Modular Context-Sensitive and Aspect-Oriented Processes with Dynamic Condition Response Graphs

Thomas Hildebrandt\textsuperscript{1} Raghava Rao Mukkamala\textsuperscript{1} Tijs Slaats\textsuperscript{1,2} Francesco Zanitti\textsuperscript{1}

\textsuperscript{1} IT University of Copenhagen, Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
\textsuperscript{2} Exformatics A/S, Lautrupsgade 13, 2100 Copenhagen, Denmark
\{rao, hilde, tslaats, zanitti\}@itu.dk

Abstract
We propose the recently introduced declarative and event-based Dynamic Condition Response (DCR) Graphs process model as a formal basis for modular implementation of context-sensitive and aspect-oriented processes. The proposal is supported by a new join operator allowing modular composition and refinement of DCR Graphs. We give small illustrative examples of DCR Graphs defining context-sensitive processes where context-events dynamically enable and disable the need for authentication and the join operator is used to add authentication to a process. Finally, we discuss the use of formal verification to ensure that processes satisfy safety and liveness properties, and define two liveness properties (deadlock freedom and liveness) that can be verified directly on the state graph for DCR Graphs.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory - Semantics, Syntax
General Terms Design, Languages, Reliability, Verification

1. Introduction
The terms context-sensitivity, context-awareness and context-dependency are generally used to describe systems that adapt their behavior according to changes in their context. Changes in the context are naturally described as events, either provided by sensors, user inputs or messages from other programs. This suggests the application of event-driven programming to implement context-aware systems. Event-driven systems are normally based on the publish-subscribe pattern. That is, at any time the system subscribes to specific (patterns of) events. Whenever a pattern has been detected, the system reacts by performing a block of code, possibly publishing new events.

In the present paper we propose to use the recently introduced event-based declarative process model, Dynamic Condition Response Graphs (DCR Graphs) \cite{3, 6, 7, 13}, as a formal basis for modular and aspect-oriented construction of context-sensitive systems. The modular and aspect-oriented construction is facilitated by a new, general join operator. We give small examples that illustrate how the join operator can be used to merge processes, and discuss how merging is similar to adding advices and weaving of aspects in AOP. The examples also show how the join operator can be used to refine and more generally adapt behavior of a process, i.e. by replacing an event or a complete sub process with a new process.

It should be stressed that the proposal in the present paper is very initial work. The DCR Graph model was developed with another application area in mind as a foundation for flexible workflow languages as part of the second author’s PhD project \cite{12}. It has then subsequently been continued in the (ongoing) PhD projects of the third and fourth authors focusing on respectively developing a formal foundation for the implementation of flexible, cross-organizational workflow systems \cite{5} and developing a process-oriented event-based programming language (PEPL) \cite{4} for context-sensitive services based on DCR Graphs.

In Fig. 1 below we give a concrete example of a context-sensitive authorization process that we will use as running example. The example is inspired by a concrete case study of an oncology workflow process at a danish hospital \cite{11}.

The graph consists of five events (shown as boxes). Each event corresponds to the (possibly repeated) execution of an activity, which is indicated by a label assigned to the event and shown inside the box. The activities in the example are thus action, authorize, reauthorize, normal and emergency.

One way to think of a DCR Graph is as an event-driven reactive program, where the code-blocks usually executed when an event pattern is detected have been replaced by specifications of response events, specifying activities that must eventually be executed (by the process or the environment) whenever possible. That is, the occurrence of an event schedules other events that must be executed in the future in order for the process to progress. The response events are specified in the graph by the response relation, indicated

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig1}
\caption{Context-sensitive authorization process as DCR Graph.}
\end{figure}
graphically by the arrow with a bullet at the source and coloured blue to make it more easily visible. In the example we thus have a response relation from the reauthorize to the authorize event.

Moreover, every event can have a list of condition events and milestone events. In order for an event to be enabled, i.e., its activity to be executable, its condition events must have been executed at least once in the past and its milestone events must not currently be scheduled for execution. In the example, the authorize event is a condition for the reauthorize event and a milestone for the action event. The condition relation means in this example that reauthorize can not be executed if authorize has not been executed at least once in the past. The milestone relation means that the action event is blocked if an authorize event is scheduled for execution.

