12,7,6] or approximations applicable only in special cases of graph structure [5,8]. We investigate in this paper an alternative approximation approach using "Genetic Algorithms" [4] to heuristically find such paths, or give up if a time-bound expires.

We make in this paper the following contributions.

1. We provide a Genetic Algorithm for approximating DCR Event Reachability, including an implementation;
2. We report on a comparison with the existing brute-force solver reported in [7,6] on both artificial and real-world models.

While the genetic algorithm is inferior to the brute-force one on examples constructed specifically to be difficult for the former, it is remarkably effective on the real-world examples: It finds paths where the brute-force algorithm times out, and even when both find a path, the genetic algorithm does so several orders of magnitude faster than the brute-force one. Moreover, at least for initial state models, the real-world examples (with one exception) does not contain unreachable events.

We conclude that the Genetic Algorithm appears to be a quite practical approach to approximating Event Reachability in DCR graphs.

2 Dynamic Condition Response Graphs

DCR Graphs is a declarative notation for modelling processes superficially similar to DECLARE [21,20] or temporal logics such as LTL [22]. One notable difference is that DCR graphs model constraints between so-called events labelled by activities, whereas in DECLARE and LTL, constraints are defined directly between activities. This indirection adds expressive power: DCR graphs model the union of regular and omega-regular languages [10].

The present paper makes two restrictions on DCR graphs: (1) we consider only finite executions, and (2) we assume each event is labelled by a unique activity. None of these restrictions are controversial, the latter is pervasive in the literature on DCR graphs. Because of the latter assumption we do not distinguish between (DCR) "events" and "activities" in the sequel, speaking only of "events".

Definition 2 (DCR Graph [13]). *A* DCR graph *is a tuple* $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ *where*

- E *is a finite set of (activity 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 $(\bullet\!\rightarrow)$*,* inclusions $(\rightarrow\!+)$*,* exclusions $(\rightarrow\!\%)$ *and* milestones $\rightarrow\!\diamond$.
- 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.*

The marking of a DCR graph is its run-time state: it records which events have been executed (Ex), which are currently included (In) in the graph, and which are required to be executed again in order for the graph to be accepting (Re). We first define when an event is enabled.

Notation. 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. In particular, we write $\mathsf{M}(e)$ for the triple of boolean values $(e \in \mathsf{Ex}, e \in \mathsf{Re}, e \in \mathsf{In})$. We write $(\rightarrow\!\bullet\, e)$ for the set $\{e' \in \mathsf{E} \mid e' \rightarrow\!\bullet\, e\}$, write $(e \bullet\!\rightarrow)$ for the set $\{e' \in \mathsf{E} \mid e \bullet\!\rightarrow e'\}$ and similarly for $(e\!\rightarrow\!+)$, $(e\!\rightarrow\!\%)$ and $(\rightarrow\!\diamond\, e)$.

Definition 3 (Enabled events [13]). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *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)* $e \in \mathsf{In}$ *and (b)* $\mathsf{In} \cap (\rightarrow\!\bullet\, e) \subseteq \mathsf{Ex}$ *and (c)* $(\mathsf{Re} \cap \mathsf{In}) \cap (\rightarrow\!\diamond\, e) = \emptyset$*.*

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

Executing an enabled event e of a DCR Graph with marking $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ results in a new marking where (a) the event e is added to the set of executed events, (b) e is removed from the set of pending response events, (c) the responses of e is added to the pending responses, (d) the events excluded by e is removed from included events, and (e) the events included by e is added to the included events.

From this we can define the language of a DCR Graph as all finite sequences of events ending in a marking with no event both included and pending.

Definition 4 (Language of a DCR Graph [13]). *Let* $G_0 = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *be a DCR graph with marking* $\mathsf{M}_0 = (\mathsf{Ex}_0, \mathsf{Re}_0, \mathsf{In}_0)$*. A* trace *of* G_0 *is a finite sequence of events* $e_0, \ldots, e_n$ *such that for* $0 \leq i \leq n$*, (i)* e_i *is enabled in the marking* $M_i = (\mathsf{Ex}_i, \mathsf{Re}_i, \mathsf{In}_i)$ *of* G_i*, and (ii)* G_{i+1} *is a DCR Graph with the same events and* *relations as G_i but with marking* $(\mathsf{Ex}_{i+1}, \mathsf{Re}_{i+1}, \mathsf{In}_{i+1}) = (\mathsf{Ex}_i \cup \{e_i\}, (\mathsf{Re}_i \backslash \{e_i\}) \cup (e_i \bullet\!\!\rightarrow), (\mathsf{In}_i \backslash (e_i \rightarrow\%)) \cup (e_i \rightarrow +))$.

