2013 17th IEEE International Enterprise Distributed Object Computing Conference

Towards Trustworthy Adaptive Case Management with Dynamic Condition Response Graphs

Raghava Rao Mukkamala
IT University of Copenhagen
Rued Langgaardsvej 7
2300 Copenhagen, Denmark
rao@itu.dk

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

Tijs Slaats
IT University of Copenhagen and
Exformatics A/S
2100 Copenhagen, Denmark
tslaats@itu.dk

***Abstract*—We describe how the declarative Dynamic Condition Response (DCR) Graphs process model can be used for trustworthy adaptive case management by leveraging the flexible execution, dynamic composition and adaptation supported by DCR Graphs. The dynamically composed and adapted graphs are verified for deadlock freedom and liveness in the SPIN model checker by utilizing a mapping from DCR Graphs to PROMELA code. We exemplify the approach by a small workflow extracted from a field study at a danish hospital.**

***Keywords*-Adaptive Case Management, Declarative Business Processes, Verification**

I. Introduction

It has been recognized early [2], [29], that supporting dynamic (i.e. run-time) changes of process descriptions is one of the key challenges in workflow management systems. The challenge has been receiving increasing interest recently as a consequence of the demand for efficient IT systems supporting so-called *adaptive case management* (ACM) processes [14], [23], [24], [27], characterized by being *unpredictable* and *individual* in nature and being carried out by knowledge workers.

Healthcare services are typical sources of case management processes that exercise the challenges of evolutionary changes and being unpredictable and individual in nature. And the lack of support for dynamic adaptation and composition of processes is indeed one of the limiting factors for the usage in practice of the many standardized treatment plans and clinical guidelines being defined around the world [15], [16].

Changing a process description while process instances are executing may cause side effects such as un-intentional repetition or skipping of tasks and introduction of deadlocks and livelocks. This situation is referred to as the *dynamic change bug*. Elimination of the dynamic change bug calls for the use of formal process models and development of verification techniques that support dynamic changes and analysis for deadlocks and livelocks. In particular, formal *declarative* models [4], [12], [31] have been put forward as offering more flexibility in execution than the traditional approaches using explicit flow-graphs.

In the present paper, we propose an approach to specification and execution of trustworthy adaptive case management processes based on Dynamic Condition Response Graphs (DCR Graphs) [4], [6]–[8], [19]. DCR Graphs is a formal, declarative process modeling language developed in the Trustworthy Pervasive Healthcare Services (TrustCare.dk) research project as part of the first author's PhD project [17], and currently being embedded in the Exformatics case management tools as part of the industrial PhD project carried out by the last author. A brief overview of the DCR Graphs Tools implemented in Exformatics is presented in [26] and a graphical editor for DCR Graphs can be downloaded from [25].

A DCR Graph specifies a process as a set of labelled events related by five different relations specifying the constraints on the execution of events. The label of an event typically indicates the name of an atomic activity and by whom/which role the activity can be executed, while the constraints declare rules for the ordering of events.

As a running example, we will consider a simple healthcare process inspired by a previous field study at a danish hospital [15]. A fragment of the process is shown as a DCR Graph in Fig. 1 below.

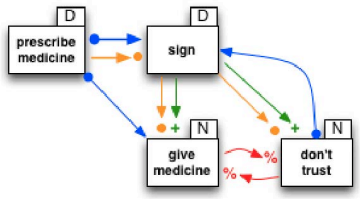


Figure 1. Prescribe medicine process fragment

The graph consists of four events shown as boxes labelled by the name of the activity (prescribe medicine, sign, give medicine, and don't trust) and the roles of whom can perform the activities, in this example either (D)octor or (N)urse. Intutivively, the process allows a doctor to prescribe medicine (any number of times) and subsequently being required to certify the prescription by a signature. The nurse

1541-7719/13 $26.00 © 2013 IEEE
DOI 10.1109/EDOC.2013.22

must then give the medicine (to a patient) or alternatively indicate that the prescription is not trusted. In the latter case the doctor is required to sign again (possibly after making a new prescription).

The prescribe medicine event is related to the sign event by a *condition* relation ($\rightarrow\bullet$), which declares that the sign event can not happen before at least one prescribe medicine event has happened. Similarly, the sign event is related to the give medicine and don't trust events, meaning that the sign event must have happened before the give medicine and don't trust events can happen.

Dually, the prescribe medicine event is related to the sign and give medicine events by a *response* relation ($\bullet\rightarrow$). The response relation declares that if the prescribe medicine event happens during an execution, it must eventually be followed (as a response) by a sign event and a give medicine event for the execution to be accepting (completed). But note, a single sign event and give medicine event can fulfill the response requirement of several preceeding prescribe medicine events. For instance, an execution starting with two prescribe medicine events and then a sign event is possible (because the condition for the sign event is fullfilled) but not (yet) completed since the give medicine event is a pending response. Now, if the execution is continued with a give medicine event then it is completed. It may still however continue, e.g. with a new prescribe medicine event. In this case the execution is no longer completed, since sign and give medicine are again pending responses.

The give medicine and don't trust events are related to each other by the *exclude* relation ($\rightarrow\%$). The exclude relation from give medicine to don't trust declares that the don't trust event will be excluded from the process if give medicine is executed. Similarly, the exclude relation from don't trust to give medicine declares that the give medicine event will be excluded from the process if don't trust is executed. That is, the two events are mutually exclusive. However, sign is related to give medicine and don't trust by an *include* relation ($\rightarrow+$), which means that whenever sign happens, the two events give medicine and don't trust are included again (if they were excluded). Note that condition relations from an excluded event are not considered, and if an excluded event is required to be executed (as a response), this requirement is also ignored as long as the event is excluded.

The intuition is that give medicine will be executed if the nurse trusts the prescription and don't trust if the nurse does not trust the prescription. In the latter case, the sign event is required to be executed again due to the response relation from don't trust to sign. In that case, the doctor will check his prescription, and may make new prescriptions but must sign again, whereafter the choice of giving the medicine or not trusting is made possible again by the inclusion relation.

