From Dynamic Condition Response Structures to Büchi Automata

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

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

***Abstract*—Recently we have presented distributed *dynamic condition response structures* (DCR structures) as a declarative process model conservatively generalizing labelled event structures to allow for finite specifications of repeated, possibly infinite behavior. The key ideas are to split the causality relation of event structures in two dual relations: the condition relation and the response relation, to split the conflict relation in two relations: the dynamic exclusion and dynamic inclusion, and finally to allow configurations to be multi sets of events. In the present abstract we recall the model and show how to characterise the execution of DCR structures and the acceptance condition for infinite runs by giving a map to Büchi-automata. This is the first step towards automatic verification of processes specified as DCR structures.**

I. 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. Several authors have noted that it could be an advantage to also use a declarative approach to specify workflow and business processes [3], [8], [9]. An important motivation for considering a declarative approach is to achieve more flexible process descriptions [9]. The increased flexibility is obtained in two ways: Firstly, imperative descriptions tend to over-constrain the control flow, since one does not think of all possible ways of fulfilling the intended constraints, whereas the declarative modes specify the process to the minimum extent. Secondly, changing or adding a new constraint to an imperative process description may require that the process code is completely rewritten, while the declarative approach just requires the constraint to be changed or added.

A drawback of the declarative approach however, is that the implicit definition of the control flow makes the flow less easy to perceive for the user or to compute by the execution engine. At each state, one has to solve the set of constraints to figure out what are the next possible events. It becomes even worse if you are not only interested in knowing the immediate next event, but also want to get an overview of the complete run of the process.

This motivates researching the problem of finding an expressive declarative process model language that can be easily visualized by the end user, allows an effective run-time scheduling and can be mapped easily to a state based model if an overview of the flow graph is needed. In [4] we have proposed a new such declarative process model language called *dynamic condition response structures* (DCR). The model is inspired by the declarative *process matrix* model language [6], [7] used by our industrial partner, Resultmaker A/S, and (labelled) prime event structures [10]. Indeed, it is formally a conservative generalization and strict extension of both event structures and the core primitives of the process matrix model language. In [4] we gave a labelled transition semantics for DCR structures and formalized acceptance for finite runs. In the present abstract we first recall the model and then show how to characterize the execution of DCR structures and the acceptance condition for infinite runs by giving a map to Büchi-automata. This is the first step towards automatic verification of processes specified as DCR structures.

II. Dynamic Condition Response Structures

An event structure [10] can be regarded as a minimal, declarative model for concurrent processes. A labelled prime event structure consists of a set of events, a causality (partial order) relation between events, a binary conflict relation, and a labeling function describing the observable action name of each event. The causality relation states which events are caused by the previous events (satisfying a condition of finite cause), or dually, which events are preconditions for the execution of an event. The conflict relation states which events can not happen in the same execution, satisfying a condition of hereditary conflict.

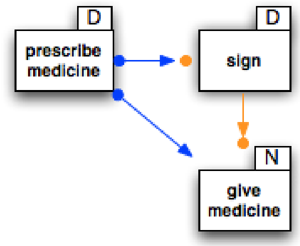
To use event structures as an execution language for concurrent reactive systems, several aspects are missing. In particular, each event can only be executed once and there is no notion of acceptance condition (except taking all or only maximal traces to be accepting). Instead, models such as Petri Net, (asynchronous) transition systems or Büchi-automata have been used as execution languages and the model of event structures as an abstract denotational model. In the paper [4], we have shown how to extend the declarative model of event structures to allow for finite specifications of repeated, possibly infinite behavior. Firstly, we allow each event to happen many times and replace the symmetric conflict relation by an asymmetric relation which *dynamically* determines which events are included in or excluded from the structure. Secondly, the causality relation is split in two relations (not necessarily partial orders): A *condition* relation stating which events must have happened before an event and a *response* relation stating which events must happen after (as a response to) an event. Note that this is slightly more general than the may and must transitions of Modal Transition Systems [5], since we do not require the response relation to be a subset of the condition relation. We then defined a run of a DCR stucture as a sequence of events satisfying that if an event e is executed, it must be currently included and all its currently included conditions have been executed previously. We defined a finite run to be accepting if it has no pending responses. Finally, since we wanted to apply the model to workflow processes we defined *distributed* DCR structures by adding a set of roles assigned to persons/processors and actions. Being based on essentially only four relations between events, our model can be simply visualized as a directed graph with events (labelled by activities and roles) as nodes and four different kinds of arrow as exemplified in Fig. 1. We use the same notation for condition and response relations as [9]. Note that the work *ibid.* differs from our approach by allowing any LTL template (thus increasing the complexity), but not allowing the dynamic inclusion/exclusion relations.