We call such a trace accepting *if for all* $0 \leq i \leq n$ *we have* $\mathsf{Re}_{i+1} \cap \mathsf{In}_{i+1} = \emptyset$. *The* language $\mathsf{lang}(G_0)$ *of* G_0 *is then the set of all such accepting traces.*

A small example graph is shown in Figure 1. This example has five events, A, B, C, D, and *Goal*. The yellow arrows indicate condition relations: because there is a yellow arrow from A to D, A must be executed before we can execute D, and D must be executed before we can execute *Goal*.The red arrows indicate exclusion relations: executing B excludes A, making A not executable and voiding the condition from A to D.

The event *Goal* is not initially enabled for execution: it is prevented by the condition to D. D is similarly not enabled, prevented by the condition to A. One way to reach *Goal* it thus the sequence A, D, *Goal*. Alternatively, one may exploit the exclusion arrow from C to void the condition from D, arriving at the sequence C, *Goal*.

Since an event in a DCR graph can be executed multiple times, there are infinitely many ways to reach the event *Goal*; e.g., the sets of strings characterised by the regular expressions $A^+D(A|D)^+ \mathit{Goal}$ or $(B|A|C)^*C\,\mathit{Goal}$.

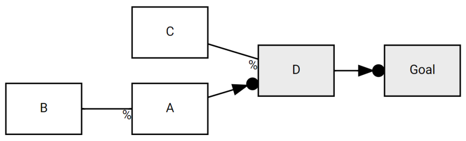

Fig. 1. Example DCR graph

Previous approaches to event reachability in DCR graphs have simply explored the state space of a DCR model [7,6]. Since the state of a DCR event is fully characterised by its three boolean attributes (executed, included, pending), an event can therefore be in at most 2^3 different states; a DCR graph with n events can therefore maximally have 2^{3n} markings. Not all of these markings will generally be reachable in a DCR graph, however, experience suggests that for real-life graphs, the state space is large enough to prohibit brute force approaches [12,7].

3 Genetic Algorithms

Genetic algorithms (GA) imitate natural selection. Solutions to a problem are structured as individuals with one or more chromosomes[1] consisting of genes.

[1] We will assume that individuals only have one chromosome, as most problems can be suitably modeled as individuals with only one chromosome.

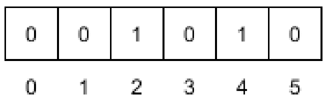

Fig. 2. A possible instantiation of a chromosome in *max-ones.*

This class of algorithms is often used where no efficient algorithm exists, but we have some notion of what makes a partial solution good.

A GA creates a list of individuals, the initial population or generation 1, with randomly generated chromosomes. Individuals are ranked by their fitness, based on how good the solution encoded in their chromosome is. We then select two or more individuals (parents) based on their rank and create a child whose chromosome is the crossover of the parents chromosomes. We create children until we reach a specified number of individuals, commonly double the initial population. At this point we proceed to kill off the worst individuals until we reach our initial population size. The remaining individuals constitute generation 2 of individuals.

This process is repeated until a stop condition is met. Common stop conditions are a solution is found or n number of generation have been generated. When it is unknown whether or not a solution is optimal, we can also set an optimization threshold oT and run the algorithm until the best individual improves less than oT in some amount of generations.

To implement a GA, we will need to implement the main building blocks of any genetic algorithm: a fitness function, a selection/ranking function, a crossover function and a mutation function. But first we have to decide how a gene and chromosome should be represented.

If our problem is to construct a bit string of length 6 with the maximum number of 1's[2], our genes will be either a 1 or a 0, and our chromosome will be a list or 1's and 0's of length 6 as seen in figure 2. We will call this problem *max-ones* and use it through out this section.

Fitness Function The fitness function of a GA takes an individual as input and return a value based on how good a solution its chromosome is. For a GA for *max-ones* a good fitness function simply returns the number of ones in a string. Such a fitness function would return 2 for the chromosome in figure 2.