A key feature of DCR Graphs is that the operational semantics indicated above can be formalized by representing the state of a process by a *marking* of the graph. The marking consists of three finite sets of events, $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, representing respectively the previously executed events, the events that are required to be executed in the future (as responses) or excluded, and the currently included events. This information is sufficient to infer enabledness of an event from the relations of the graph and to infer if an execution is completed. If we again consider the execution starting with two prescribe medicine events and then a sign event, this execution leads to the marking $(\{\mathsf{prescribe\ medicine}, \mathsf{sign}\}, \{\mathsf{give\ medicine}\}, \mathsf{E})$, where $\mathsf{E} = \{\mathsf{prescribe\ medicine}, \mathsf{sign}, \mathsf{give\ medicine}, \mathsf{don't\ trust}\}$ is the set of all events in the graph. Continuing the execution by the give medicine event then leads to the marking $(\{\mathsf{prescribe\ medicine}, \mathsf{sign}, \mathsf{give\ medicine}\}, \emptyset, \mathsf{E}\backslash\{\mathsf{don't\ trust}\})$.

In [17], [19] it is shown that the operational semantics of DCR Graphs can be mapped to Büchi-automata [19]. This makes it possible to formally verify temporal properties of the processes, and in particular deadlock freedom and liveness, using standard tools as for instance the SPIN verification tool [10], [11], [18].

A. DCR Graphs for ACM

The new contribution of the present paper is to describe how DCR Graphs and formal verification of such can be used for trustworthy, adaptive case management. Due to its emergent nature, visibility and control of an ACM process can only be achieved in the context of the execution of a process instance [27]. Therefore, case/knowledge workers continuously adapt the process activities to achieve their (sub)goals successfully [20]. At the same time, due to frequent adaptation, a process may end up in a situation, where it is no longer possible to achieve the overall goal of the process. We primarily use the term trustworthy to indicate that the adapted processes represented as DCR Graphs can be verified before execution is continued. Ideally, the application of formal verification techniques will not only enhance the trustworthiness of the processes, but also help knowledge workers in making suitable adaptive changes.

We demonstrate below, that the declarative nature of DCR Graphs makes it well suited for handling run-time changes and thus the emergent nature of an ACM process. Declarative models are usually considered harder to perceive than imperative process models based on explicit flow graphs. However, the simple representation of the run-time state by a marking on the DCR Graph and the verification step, help to perceive the meaning (and state) of the process and to ensure that the process execution can still be completed, i.e. the goal of the process can be met.

Fig. 2 below illustrates the normal iteration cycle through three phases of a trustworthy execution of a DCR Graph.

The execution starts in the phase *model & adapt*, where an ACM process is modelled either from scratch or by

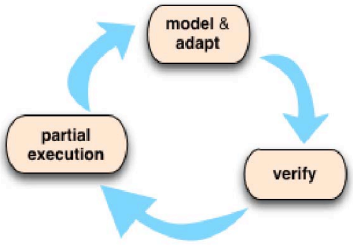
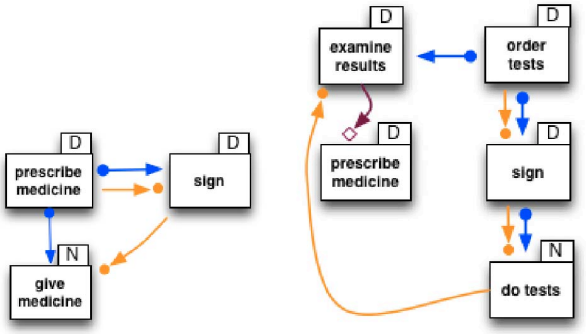


Figure 2. Execution phases of a DCR Graph for ACM

selecting one or more DCR Graphs process fragments (e.g. provided in repositories) which are composed and adapted. In dynamic environments, process knowledge can be local and fragmentary, confined to a certain situation or context. Process Fragments [1] represent a notion of partial and local knowledge, which can be integrated or composed dynamically at design time or run-time. Adopting the notion of process fragments, DCR Graphs can be used as process fragments to represent a partial perspective of a complex process, which can be combined through dynamic composition. Formally speaking, there is no difference between a normal DCR Graph and a DCR Graph representing a process fragment, except that the fragment DCR Graph might represent a subgoal or partial functionality like reusable templates. Fig 3 shows two such fragments for our healthcare example, to be explained in Sec. IV.



(a) prescribe medicine (b) order tests

Figure 3. DCR Graph fragments

In the next phase, *verify*, formal verification techniques will be applied to the process. For instance, one could verify that the process does not allow deadlocks, always allows progress and to achieve the overall goal, i.e. completion by (continued) execution or exclusion of all pending response events.

After successful verification of the process, the execution can proceed to the next phase, *partial execution*, where the process can be executed further until it reaches a point where further adaptation is required and the execution moves to the model & adapt phase, starting a new iteration. However, note that (in the spirit of ACM) it should also be possible to go back to the modelling phase after verification, to skip verification after the modelling phase, or go back to the verify phase after the partial execution.

In this way, an ACM process represented as DCR Graphs can be modelled, verified and executed iteratively, where the emergence of new knowledge can be used by the case workers to adapt the processes at run-time.

However, one may ask why the proposed approach can not simply prevent adoptions that will lead to dead/livelocks. We believe that the general approach allowing to compose/adapt and have intermediate models containing potential deadlocks/livelocks would be valuable as it gives more flexibility in the modeling and adaptation phase. It can be seen as analogous to be able to write a program that does not type check, and then correct the errors, as opposed to only be able to add code parts that lead to a well-typed program.

B. *Structure of paper*

The rest of the paper is structured as follows. We discuss related work in Sec. II whereas in Sec. III we further elaborate the idea of using DCR Graphs for ACM by our healthcare example. In Sec. IV we then formally define the adaptation operations on DCR Graphs, after recalling the formal definition of DCR Graphs and their execution. In Sec. V we then formally define what it means for a DCR Graph to be deadlocked and live, introducing new notions of *strongly* deadlock free and live processes which guarantee progress even if only events that are required as responses are executed. This is in particular relevant if the execution of the DCR Graphs is distributed on different peers (e.g. according to the different roles) as considered in [7]. In Sec. VI we then describe how to verify safety and liveness properties on DCR Graphs (as defined in Sec. V) using the SPIN model checking tool and based on the mapping of DCR Graphs to Büchi-automata [19]. Finally, Sec. VII concludes the paper.