In this short paper (and poster) we consider infinite runs, which are defined to be accepting if no event from some point in the execution, is required as response and included continuously without ever being executed. This relates to the elegant definition of fair runs in true concurrency models investigated in [2], but differs in that we distinguish between required events (responses) and allowed events (events for which all condition events have been executed previously).

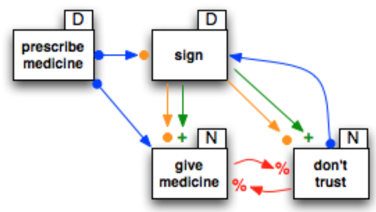
Before giving the formal definitions let us consider a small part of a hospital workflow extracted from a study of paper-based oncology workflow at danish hospitals [6], with a rule stating that the doctor must sign after adding medical prescription to the patient record. A naive imperative process description may instruct the doctor first to prescribe medicine and then sign it. In this way the possibility of adding several prescriptions before or after signing is lost, even if it is perfectly legal according to the declaratively given rule. With respect to the second type of flexibility, consider adding the rule that a nurse should give the prescribed medicine to the patient, but it is not allowed before the patient record has been signed. A simple imperative solution may be to just add a command in the end of the program instructing the nurse to give the medicine. Perhaps we remember to insert a loop to allow that the nurse give the medicine repeatedly. But the nurse should be allowed to give medicine as soon as the first signature is put and the doctor should also be allowed to add new prescriptions after or even at the same time as the nurse gives the medicine. So, the most flexible imperative description should in fact spawn a new thread for the nurse after the first signature has been given.

The example described above is modeled using DCR graphical notation in figure 1(a). It contains three events uniquely labelled (and thus identified) by the actions: prescribe medicine, sign and give medicine. The events are also labelled by the assigned roles (D for Doctor and N for Nurse). The arrow •→• 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 happens, subsequently at some point of time, the sign event must happen for the flow to be accepted. Similarly, the response relation between prescribe medicine and give medicine enforces that, if the prescribe medicine event happens, subsequently later the give medicine event must happen for the flow to be accepted. Finally, the condition relation between sign and give medicine enforces that the sign 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 chose 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 is modeled in Figure 1(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



(a) Prescribe Medicine Example



(b) Prescribe Medicine Example With Check

Figure 1. DCRS example in graphical notation

the doctor re-executes the sign action. Executing the don't trust action will exclude give medicine and make the sign as pending response. So the only way to execute the give medicine action is to re-execute thesign 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.

We are now ready for the formal definitions.

Definition 1: A *dynamic condition response structure* (DCR) is a tuple $D = (\mathsf{E}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \pm, l)$ where

(i) E is the set of events

(ii) Act is the set of actions

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

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

(v) $\pm : \mathsf{E} \times \mathsf{E} \rightarrow \{+, \%, *\}$ is the *dynamic inclusion/exclusion* relation.

(vi) $l : \mathsf{E} \rightarrow \mathsf{Act}$ is a labelling function mapping events to actions.

Finally, we define *distributed* dynamic condition response structures by adding roles and principals.

Definition 2: A *distributed* dynamic condition response structure(DDCR) is a tuple

$$(\mathsf{E}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$$

where $(\mathsf{E}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \pm, l)$ is a dynamic condition response structure, R is a set of *roles*, P is a set of *principals* (e.g. persons/processors/agents) and $\mathsf{as} \subseteq (\mathsf{P} \cup \mathsf{Act}) \times \mathsf{R}$ is the role assignment relation to executors and actions.

For a *distributed* DCR, the role assignment relation indicates the roles of principals and which roles gives permission to execute which actions. As an example, if $Peter \,\mathsf{as}\, Doctor$ and $Sign \,\mathsf{as}\, Doctor$ (for $Peter \in \mathsf{P}$ and $Doctor \in \mathsf{R}$, then $Peter$ can do the $Sign$ action having the role as $Doctor$.