Selection Function One of the common ways to select individuals from a population is to first sort the population by fitness. Then we set the accumulated selection chance of each individual to

$$individual[n].AccSelectionChance = \sum_{i=0}^{n} \frac{individual[i].Fitness}{population.TotalFitness}$$

[2] While this problem may seem trivial, because we already know the answer, one can think of the 1's as the binary representation of yes to the answer of a more complex question. Our problem therefore expands into: What is the maximum number of questions I can get a yes from.

The accumulated selection chance of the last individual will be 1. We then randomly select a value between 0 and 1 and select the last individual with an accumulated selection chance smaller than the random number. We have a greater chance of choosing individuals with better fitness score, but occasionally we get a worse individual, which is great for escaping local maximums in optimization algorithms [4].

Some selection functions will always retain the top $5 - 10\%$ of individuals, normally called the elite. This ensures that the top individuals never get worse.

Crossover Function A crossover function in most cases takes two individuals (it may take an arbitrary number) as input and returns a new individual that is the child of the two parents. The crossover function depends greatly on which problem we are solving, and is sometimes trial and error. The most common way is to select one or more crossover points in each individuals chromosome and split the chromosome at these points and glue the parts from both parents together.

Mutation Function The last of the main functions used in a GA is the mutation function. This function is called from inside the crossover function before the newly created child is returned. The mutation function changes one or more genes with a certain mutation probability.

4 GA to Solve Reachability in DCR Graphs

To solve reachability in DCR graphs, we must find a sequence of events to execute before the goal event is in an executable state, so a gene will be the id of an event to execute. A chromosome will be lists of these ids.

Fitness function Looking at Definition 3, we know that an event G can be executed if:

1. G is included.
2. All events that have a condition to G have been executed or excluded.
3. All events that have a milestone to G are excluded or are not pending.

Therefore, we score an event X when:

1. X has an include to G. This score is only considered if G is excluded.
2. a condition to G. This score is only considered if X has not been executed previously. Execution of an event Y that has a condition to X must also be given a score and so on.
3. a milestone relation to G. This score is only considered if the event is pending. Execution of Y that has milestone X must also be given a score, in the same way as the condition above.
4. an exclude relation to events that would have given some score according to 2 or 3.

(1) is easy to check. We achieve (2) and (3) by performing a graph search in the graph of events and relations from the G event following only condition and milestone relations and store a score on the events we encounter and return that score in the fitness function when the event is executed.

Before we consider (4) we first need to recall the example graph in Figure 1. We see that it is more beneficial to execute C than B: Executing C instantly enables execution of G, while executing B also requires executing D to enable execution of G. The fitness function must therefore reflect the distinction that C is better to execute than B. This is implemented by performing the graph search used for 2 and 3 with a BFS and decrease the score the further we are from the G event. Excluding an event that has a condition or milestone score according to 2 or 3 should give the same score as executing the event. The BFS ensures that execution or exclusion of an event that is closer to the G event returns a higher score than events further away.

Excluding an event should then give the combined condition and milestone score if the events that become excluded fulfill the requirements e.g. for condition score that the event has never been executed and for milestone score that the events is pending.

The *selection function* is implemented exactly as described in Section 3, by calculating the accumulated selection chance and then selecting a parent by chance.

The *crossover function* is implemented by choosing alternating genes from the two selected parents. If we reach the end of one parents chromosome, we just append the rest of the genes of the second parent to the child's chromosome. This is equivalent to having a crossover point after every gene[3].

The *mutation function* has three equiprobable mutations:

1. Change a gene (execution) to another random execution value.
2. Append a gene with a random value to the end of the chromosome.
3. Remove a gene at a random position in the chromosome.

These mutations allow chromosomes to contain genes that represent executions of events that are not executable. We hypothesize that it is beneficial to retain these invalid executions, as they may become valid executions in the future if a preceding execution mutates.

The last parameters of the genetic algorithm are the mutation probability, which is set to 10% (we tried 5% with similar results), the initial population size which is set to 50 and the starting length the individuals' chromosomes which is set to 2. The rather high mutation probability is necessary, as difficult to find paths require more than 2 preceding executions, so we must force the chromosomes to grow rather often.

The low initial population size allows us to generate new generations quicker. Higher initial population sizes were tried, but generally performed worse. The starting length the individuals' chromosomes is set low, as the algorithm provides better solutions faster for events where few preceding executions are necessary.