II. Related Work

The issue of dynamic change [2] in workflow systems has been investigated thoroughly for Petri net and graph based models. In [29] Van der Aalst described an approach to find change regions in WorkFlow nets, which represent the parts of a model that are effected by a change. He also proved that a change can be safely applied to a part outside the change region, simultaneously preserving the soundness of a workflow instance. In [22] Reichert et al. presented a framework for the support of adaptive changes in the graph-based workflow model ADEPT. They developed a complete and minimal set of change operations that will allow for modifying an ADEPT workflow at run-time, while still preserving its consistency and correctness.

The previous work often take as a correctness criteria in previous approaches [2], [22], [29], that the state of the instance after applying the change, could have been reached from the initial state by replaying the past run. We find that only allowing changes that are consistent with the past history too strong for ACM. Instead we advocate recording the change as part of the execution sequence.

Recent studies [24], [28] have indicated that BPMN-like languages are not suitable for ACM. One reason is that the processes are described as procedures. Procedures tend to over-specify the processes, and also, the changes one can apply must be formulated as changes to the procedure. On the other hand, declarative workflow models [4], [12], [23], [31], including DCR Graphs, have been proposed as a alternative to traditional workflow models to handle unpredictability and emergent nature of ACM processes. Here process are described by declaring the constraints and goals, which usually under-specify the process and supports changes to the constraints and goals. Declare [30] is a constraint-based declarative workflow model formalized using linear time logic (LTL). Similar to DCR Graphs, Declare also supports adaptive changes such as add/remove constraints and activities, however, since they can not be interpreted in an immediate state, it is required that the trace of the past execution satisfies the LTL formulae corresponding to the change. That is, as discussed above, only changes that are consistent with the run so far are allowed.

A declarative approach using *Guard-Stage-Milestone* [12] based on ECA-like rules for specification of life cycles on Business artifacts was proposed in the recent years. To the best of our knowledge, no work on adaptive changes for the GSM model has been published yet. However, it has been advocated as a model for ACM due to its rich data-centric approach and declarative nature and forms the basis for the recent Case Management Model And Notation (CMMN) [21] proposed by OMG, which includes support for dynamic changes. Also the IBM case manger [3] includes some support for dynamic changes. Furthermore, a modeling approach based on Declarative Configurable Process Specifications [24] is being developed for automated support of case management processes. Using declarative modeling, their model supports process adaptability by using configurable data objects and context based configuration rules, but does not support process run-time adaptation when compared to DCR Graphs.

The use of SPIN for verification of business processes was studied earlier. In [13], authors used SPIN to verify business processes by translating an imperative process specification into state machine description in Promela. In this paper, we translated a declarative process specification in DCR Graphs to Promela by mapping it to Bučhi-automata [19].

Finally, we have recently proposed a *join* operator [8] for modular composition and refinement of DCR Graphs and to use it as a formal basis for modular implementation of context sensitive and aspect oriented processes. The *compose* and *change* operations can be derived from the *join* operator, however, in this paper we have chosen to define the adaptation operations directly to make them more straightforward and easier to understand.

III. DCR Graphs for ACM by Example

In this section, we will discuss the adaptation operations for DCR Graphs and exemplify the adaptiveness of DCR Graphs for ACM using the healthcare example.

As adaptation operations we consider the operations of *adding/removing an event*, *adding/removing a constraint between two events*, *changing an event*, *relabelling an event*, *adapting the marking* and *forcing execution* of a non-enabled event. The operations of adding events (to the graph or the marking) and constraints are facilitated by a general compositition operation, which simply takes the union of two graphs as described formally in the next section.

As an example of an ACM process, consider a healthcare workflow where a doctor during the initial consultation realizes that some medical tests are needed before giving the medicine to a patient. In the initial modelling phase, the doctor thus selects in a repository the prescribe medicine DCR Graph fragment in Fig. 3(a) and the order tests process fragment in Fig. 3(b) and compose them. The doctor then proceeds to execute the order test event resulting in the DCR Graph instance shown in Fig. 4.

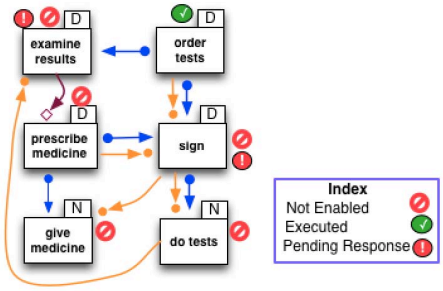


Figure 4. Composed prescribe medicine example with live lock.

The tick mark in the green circle on order tests represents that order tests has been executed, i.e. it is in the executed events set of the marking. Additionally, due to the response relation from order tests to the examine tests and sign events, they have a pending response, i.e. they are in the response events set of the marking. This is indicated by red circle with an exclamation mark. Now, the arrow from examine tests to prescribe medicine having a diamond at its head is the last constraint that we have not yet explained, called the *milestone* constraint. The milestone constraint disables the target event (in this case prescribe medicine) if the source event has a pending response and is included, as it is the case in Fig. 4. The intuition is that the source event must be in a completed state (the milestone reached)

before the target event can be executed. Disabled events are indicated with red stop signs. Note that the sign, examine tests and give medicine events are also not enabled because of the condition constraints from prescribe medicine, do tests and sign respectively.

Looking carefully at the DCR Graph in Fig. 4, one may notice that, in order to sign for ordering the tests, the prescribe medicine event must have been done first, which requires the examine tests has been done first (to remove the pending response), which then requires that do tests event has to be done first, because of the condition relation from do tests to examine tests. Alas, the do tests event is blocked by the sign step. In other words, we have a cycle of events blocking each other, in which two of the events are actually required to be executed to complete the workflow. The DCR Graph is not deadlocked, since the order tests event can be repeatedly executed, but this will not change the marking and thus not allow the doctor to make further progress. Hence, the DCR Graph is live locked as explained in Def. 9, as it will never be able to execute or exclude the pending response events (sign and examine tests).