We now show how to characterize the execution and acceptance criteria for infinite runs for DCR structures by giving a mapping from finite DCR structures to Büchi-automata. Recall that the acceptance criteria is that "no event from some point in the execution is required as response and included continuously without ever being executed". The mapping 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 DCR structure. 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 3: For a finite distributed DCR $D = (\mathsf{E}, \mathsf{Act}, \rightarrow\!\bullet, \bullet\!\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$ where $\mathsf{E} = \{e_1, \ldots, e_n\}$ we define the corresponding Büchi-automaton $Aut(D)$ to be the tuple $(S, s, \rightarrow \subseteq S \times \mathsf{Act} \times S, F)$ where $S = \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \{1, \ldots, n\} \times \{0, 1\}$ is the set of states and $s = (\emptyset, \mathsf{E}, \emptyset, 1, 1) \in S$ is the initial state and $F = \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \mathcal{P}(\mathsf{E}) \times \{1, \ldots, n\} \times \{1\}$ is the set of accepting states. $\rightarrow \subseteq S \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{R}) \times S$ is the transition relation given by

$$(E, I, R, i, j) \xrightarrow{\text{(p,a,r)}} (E \cup \{e\}, I', R', i', j') \text{ where}$$

(i) $e \in I$, $l(e) = a$, $p \,\mathsf{as}\, r$, and $a \,\mathsf{as}\, r$

(ii) $\{e' \in I \mid e' \rightarrow\!\bullet\, e\} \subseteq E$

(iii) $I' = (I \cup \{e' \mid \pm(e, e') = +\}) \setminus \{e' \mid \pm(e, e') = \%\}$

(iv) $R' = (R \setminus \{e\}) \cup \{e' \mid e \bullet\!\rightarrow e'\}$

(v) $j' = 1$ if

a) $I' \cap R' = \emptyset$ or

b) $min(M) \in (I \cap R \backslash (I' \cap R')) \cup \{e\}$ or

c) $M = \emptyset$ and $min(I \cap R) \in (I \cap R \backslash (I' \cap R')) \cup \{e\}$

otherwise $j = 0$.

(vi) $i' = rank(min(M))$ if $min(M) \in (I \cap R \backslash (I' \cap R')) \cup \{e\}$ or else

(vii) $i' = rank(min(I \cap R))$ if $M = \emptyset$ and $min(I \cap R) \in (I \cap R \backslash (I' \cap R')) \cup \{e\}$ or else

(viii) $i' = i$ otherwise.

for $M = \{e \in I \cap R \mid rank(e) \rangle\, i\}$.

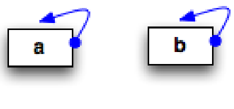
The set E in each state of the automaton records the events that have been executed. The set I records the events that are currently included. The set R records the pending responses. The index i is used to make sure that no event stays forever included and in the response set without being executed. Finally, the flag j indicates if the state is accepting or not.

Condition (i) captures that the executed event must be currently included (i.e. in the set I), and record in the label the principal, action and role. Condition (ii) captures that all the currently included conditions for the executed event must have been executed. Condition (iii) captures the dynamic inclusion and exclusion of events. Condition (iv) removes the currently executed event from the pending response set R and adds new pending responses (which may include the currently executed event as we will see below). Condition (v) defines when a state is accepting. Either (va) there are no pending responses in the resulting state which are also included. Or (vb), the included pending response with the minimal rank above the index i was excluded or executed. Or (vc), there is no included pending response with rank above the index i and the included pending response with the minimal rank was excluded or executed. Condition (vi) records the new rank if the resulting state is accepting according to condition (vb). Condition (vii) records the new rank if the resulting state is accepting according to condition (vc).

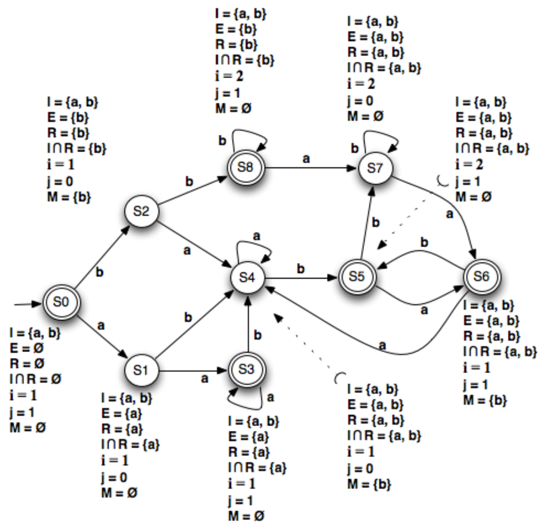
To give a simple example of the mapping consider the DCR in Fig. 2(a) and the corresponding generalized Büchi-automaton in Fig. 2(b). The structure consists of two events, a and b, having themselves as responses. The accepting runs are all infinite runs which contain either none or an infinite number a events and similarly for b events.

