Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs

Thomas T. Hildebrandt Raghava Rao Mukkamala
{hilde,rao}@itu.dk
IT University of Copenhagen
Programming, Logic and Semantics Group
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark

We present *Dynamic Condition Response Graphs* (DCR Graphs) as a declarative, event-based process model inspired by the workflow language employed by our industrial partner and conservatively generalizing prime event structures. A dynamic condition response graph is a directed graph with nodes representing the events that can happen and arrows representing four relations between events: condition, response, include, and exclude. Distributed DCR Graphs is then obtained by assigning roles to events and principals. We give a graphical notation inspired by related work by van der Aalst et al. We exemplify the use of distributed DCR Graphs on a simple workflow taken from a field study at a Danish hospital, pointing out their flexibility compared to imperative workflow models. Finally we provide a mapping from DCR Graphs to Büchi-automata.

1 Introduction

A key difference between declarative and imperative process languages is that the control flow for the first kind is defined *implicitly* as a set of constraints or rules, and for the latter is defined *explicitly*, e.g. as a flow diagram or a sequence of state changing commands.

There is a long tradition for using declarative logic based languages to schedule transactions in the database community, see e.g. [15]. Several researchers have noted [12, 23, 24, 7, 3] that it could be an advantage to use a declarative approach to achieve more flexible process descriptions in other areas, in particular for the specification of case management workflow and ad hoc business processes. The increased flexibility is obtained in two ways: Firstly, since it is often complex to explicitly model all possible ways of fulfilling the requirements of a workflow, imperative descriptions easily lead to over-constrained control flows. In the declarative approach any execution fulfilling the constraints of the workflow is allowed, thereby leaving maximal flexibility in the execution. Secondly, adding a new constraint to an imperative process description often requires that the process code is completely rewritten, while the declarative approach just requires the extra constraint to be added. In other words, declarative models provide flexibility for the execution at run time and with respect to changes to the process.

As a simple motivating example, consider a hospital workflow extracted from a real-life study of paper-based oncology workflow at danish hospitals [19, 21]. As a start, we assume two events, *prescribe* and *sign*, representing a doctor adding a prescription of medicine to the patient record and signing it respectively. We assume the constraints stating that the doctor must sign after having added a prescription of medicine to the patient record and not to sign an empty record. A naive imperative process description may simply put the two actions in sequence, *prescribe;sign*, which allows the doctor first to prescribe medicine and then sign the record. In this way the possibilities of adding several prescriptions before or after signing and signing multiple times are lost, even if they are perfectly legal according to the

Submitted to:
PLACES 2010

constraints. The most general imperative description should start with the prescribe event, followed by a loop allowing either sign or prescribe events and only allow termination after a sign event. If the execution continues forever, it must be enforced that every prescription is eventually followed by a sign event.

With respect to the second type of flexibility, consider adding a new event *give*, representing a nurse giving the medicine to the patient, and the rule that a nurse must give medicine to the patient if it is prescribed by the doctor, but not before it has been signed. For the most general imperative description we should add the ability to execute the *give* event within the loop after the first *sign* event and not allow to terminate the flow if we have had a *prescribe* event without a subsequent *give* event. So, we have to change the code of the loop as well as the condition for exiting it.

In [4, 22], van der Aalst and Pesic propose to use Linear-time Temporal Logic (LTL) as a declarative language for describing the constraints of the workflow. LTL allows for describing a rich set of constraints on the execution flow. In particular, the first example workflow above is expressed as $(\mathbf{F}\text{Prescribe} \implies \neg\text{Sign}\,\mathbf{U}\,\text{Prescribe}) \wedge (\mathbf{G}(\text{Prescribe} \implies \mathbf{F}\text{Sign}))$, in words: "(Future Prescribe implies (not Sign Until Prescribe)) and (Globally (Prescribe implies Future Sign))". The expression becomes slightly more readable if the past modality is used: $(\mathbf{G}(\text{Sign} \implies \mathbf{P}\text{Prescribe}) \wedge (\mathbf{G}(\text{Prescribe} \implies \mathbf{F}\text{Sign}))$, in words: "(Globally (Sign implies Past Prescribe)) and (Globally (Prescribe implies Future Sign))". Since the notation of LTL is likely to be too difficult to use directly by the end user it is suggested to use a graphical notation for common patterns of temporal constraints which are then compiled to LTL. The example is then a combination of a *preceedence* pattern, $(\mathbf{G}(\text{Sign} \implies \mathbf{P}\text{Prescribe})$ and a *response* pattern $(\mathbf{G}(\text{Prescribe} \implies \mathbf{F}\text{Sign}))$. However, this approach suffers from the fact that the subsequent tools for execution and analysis will refer to the LTL expression (or further compilations to e.g. Büchi automata) and not the graphical notation. Also, the full generality of LTL may lead to a poor execution time.

This motivates researching the problem of finding an expressive declarative process language where both the constraints as well as the run time state can be easily visualized and understood by the end user and also allows an effective execution. We believe that the declarative process model language of *dynamic condition response graphs* and its graphical representation proposed in this paper is a promising candidate. The model is inspired by and a conservative generalization of the declarative *process matrix* model language [19, 21] used by our industrial partner and prime event structures [26].

We present distributed dynamic condition response graphs as a sequence of three generalizations of prime event structures. A prime event structure can be regarded as a minimal, declarative model for concurrent processes. It consists of a (possibly infinite) set of *events* (that can happen at most once), a (partial order) *causality relation* between events corresponding to the precedence LTL pattern above and a *conflict relation* stating which events can not happen in the same execution.

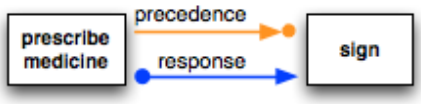


Figure 1: Prescribe and Sign Example

The first generalization, named *condition response event structures*, is obtained by adding a *response* relation between events and a set Re of *initially required response events*. The initially required response events can be regarded as goals that must be fulfilled (or falsified) in order for an execution to be accepting. That is, for any event $e \in \mathsf{Re}$, either e must eventually happen or it must become in conflict with an event that has happened in the past. The response relation in some sense corresponds to the response LTL pattern above as a dual relation to the usual causality relation: If an event b is a response to an event a then b must happen at some point after event a happens or become in conflict. However, note that the response pattern does not allow for conflicts. Operationally, as we will see in the following section, one can think of the event b as being added to the

set Re of required responses when a happens.

Next we generalize condition response event structures by allowing each event to happen many times and replacing the symmetric conflict relation by an asymmetric relation which *dynamically* determines which events are included in or excluded from the structure. To allow the graphs to represent intermediate run time state (e.g. like the marking of a Petri Net) we also add sets I and E of respectively included and executed events and refer to the triple of sets of pending responses, included and executed events as the *marking* of the graph. This results in the model of *Dynamic Condition Response Graphs*, short DCR Graphs.