In the example, the event emergency excludes the authorize and reauthorize events and the event normal includes the two events. Note we use the nested DCR Graphs notation introduced in [6], a relation to a box around events is simply a short hand for having the relation to all the sub events. Now, if an emergency event happens (e.g. representing that the condition of the patient becomes critical), then the events authorize and reauthorize are excluded. This means that even if authorize is scheduled for execution and a milestone for action, then it is not disabling the action event. However, if the normal event happens, then authorize and reauthorize are again included.

The five relations (condition, response, milestone, include and exclude) completely describe the dynamic behavior of the process. Moreover, the state of a running process can be described by a triple of finite sets of events, (Ex, Re, In) similar to markings of Petri Nets [15, 16]. The set Ex records the previously executed events, the set Re describes the events scheduled for execution, and the set In denotes the currently included events. The set Re thus contains events schedules as responses to events that have been executed, but we also allow to define that some events are scheduled in the initial marking of the graph.

We graphically visualize a marking (Ex, Re, In) by adding a (green) checkmark to every box for events in the Ex set, a (red) exclamation mark to every box for events in the Re set, and making the border dashed of boxes for events which are not in the In set. A run of a DCR Graph is then a (possibly infinite) sequence of markings, where the \((n + 1)\)th marking is obtained by executing one of the enabled events in the \(n\)th marking (adding it to the set Ex) and updating the Re and In sets according to the relations.

An example run of the process in Fig. 1 is shown in Fig. 2. In the initial marking (at the top), nothing has been executed, the events action and authorize are scheduled, and every event is included. This state is described by the marking \((\emptyset, \{\text{action, authorize}\}, E)\), where E is the set of all five events. In this state, only the events authorize, normal and emergency are enabled. The event action is blocked because it has authorize as milestone and authorize is scheduled for execution. The event reauthorize is blocked because it has authorize as condition, and this event has never been executed.

Now, if the emergency event is executed then the state changes to the marking \((\{\text{emergency}\}, \{\text{action, authorize}\}, E\backslash\{\text{authorize, reauthorize}\})\), shown at the graph in the middle of Fig. 2. That is, emergency is recorded as executed, whereas action and authorize are still scheduled, but authorize and reauthorize are excluded. This implies that authorize is not longer considered as milestone for action, which therefore is enabled. The marking shown at the bottom of Fig. 2 shows the state if the event action is executed.

Note that action can be executed several times. However, if normal is now executed, i.e. if the patient condition is no longer critical, then system moves to the marking shown in Fig. 3, where authorize and reauthorize are included again.

Figure 2. An example run of the authorization process.

Figure 3. If the normal event is executed, then authorize is again required before action can be executed.
authorize is executed or excluded because the event emergency is executed again.

Runs may be finite or infinite. We say that a run is accepting if whenever an event is scheduled, it eventually gets excluded or executed. As shown in [12, 13], DCR Graphs can be mapped to the standard Büchi-automata model, characterizing all runs as well as fair runs as the accepting runs, and subsequently verified for safety and liveness properties using the SPIN model checker [9]. However, it can also be represented directly (and more compactly) as a so-called transition system with responses [2], which is simply a labelled transition system where each state is annotated by a set of labels, referred to as the response actions. The accepting runs of a transition system with responses is then the finite or infinite runs where whenever a label is included in the response set of an intermediate state in the run, it will be excluded from the response set in a subsequent state or executed.

It is worth stressing that there is no explicit sequencing of commands in the DCR Graphs process model. This makes it possible to weave, refine and adapt processes by the general join operator introduced in Sec. 3. If a new event i is joined as a milestone for some existing events in a process, and such that this new event is scheduled whenever the existing events are scheduled as responses, then the new event must be executed before the existing events become enabled. Thus, the new event is similar to an advice in AOP, that must be executed before the targeted set of existing events which correspond to join points.