The process instance can be adapted in many ways to remove the live-lock. One way to solve the problem is to force execution of the disabled sign event. However, in fact, the live lock happens because of a modeling error. The doctor should have two separate sign events, one for prescribe medicine and one for order tests. This can again be achieved in many ways. A simple way is to rename the sign events to fresh event names before composing the graphs, which would result in the DCR Graphs in 5 (again after the execution of order tests). Verification of this graph shows that it is deadlock and livelock free.

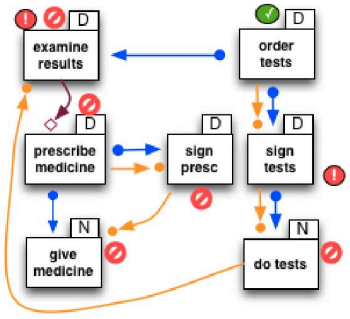


Figure 5. Adapted prescribe medicine example

IV. Adaptive DCR Graphs Formally

In this section we first recall from [17] the formal definitions of DCR Graphs and then give the formal definitions of the new adaptation operations. We employ the following notation: We assume infinite sets $\mathbb{E}$ and $\mathbb{L}$ for events and labels respectively. For a set E we write $\mathcal{P}(E)$ for the power set of E (i.e. set of all subsets of E). For a binary relation $\rightarrow \subseteq E \times E$ and a subset $\xi \subseteq E$ we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{e \in E \mid (\exists e' \in \xi \mid e \rightarrow e')\}$ and the set $\{e \in E \mid (\exists e' \in \xi \mid e' \rightarrow e)\}$ respectively, and abuse notation writing $\rightarrow e$ and $e \rightarrow$ for $\rightarrow \{e\}$ and $\{e\} \rightarrow$ respectively when $e \in E$.

A. Basic Definitions

Formally, a DCR Graph is defined as follows.

Definition 1: A Dynamic Condition Response Graph (DCR Graph) G is a tuple $(\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, where

(i) $\mathsf{E} \subset \mathbb{E}$ is a finite set of *events*,
(ii) $\mathsf{M} \in \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E})$ is the *marking*,
(iii) $\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\% \subseteq \mathsf{E} \times \mathsf{E}$ is the *condition*, *response*, *milestone*, *include* and *exclude* relation respectively.
(iv) $\mathsf{L} \subset \mathbb{L}$ is the finite set of *labels* and $l : \mathsf{E} \rightarrow \mathcal{P}(\mathsf{L})$ is a labeling function mapping events to sets of labels.

As explained in the introduction, the marking (ii) represents the state of the DCR Graph and the five binary relations over the events (iii) define the constraints on the events and dynamic inclusion and exclusion. Finally, each event is mapped to a set of labels (iv). In our running example simply assign a singleton set containing a pair consisting of the name of the activity and the role able to perform the activity.

In Def. 2 we formally define when an event e of a DCR Graph is *enabled* for a marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, written $\mathsf{M} \vdash_G e$. To be enabled, the event e must be included, i.e. $e \in \mathsf{In}$, all the included events that are conditions for it must be in the set of executed events, i.e. $(\mathsf{In} \cap \rightarrow\bullet e) \subseteq \mathsf{Ex}$, and none of the included events that are milestones for it can be in the set of scheduled response events, i.e. $(\mathsf{In} \cap \rightarrow\diamond e) \subseteq \mathsf{E} \backslash \mathsf{Re}$.

We then further define the new marking $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$, resulting from executing an event e in the marking M. Firstly, the event e is added to the set of executed events, i.e. $\mathsf{Ex}' = (\mathsf{Ex} \cup \{e\})$. Secondly, the event e is removed from the set of scheduled responses and all events that are a response to the event e are added, i.e. $\mathsf{Re}' = ((\mathsf{Re} \backslash \{e\}) \cup e\bullet\rightarrow)$. Note that if an event is a response to itself, it will be removed and immediately added again, and thus remain in the set of scheduled responses after its execution. Finally, all the events that are excluded by e are removed from the included events set, and all the events that are included by e are added, i.e. $\mathsf{In}' = (\mathsf{In} \setminus e\rightarrow\%) \cup e\rightarrow+$.

Definition 2: For a Dynamic Condition Response Graph $G = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, and $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, we define that an event $e \in \mathsf{E}$ is *enabled*, written $M \vdash_G e$, if $e \in \mathsf{In} \wedge (\mathsf{In} \cap \rightarrow\bullet e) \subseteq \mathsf{Ex} \wedge (\mathsf{In} \cap \rightarrow\diamond e) \subseteq \mathsf{E} \backslash \mathsf{Re}$. The result of executing the event e in the marking M of a DCR Graph G is the marking $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \oplus_G e =_{def} \big(\mathsf{Ex} \cup \{e\}, (\mathsf{Re} \setminus \{e\}) \cup e\bullet\rightarrow, (\mathsf{In} \setminus e\rightarrow\%) \cup e\rightarrow+\big)$.

Having defined when events are enabled for execution and the effect of executing an event we define in Def. 3 the notion of finite and infinite executions and when they are accepting (or completed). Intuitively, an execution is accepting if any required, included response in any intermediate marking is eventually executed or excluded in a subsequent marking during the execution.

We further define the subset of executions in which only events that are required as responses are executed, which we refer to as *must* executions. The reason for considering must executions is that if a process is deadlock and livelock free even when restricted to must executions, then we are guaranteed progress, even if the participants in any step only perform activities that are required as responses.

Definition 3: For a Dynamic Condition Response Graph $G = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ we define an *execution* of G to be a (finite or infinite) sequence of pairs of events and labels: $\bar{e} = (e_0, a_0), (e_1, a_0), \ldots$ such that $a_i \in l(e_i)$ and $\mathsf{M}_i \vdash_G e_i$ for $\mathsf{M}_0 = \mathsf{M}$, $\mathsf{M}_{i+1} = \mathsf{M}_i \oplus_G e_i$.