Finally, we reach the model of *Distributed Dynamic Condition Response Graphs* allowing for role based *distribution* by adding a set of *principals* and a set of *roles* assigned to both principals and events, and define that an event can only be executed by a principal assigned one of the roles assigned to the event.

Being based on only four relations between events (condition, response, include, exclude) and the role assignment, the distributed dynamic condition response graphs can be simply visualized as a directed graph with a box for each event as nodes and four different kinds of arrows. We base our graphical notation for the condition and response relations on the notation suggested in [3] for precendence and response LTL patterns, since they coincide when no events are excluded. The inclusion and exclusion relations are denoted by arrows with a $+$ and $\%$ sign at the head respectively. We label each node with the activity of the event and add a small box to the top containing the roles that can execute the event. We annotate the graph by the marking, showing if an event is required as a response by adding a small exclamation mark, if it has happened in the past by a small check sign, and if it is excluded by making the box dashed. In addition we found it useful to show (by a small no-entry sign) if an event is blocked by an unfulfilled condition event, even though this information can be inferred from the condition relations and the currently included and executed events. We formalize the execution of dynamic condition response graph as a labelled transition system, which is finite state if the graph is finite. Indeed, the states of transition system will be markings consisting of triples of sets of executed, included, and required response events. We define a (finite or infinite) run of the labelled transition system to be accepting if no response event is forever continuously included and pending. We end by characterizing the execution semantics by providing a mapping of dynamic condition response graph to Büchi-automata.

The rest of the paper is structured as follows. In Sec. 2 below we recall the definition of prime event structures and introduce condition response event structures as the first generalization. We show how the response relation allows to represent the notion of weak fairness. In Sec. 3 we introduce the model of dynamic condition response graph (DCR Graphs) and distributed DCR Graphs. In Sec. 4 we provide a mapping from DCR Graphs to Büchi-automata with τ-events. In Sec. 5 we briefly address related work. Finally, we conclude and discuss current and future work in Sec. 6.

This paper replaces and extends the work presented in the two previous short papers [16] and [20]. The paper [16] introduced condition response event structures and dynamic condition response structures, which are essentially dynamic condition response graph without markings.The paper [20] provided a mapping from dynamic condition response structures to Büchi automata, but only capturing acceptance for the infinite runs. The mapping from dynamic condition response graphs to Büchi automata provided in the present paper characterizes also the acceptance of finite runs by introducing silent (τ) transitions in the Büchi automata.

2 Condition Response Event Structures

As an intermediate step towards dynamic condition response graphs, we generalize prime event structures to allow for a notion of *progress* based on a response relation. This model is interesting in itself as an extensional event-based model with progress, abstracting away from the intentional representation of repeated behavior. In particular we show that it allows for an elegant characterization of weakly fair runs of event structures.

First let us recall the definition of a prime event structure and configurations of such [26].

Definition 1 *A labeled* prime event structure *(*ES*) is a 5-tuple* $E = (\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *where*

(i) E *is a (possibly infinite) set of events*

(ii) Act *is the set of actions*

(iii) $\leq \subseteq \mathsf{E} \times \mathsf{E}$ *is the* causality *relation between events which is a partial order*

(iv) $\# \subseteq \mathsf{E} \times \mathsf{E}$ *is a binary* conflict *relation between events which is irreflexive and symmetric*

(v) $l : \mathsf{E} \to \mathsf{Act}$ *is the* labeling *function mapping events to actions*

The causality and conflict relations must satisfy the conditions that

1. $\forall e, e', e'' \in \mathsf{E}. e\#e' \leq e'' \implies e\#e''$
2. $\forall e \downarrow = \{e' \mid e' < e\}$ *is finite for any* $e \in \mathsf{E}$.

A configuration *of E is a set* $c \subseteq \mathsf{E}$ *of events satisfying the conditions*

1. conflict-free*:* $\forall e, e' \in c. \neg e\#e'$
2. downwards-closed*:* $\forall e \in c, e' \in \mathsf{E}. e' \leq e \implies e' \in c$

A run ρ *of E is a (possibly infinite) sequence of labelled events* $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *such that for all* $i \geq 0. \cup_{0 \leq j \leq i} \{e_j\}$ *is a configuration.*

A run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ *is* maximal *if any enabled event eventually happen or become in conflict, formally* $\forall e \in \mathsf{E}, i \geq 0. e \downarrow \subseteq (e_i \downarrow \cup \{e_i\}) \implies \exists j \geq 0. (e\#e_j \vee e = e_j)$.

Action names $a \in \mathsf{Act}$ represent the actions the system might perform, an event $e \in \mathsf{E}$ labelled with a represents occurrence of action a during the run of the system. The causality relation $e \leq e'$ means that event e is a prerequisite for the event e' and the conflict relation $e\#e'$ implies that events e and e' both can not happen in the same run, more precisely one excludes the occurrence of the other. The definition of maximal runs follows the definition of weak fairness for concurrency models in [8] and is equivalent to stating that the configuration defined by the events in the run is maximal with respect to inclusion of configurations.

We now generalize prime event structures to *condition response event structures*, by adding a dual *response* relation $\bullet\!\!\rightarrow$, such that $\{e' \mid e \bullet\!\!\rightarrow e'\}$ is the set of events that must happen (or be in conflict) after the event e has happened for a run to be accepting. The resulting structures, named *condition response event structures*, in this way add the possibility to state progress conditions. We also introduce a subset of the events Re of *initial* responses, which are events that are initially required eventually to happen (or become in conflict). In this way the structures can represent the state after an event has been executed. As we will see below, it also allows us to capture the notion of maximal runs.

Definition 2 A labeled *condition response event structure* (CRES) over an alphabet Act is a tuple $(\mathsf{E}, \mathsf{Re}, \mathsf{Act}, \rightarrow\!\!\bullet, \bullet\!\!\rightarrow, \#, l)$ where

(i) $(\mathsf{E}, \rightarrow\bullet, \#, l)$ is a labelled prime event structure, referred to as the underlying event structure

(ii) $\bullet\rightarrow \subseteq \mathsf{E} \times \mathsf{E}$ is the *response* relation between events, satisfying that $\rightarrow\bullet \cup \bullet\rightarrow$ is acyclic.

(iii) $\mathsf{Re} \subseteq \mathsf{E}$ is the set of *initial responses*.

We define a configuration c and run ρ of a CRES to be a respectively a configuration and run of the underlying event structure. We define a run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ to be *accepting* if $\forall e \in E, i \geq 0. e_i \bullet\rightarrow e \implies \exists j \geq 0. (e \# e_j \vee (i < j \wedge e = e_j)$ and $\forall e \in R. \exists j \geq 0. (e \# e_j \vee e = e_j)$. In words, any pending response event must eventually happen or be in conflict.