After briefly recalling the formal definition of DCR Graphs in Sec. 2, we exemplify this aspect-oriented modularity in Sec. 3 where the DCR Graph shown in Fig. 1 is joined with another intermediate state in the run, it will be excluded from the response set in a subsequent state or executed.

We end by briefly summarizing the techniques for formal verification and distribution of DCR Graphs developed so far in the above mentioned research projects and briefly touch on related work.

2. DCR Graphs

In this section we give the formal definitions of DCR Graphs. We employ the following notation.

**Notation:** For a set \( E \) we write \( P(E) \) for the power set of \( E \) (i.e. set of all subsets of \( E \)). For a binary relation \( \rightarrow \subseteq E \times E \) and a subset \( \xi \subseteq E \) we write \( \rightarrow \xi \) and \( \rightarrow\xi \) for the set \( \{ e \in E \mid (\exists e' \in \xi \mid e \rightarrow e') \} \) and the set \( \{ e \in E \mid (\exists e' \in \xi \mid e' \rightarrow e) \} \) respectively, and abuse notation writing \( e \rightarrow \rightarrow \rightarrow e \rightarrow \leftrightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \right.
In particular the join operation allows in special cases for basic union of graphs and refinement of events, that is, substituting an event by an entire new sub-graph. Before giving the formal definition, we will give an example of how to join a process graph with another, where one event in the former graph is refined by two events in the latter process.

The DCR Graph $G_{cr}$ in Fig. 4 below shows a process for a context-sensitive request/reply pattern.

**Figure 4. A context-sensitive request/reply process**

If a request event happens, the reply is scheduled as a response. The reply event is excluded if it is executed, or a cancel event subsequently happens. However, it will be included (and scheduled) again if a new request event happens. As for the example given in the introduction, the cancel event may be triggered by some change of conditions in the context of the process.

We can now use the join operation to define a context-sensitive request/reply process where the reply needs (context-sensitive) authorization as defined in the DCR Graph $G_{auth}$ given in Fig. 1 in the introduction.

The intention is to merge the two process graphs, but such that the reply event in $G_{cr}$ is replaced by the authorize and action events in $G_{auth}$. The result is the graph shown in Fig. 5 and formally defined as $G_{rrauth} = G_{cr} \triangleright\triangleright G_{auth}$ for $\ll = \{\{\text{reply}, \{\text{authorize}, \text{action}\}\}\}$ and $\triangleright\triangleright = \emptyset$.

**Figure 5. Joining context-sensitive request/reply and authorization processes**

Informally, the two event sets have been joined, replacing the reply event in the graph $G_{cr}$ with the two events authorize and action in the $G_{auth}$ graph, while inheriting all relations between the replaced event and other events in $G_{cr}$. Recall, that relations to the box around authorize and action is merely a convinient way to represent relations to both authorize and action.

To ease readability we may adopt a programming language notation for the join operation, writing the above join as follows.

```
Listing 1. Using Join to Authorize a Reply
CSRequestReplyAuth =
  join CSRequestReply and CSAuthorization
where reply < {authorize, action}
```

Below we give the formal definition of the join operator.

**Definition 6.** Assume DCR Graphs $G_i = (E_i, M_i, \rightarrow_i, \leadsto_i, L_i)$ where $M_i = (Ex_i, Re_i, Lm_i)$ for $i = 1, 2$ and join relations $\ll_i : E_1 \rightarrow \mathcal{P}(E_2)$ and $\triangleright_i : E_2 \rightarrow \mathcal{P}(E_1)$.

The join of $G_1$ and $G_2$ relative to $\ll$ and $\triangleright$ is defined as $G_{1} \ll\triangleright G_{2} = (E, M, \rightarrow, \leadsto, \rightarrow\ll, \rightarrow\triangleright, \leadsto\ll, \leadsto\triangleright, L_i)$, where

(i) $\forall e \in \text{dom}(\ll)$. $\ll(e) \cap \text{dom}(\triangleright) = \emptyset$ and $\forall e \in \text{dom}(\triangleright)$. $\ll(e) \cap \text{dom}(\ll) = \emptyset$

