Distributed Dynamic Condition Response Structures

Thomas 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

Abstract

We present *distributed dynamic condition response structures* as a declarative process model inspired by the workflow language employed by our industrial partner and conservatively generalizing labelled event structures. The model adds to event structures the possibility to 1) finitely specify repeated, possibly infinite behavior, 2) finitely specify fine-grained acceptance conditions for (possibly infinite) runs based on the notion of responses and 3) distribute events via roles. We give a graphical notation inspired by related work by van der Aalst et al and formalize the execution semantics as a labelled transition system. Exploration of the relationship between dynamic condition response structures and traditional models for concurrency, application to more complex scenarios, and further extensions of the model is left to future work.

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. Several authors have noted that it could be an advantage to also use a declarative approach to specify workflow and business processes [4, 8, 9, 5, 1]. An important motivation for considering a declarative approach is to achieve more flexible process descriptions [10]. 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. Secondly, 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 extra constraint to be added.

As a simple example, consider a hospital workflow with a single rule stating that the doctor must sign after having added a prescription of medicine 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. For the simple imperative solution, one may be led to just adding 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. One may argue that the rules are too lax in this setting, i.e. that one would need stricter rules to govern the medication. However, besides the fact that this example is indeed extracted from a real-life study of paper-based oncology workflow at danish hospitals [6, 7], the main point is that this is an example of how workflows in general often are intended to be lax and flexible, not this workflow in particular.

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 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 this paper, we propose a new such declarative process model language called *dynamic condition response structures*. The model is inspired by the declarative *process matrix* model language [6, 7] used by our industrial partner and (labelled) prime event structures [12]. Indeed, it is formally a conservative generalization and strict extension of both event structures and the core primitives of the process matrix model language.

An (labelled, prime) event structure in some sense can be regarded as a minimal, declarative model for concurrent processes. It consists of a set of events, a causality (partial order) relation between events stating which events are caused by the previous events (or dually, which events must have preceded the execution of an event), a conflict relation stating which events can not happen in the same execution and finally a labeling function describing the observable action name of each event.

To be used as an execution language for workflow or concurrent (multi-processor) systems several aspects are missing however. In this paper we consider three of these aspects: Firstly, we need some compact, still declarative, way to model *repeated*, possibly infinite behavior. In an event structure each event can only be executed once. Secondly, it must be possible to specify that only *some* of the partial (or infinite) computations are *acceptable*. Event structures have no notion of acceptance condition. Finally, we need to be able to describe a *distribution* of events on agents/persons/processors.

To address these aspects, we propose a number ways to generalize event structures. 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. We can then define runs to be acceptable if no response event from some point in the execution is executable continuously without ever being executed. This relates to the elegant definition of fair runs in true concurrency models investigated in [2]. Finally, we define *distribution* by adding a set of roles assigned to persons/processors and actions.

Being based on essentially only four relations between events, the model can be simply visualized as a directed graph with events (labelled by activities and roles) as nodes and four different kinds of arrows. We found that our condition and response relations were two of the core LTL templates used in [10] and thus decided to base our graphical notation on the one suggested in [10].

We also provide a relatively simple mapping to the state based model of labelled transition systems, which formalizes the semantics. We show how run-time scheduling for workflows with finite runs can easily be supported by identifying accepting states in the labelled transition system This gives a finite state automaton that reflects the run-time scheduling of the process matrix model used by our industrial partner. We leave the treatment of *infinite* runs for future work.

The main advantage of the dynamic condition response structures compared to the related work based on Event logics, Concurrent transactional logic and temporal logics such as LTL explored in [11, 10, 4, 3] is that the latter logics are more general and thus, we claim, more complex to visualize and understand by people not trained in logic.

2 Distributed Dynamic Condition Response Structures

Let us first recall the definition of a prime event structure and configurations of such [12].