[3] Other types of crossover functions were tried, however none achieved better performance.

Graph	Events[a]	Relations[b]	Comment
REAL-WORLD MODELS			
Dreyer	31(4)	205	Executable model running Dreyer systems [24,11,12]
BigBelt	65(3)	293	Danish Railways emergency response processes [9]
CreateCase	12	24	Used in SPIN-based model-checking case-study [17]
6330	44(1)	135	Evaluation form for a Danish Arbitration Court [23]
7180	49	275	Complaints process from property evaluation
5919	25(3)	134	Application process of a major Danish pension fund
ARTIFICIAL MODELS			
hard	25	42	Hard for state-space exploration
harder	29	49	Even harder for state-space exploration
milestones	8	56	Has paths exponentially larger than the model itself

[a] Number of events with no self-conditions; number with self-condition in parenthesis.
[b] Number of relations after flattening out nestings [14].

Real-world models (except BigBelt) kindly provided by Danish vendors of process technology, Exformatics A/S and DCR Solutions A/S.

Table 1. Overview of graphs used in experiments

Post-processing The fitness function above allows paths that actually cannot execute under DCR semantics. We remove such events before returning a solution, and they do not contribute to the fitness score; however, they are important because they may become legal after mutation.

5 Experiments

We evaluated the genetic algorithm on six real-world models and three artificial ones, comparing results with those of the brute-force state exploration tool dcri [6,7].

We list the 9 models in Table 1; the models are available at `https://itu.dk/people/debois/ai4bpm-18`. The real-world models were kindly provided by Exformatics A/S and DCR Solutions A/S, except for BigBelt, which was a result of the case study [9]. We note that the BigBelt model is the largest publically DCR model available, and that the Dreyer model is known to be infeasible for the dcri tool. The artificial models “hard” and “harder” was constructed by dcri authors and pre-date the present study. Finally, we are grateful to Tijs Slaats for suggesting the model “milestones” was as a model that has minimal paths exponentially larger than the model itself.

The test were run on a machine equipped with an Intel Core i7-6700 processor. The genetic algorithm used the same amount of memory around 18-20 MB on all graphs. Because the GA is non-deterministic, reported timings are the average over 100 queries; because dcri is deterministic, reported timings are the result of a single run.

Graph	10 s timeout				60 s timeout			
	Total (s)	Max (s)	Avg. (s)	Misses (%)	Total (s)	Max (s)	Avg. (s)	Misses (%)
GENETIC ALGORITHM								
Dreyer	36.18	DNF	1.17	6.06	49.82	DNF	1.61	0.06
CreateCase	0.28	1.24	0.02	0	–	–	–	–
BigBelt	6.45	2.78	0.10	0	–	–	–	–
6330	0.70	0.19	0.02	0	–	–	–	–
7180	61.46	8.24	1.25	0	–	–	–	–
5919	4.47	DNF	0.15	0.20	6.18	10.25	0.23	0
BRUTE-FORCE (dcri)								
Dreyer	154.00	DNF	4.97	35.48	595.98	DNF	19.23	25.80
CreateCase	0.08	0.05	0.01	0	–	–	–	–
BigBelt	386.57	DNF	5.94	47.69	1405.40	DNF	21.62	26.15
6330	6.00	1.23	0.14	0	–	–	–	–
7180	247.00	DNF	5.06	40.81	966.64	55.12	19.73	0
5919	45.01	DNF	1.80	16.00	155.94	32.91	6.00	0

Table 2. Comparison of results, Real-world models

Real-world model timing results are reported in Table 2. For both algorithms, on each graph, we ran the algorithm on each event of the graph[4], with a timeout of 10 seconds. Then, for those graphs where one or more event was not reached, we the search for each event, this time with a 60 second per event timeout.

Each row in the table reports the aggregate time consumption of consecutively searching for each event in the model. The columns in the table are:

1. "Total", the total time spent searching for paths for each event in the graph.
2. "Max", the maximum time spent on a single event (i.e., on the most difficult event); or DNF (Did Not Finish) in case of timeouts.
3. "Avg.", the average time spent per event.
4. "Misses", the percentage of queries failing to find a path.

When the 10-second timeout search succeeded on all events, we did not repeat the search with a 60-second timeout, hence the cells containing "–".