(ii) $E = E_1 \cup E_2$. $\forall e_1 \in E_1 \setminus \text{dom}(\ll)$ and $\forall e_2 \in E_2 \setminus \text{dom}(\triangleright)$

(iii) $M = (Ex, Re, Lm)$, where: $Ex = Ex_1 \cap E_1 \cup Ex_2 \cap E_2$. $Lm = Lm_1 \cap E_1 \cup Lm_2 \cap E_2$ and $Re = (Re_1 \cup E_1 \cap Lm_1) \cup (Re_2 \cup E_2 \cap Lm_2)$

(iv) $\rightarrow = \ll^{-1} \cup \rightarrow\ll \cup \rightarrow\triangleright \cup \triangleright\ll \cup \triangleright\triangleright$

(v) $L = L_1 \cup L_2$

(vi) $l(e) = \begin{cases} l_1(e) \cup l_2(e) & \text{if } e \in E_1 \cap E_2 \\ l_1(e) & \text{if } e \in E_1 \setminus E_2 \\ l_2(e) & \text{if } e \in E_2 \setminus E_1 \end{cases}$

The first condition, (i), guarantees that there are no circular refinements. That is, an event in $G_1$ can not be refined by an event in $G_2$ which is also refined by an event in $G_1$ and vice versa. The second condition, (ii), states that the set of events in the joined graph consists of all the events of the two graphs, that have not been refined by events in the other graph. The third condition, (iii), defines the join of the markings of the graphs. It simply inherits the markings from the two graphs. Note that the fact that markings are also joined means that we can also apply the join operator on computing processes. Condition (iv) defines the extension of the relations to the refining events. The relation $\ll\leq$ is the reflexive closure of the left join $\ll$ relation, i.e. $e \leq e'$ if and only if $e \ll e'$. Similarly, $\triangleright\leq$ is the reflexive closure of the right join $\triangleright$ relation, i.e. $e \leq e'$ if and only if $e \triangleright e'$. Consequently, $\ll\leq\ll$ and $\triangleright\leq\triangleright$ are the relational composition of $\ll\leq\ll$ and $\triangleright\leq\triangleright$. The relation $\ll\leq\ll$ is the relational composition of $\ll\leq\ll$ and $\triangleright\leq\triangleright$, i.e. $e \ll e'$ if $e \rightarrow_1 e''$ or $e \rightarrow_2 e''$ or $e'' \ll e'$ or $e'' \triangleright e'$.

Considering our example in Fig. 5, this definition implies that e.g. cancel will exclude both action and authorize. But note that the self-exclude relation on reply, enforcing the "linearity" constraint that only one reply can only be carried out for each request, is not kept. This is because relations between events that are replaced, and thus in particular self-relations are not kept. The rationale for not keeping these relations are that a join should allow for removing constraints on the refined events. To keep the property that the action can only be carried out once for each request, the refining graph $G_{auth}$ should have an exclude relation from action to itself.

Finally, conditions (v) and (vi) define the label set as the union of the two label sets, and the labeling of events by the original label for events that are not shared and the union of the label set
for shared events. Since we have not used the label function in the present paper it can safely be ignored. For the curious reader, the labeling function allows to have distinct events with the same "external" label, which is useful for some practical applications, as well as for proving that every Büchi-automaton can be represented by a DCR Graph. This means in particular that the DCR Graphs model is more expressive that LTL.

As indicated in the beginning of the section, we may derive operations \(G \setminus e, G[e \triangleright G']\) and \(G \cup G'\) for respectively discarding an event \(e\), refining an event \(e\) by \(G'\) and taking the union of two graphs \(G\) and \(G'\) as special cases of the join operator.

**Definition 7.** For a DCR Graph \(G = (E_1, M_1, \rightarrow \ell_1, \cdot \rightarrow \ell, \rightarrow \ell_2, \rightarrow \ell_3, L_1, L, l)\) for \(i \in \{1, 2\}\) and \(e \in E_1\) define

- \((\text{Discard})\) \(G \setminus e = G_1 \triangleright G_2\) where \(\triangleleft (e, \emptyset)\) and \(G_0\) is the empty DCR Graph, i.e. the DCR Graph with no events,
- \((\text{Refine})\) \(G[e \triangleright G_2] = G_1 \triangleright G_2\) where \(\triangleleft (e, E_2)\) and \(\triangleright (\emptyset, \emptyset)\),
- \((\text{Union})\) \(G_1 \cup G_2 = G_1 \triangleright G_2\) where \(\triangleleft (\emptyset, \emptyset)\) and \(\triangleright (\emptyset, \emptyset)\).