Definition 1. *A labeled* prime event structure *(*ES*) over an alphabet Act is a 4-tuple* $(E, \leq, \#, l)$ *where*

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

(ii) $\leq \subseteq E \times E$ is the causality relation between events which is partial order

(iii) $\# \subseteq E \times E$ is a binary conflict relation between events which is irreflexive and symmetric

(iv) $l : E \rightarrow Act$ is the labeling function mapping events to actions

Action names $a \in Act$ represent the actions the system might perform, an event $e \in E$ labelled with a represents occurrence of action a during the possible 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 causality and conflict relations satisfy the conditions that $e\#e' \leq e'' \implies e\#e''$ and $\{e' \mid e' \leq e\}$ is finite for any $e \in E$. A configuration c is a set of events such that,

(i) conflict-free: $\forall e, e' \in c. \neg e\#e'$

(ii) downwards-closed: $\forall e \in c, e' \in E. e' \leq e \implies e' \in c$

We define a run of a labelled event structure to be a sequence of labelled events $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ such that $\{e \mid e \leq e_0\} = \emptyset$ and for all $i \geq 0. \cup_{0 \leq j \leq i} \{e_j\}$ is a configuration.

As an intermediate step towards dynamic condition response structures we generalize prime event structures to (prime) condition response event structures by replacing the causality relation with two relations: the condition and the response relation, as described in the introduction.

Definition 2. A labeled *condition response event structure* (CRES) over an alphabet Act is a tuple $(E, \leq_C, \leq_R, \#, l)$ where

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

(ii) $\leq_C \subseteq E \times E$ is the *condition* relation between events which is partial order

(iii) $\leq_R \subseteq E \times E$ is the *response* relation between events, satisfying that $\leq = \leq_C \cup \leq_R$ is a partial order

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

(v) $l : E \rightarrow Act$ is the labeling function mapping events to actions

The *condition* relation imposes a precedence relation between events. For example, if two events are related by the *condition* relation $e \leq_C e'$, then event e must have happened before event e' can happen. As for the causality relation in prime event structures we require that $e\#e' \leq e'' \implies e\#e''$ and $\{e' \mid e' \leq e\}$ is finite for any $e \in E$. We define configurations and runs as for prime event structures, except that a configuration of a CRES is only required to be downwards closed with respect to the *condition* relation. That is, a configuration c of a CRES is a set of events such that,

(i) conflict-free: $\forall e, e' \in c. \neg e\#e'$

(ii) downwards-closed: $\forall e \in c, e' \in E. e' \leq_C e \implies e' \in c$

The *response* relation is in some sense dual to the condition relation and allows for defining an acceptance condition for runs: We define a run $(e_0, l(e_0)), (e_1, l(e_1)), \ldots$ to be accepting if $\forall i \geq 0.e_i \leq_R e \implies \exists j \geq 0.(e\#e_j \vee (i < j \wedge e = e_j)$. In words, any pending response event must eventually happen or be in conflict.

If one as it is usually the case consider any run of a prime event structure to be accepting, 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* $(\mathsf{E}, \leq, \#, l, \mathsf{Act})$ *has the same runs as the accepting runs of the* CRES *structure (*$\mathsf{E}, \mathsf{Act}, \leq_C, \leq_R, \#, l, \mathsf{Act}$*) where* $\leq_C = \leq$, $\leq_R = \emptyset$

We now go on to generalize the model to allow events to be executed several times. This also leads to a relaxation of the constraints on the condition and response relations and changing the conflict relation to a dynamic exclusion and inclusion of events.

Definition 3. 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.

The condition and response relations in DCR are the same as corresponding relations from CRES, except that they are not constrained in any way. In DCR, we have used a slightly different symbols for condition and response relations in order to be consistent with the graphical notation of DCR model. The *dynamic inclusion/exclusion* relation allows events to be included and excluded dynamically in the process. We will use the notation $e \rightarrow + \, e'$ for $\pm(e, e') = +$ and similarly write $e \rightarrow \% \, e'$ for $\pm(e, e') = \%$. The relation $e \rightarrow + \, e'$ expresses that, whenever event e happens, it will include e' in the process. On the other hand, $e \rightarrow \% \, e'$ expresses that when e happens it will exclude e' from the process.