A prime event structure can trivially be regarded as a condition response event structure with empty response relation. This provides an embedding of prime event structures into condition response event structures which preserves configurations and runs.

Proposition 1 *The labelled prime event structure* $(\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *has the same runs as the* CRES $(\mathsf{E}, \emptyset, \mathsf{Act}, \leq, \emptyset, \#, l)$ *for which all runs are accepting.*

We can also embed event structures into CRES by considering every condition to be also a response and all events with no conditions to be initial responses. This characterizes the interpretation in [8] where only *maximal* runs are accepting. In other words, the embedding captures the notion of weakly fair execution of event structures.

Proposition 2 *The labelled prime event structure* $(\mathsf{E}, \mathsf{Act}, \leq, \#, l)$ *has the same runs and* maximal *runs as respectively the runs and the accepting runs of the* CRES $(\mathsf{E}, \{e \mid e \downarrow = \emptyset\}, \mathsf{Act}, \leq, \leq, \#, l)$.

3 Distributed Dynamic Condition Response Graphs

We now go on to generalize condition response event structures to dynamic condition response graphs (DCR Graphs). As opposed to event structures, a dynamic condition response graph allows events to be executed multiple times and there are no constraints on the condition and response relations. This allows for finite representations of infinite behavior, but also for introducing deadlocks. Moreover, the conflict relation is generalized to two relations for dynamic exclusion and inclusion of events, which is more appropriate in a model where events can be re-executed and has shown useful in practice as a primitive for skipping events and constraints.

Definition 3 *A* dynamic condition response graph *is a tuple* $G = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ *where*

(i) E *is the set of events*

(ii) $\mathsf{M} \in \mathscr{M}(G) = \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E})$ *is the* marking *and* $\mathscr{M}(G)$ *is the set of all markings*

(iii) Act *is the set of actions*

(iv) $\rightarrow\bullet \subseteq \mathsf{E} \times E$ *is the* condition *relation*

(v) $\bullet\rightarrow \subseteq \mathsf{E} \times E$ *is the* response *relation*

(vi) $\pm : \mathsf{E} \times \mathsf{E} \rightharpoonup \{+, \%\}$ *defines the* dynamic inclusion/exclusion *relations by* $e \rightarrow + e'$ *if* $\pm(e, e') = +$ *and* $e \rightarrow \% e'$ *if* $\pm(e, e') = \%$.

(vii) $l : \mathsf{E} \rightarrow \mathsf{Act}$ *is a labelling function mapping every event to an action.*

We let DCR Graphs *refer to the model of dynamic condition response graphs.*

The condition and response relations in DCR Graphs are similar to the corresponding relations in CRES, except that they are not constrained in any way. In particular, we may have cyclic relations. The marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}) \in \mathscr{M}(G)$ consists of three sets of events, capturing respectively which events have *previously been executed* (Ex), which events are *pending responses required to be executed or excluded* (Re), and finally which events are currently *included* (In). The set of pending responses Re of DCR Graphs thus plays the same role as the set of initial responses in the CRES.

The *dynamic inclusion/exclusion* relations $\rightarrow+$ and $\rightarrow\%$, represented by the (partial map) $\pm : \mathsf{E} \times \mathsf{E} \rightharpoonup \{+, \%\}$, allow events to be included and excluded dynamically in the graph. The intuition is that only the currently included events are considered in evaluating the constraints. This means that if event a has event b as condition, but event b is excluded from the graph then it is no longer required for a to happen. Also, if event a has event b as response and event b is excluded then it is no longer required to happen for the flow to be acceptable. Formally, the relation $e \rightarrow+ e'$ expresses that, whenever event e happens, it will include e' in the graph. On the other hand, $e \rightarrow\% e'$ expresses that when e happens it will exclude e' from the graph.

We define the execution semantics of DCR Graphs by a labelled transition system with markings as states and define the set of accepting runs by requiring that no event must be continuously included and pending.

Definition 4 *For a dynamic condition response graph $G = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l)$ we define the corresponding labelled transition systems $T(G)$ to be the tuple $(\mathscr{M}(G), \mathsf{M}, \rightarrow \subseteq \mathscr{M}(G) \times \mathsf{Act} \times \mathscr{M}(G))$ where $\mathscr{M}(G)$ is the set of markings of G, $\mathsf{M} \in \mathscr{M}(G)$ is the initial marking, $\rightarrow \subseteq \mathscr{M}(G) \times (\mathsf{E} \times \mathsf{Act}) \times (G)$ is the transition relation given by $\mathsf{M}' \xrightarrow{(e,a)} \mathsf{M}''$ where*

(i) $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ *is the marking before transition*

(ii) $\mathsf{M}'' = (\mathsf{Ex}' \cup \{e\}, \mathsf{Re}'', \mathsf{In}'')$ *is the marking after transition*

(iii) $e \in \mathsf{In}'$ *and* $l(e) = a$

(iv) $\{e' \in \mathsf{In}' \mid e' \rightarrow\bullet\ e\} \subseteq \mathsf{Ex}'$

(v) $\mathsf{In}'' = (\mathsf{In}' \cup \{e' \mid e \rightarrow+ e'\}) \setminus \{e' \mid e \rightarrow\% e'\}$

(vi) $\mathsf{Re}'' = (\mathsf{Re}' \setminus \{e\}) \cup \{e' \mid e \bullet\rightarrow e'\}$

We define a run $(e_0, a_0), (e_1, a_1), \ldots$ of the transition system to be a sequence of labels of a sequence of transitions $\mathsf{M}_i \xrightarrow{(e_i,a_i)} \mathsf{M}_{i+1}$ starting from the initial marking. We define a run to be accepting if $\forall i \geq 0, e \in \mathsf{Re}^i . \exists j \geq i . (e = e_j \vee e \notin \mathsf{In}^{j+1})$. In words, a run is accepting if no response event is continuously included and pending without it happens or become excluded.

The first two items in the above definition are markings before and after transition. The third item expresses that only events e that are currently included can be executed. A requirement saying that all currently included condition events for e should have been executed previously is expressed in iv. The next two items are the updates to the sets of included events and pending responses respectively. Note that an event e' can not be both included and excluded by the same event e, but an event may exclude itself. Also an event may trigger itself as a response and/or has itself as condition.

If one only want to consider finite runs, which is common for workflows, the acceptance condition degenerates to requiring that no pending response is included at the end of the run. This corresponds to defining all states where $\mathsf{Re} \cap \mathsf{In} = \emptyset$ to be accepting states and define the accepting runs to be those ending in an accepting state. If infinite runs are also of interest (as e.g. for reactive systems and the LTL

logic) the acceptance condition can be captured by a mapping to a Büchi-automaton with τ-events which we give in Sec. 4 below.