As an example of the discard operation, we may remove the ability to cancel requests in the graph \(G_{\text{req}}\) in Fig. 4 by discarding the cancel event, taking the graph \(G_{\text{req}}\)\text{cancel}.

As an example of the refine operation, we may add the "linearity" constraint to action in \(G_{\text{req}}\), i.e. that it can be executed at most once for every request. This is done with a refine operation \(G_{\text{req}}:\text{action} \triangleright G_{\text{inact}}\), where \(G_{\text{inact}}\) is the DCR Graph \(\{(\text{action}), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset, \emptyset), (\text{action}, \emptyset, \emptyset), (\emptyset, \emptyset)\},\) i.e. the graph with an empty marking and a single event action related to itself by the exclude relation, and no other relations.

### 3.1 Aspect Oriented and Modular DCR Graphs

In an aspect oriented language based on DCR Graphs, we propose using the join operator to define an operator for adding "before" advices for a subset of events (the joincut). This could for instance be done by refining every event \(e\) in the joincut by a DCR Graph that adds an advice sub process as a milestone before the event \(e\), ad e.g. the authorize event before the action event in Fig. 1. Then, every time an event \(e\) in the join cut is scheduled for execution, every event in the advice sub process will also be scheduled for execution, and must be executed before executing \(e\) due to the milestone relation.

Dually, an "after" advice may be added to an event \(e\) by joining a graph that has a response, condition and include relation from \(e\) to the advice sub process, like for the request and reply events in the \(G_{\text{req}}\) graph in Fig. 4. This ensures that the advice must be carried out once after the event \(e\). These two operations can then be combined to have both "before" and "after" advices.

Toggling of advices, e.g. as a result of context-events, can be achieved by joining in events that include and exclude the advice sub processes, e.g. like the normal and emergency events in Fig. 1. One can further constrain when an advice can be toggled as in Fig. 4, where the reply action can be cancelled, but only after the request has happened.

Finally, the join operator can also be considered as a general way of allowing modular definition of DCR Graphs. However, it should be stressed, that the join operator provide no guarantees for preserving safety and liveness properties. Indeed, it is easy to join graphs and achieve a circular condition dependency between events that may lead to a deadlock state if one of the events are required as response (and can not be excluded). Similarly, the join operator may introduce the possibility of a life lock, i.e. a DCR Graph with an infinite run that is not accepting and has no way of breaking out of the loop.

It is possible to verify safety and liveness properties of the composed DCR Graphs, e.g. by mapping the DCR Graph to a Büchi-automaton and verify the properties using the SPIN model checker as shown in [12]. This is however time consuming when the size of the processes grow. A more efficient verification currently explored is to carry out the verification on the corresponding transition systems with responses. Finally, an even more efficient guarantee of well-behaved modular composition, which we are currently investigating, is to define a notion of behavioral type for DCR Graphs based on the work on session types, which then guarantee that 1) well-typed graphs are safe and live, and 2) if compatible, well-typed DCR Graphs are joined, then the resulting graph is again well-typed.

### 4. Conclusion

We have proposed the declarative, event-based DCR Graphs model as a foundation for modular construction of context-sensitive, aspect-oriented processes. Concretely, we showed how the dynamic exclusion and inclusion primitives of DCR Graphs allow to turn the relevance of events as conditions/milestones and responses for any other event in the model on and off. It was exemplified by a simple authorization process fragment, where the need for authorization can be toggled by events signaling whether the context situation is an emergency or normal. Another example was a request/reply process fragment, where the reply can be cancelled by a cancel event in the context. Moreover, we presented a new join operator for DCR Graphs, allowing both modular composition and refinement, exemplified by joining the request/reply process and the authorization process, by which the reply was refined into two events, an action event (i.e. representing the reply) and an authorization event, and to the remaining events and relations for authorization were inherited.