We make the execution semantics precise below by giving a mapping to a labelled transition system with an acceptance condition on runs defined as described in the introduction.

A CRES can be represented as a DCR by making every event excluding itself and encoding the conflict relation by making any two conflicting events mutually exclude each other.

For example, consider a CRES with two conflicting events e, e' as shown in figure 1(a). This CRES can be represented as a DCR using the *exclude* relation as shown in the figure 1(b). The mutual *exclude* relation on events e, e' will ensure that, only one of the events can happen and similarly self *exclude* relation on the events will enforce that any event can happen only once.

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

Definition 4. A *distributed* dynamic condition response structure(DDCR) is a tuple

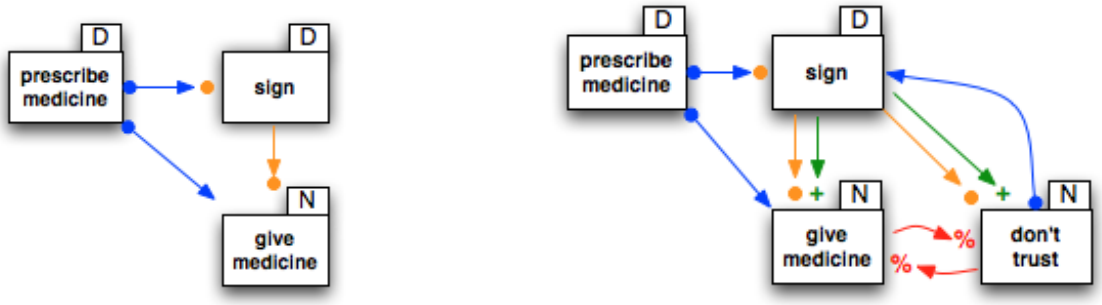
$$(\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$$

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.

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

Figure 1: Conflict relation in graphical notation

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



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

Figure 2: DCRS example in graphical notation

Now, figure 2(a) shows the small example workflow from the introduction graphically. It contains three events uniquely labelled (and thus identified) by the actions: 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 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 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 can be modeled as shown in Figure 2(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.

We now define the semantics of distributed DCRs by giving a map to a labelled transition system and define the set of accepting runs. The states of the transition semantics will be triples (E,I,R) where $E \subseteq \mathsf{E}$ represents the set of happened events, $I \subseteq \mathsf{E}$ represents the set of currently included events, and R represents the set of pending responses.

Definition 5. For a distributed DCR $D = (\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l, \mathsf{R}, \mathsf{P}, \mathsf{as})$ we define the corresponding labelled transition systems $T(D)$ to be the tuple $(S, (\emptyset, \mathsf{E}, \emptyset), \rightarrow \subseteq S \times \mathsf{Act} \times S)$ where $S = \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E}) \times \mathscr{P}(\mathsf{E})$ is the set of states, $(\emptyset, \mathsf{E}, \emptyset) \in S$ is the initial state, $\rightarrow \subseteq S \times (\mathsf{P} \times \mathsf{Act} \times \mathsf{R}) \times S$ is the transition relation given by

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

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

We define the runs $(e_0,(p_0,a_0,r_0)),(e_1,(p_1,a_1,r_1)),\ldots$ of the transition system to be the sequences of labels of a sequence of transitions $(E_i,I_i,R_i) \xrightarrow{(e_i,(p_i,a_i,r_i))} (E_{i+1},I_{i+1},R_{i+1})$ from the initial state. We define such a run to be accepting if $\forall i \geq 0. e \in R_{i+1} \implies \exists j. i < j \wedge (e = e_j \vee e \notin I_j)$. In words, a run is accepting if no pending response event from one point in the run is continuously included without happening.