Note that whereas dcri deterministically either always finds a path or never does, the GA randomly finds or fails to find certain paths. Also note that for GA, "Total" is the average over all runs, whereas "Max" is maximum, so it is possible for "Max" to be larger than "Total".

Finally, we note that the GA does not consistently miss: Every event was reached in the majority of runs.

[4] A common modelling trick for DCR models is to have inert events that are prevented from ever executing by having a condition to themselves. We did not search for paths to such events.

Graph	Goal	GA (s)	dcri (s)
hard	GOAL	0.15	976.00
harder	GOAL	0.32	DNF
milestones	A_8	257.39	0.03

Table 3. Comparison of results, Artificial models (50 m timeout)

Artificial model timing results are reported in Table 3. We ran both tools with timeouts of 50 minutes, searching only for a designated goal event.

6 Discussion

Real-world models. For *all* real-world models, the GA finds solutions strictly more often than dcri.

1. With the 10-second timeout, of the 6 models, GA fails on 2, missing 6% and 0.2%; whereas dcri fails 4, missing 16%, 35%, 41%, and 48%.
2. With the 60-second timeout, GA fails on 1 model, missing only 0.06% on Dreyer; whereas dcri fails on 2, ca. 25% of events on Dreyer and BigBelt.

On all but the "CreateCase" model, the GA is also faster:

1. Total time for the GA algorithm is between 4 and 12 times smaller than dcri at the 10-second timeout. It is always an order magnitude smaller than dcri.
2. Average time for the GA algorithm is between 4 and 594 times smaller than dcri at the 10-second timeout.

Timing results differ on CreateCase likely because the model is small enough to be in the "sweet spot" for a brute force solution. We see in Table 1 that CreateCase is indeed the smallest model, both in terms of number of events and number of relations. In particular, it has an order of magnitude fewer relations than other models, and so likely has a much smaller state-space.

In summary, our data indicates that for real-world models, the GA is universally better on all but very small models.

Artificial models. As expected, dcri performed very poorly on the models "hard" and "harder" designed to have large state spaces, solving "hard" after ca. 16 minutes and failing to terminate on "harder". Perhaps surprisingly, the GA performs excellently on both, solving either in substantially less than a second.

The "milestones" model consists of 8 events $\{A_1 \dots A_8\}$. Each event has a milestone to all events with a higher number and a response to all events with a lower number. This results in an execution pattern where the executions necessary to execute A_{n-1} must be repeated to execute A_n; it easy to prove by induction that in a graph of this shape, the shortest path to executing A_k has length 2^{k-1}, so executing A_8 requires a path of length 127.

However, the state-space of this model is tiny: Each event in the graph can be in only 3 distinct states, namely "not executed and pending", "executed and not pending" or "executed and pending"; with 8 events, it follows that the state-space contains at most a tiny $3^8 = 6461$ possible states, making it easy to solve for dcri. The GA struggles with this model, only finding a path in ca. 4.2 minutes. The longer the shortest path, the more often the GA has to randomly find a good execution. Furthermore the infinite execution pattern $(A_1A_2)^+$ will return a higher fitness score the more times it is run.

The paths discovered by the GA in the real-world graphs is generally in the single digits for "BigBelt" and around 20 for "Dreyer". However, these graphs contain many more events and relations—cf. Table 1—and therefore presumably exponentially more states to explore, making them more challenging for dcri.

GAs are not an exact science: a small change to the fitness, mutation or crossover might substantially change results. We have set correct parameters for these functions by trial and error, and it is likely that a more efficient implementation can be found. Furthermore smaller optimizations can possibly make the performance of the algorithm better. One example is to implement the events as bit vectors as done in other DCR engines [16].

Shortest paths. We note that our GA algorithm does not necessarily find the shortest path; not even typically. For example, refer to the graph in Figure 1: The current fitness function would give executions $\langle B,\ D\rangle$ the combined condition scores of A and D while executing C only would give the condition score of D. Therefore in the eyes of the GA, the path $\langle B,\ D,\ Goal\rangle$ is better than $\langle C,\ Goal\rangle$.