A CRES can be represented as a dynamic condition response graph by making every event exclude itself and encode the conflict relation by defining any two conflicting events to mutually exclude each other as shown in figure 2(b).

Proposition 3 *The* CRES $(\mathsf{E},\mathsf{Re},\mathsf{Act},\rightarrow\bullet,\bullet\rightarrow,\#,l)$ *has the same runs and accepting runs as the dynamic condition response graph* $(\mathsf{E},\mathsf{M},\mathsf{Act},\rightarrow\bullet,\bullet\rightarrow,\pm,l)$ *where marking* $\mathsf{M}=(\emptyset,\mathsf{Re},\mathsf{E}),\pm(e,e')=\%$ *if* $e=e'$ *or* $e\#e'$ *and undefined otherwise.*

(a) # relation in CRES (b) Encoding of # in DCR Graphs

Figure 2: Encoding conflicting events in CRES as mutual excluding events in DCR Graphs

We now define *distributed* dynamic condition response graphs by adding roles and principals.

Definition 5 *A* distributed *dynamic condition response graph is a tuple (G,* Roles*,* P*,* as*) where*

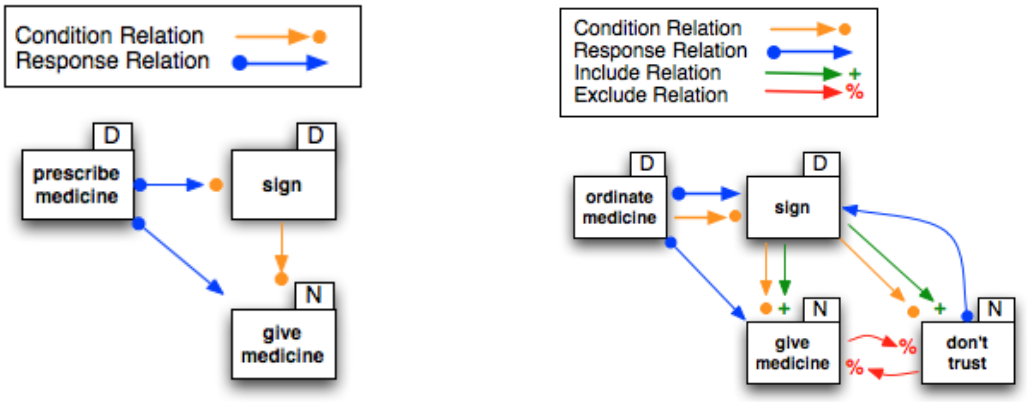
1. $G=(\mathsf{E},\mathsf{M},\mathsf{Act},\rightarrow\bullet,\bullet\rightarrow,\pm,l)$ *is a dynamic condition response graph,*
2. Roles *is a set of* roles*,*
3. P *is a set of* principals *(e.g. persons or processors) and*
4. $\mathsf{as}\subseteq(\mathsf{P}\cup\mathsf{Act})\times\mathsf{Roles}$ *is the role assignment relation to principals and actions.*

For a *distributed* DCR Graphs, the role assignment relation indicates the roles (access rights) assigned to principals and which roles gives right to execute which actions. As an example, assume $Peter\in\mathsf{P}$ and $Doctor\in\mathsf{Roles}$, then if $Peter\,\mathsf{as}\,Doctor$ and $Sign\,\mathsf{as}\,Doctor$ then $Peter$ as a doctor can sign as a doctor.

This is formalized by defining the labelled transition semantics for a *distributed* dynamic condition response graph $D=(G,\mathsf{Roles},\mathsf{P},\mathsf{as})$ to have the same states as the underlying dynamic condition response graph G, and the transitions $\rightarrow\subseteq\mathscr{M}(G)\times\mathsf{E}\times(\mathsf{P}\times\mathsf{Act}\times\mathsf{Roles})\times\mathscr{M}(G)$ defined by $\mathsf{M}'\xrightarrow{(e,(p,a,r))}\mathsf{M}''$ if $p\,\mathsf{as}\,r$ and $a\,\mathsf{as}\,r$ and $\mathsf{M}'\xrightarrow{(e,a)}\mathsf{M}''$ in the underlying dynamic condition response graph. We define a run to be (finite or infinite) sequence of labels $(e_0,(p_0,a_0,r_0))(e_1,(p_1,a_1,r_1))\ldots$ of a sequence of transitions $\mathsf{M}_i\xrightarrow{(e_i,(p_i,a_i,r_i))}\mathsf{M}_{i+1}$ starting from the initial marking. We define a run to be accepting if the underlying run of the DCR Graphs is accepting.

We are now ready to give the small example workflow from the introduction graphically as a distributed dynamic condition response graph shown in Fig. 3(a). It contains three events: prescribe medicine (the doctor calculates and writes the dose for the medicine), sign (the doctor certifies the correctness of the calculations) and give medicine (the nurse administers medicine to patient). The events are also labelled by the assigned roles (D for Doctor and N for Nurse).

The arrow $\bullet\rightarrow\bullet$ between prescribe medicine and sign indicates that the two events are related by both the condition relation and the response relation. The condition relation means that the prescribe medicine event must happen at least once before the sign event. The response relation enforces that, if the prescribe medicine event happen, subsequently at some point the sign event must happen for the flow to be accepted. Similarly, the response relation between prescribe medicine and give medicine



(a) Prescribe Medicine Example (b) Prescribe Medicine Example With Check

Figure 3: DCRS example in graphical notation

enforces that, if the prescribe medicine event happen, subsequently at some point the give medicine event must happen for the flow to be accepted. Finally, the condition relation between sign and give medicine enforces that the signature event must have happened before the medicine can be given. Note the nurse can give medicine many times, and that the doctor can at any point choose to prescribe new medicine and sign again. (This will not block the nurse from continue to give medicine. The interpretation is that the nurse may have to keep giving medicine according to the previous prescription).

The dynamic inclusion and exclusion of events is illustrated by an extension to the scenario (also taken from the real case study): If the nurse distrusts the prescription by the doctor, it should be possible to indicate it, and this action should force either a new prescription followed by a new signature or just a new signature. As long the new signature has not been added, medicine must not be given to the patient.

This scenario can be modeled as shown in Fig. 3(b), where one more action don't trust is added. Now, the nurse have a choice to indicate distrust of prescription and thereby avoid give medicine until the doctor re-execute sign action. Executing the don't trust action will exclude give medicine and makes the sign as pending response. So the only way to execute give medicine action is to re-execute sign action which will then include give medicine. Here the doctor may choose to re-do prescribe medicine followed by sign actions (new prescription) or simply re-do sign. The transition system for finite runs for the prescribe medicine example from Fig 3(a) is shown in the Fig. 4.