A key point is that aspects are not turned on and off when processes are initiated, but asynchronously at any point during the execution of the process. Another key point is that the formal semantics allows us to verify if the defined process, e.g. obtained by joining processes, have safety or liveness problems. As shown in [12, 13], DCR Graphs can be mapped to Büchi-automata. In [12] an implementation of this mapping and its application to verify safety and liveness properties of DCR Graphs using the SPIN model-checker are described. Recently we have developed a model checker that allows to verify properties directly on DCR Graphs. It avoids the translation to Büchi-automata and seems more efficient, however this is still work in progress. In [17] we have developed a technique for distributing DCR Graphs, which is somehow reverse to the composition operator. It allows to divide a DCR Graph in a collection of (not necessarily disjoint) sub graphs, which can then be executed at different locations. Finally, the fourth author has developed a prototype implementation of PEPL [4] where events are extended with data, and the third author has implemented a workflow engine at Exformatics A/S based on DCR Graphs with data.

**Related work:** It goes beyond the scope of the present paper to give a comprehensive overview of related work. Context-oriented programming (e.g. [8, 18]) introduces the notion of layers in normal object-oriented, block structured programming languages such as Java, allowing features (aspects) to be activated or deactivated depending on the current context. The selection of active layers are typically done using the with primitive when methods are invoked. In [10], the event-driven and context-oriented approaches are combined, allowing events triggered in other threads to activate and de-activate layers somewhat similar to the include and exclude primitives of DCR Graphs.

The use of declarative primitives for the description of processes, and workflow processes in particular, is also treated in the work on Declare [19, 20]. Declare is similar to DCR Graphs in
that processes are specified by temporal relations between events, indeed, the condition and responses relations are specified using the same graphical notation. Another related approach would be to specify processes using a temporal logic like LTL [14, 17] or CTL [1]. In [14] is shown that primitives similar to those used in DCR Graphs can indeed be represented in LTL. However, neither Declare, LTL nor CTL have explicit operators for the inclusion and exclusion of events as in DCR Graphs. Consequently, toggling of aspects can not simply be added by joining additional constraints. It is instead necessary to rewrite the logical formula (or Declare process) in a non trivial way. As an example, consider the request-reply-cancel pattern in Fig. 4. The request-reply-reply pattern alone would typically be expressed in LTL by a formula like $G(request \implies F\text{reply}) \land (F\text{reply} \implies \neg reply\{request\}) \land G(reply \implies N(F\text{reply} \implies \neg reply\{request\}))$, where $G$ reads Generally, $F$ reads Future, $U$ reads Until and $N$ is the Next operator. That is, 1) it is generally the case that if a request happens, then a reply happens in the future, 2) if a reply happens, then that reply can not happen before at least one request has happened, and 3) if a reply happens, then if another reply happens some time in the future, it will not happen before a new request has happened. Now, to add the cancel option, it is necessary to rewrite the formula, it is not possible simply to add new constraints e.g. as a conjunction. This is because 1) a reply is not required if a cancel event happens, i.e. one must change the first conjunct to $G(request \implies F(reply \lor cancel))$, 2) after a cancel event, a reply should not occur until we have seen a new request, i.e. the third conjunct becomes $G(reply \lor cancel \implies N(F\text{reply} \implies \neg reply\{request\}))$ and 3) a cancel event should not occur before the first request has happened, i.e. we should add an additional conjunct $(Fcancel \implies \neg cancel\{request\})$. In general, if the LTL formula does not have this specific form as given by DCR Graphs, we claim that it would be non-trivial to define how formulas should be changed to toggle aspects on and off, and at least not as simple as the join operator of DCR Graphs proposed in the present paper.

Acknowledgments

This research is supported by IT University of Copenhagen, Exformatics A/S, an Industrial PhD grant and the Danish Council for Strategic Research through the Jingling Genies project, grant no.: 2106-080046. We would like to thank the anonymous reviewers for careful reviews and suggestions.

References