The first item in the above definition expresses that, only events e that are currently included, can be executed, and to give the label (p,a,r) the label of the event must be a, p must be assigned to the role r, which must be assigned to a. The second item requires that all condition events to e which are currently included should have been executed previously. The third and fourth items are the updates to the sets of included events and pending responses respectively.

If one only want to consider finite runs, which is sometimes the case in the workflow community, 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 $R \cap I = \emptyset$ 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 criteria can be captured by a mapping to a Büchi-automaton. The construction is not straightforward and we leave it for future work to study it in detail.

(We define the transition system, runs and acceptance condition for a non-distributed DCR as for a distributed DCR except there are no principals and roles.)

We can then state the result that the representation of CRES as DCR exemplified in figure 1(b) provides an embedding preserving accepting runs.

Proposition 2. *The condition response event structure* $(\mathsf{E}, \leq_C, \leq_R, \#, l, \mathsf{Act})$ *has the same accepting runs as the accepting runs of the* DCR *structure (*$\mathsf{E}, \mathsf{Act}, \rightarrow\bullet, \bullet\rightarrow, \pm, l$*) where* $\rightarrow\bullet = \leq_C$, $\bullet\rightarrow = \leq_R$, $\forall e,e' \in \mathsf{E}. \pm(e,e') = \%$ *if* $e = e'$ *or* $e \# e'$ *and otherwise* $\pm(e,e') = *$.

3 Conclusion and Future Work

We presented a declarative process model derived as a sequence of relatively simple generalizations of labelled event structures inspired by the workflow language employed by our industrial partner. The first generalization is to split the causality relation of event structures into two dual relations, a *condition* relation $\rightarrow\bullet$ such that $\{e' \mid e' \rightarrow\bullet\ e\}$ is the set of events required to have happened before the event e can happen and a *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. The final extension allows to finitely specify repeated, possibly infinite behavior and acceptance conditions for runs by allowing *multiple execution*, and *dynamic inclusion* and *exclusion* of events and allows for distribution of events via roles. We presented a graphical notation inspired by related work by van der Aalst et al, and gave a mapping to labelled transition systems with an acceptance condition on runs based on the response relation. We remarked that if one only considers finite runs, the acceptance condition can be captured by defining a set of accepting states in the labelled transition system and defining a run to be accepting if it ends in an accepting state. Moreover, we remarked that for infinite runs the accepting condition can be captured by a mapping to a Büchi-automaton, but leave the detailed study of this construction to future work. Also, future work will consider a more detailed comparison between dynamic condition response structures and existing models for concurrency, including the relation to the work in [2]. We also plan to study more complex scenarios and workflow patterns, other acceptance conditions, distributed scheduling, and extensions of the model, notably with time, nested sub structures, soft constraints, and compensation/exceptions.

Acknowlegments

This research is supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project. Danish Research Agency, Grant # 2106-07-0019 (www.TrustCare.eu).

References

[1] Christoph Bussler and Stefan Jablonski. 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*, pages 53–59, Feb 1994.

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

[3] Nihan Kesim Cicekli and Ilyas Cicekli. Formalizing the specification and execution of workflows using the event calculus. *Information Sciences*, 176(15):2227 – 2267, 2006.

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

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

[6] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *Proceedings of 2nd International Workshop on Process-oriented information systems in health- care (ProHealth 08)*, pages 336–347, Milan, Italy, September 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 1st International Workshop on Dynamic and Declarative Business Processes, 2008*, pages 36–43, 2008.

[8] Pinar Senkul, Michael Kifer, and Ismail H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *In VLDB*, pages 694–705, 2002.

[9] Munindar P. Singh, Greg Meredith, Christine Tomlinson, and Paul C. Attie. An event algebra for specifying and scheduling workflows. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 53–60. World Scientific Press, 1995.

[10] 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.

[11] Wil M.P van der Aalst and Maja Pesic. A declarative approach for flexible business processes management. In *Proceedings of Workshop on Dynamic Process Management (DPM 2006)*, volume 4103 of *LNCS*, pages 169–180. Springer Verlag, 2006.

[12] 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.