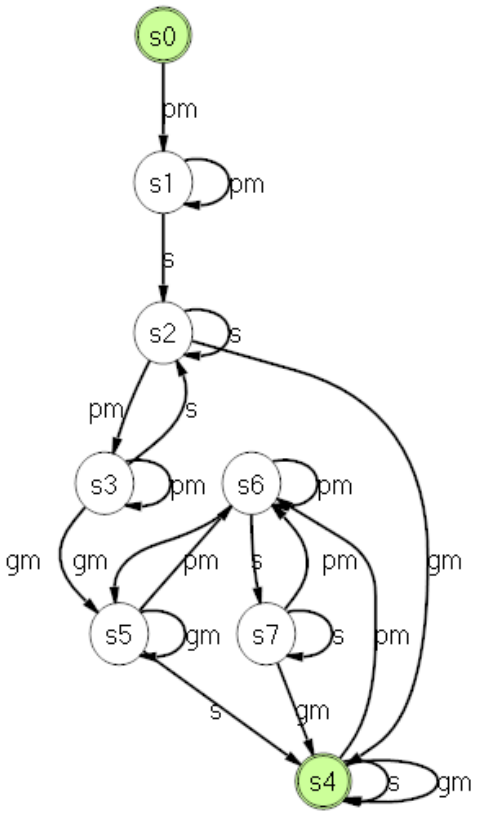
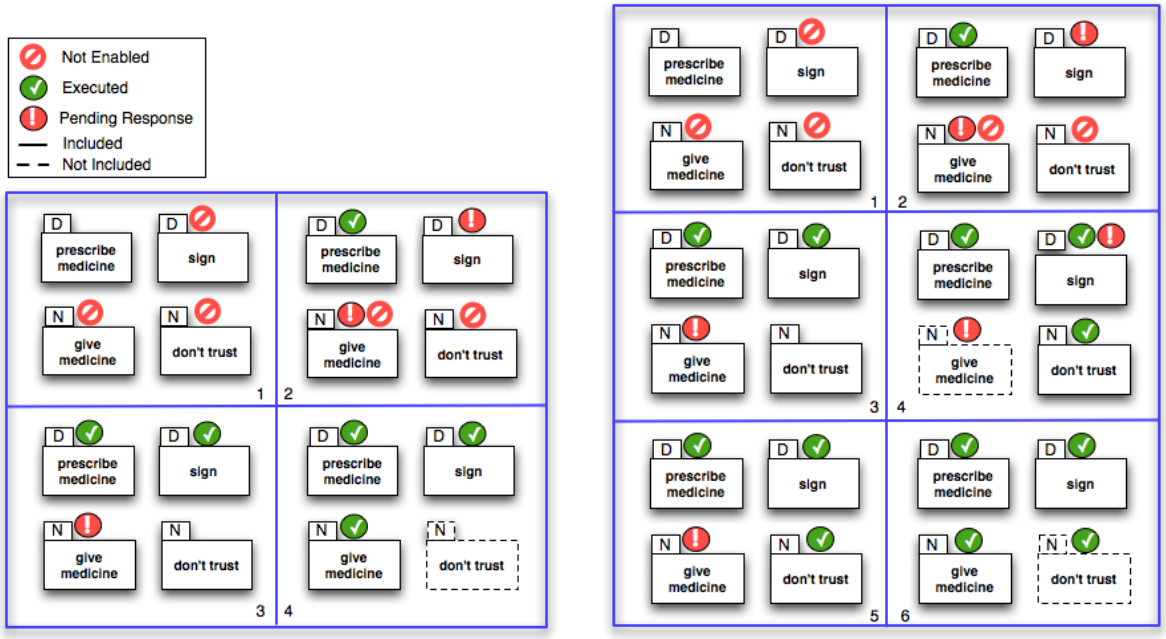


Figure 4: Transition system for DCR graph from Fig 3(a)

In Fig. 5 below we propose a graphical notation that illustrates the run-time information during two different runs of the extended scenario in Fig. 3(b). We show the events as boxes just as in the graphical notation for the dynamic condition response graph and use three different small icons (Ø, √,!) above the boxes to show

if the event is enabled (i.e. not blocked by any conditions), if it has been executed (i.e. included in the set E in the marking), and if it is required as a response (i.e. included in the set R in the marking). We indicate that an event is excluded (i.e. not included in the set I in the marking) by making the box around the event dashed.



(a) Prescribe Medicine Example

(b) Prescribe Medicine Example Don't trust

Figure 5: DCR Graphs Runtime state graphical notation

Fig 5(a) shows the four states of a run in the workflow process in Fig. 3(b), starting in the initial state where all events except prescribe medicine is blocked. The second state is the result of executing prescribe medicine, now showing that sign and give medicine are required as responses and that sign is no longer blocked. The third state is the result of executing the sign event, which enables give medicine and don't trust. Finally, the fourth state is the result of executing the give medicine event, excluding the don't trust event.

Similarly, Fig. 5(b) shows the six states of a run where the nurse executes don't trust in the third step, leading to a different fourth state where give medicine is excluded (but still required as response if it gets included again) and sign is required as response. The fifth state shows the result of the doctor executing sign, which re-includes give medicine, which is then executed, leading to the final state where all events have been executed, and don't trust is excluded.

4 From DCR Graphs to Büchi-automata

In this section, we show how to characterize the acceptance condition for DCR Graphs by a mapping to the standard model of Büchi-automata. Recal that a Büchi-automaton is a finite state automaton accepting only infinite runs, and only the runs that pass through an accepting state infinitely often. Acceptance of finite runs can be represented in the standard way by introducing a special silent event, e.g. a τ-event, which may be viewed as a delay. If an infinite accepting run contains infinitely many delays it then represent an accepting run containing only a finite number of (real) events. We define a Büchi-automaton

with τ-event as follows.

Definition 6 *A* Büchi-automaton with τ-event *is a tuple* $(S, s, Ev_\tau, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ *where* S *is the set of* states, $s \in S$ *is the* initial state, Ev_τ *is the set of* events *containing the special event* τ, $\rightarrow \subseteq S \times Ev_\tau \times S$ *is the* transition relation, *and* F *is the set of* accepting states. *A (finite or infinite) run is a sequence of labels not containing the* τ *event that can be obtained by removing all* τ *events from a sequence of labels of transitions starting from the initial state. The run is* accepting *if the sequence of transitions passes through an accepting state infinitely often.*

The mapping from DCR Graphs to Büchi-automata is not entirely trivial, since we at any given time may have several pending responses and thus must make sure that all of them are eventually executed or excluded. To make sure we progress, we assume any fixed order of the finite set of events E of the given dynamic condition response graph. For an event $e \in E$ we write $rank(e)$ for its rank in that order and for a subset of events $E' \subseteq E$ we write $min(E')$ for the event in E' with the minimal rank.