7 Conclusion

We have implemented event reachability for DCR graphs using a Genetic Algorithm, and evaluated the resulting algorithm against an existing brute-force solutions. On medium-sized real-world models and up, the Genetic Algorithm both finds a solution more often, and does so more quickly.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems and Computers 08, 21–66 (Feb 1998)
2. Aalst, W.M.P.v.d., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Web Services and Formal Methods. pp. 1–23. LNCS, Springer, Berlin, Heidelberg (Sep 2006)
3. Aalst, W.M.P.v.d.: Verification of Workflow Nets. In: Proceedings of the 18th International Conference on Application and Theory of Petri Nets. pp. 407–426. ICATPN '97, Springer-Verlag, London, UK, UK (1997)
4. Anderson-Cook, C.M.: Practical genetic algorithms (2005)
5. Basin, D.A., Debois, S., Hildebrandt, T.T.: In the Nick of Time: Proactive Prevention of Obligation Violations. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016. pp. 120–134. IEEE Computer Society (2016)

6. Debois, S., Hildebrandt, T.: The DCR Workbench: Declarative Choreographies for Collaborative Processes. In: Behavioural Types: from Theory to Tools, pp. 99–124. Automation, Control and Robotics, River Publishers (Jun 2017)
7. Debois, S., Hildebrandt, T., Marquard, M., Slaats, T.: Hybrid Process Technologies in the Financial Sector: The Case of BRFkredit. In: Business Process Management Cases, pp. 397–412. Management for Professionals, Springer, Cham (2017)
8. Debois, S., Hildebrandt, T., Slaats, T.: Replication, refinement & reachability: complexity in dynamic condition-response graphs. Acta Informatica pp. 1–32 (2017)
9. Debois, S., Hildebrandt, T.T., Sandberg, L.: Experience Report: Constraint-Based Modelling and Simulation of Railway Emergency Response Plans. In: ANT '16 / SEIT '16: Affiliated Workshops. pp. 1295–1300 (2016)
10. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, Refinement & Reachability: Complexity in Dynamic Condition-Response Graphs. Acta Informatica (2017)
11. Debois, S., Hildebrandt, T.T., Slaats, T., Marquard, M.: A Case for Declarative Process Modelling: Agile Development of a Grant Application System. In: EDOC Workshops 2014. pp. 126–133. IEEE Computer Society (2014)
12. Debois, S., Slaats, T.: The Analysis of a Real Life Declarative Process. In: SSCI '15. pp. 1374–1382. IEEE (2015)
13. Hildebrandt, T., Mukkamala, R.R.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In: Post-proceedings of PLACES 2010. EPTCS, vol. 69, pp. 59–73 (2010)
14. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T.: Nested Dynamic Condition Response Graphs. In: FSEN 2011, Revised Selected Papers. LNCS, vol. 7141, pp. 343–350. Springer (Apr 2011)
15. Hull, R., Masellis, E.D.R.D., Fournier, F., Gupta, M., Heath, F., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N., Vaculín, R.: A Formal Introduction to Business Artifacts with Guard-Stage-Milestone Lifecycles (2011)
16. Madsen, M.F., Gaub, M., Høgnason, T., Kirkbro, M.E., Slaats, T., Debois, S.: Collaboration among adversaries: Distributed workflow execution on a blockchain. In: 2018 Symposium on Foundations and Applications of Blockchain (2018)
17. Mukkamala, R.R., Hildebrandt, T., Slaats, T.: Towards Trustworthy Adaptive Case Management with Dynamic Condition Response Graphs. In: Proceedings of the 17th IEEE International EDOC Conference, EDOC 2013. pp. 127–136 (2013)
18. Object Management Group: Case Management Model and Notation. Tech. Rep. formal/2014-05-05, Object Management Group (May 2014), version 1.0
19. Object Management Group BPMN Technical Committee: Business Process Model and Notation, Version 2.0 (2013)
20. Pesic, M., Schonenberg, H., Aalst, W.M.P.v.d.: DECLARE: Full Support for Loosely-Structured Processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007). pp. 287–287 (Oct 2007)
21. Pesic, M., Van der Aalst, W.M.: A declarative approach for flexible business processes management. In: Business Process Management. pp. 169–180 (2006)
22. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (FOCS). p. 46?57 (1977)
23. Rasmus Strømsted, Hugo A. Lopez, Søren Debois, Morten Marquard: Dynamic Evaluation Forms using Declarative Modeling. In: BPM '18 (Industry track) (2018), submitted for publication
24. Slaats, T., Mukkamala, R.R., Hildebrandt, T.T., Marquard, M.: Exformatics Declarative Case Management Workflows as DCR Graphs. In: BPM '13. pp. 339–354 (2013)