Assuming $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{In}_i, \mathsf{Re}_i)$ we say the execution $\bar{e}$ is a *must* execution if for all $(e_i, a_i) \in \bar{e}$, $e_i \in \mathsf{Re}_i$ and an *accepting* (or completed) execution if for all $(e_i, a_i) \in \bar{e}$, $\forall e \in \mathsf{In}_i \cap \mathsf{Re}_i . \exists j \geq i . e_j = e \vee e_j \rightarrow\% e$. Let $\mathsf{exe}_{\mathsf{M}}(G)$, $\mathsf{mexe}_{\mathsf{M}}(G)$, $\mathsf{acc}_{\mathsf{M}}(G)$and $\mathsf{macc}_{\mathsf{M}}(G)$ denote respectively the set of all executions, all must executions, all accepting executions, and all accepting must executions of G starting in marking M.

B. Adaptation Operations on DCR Graphs

In order to support adaptive changes for case management, we define three operations on DCR Graphs: *compose*, *change* and *discard*.

The first operation *compose* is a binary composition of two DCR Graphs, where we glue together (take the union of) the events, constraints, labels and markings of both the DCR Graphs as formally defined in Def. 4. Note that the *compose* operation also glues the markings of DCR Graphs, therefore it really defines composition of process instances.

Definition 4: Let $\mathsf{G}_i = (\mathsf{E}_i, \mathsf{M}_i, \rightarrow\bullet_i, \bullet\rightarrow_i, \rightarrow\diamond_i, \rightarrow+_i, \rightarrow\%_i, \mathsf{L}_i, l_i)$, $\mathsf{M}_i = (\mathsf{Ex}_i, \mathsf{Re}_i, \mathsf{In}_i)$ for $i \in \{1, 2\}$. Then $\mathsf{G}_1 \oplus \mathsf{G}_2 = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, where

(i) $\mathsf{E} = (\mathsf{E}_1 \cup \mathsf{E}_2)$
(ii) $\mathsf{M} = (\mathsf{Ex}_1 \cup \mathsf{Ex}_2, \mathsf{Re}_1 \cup \mathsf{Re}_2, \mathsf{In}_1 \cup \mathsf{In}_2)$,
(iii) $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ for each $\rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%\}$
(iv) $l(e) = l_1(e) \cup l_2(e)$ and $\mathsf{L} = \mathsf{L}_1 \cup \mathsf{L}_2$

In Def. 5 we define event substitution operation on DCR Graphs, which is used for renaming of events. We use the shorthand $e''[e \mapsto e']$ to refer to the event e' if $e'' = e$ and e'' otherwise. First, the new event is substituted in the set of events (i) and labeling function is updated accordingly (ii). Further, all the constraints sets (iii) and the sets in the marking (iv) are updated accordingly for the event substitution. Note that there are no restrictions on the new name for the event, therefore if the new name already exists in the DCR Graph, then substitution operation allows merging of events.

Definition 5: Let $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ and $e \in \mathsf{E}, e' \in \mathbb{E}$. The *event substitution operation* is defined as $\mathsf{G}[e \mapsto e'] = (\mathsf{E}', \mathsf{M}', \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L}, l')$ where

(i) $\mathsf{E}' = \mathsf{E} \backslash \{e\} \cup \{e'\}$
(ii) $(e''[e \mapsto e'], a) \in l'$ if $(e'', a) \in l$
(iii) $\forall \rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%\} . e_1[e \mapsto e'] \rightarrow' e_2[e \mapsto e']$ if $e_1 \rightarrow e_2$
(iv) $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ and $\forall R \in \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\} . e''[e \mapsto e'] \in R'$ if $e'' \in R$

The *change* operation can be used for renaming labels as formally defined in Def. 6. First, an event substitution operation is applied (i) and then new labels are added to the set of labels. Finally, the labeling function is updated (ii) for the new labels.

Definition 6: Let $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$, $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ and $e \in \mathsf{E}, e' \in \mathbb{E}, A \subset \mathbb{L}$. The *change event operation* is defined as $\mathsf{G}[e \mapsto (e', A)] = (\mathsf{E}', \mathsf{M}', \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L} \cup A, l'')$ where

(i) $\mathsf{G}[e \mapsto e'] = (\mathsf{E}', \mathsf{M}', \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L}, l')$
(ii) $l''(e'') = \begin{cases} A & \text{if } e'' = e \\ l'(e'') & \text{otherwise} \end{cases}$

In the Def. 7, we introduce three overloaded versions of the *discard* operation to delete: an event from a DCR Graph (a), a constraint from a DCR Graph (b) and an event from a marking (c). Discarding an event from a DCR Graph will delete it from the set of events along with its label mapping from the labeling function (ai), additionally, it will also be removed from all the sets in the marking (aii) and finally all the constraints from and to the event will also be deleted from the respective constraints sets (aiii). Similarly, discarding a constraint from a DCR Graph will delete it from the respective constraint set (b), where as discarding an event from a set in the marking of a DCR Graph is simply removing that event from the set (c).

Definition 7: Let $\mathsf{G} = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ with $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. We define three *discard* operations by

(a) $\mathsf{G} \ominus e = (\mathsf{E}', \mathsf{M}', \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L}, l')$ where

(i) $\mathsf{E}' = \mathsf{E} \setminus \{e\}, l' = l \setminus \{(e, l(e))\}$
(ii) $\forall R \in \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\} . R' = R \setminus \{e\}$
(iii) $\forall \rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%\}.$ $\rightarrow' = \rightarrow \setminus \{(e, e'), (e', e), (e, e) \mid e' \in \mathsf{E}\}$

(b) $\mathsf{G} \ominus (e \rightarrow_c e') = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet', \bullet\rightarrow', \rightarrow\diamond', \rightarrow+', \rightarrow\%', \mathsf{L}, l')$ where $\rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%\}$, and $\rightarrow' = \begin{cases} \rightarrow \setminus \{(e, e')\} & \text{if } \rightarrow_c = \rightarrow \\ \rightarrow & \text{otherwise} \end{cases}$

(c) $\mathsf{G} \ominus (e, R) = (\mathsf{E}, \mathsf{M}', \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ where M' is the obtained by removing the event from a set $R \in \{\mathsf{Ex}, \mathsf{Re}, \mathsf{In}\}$ in M.