Definition 7 For a finite distributed dynamic condition response graph $G = (\mathsf{E}, \mathsf{M}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, \mathsf{Roles}, \mathsf{P}, \mathsf{as})$ where $\mathsf{E} = \{e_1, \ldots, e_n\}$ and $rank(e_i) = i$, we define the corresponding Büchi-automaton with τ-event to be the tuple $B(G) = (S, s, \rightarrow \subseteq S \times Ev_\tau \times S, F)$ where

- $S = \mathscr{M}(G) \times \{1, \ldots, n\} \times \{0, 1\}$ is the set of states,
- $Ev_\tau = (\mathsf{E} \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{Roles})) \cup \{\tau\}$ is the set of events,
- $s = (\mathsf{M}, 1, 1)$ if $I \cap \mathsf{Re} = \emptyset$, and $s = (\mathsf{M}, 1, 0)$ otherwise
- $F = \mathscr{M}(G) \times \{1, \ldots, n\} \times \{1\}$ is the set of accepting states and
- $\rightarrow \subseteq S \times Ev_\tau \times S$ is the transition relation given by

$$(\mathsf{M}', i, j) \xrightarrow{\tau} (\mathsf{M}', i, j') \text{ where}$$

(a) $j' = 1$ if $\mathsf{In}' \cap \mathsf{Re}' = \emptyset$ otherwise $j' = 0$.

and

$$(\mathsf{M}', i, j) \xrightarrow{(\mathrm{e},(\mathrm{p},\mathrm{a},\mathrm{r}))} (\mathsf{M}'', i', j') \text{ where}$$

(i) $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ and $\mathsf{M}'' = (\mathsf{Ex}' \cup \{e\}, \mathsf{Re}'', \mathsf{In}'')$

(ii) $\mathsf{M}' \xrightarrow{(\mathrm{e},(\mathrm{p},\mathrm{a},\mathrm{r}))} \mathsf{M}''$ is a transition of $T(D)$

(iii) $j' = 1$ if

(a) $\mathsf{In}'' \cap \mathsf{Re}'' = \emptyset$ or

(b) $min(M_r) \in (\mathsf{In}' \cap \mathsf{Re}' \setminus (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ or

(c) $M_r = \emptyset$ and $min(\mathsf{In}' \cap \mathsf{Re}') \in (\mathsf{In}' \cap \mathsf{Re}' \setminus (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$

otherwise $j' = 0$.

(iv) $i' = rank(min(M_r))$ if $min(M_r) \in (\mathsf{In}' \cap \mathsf{Re}' \setminus (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ or else

(v) $i' = rank(min(\mathsf{In}' \cap \mathsf{Re}'))$ if $M_r = \emptyset$ and $min(\mathsf{In}' \cap \mathsf{Re}') \in (\mathsf{In}' \cap \mathsf{Re}' \setminus (\mathsf{In}'' \cap \mathsf{Re}'')) \cup \{e\}$ or else

(vi) $i' = i$ otherwise.

for $M_r = \{e \in \mathsf{In}' \cap \mathsf{Re}' \mid rank(e) > i\}$.

In the marking M, the set Ex records the events that have been executed, where as In and Re records the events that are currently included and pending responses respectively. The index i is used to make sure that no event stays forever included and in the pending response set without being executed. Finally, the flag j indicates if the state is accepting or not.

Condition (a and iii) defines when a state is accepting. Either there are no included pending responses in the resulting state (iiia) or the included pending response with the minimal rank above the index i was either excluded or executed (iiib). Alternatively, if the set of included pending responses with rank above the index i is empty and the included pending response with the minimal rank is excluded or executed (iiic), then also the resulting state will be accepting. Condition (iv) records the new rank if the resulting state is accepting according to condition (iiib) and similarly when the state is accepting according to condition (iiic), the condition (v) records the new rank.

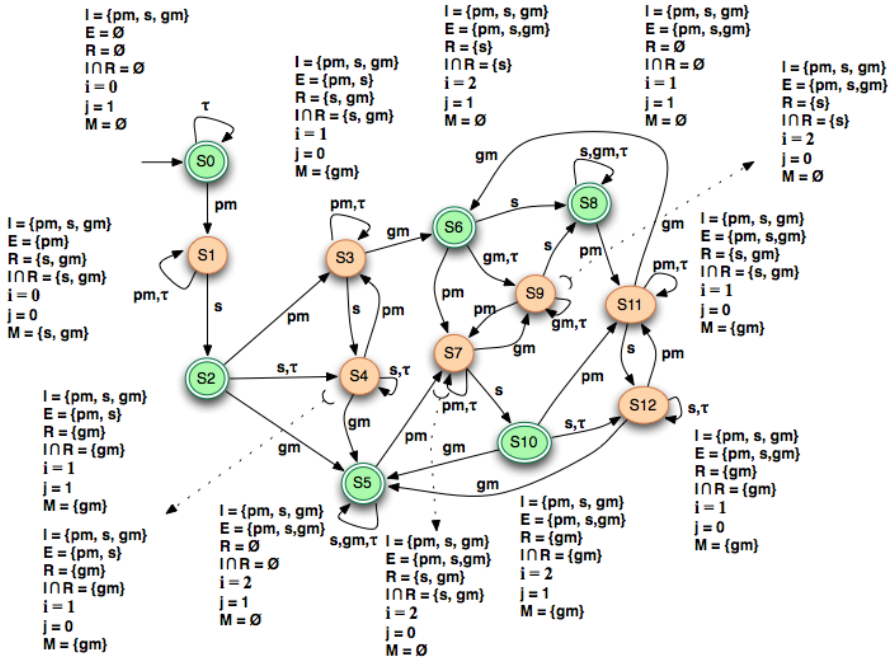


Figure 6: The Büchi-automaton for DCR Graph from Fig. 3(a) annotated with state information

To give a simple example of the mapping, let us consider the dynamic condition response graph in Fig. 3(a) and the corresponding Büchi-automaton in Fig. 6.

The key point to note is that the automaton enters an accepting state if there is no pending responses, or if the pending response which is the minimal ranked event according to the index i is executed or excluded. State $S7$ and $S11$ illustrate the use of the rank: Both states have the two events s (having rank 1) and gm as pending responses. In state $S7$ only executing event s leads to an accepting state ($S10$). The result of executing event gm is to move to state $S9$ which is not accepting. Dually, in state $S11$ only executing event gm leads to an accepting state ($S16$). The result of executing event s is to move to state $S12$ which is not accepting.

Fig. 7 shows a stratified view of the automaton, dividing the state sets according to the rank i in order to emphasize the role of the rank in guaranteeing progress.

We end by stating the main theorem that the mapping from dynamic condition response graph to Büchi-automata characterizes the execution semantics.

Theorem 1 *For a finite distributed dynamic condition response graph D the Büchi-automaton with* τ-

event $B(D)$ has the same runs and accepting runs as D.