The key point to note is that the automaton enters an accepting state if a pending response is executed, but *only if it is the minimal ranked event according to the index i*. So, if a is executed in state $S4$ we do not enter an accepting state, even if a is a pending response, because event b is also a pending response and it is the one to be executed according to the rank i. Dually, in state $S7$ only an a event will take us to an accepting state, even though b is also a pending response.

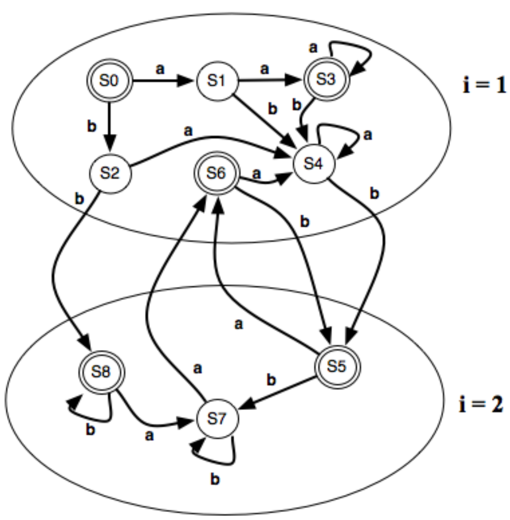
Fig. 2(c) shows a stratified view of the automaton, dividing the state sets according to the rank i in order to emphazise the role of the rank in guaranteeing that both a and b events get executed infinitely often if they are executed at least once.



(a) DCRS Graphical Notation



(b) Büchi automaton



(c) Stratified automaton

Figure 2. Fair interleaving of two events repeated infinitely (if ever executed)

We end by stating the soundness and completeness of the mapping from DCR structures to Büchi-automata.

Theorem 1: For a finite distributed DCR structure D the Büchi-automaton $Aut(D)$ has exactly the valid runs of D and accepts the infinite runs in which no event from some point in the execution is required as response and included continuously without ever being executed.

III. Conclusion and Future Work

The model of DCR structures is a declarative process model introduced and studied for finite runs in [4]. It is derived as a simple generalization of labelled event structures that adds the possibility to specify repeated, possibly infinite behavior and acceptance conditions for infinite runs by allowing events to be executed several times by default, generalizing the conflict relation to a dynamic inclusion/exclusion relation, and employing a response relation dual to that of precedence. In this paper, we have defined a natural notion of acceptance condition for *infinite* runs of DCR structures and showed how to characterize it by a mapping to Büchi automata. The mapping 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. We believe that the DCR model and its graphical presentation is much simpler to understand than Büchi-automata, e.g. as illustrated by comparing Fig. 2(a) and Fig. 2(b). Also, the declarative nature of the model provides the required flexibility in execution and with respect to addition/change of constraints which is not provided by Büchi-automata or Petri Nets. Finally, its execution is much simpler to understand and describe than Linear-time Temporal Logic (LTL) which is the basis of the work in [9].

For future work we will consider a more detailed comparison between DCR and existing models for concurrency, including the relation to LTL and the work in [2], [9], [1]. We also plan to develop an algebra and simulation relations for DCR structures, and study distributed scheduling and extensions to the model, notably with time, nested sub structures, soft constraints, and compensation/exceptions.

References

[1] B. Bérard, F Cassez, S. Haddad, D. Lime, and O.H. Roux. Comparison of the expressiveness of timed automata and time petri nets. In *Proceedings of 3rd Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, pages 211–225, 2005.

[2] Allan Cheng. Petri nets, traces, and local model checking. In *Proceedings of AMAST*, pages 322–337, 1995.

[3] Hasam Davulcu, Michael Kifer, C. R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proceedings of ACM SIGACT-SIGMOD-SIGART*, pages 1–3. ACM Press, 1998.

[4] Thomas Hildebrandt and Raghava Rao Mukkamala. Distributed dynamic condition response structures. In *Pre-proceedings of International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 10)*, March 2010.

[5] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[6] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *Proceedings ProHealth 08 workshop*, 2008.

[7] Raghava Rao Mukkamala, Thomas Hildebrandt, and Janus Boris Tøth. The resultmaker online consultant: From declarative workflow management in practice to LTL. In *Proceeding of DDBP*, 2008.

[8] Munindar P. Singh, Greg Meredith, Christine Tomlinson, and Paul C. Attie. An event algebra for specifying and scheduling workflows. In *Proceedings of DASFAA*, pages 53–60. World Scientific Press, 1995.

[9] 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.

[10] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.