V. Safety and Liveness

A deadlock state of a DCR Graph is a marking where there is an included, required response but no enabled events. Thus, a DCR Graph is *deadlock free* if and only if for any reachable marking, there is either an enabled event or no included required responses. It is *strongly* deadlock free if and only if for any reachable marking there is either an enabled event which is also a required response or no included required responses. As exemplified below, strongly deadlock freedom guarantees progress even if the execution of the DCR Graphs is distributed (e.g. according to the different roles) and every peer only executes events that are required as responses.

Definition 8: Let $\mathcal{M}_{\mathsf{M}\rightarrow^*}(G)$ denotes the set of all reachable markings from M. For a dynamic condition response graph $G = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ we define that G is *deadlock free*, if $\forall \mathsf{M}' = (\mathsf{Ex}', \mathsf{In}', \mathsf{Re}') \in \mathcal{M}_{\mathsf{M}\rightarrow^*}.(\exists e \in \mathsf{E}.\mathsf{M}' \vdash_G e \vee (\mathsf{In}' \cap \mathsf{Re}' = \emptyset))$. We say that G' is *strongly deadlock free*, if $\forall \mathsf{M}' = (\mathsf{Ex}', \mathsf{In}', \mathsf{Re}') \in \mathcal{M}_{\mathsf{M}\rightarrow^*}.(\exists e \in \mathsf{Re}'.\mathsf{M}' \vdash_G e \vee (\mathsf{In}' \cap \mathsf{Re}' = \emptyset))$.

If G is the DCR Graph in Fig. 1, then G is both deadlock free and also strongly deadlock free. However, the adapted graph $G \ominus (\mathsf{prescribe\ medicine} \bullet\rightarrow \mathsf{sign})$ in which the response relation from prescribe medicine to sign is discarded will only be deadlock free, but not strongly deadlock free. If the doctor starts by prescribing medicine, then there will be a pending response on give medicine (but not on sign), therefore there will be no enabled event which also is a pending response. The workflow is not deadlocked since the sign activity may be executed, even though it is not required as a response. The workflow is not in an accepting state either, since there is a required response for give medicine. However, if every participant only does what is required, i.e. scheduled as a response, the workflow will never progress to a completed state. This may in particular be a problem if the execution of the workflow is distributed, e.g. according to the roles, as supported by the algorithm given in [7]. If the doctor only sees activities assigned to the doctor role, she may never sign after doing a prescription if it is not required as response. However, the nurse will be required to perform the give medicine activity, but it is not enabled since the sign must have been done first.

Note that deadlock freedom only guarantees that the process can make some progress, but not that it can proceed a long an accepting (completed) execution. A DCR Graph is defined to be *live* if and only if, in every reachable marking, it is always possible to continue along an accepting run (i.e. eventually execute or exclude any of the pending responses). We defined that it is *strongly live* if and only if, from any reachable marking there exists an accepting must execution.

Definition 9: For a dynamic condition response graph $G = (\mathsf{E}, \mathsf{M}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\diamond, \rightarrow+, \rightarrow\%, \mathsf{L}, l)$ we define that the DCR Graph is *live*, if $\forall \mathsf{M}' \in \mathcal{M}_{\mathsf{M}\rightarrow^*}.\mathsf{acc}_{\mathsf{M}'}(G) \neq \emptyset$, and *strongly live*, if $\forall \mathsf{M}' \in \mathcal{M}_{\mathsf{M}\rightarrow^*}.\mathsf{macc}_{\mathsf{M}'}(G) \neq \emptyset$,

The give medicine example G in Fig. 1 is again both live and strongly live, and $G \ominus (\mathsf{prescribe\ medicine} \bullet\rightarrow \mathsf{sign})$ will be live, but not strongly live.

VI. Verification of DCR Graphs

This sections describes how the safety and liveness properties on DCR Graphs can be verified by using the Spin [10] model checking tool. Spin supports verification of properties for asynchronous process models and distributed systems, specified in the language called PROMELA.

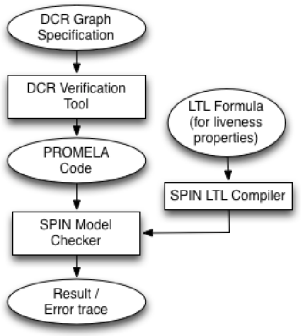


Figure 6. Verification of DCR Graphs using Spin tool

Fig. 6 shows the overall methodology of the verification of DCR Graphs using Spin. The DCR verification tool accepts a DCR Graph specification as an XML file and generates the necessary PROMELA code, compiles it and verifies it using Spin. The DCR Graph verification tool is available through a web interface [18].

The properties to be verified can be expressed as a Linear Temporal Logic (LTL) formula in the tool. The Spin LTL compiler generates a finite automaton for the negation of the formula, referred to as a *never claim*. Similarly, a finite automaton is generated for the DCR Graph model specified in PROMELA code. Finally, the Spin verifier searches for an acceptance cycle in the synchronous product of the two automata. In case the verifier finds an acceptance cycle, it reports an error by providing a trace for the property violation.

A. DCR Graphs to PROMELA

In the encoding of a DCR Graph to PROMELA, we employ the mapping from DCR Graphs to Büchi automata from [9], [19], as liveness properties are to be verified over infinite runs. The event names of a DCR Graph are mapped to integers, as PROMELA does not have support for strings. The constraint sets and marking of a DCR Graph are encoded as *arrays*, as PROMELA does not support sets. Furthermore, the language only supports fixed size arrays,

therefore we have defined an event set of a DCR Graph as a bit array, where the index of an array represents the integer code of an event and the value (0 or 1) at that index defines whether the event is part of the set or not. The marking (M) of a DCR Graph is encoded as three bit arrays.

The PROMELA language supports one-dimensional arrays only. Therefore the constraints sets of a DCR Graph are defined by using typedef for two-dimensional arrays where the indices of the array are the integer codes of events and the values (0 or 1) represents whether the constraint exists from the first to the second event as shown in Fig. 7.

The assignment *included[2] = 1;* defines that the event *gm* (with integer code = 2) is part of the *included* set. Since all data types of PROMELA are initialized to 0 by default, all the events which are not explicitly mentioned in the initial marking or specification are not included.