5 Related Work

There exists many different approaches to formally specify and enact business processes/workflows. As it is not possible to provide a complete overview of all related work, here we give a brief overview of some of the formalisms which are related to our work and make a comparison against them.

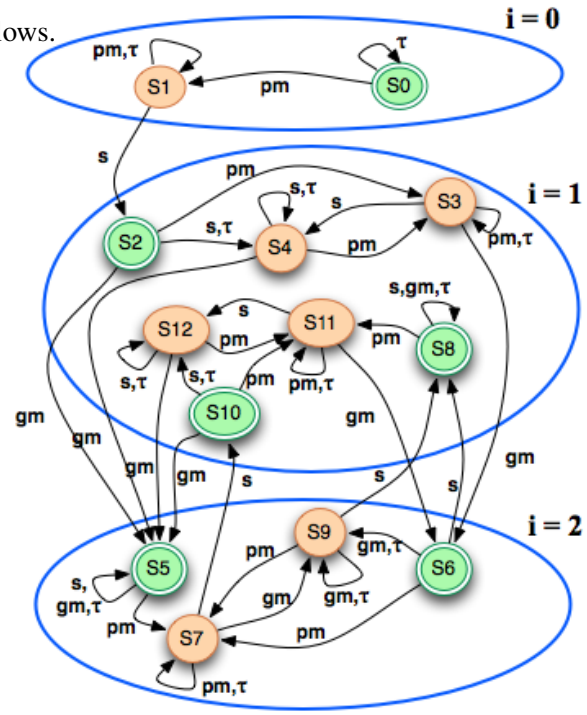


Figure 7: The Büchi-automaton with stratified view

On contrary to imperative modeling languages, the authors in [4, 3] have proposed *ConDec*, a declarative language for modeling and enacting the dynamic business processes based on Linear Temporal Logic (LTL). In [5], the authors have proposed Declarative Service Flow language (DecSerFlow) to specify, enact and monitor service flows, which is a sister language for *ConDec*. Both the languages share the same concepts and are supported in the *Declare* [3] tool. They specifies *what* should be done, instead of specifying *how* it should be done, there by leaving more flexibility to users. The enactment in both the languages is defined by translating the constraints specified in LTL, into a Buchi automaton and executing the workflow/service by executing the referring Buchi automaton. LTL being a very expressive language, the Declare tool suffers from efficiency problems in executing models with large specification [3]. Even though our approach is inspired from the *ConDec, DecSerFlow* models [4, 5] in respect of using constraints based approach to model processes, our model has fewer primitives than LTL and that makes it more suitable for expressing and executing declarative workflows.

In [9, 10] the authors have proposed *Event Calculus* [18] as a logic-based methodology for specification and execution of workflows. *Event Calculus* [18] is a logic programming formalism for representing events and their effects in the context of database applications. In their approach, the authors have expressed the basic control flow primitives as a set of logical formulas and used axioms of *Event Calculus* to specify activity dependency execution and agent assignments rules. Their workflow model also supports enactment and iteration of activities, but does not support verification of global and temporal constraints on workflow activities. Also, their approach is limited to imperative/procedural workflow modeling languages.

Concurrent Transaction Logic (CTR) used in [12] as a language for specifying, analysis, scheduling and verification of workflows. In their framework, the authors have used CTR formulas for expressing the local and global properties of workflows and reasoning about the workflows has been done with the help of proof theory and semantics of logic. In [23], the authors have used Concurrent Constraint Transaction Logic (CCTR) which is flavor of CTR integrated with Constraint Logic Programming for scheduling workflows. Like the other logic programming systems, the authors in [12, 23] have used the

proof theory of CTR as run-time environment for enactment of workflows. The CTR approach mainly aims at developing an algorithm for consistency checking and verification of properties of workflows, but only limited to imperative modeling languages.

Petri nets has been the major formalism that has been studied and used extensively in the domain of workflows and business processes. There are many researchers [2, 13, 1]who have used Petri-nets and its extensions of to model workflows and explored the problem of soundness in workflows and its correctness criteria.

Our formal model DCR Graphs also relates to the declarative approaches used by *Guard-Stage-Milestone* (GSM) model [17] by Hull et al, presented as an invited talk at the WS-FM 2010 workshop. The GSM meta-model uses declarative approach for specification of life cycles, which is part of research on data driven artifact-centric business processes [6, 14, 11], carried out by the IBM Research. The operational semantics of GSM are based on Event-Condition-Action (ECA) rules, and provide a basis for formal verification and reasoning.

6 Conclusion and Future Work

We have presented dynamic condition response graphs, short DCR Graphs, as a new declarative, event-based workflow process model inspired by the workflow language employed by our industrial partner [21]. We have demonstrated the use and flexibility of the model on a small example taken from a field study on danish hospitals [19] and proposed a graphical notation for presenting both the processes and their run-time state.

The model was presented as a sequence of generalizations of the classical model for concurrency of prime event structures [26]. The first generalization introduced a notion of progress to event structures by replacing the usual causal order by two dual relations, a *condition* relation $\rightarrow\bullet$ expressing for each event which events it has as preconditions and a *response* relation $\bullet\rightarrow$ expressing for each event which events that must happen (or be ruled out) after it has happened. We prove in particular, that the resulting model, named *condition response event structures* can express the standard notion of weak concurrency fairness.

The next generalization is to allow for finite representations of infinite behaviours by allowing *multiple execution*, and *dynamic inclusion* and *exclusion* of events, resulting in the model of *dynamic condition response graphs*.

Finally, we extended the model to allow distribution of events via roles and presented a graphical notation inspired by related work by van der Aalst et al. [4, 3], but extended to include information about the run-time state (e.g. markings).

We prove that all generalizations conservatively contain the previous model. Moreover, we provide a mapping from dynamic condition response graphs to Büchi-automata characterising the acceptance condition for finite and infinite runs.

One key advantage of the dynamic condition response graphs compared to the related work explored in [4, 3, 12, 10] is that the latter logics are more complex to visualize and understand by people not trained in logic. Another advantage, illustrated in the given mapping to Büchi-automata and our graphical visualization of the run time state, is that the execution of dynamic condition response graphs can be based on a relatively simple information about the run-time state, which can also be visualized directly as annotations (marking) on the graph. We have implemented a prototype engine and mapping to the input format for the SPIN model checker, and are currently working on implementing a simulator for DCRS able to visualize the state graphically in this way.

Current and future work include studying extensions of the DCR Graphs model with time, exceptions, nesting (sub processes) and data (as publish/subscribe events to changes of data) and the relationship between DCR Graphs and Petri Net with time and infinite behavior [25]. Also we are investigating the expressiveness of the model compared to LTL and the work in [4]. Along the line of work in [27] we investigate the definition of interfaces between concurrently interacting dynamic condition response graphs, a (categorical) theory of simulations and defining unfolding (forgetful) mappings from DCR Graphs to CRES and from CRES to event structures. We expect the theory to be useful in achieving compositional design and verification of workflow processes, as well as studying the impact of adapting or adding new interacting workflow processes to a pool of processes. Finally we intend to explore the relation to the recent work on event-based business processes.

Acknowlegments

This research is supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project and the Computer Supported Mobile Adaptive Business Processes (CosmoBiz) project. Danish Research Agency, Grants # 2106-07-0019 and # 274-06-0415 (www.TrustCare.eu and www.CosmoBiz.dk).

References

[1] W. van der Aalst, K. van Hee, A. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve & M. Wynn (2011): *Soundness of workflow nets: classification, decidability, and analysis*. Formal Aspects of Computing 23, pp. 333–363.

[2] W. M. P. van der Aalst (1998): *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers 8(1), pp. 21–66.

[3] Wil M. P. van der Aalst, Maja Pesic & Helen Schonenberg (2009): *Declarative workflows: Balancing between flexibility and support*. Computer Science - R&D 23(2), pp. 99–113. Available at `http://dx.doi.org/10.1007/s00450-009-0057-9`.

[4] Wil M.P van der Aalst & Maja Pesic (2006): *A Declarative Approach for Flexible Business Processes Management*. In: Proceedings DPM 2006, LNCS, Springer Verlag.

[5] Wil M.P van der Aalst & Maja Pesic (2006): *DecSerFlow: Towards a Truly Declarative Service Flow Language*. In: M. Bravetti, M. Nunez & Gianluigi Zavattaro, editors: Proceedings of Web Services and Formal Methods (WS-FM 2006), LNCS 4184, Springer Verlag, pp. 1–23.

[6] Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu & Jianwen Su (2007): *Towards formal analysis of artifact-centric business process models*. In: In preparation, pp. 288–304.

[7] Christoph Bussler & Stefan Jablonski (1994): *Implementing agent coordination for workflow management systems using active database systems*. In: Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on, pp. 53–59.

[8] Allan Cheng (1995): *Petri Nets, Traces, and Local Model Checking*. In: Proceedings of AMAST, pp. 322–337.

[9] Nihan K. Cicekli & Yakup Yildirim (2000): *Formalizing Workflows Using the Event Calculus*. pp. 222+. Available at `http://www.springerlink.com/content/kgxdp8llady8t1jd`.

[10] Nihan Kesim Cicekli & Ilyas Cicekli (2006): *Formalizing the specification and execution of workflows using the event calculus*. Information Sciences 176(15), pp. 2227 – 2267. Available at `http://www.sciencedirect.com/science/article/B6V0C-4HMGKHK-1/2/8d83da5d71878f08a894fa8b9deb9933`.

[11] David Cohn & Richard Hull (2009): *Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes*. *IEEE Data Eng. Bull.* 32(3), pp. 3–9. Available at `http://sites.computer.org/debull/A09sept/david.pdf`.

[12] Hasam Davulcu, Michael Kifer, C. R. Ramakrishnan & I.V. Ramakrishnan (1998): *Logic Based Modeling and Analysis of Workflows*. In: *Proceedings of ACM SIGACT-SIGMOD-SIGART*, ACM Press, pp. 1–3.

[13] J Desel & Thomas Erwin (2000): *Modeling, Simulation and Analysis of Business Processes*. In: Wil van der Aalst, J Desel & Andreas Oberweis, editors: *Business Process Management*, *Lecture Notes in Computer Science* 1806, Springer Berlin / Heidelberg, pp. 247–288.

[14] Alin Deutsch, Richard Hull, Fabio Patrizi & Victor Vianu (2009): *Automatic verification of data-centric business processes*. In: *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, ACM, New York, NY, USA, pp. 252–267. Available at `http://doi.acm.org/10.1145/1514894.1514924`.

[15] Alvaro A. A. Fernandes, M. Howard Williams & Norman W. Paton (1997): *A logic-based integration of active and deductive databases*. *New Gen. Comput.* 15(2), pp. 205–244.

[16] Thomas Hildebrandt & Raghava Rao Mukkamala (2010): *Distributed Dynamic Condition Response Structures*. In: *Pre-proceedings of International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10)*. Available at `http://www.itu.dk/people/rao/rao_files/dcrsplacescamredver.pdf`.

[17] Richard Hull (2010): *Formal Study of Business Entities with Lifecycles: Use Cases, Abstract Models, and Results*. In: *Proceedings of 7th International Workshop on Web Services and Formal Methods*, *Lecture Notes in Computer Science* 6551.

[18] Robert Kowalski (1992): *Database updates in the event calculus*. *J. Log. Program.* 12(1-2), pp. 121–146.

[19] Karen Marie Lyng, Thomas Hildebrandt & Raghava Rao Mukkamala (2008): *From Paper Based Clinical Practice Guidelines to Declarative Workflow Management*. In: *Proceedings ProHealth 08 workshop*. Available at `http://www.itu.dk/people/hilde/Papers/ProHealth08.pdf`.

[20] Raghava Rao Mukkamala & Thomas Hildebrandt (2010): *From Dynamic Condition Response Structures to Büchi Automata*. In: *Proceedings of 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010)*. Available at `http://www.itu.dk/people/rao/rao_files/dcrsextendedabstractTase2010.pdf`.

[21] Raghava Rao Mukkamala, Thomas Hildebrandt & Janus Boris Tøth (2008): *The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL*. In: *Proceeding of DDBP*.

[22] Maja Pesic (2008): *Constraint-Based Workflow Management Systems: Shifting Control to Users*. Ph.D. thesis, Eindhoven University of Technology, Netherlands.

[23] Pinar Senkul, Michael Kifer & Ismail H. Toroslu (2002): *A Logical Framework for Scheduling Workflows Under Resource Allocation Constraints*. In: *In VLDB*, pp. 694–705.

[24] Munindar P. Singh, Greg Meredith, Christine Tomlinson & Paul C. Attie (1995): *An Event Algebra for Specifying and Scheduling Workflows*. In: *Proceedings of DASFAA*, World Scientific Press, pp. 53–60.

[25] Rüdiger Valk & Heino Carstensen (1985): *Infinite Behaviour and Fairness in Petri Nets*. In: G. Rozenberg, editor: *Advances in Petri Nets 1984*, *Lecture Notes in Computer Science* 88, Springer-Verlag, pp. 83–100.

[26] Glynn Winskel (1986): *Event Structures*. In: Wilfried Brauer, Wolfgang Reisig & Grzegorz Rozenberg, editors: *Advances in Petri Nets*, *Lecture Notes in Computer Science* 255, Springer, pp. 325–392.

[27] Glynn Winskel & Mogens Nielsen (1995): *Models for concurrency* , pp. 1–148.