PROMELA does not have procedure/function construct to structure the code, therefore the *inline* construct was used to group a sequence of statements related to one logical function as shown in Fig. 7.

```
57 inline model_specification(){
58 /* Specification of DCR Graph */
59     /* Relations */
60     condition_relation[pm].to[sign] = 1;
61     condition_relation[sign].to[gm] = 1;
62     response_relation[pm].to[sign] = 1;
63     response_relation[pm].to[gm] = 1;
64     /* Specification of the current state
65     /* Executed Actions */
66     /* Pending Response Actions */
67     /* Included Actions */
68     included[pm] = 1;
69     included[sign] = 1;
70     included[gm] = 1;
71 }
```

Figure 7. Specification of give medicine example

The main logic for verification of safety and liveness properties is shown in Fig. 8. The main process function (*proctype dcrs*) contains one *do* loop in which code from different *inline* blocks will be executed.

The *model_specification()* inline block contains the specification of a DCR Graph as described previously and the *clear_enabled_events* block clears the list of events in the enabled set. The next inline block is *Compute_enabled_events*, which loops through the events in the *included* set and verifies whether all its included condition events have been executed. Similarly, all included milestone events of an event are also checked to make sure that none of them are part of the pending *response* set. An event satisfying these checks will be added to the *enabled* events set.

The next inline block, *nondeterministic_execution()*, contains code for executing one of the events from the *enabled* set. The tool generates options for an *if* block with a guard matching to status bit of an event in the *enabled* set. During verification, Spin will evaluate all the guards and choose one of the enabled options non-deterministically, by assigning it to the variable *random_event_executed*.

```
263 active proctype dcrs()
264 {
265     /* DCR graph specification */
266     model_specification();
267     do
268     ::
269     /* Clearing away enabled set */
270     clear_enabled_events();
271     /* Compute enabled events */
272     compute_enabled_events();
273     /* Execute an event non-nondeterministically */
274     nondeterministic_execution();
275     /* Compute response sets and m-set */
276     compute_include_response_sets();
277    /* Compute minimum values */
278     compute_set_minimum();
279    /* Compute state accepting conditions  */
280     check_state_acceptance_condition();
281    /* Compute state after execution. */
282    compute_state_after_execution();
283     od;
284 end_state: printf("End state reached after %u",
285                executed_event_count);
286 }
```

Figure 8. PROMELA code for main process

B. Deadlock and Liveness

In *nondeterministic_execution()*, if none of the guards are evaluated to true, then the else block will be executed. The code in the else blocks declares a deadlock if there are any included pending responses. In the absence of included pending responses, the program jumps to *end_state* to terminate the program.

In the case of enabled events in every marking, the *else* block will never get executed and thereby the *do* loop will continue forever without breaking. Spin detects such kind of cycles and terminates the program after inspecting all the states of the automaton.

Liveness properties of a DCR Graph can be verified by specifying a *never claim* in LTL. In the tool, liveness verification is done by specifying a correctness claim as $\square\Diamond$ *accepting_state_visited* in LTL, from which Spin generates a *never claim* based on the negation of the formula.

C. Strong Deadlock freedom and Liveness

The encoding of deadlock for must executions is very much similar to that of the deadlock property introduced in the previous paragraphs. In the *nondeterministic_execution*, an additional check for *included* and *enabled* pending responses will be made, before choosing any enabled event for non-deterministic execution. In case of existence of a pending response without any *enabled* pending response, a violation of strongly deadlock freedom will be declared.

For verification of the strong liveness property on a DCR Graph, we generate an encoding for every possible reachable marking of the DCR Graph. In addition a check will be made in the non-deterministic execution of events, to make sure that the enabled events are also pending responses.

D. Evaluation of Spin Verification

Table. I shows statistics for the Spin verification of the healthcare workflow from the previous sections. The second and third columns represent the number of events and constraints in the DCR Graph. The number of reachable markings in the DCR Graph is shown in column 4. The last three columns display the statistics from Spin verification: the number of Spin program states, time taken in seconds and memory usage in megabytes respectively.

Model	DCR Graph			Spin statistics		
	E	→	states	program states	time sec.	memory MB.
prescribe medicine	3	4	13	14,741	0.04	613.04
order tests	5	7	72	231,731	0.60	759.70
prescribe + order	6	11	116	602,289	1.67	775.20
adapted prescribe + order	7	11	460	3,267,596	9.37	880.70
create case [5]	17	28	1386	15,614,513	63.1	1432.5

Table I
SPIN VERIFICATION STATISTICS

Even though Spin verification on DCR Graphs is quite useful, we have noticed certain drawbacks. First of all, the number of Spin program states grows exponentially with the number of events in a DCR Graph. For example, the *adapted prescribe medicine* example from Fig. 5 contains 7 events and 11 constraints. But as shown in Table. I, the Spin program states are more than three million, even though there are only 460 unique reachable markings in the büchi automaton for the DCR Graph. The automata construction is inherently exponential, however, a further blow-up of Spin program states is caused by the lack of good data structures in PROMELA for encoding sets and other complex types. This means that the event sets of a DCR Graphs have to be encoded as fixed size arrays and these arrays have to be iterated many times to calculate the updated markings. Moreover, every value change of a variable (e.g. loop index) is considered as unique program state in Spin. Additionally, the increase of Spin memory usage (last column) is also an alarming issue, which could be problematic in verification of larger models.

In addition to the above limitations, the output generated by Spin is also not user friendly. Especially, it is difficult for modellers to figure out the counter example from the Spin error trails. Therefore, we strongly believe that by performing the verification directly on the reachable markings of a DCR Graph, it is possible to verify much larger models and provide intuitive validation results.

VII. CONCLUSION

We have presented an approach to adaptive case management based on the recently introduced declarative process model Dynamic Condition Response (DCR) Graphs. Our work leverages three key features of DCR Graphs: 1) Its declarative nature with implicit definition of states allow for simple definitions of process composition and change, 2) its simple operational semantics based on markings of the graph allowed us to extend the definitions of process composition and change to running instances, and 3) the mapping of DCR Graphs to the SPIN model checking tool allowed us to formally verify deadlock freedom and liveness for the dynamically changed adapted models.

The definition of deadlocks for DCR Graphs is new, and exploited that the markings of DCR Graphs allow to distinguish between which events may happen now (the enabled events), and which events must eventually happen (the required future responses). This allowed us to define a deadlock as a state where some event must eventually happen, but no events may happen now. Moreover, we introduced a new notion of *strongly* deadlock freedom, which intuitively means that even in a situation where every actor only perform required actions there will be no deadlocks. We also introduced the notion of liveness and strong liveness for DCR Graphs

We found that the PROMELA language and its ability to verify both safety and liveness properties made SPIN easy to use as back-end verification tool for DCR Graphs and benchmarked the verification on a small set of examples. However, the benchmarks also show that the resulting SPIN models reach a quite large number of states for even small DCR Graphs, which indicate that there may be an advantage to implement the verification algorithms directly for DCR Graphs. This could potentially explore the partial order information of DCR Graphs and avoid constructing the interleaved transition system model.

In future work we plan to investigate a native implementation of model checking for DCR Graphs. We also plan to extend the approach to ACM presented in the present paper to DCR Graphs extended with data and nested sub graphs as defined in [17] and relate and compare our work to the GSM-approach [12].

REFERENCES

[1] Hanna Eberle, Tobias Unger, and Frank Leymann. Process fragments. In *OTM '09*, pages 398–405. Springer-Verlag, 2009.

[2] Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic change within workflow systems. In *Proceedings of conference on Organizational computing systems*, COCS '95, pages 10–21, New York, NY, USA, 1995. ACM.

[3] Wei-Dong Zhu et. al. *Advanced Case Management with IBM Case Manager*. IBM Redbooks, 2013. http://www.redbooks.ibm.com/redbooks/pdfs/sg247929.pdf.

[4] Thomas Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, volume 69 of *EPTCS*, pages 59–73, 2011.

[5] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Designing a cross-organizational case management system using dynamic condition response graphs. In *Proceedings of IEEE International EDOC Conference*, 2011.

[6] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Nested dynamic condition response graphs. In *Proceedings of Fundamentals of Software Engineering (FSEN)*, April 2011.

[7] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Safe distribution of declarative processes. In *9th International Conference on Software Engineering and Formal Methods (SEFM) 2011*, 2011.

[8] Thomas Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats, and Francesco Zanitti. Modular context-sensitive and aspect-oriented processes with dynamic condition response graphs. In *Foundations of Aspect-Oriented Languages 2013*, 2013.

[9] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *PLACES*, pages 59–73, 2010.

[10] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[11] Gerard J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[12] Richard Hull. Formal study of business entities with lifecycles: Use cases, abstract models, and results. In Tevfik Bravetti, Mario; Bultan, editor, *7th International Workshop on Web Services and Formal Methods*, volume 6551 of *Lecture Notes in Computer Science*, 2010.

[13] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Jan Springintveld. Verifying business processes using spin. In *Proceedings of the 4th International SPIN Workshop*, pages 21–36, 1998.

[14] Jana Koehler, Joerg Hofstetter, and Roland Woodtly. Capabilities and levels of maturity in it-based case management. In *Business Process Mangement (BPM)*, LNCS. Springer Verlag, 2012.

[15] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *Process-oriented information systems in healthcare (ProHealth 08)*, pages 36–43. BPM 2008 Workshops, 2008.

[16] K.M. Lyng. Clinical guidelines in everyday praxis, implications for computerization. *Journal of Systems and Information Technology*, 2009.

[17] Raghava Rao Mukkamala. *A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs*. PhD thesis, IT University of Copenhagen, June 2012. http://www.itu.dk/people/rao/phd-thesis/DCRGraphs-rao-PhD-thesis.pdf.

[18] Raghava Rao Mukkamala. Formal verification of dcr graphs using spin. http://trustcare.itu.dk/dcrgraphs-verification/verificationWebUI.aspx, 2012.

[19] Raghava Rao Mukkamala and Thomas Hildebrandt. From dynamic condition response structures to büchi automata. In *Proceedings of 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010)*, August 2010.

[20] Nicolas Mundbrod, Jens Kolb, and Manfred Reichert. Towards a system support of collaborative knowledge work. In *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, LNBIP. Springer, September 2012.

[21] Object Management Group (OMG). Case management model and notation (cmmn). http://www.omg.org/spec/CMMN/, January 2013.

[22] Manfred Reichert and Peter Dadam. A framework for dynamic changes in workflow management systems. In *Conference on Database and Expert Systems Applications*, 1997.

[23] I. Rychkova and S. Nurcan. Towards adaptability and control for knowledge-intensive business processes: Declarative configurable process specifications. In *Hawaii International Conference on System Sciences*, 2011.

[24] Irina Rychkova. Towards automated support for case management processes with declarative configurable specifications. In *BPM Workshops*. Springer Berlin Heidelberg, 2013.

[25] Tijs Slaats. Dcr graphs editor. http://www.itu.dk/research/models/wiki/index.php/DCR_Graphs_Editor, February 2013.

[26] Tijs Slaats, Raghava Rao Mukkamala, Thomas Hildebrandt, and Morten Marquard. Exformatics declarative case management workflows as dcr graphs. In *International Conference on Business Process Management (BPM2013)*, 2013.

[27] Keith D. Swenson. *Mastering the Unpredictable: How Adaptive Case Management Will Revolutionize the Way That Knowledge Workers Get Things Done*. Meghan-Kiffer Press, 2010.

[28] KeithD. Swenson. Position: Bpmn is incompatible with acm. In Marcello Rosa and Pnina Soffer, editors, *BPM Workshops*, volume 132 of *Lecture Notes in Business Information Processing*, pages 55–58. Springer Berlin Heidelberg, 2013.

[29] W. M. P. Van Der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, September 2001.

[30] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D*, 23(2):99–113, 2009.

[31] Wil M.P van der Aalst and Maja Pesic. A declarative approach for flexible business processes management. In *Proceedings DPM 2006*, LNCS. Springer Verlag, 2006.