Acta Informatica manuscript No.
(will be inserted by the editor)

Replication, Refinement & Reachability: Complexity in Dynamic Condition-Response Graphs

Søren Debois · Thomas T. Hildebrandt · Tijs Slaats

Received: date / Accepted: date

Abstract We explore the complexity of reachability and run-time refinement under safety and liveness constraints in event-based process models. Our study is framed in the DCR* process language, which supports modular specification through a compositional operational semantics. DCR* encompasses the "Dynamic Condition Response (DCR) graphs" declarative process model for analysis, execution and safe run-time refinement of process-aware information systems; including replication of sub-processes. We prove that event-reachability and refinement are NP-HARD for DCR* processes without replication, and that these finite state processes recognise exactly the languages that are the union of a regular and an ω-regular language. Moreover, we prove that event-reachability and refinement are undecidable in general for DCR* processes with replication and local events, and we provide a tractable approximation

Supported by the Velux foundation (grant 33295), the Danish Council for Independent Research (grant DFF-6111-00337) and Innovation Fund Denmark.

Søren Debois
Department of Computer Science
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S
E-mail: debois@itu.dk

Thomas T. Hildebrandt
Department of Computer Science
IT University of Copenhagen
Rued Langgaards Vej 7
2300 Copenhagen S
E-mail: hilde@itu.dk

Tijs Slaats
Department of Computer Science
University of Copenhagen
Njalsgade 128, Building 24, 5th floor
2300 Copenhagen S
E-mail: slaats@di.ku.dk

for refinement. A prototype implementation of the DCR* language is available at `http://dcr.tools/acta16`.

Keywords DCR graphs · replication · refinement · complexity

1 Introduction

Software systems today control increasingly complex processes operating in unpredictable contexts. At the same time, it is becoming more and more critical that the software behaves correctly, e.g. that it is compliant with safety, security and legal regulations. The combination of complexity, unpredictability and need for compliance has lead to a general understanding that a foundation for the implementation of modular, run-time adaptable and formally verifiable software systems is needed [44, 43, 42].

This is not least the case in the fields of *Process-Aware Information Systems* (PAIS) [43] and *Business Process Management* (BPM) [3], which constitute the context of the present work. These fields deal with systems driven by explicit process designs for the enactment and management of business processes and human workflows, and the study of formalisms for describing processes has always been central in these fields: As a vehicle for communication, it is vital that a business process model is unambiguous; as a vehicle for understanding, it is vital that it is analysable; and as a foundation for practical systems, it is vital that it can be made executable. Popular models include in particular models which specify explicit sequencing of business activities as flow graphs, such as Petri Nets and Workflow Nets [1] which are the closest formal counterpart to the industrial standard Business Process Model and Notation (BPMN) [38].

However, an approach to process implementation based on flow graphs implicitly assumes the initial design of a *pre-specified* process graph, that implements the believed best practice given the initial required set of business rules and legal constraints. This is problematic in several ways: Firstly, the explicit flow graph often imposes more constraints than necessary. Secondly, procedures, rules and regulations change or the process graph turns out not to be the desired practice anyway. For long running processes, such as control software in hardware systems that can not be stopped or mortgages of credit institutions, the changes need to be reflected in running processes. And while the graph may be initially verified to implement the given business rules and legal constraints, it does typically not represent the rules *explicitly*. It is thus very difficult, if at all possible, to identify the required changes to the flow graph.

Declarative process languages [2, 21] address this deficiency by leaving the exact sequencing of activities undefined, yet specifying the constraints processes must respect. This gives a workflow system the maximum flexibility available under the rules and regulations of the process. In practice, the caseworker or process engine is empowered to take what is considered the appropriate steps (e.g. considering resource usage) for the process and situation at

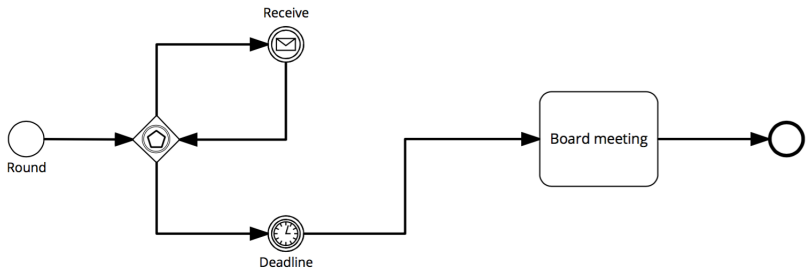

Fig. 1 Grant Application BPMN Process

hand, subject only to the constraints expressed in the process model. If the constraint language is well designed, the constraints can directly represent the business and legal regulations, making it easy to add or update constraints if the regulations change [37].

As a running example, we consider a grant application process of a funding agency that our industry partner, Exformatics, has recently implemented in a commercial solution [12]. The high-level requirements are[1]:

1. applications can be received after a round is opened and until the deadline,
2. if a round is opened, the board must eventually meet, and
3. a board meeting can only happen when a round is open (i.e. before the deadline), if at least one application was received.

A BPMN process implementing the requirements may be defined as shown in Figure 1.

The process is initiated by an event Round indicating the start of a round, followed by a loop for receiving applications until the deadline, after which a board meeting is held. However, the requirements are not explicitly represented in the process diagram, and moreover, the process introduces unnecessary, or at least unspecified constraints. For instance, no board meeting can be held before the deadline, even if applications are received, and it is not possible to reopen the round, e.g. if insufficiently many good applications were received. Of course, these possibilities may be modelled, but with the cost of making the process graph more complex. And even then, the requirements would only be implicitly represented by the diagram.

Our industry partner Exformatics employs a declarative, graphical process notation, *Dynamic Condition Response (DCR) graphs*, introduced in [21, 36] and developed further in [46, 22, 8, 37, 24, 13]. As exemplified in Figure 2 below (produced with the tool at `http://dcr.tools/`), the DCR graphs notation allows to specify the process by the four events (Receive, Deadline, Round, Board Meeting) and four relations between the events.

[1] We deviate from the implemented solution when it makes our examples clearer. In particular, many reasonable constraints—such as "the deadline can only occur once the round has begin"—have been left out in order to keep the example small.

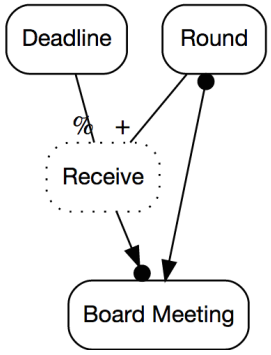

Fig. 2 Grant Application DCR Graph Process

The Receive event is dashed, representing that it is initially excluded from the process. The line from Round to Receive with a + sign at the end is an *include* relation, meaning that Receive is dynamically included if the event Round (the start of a round) happens. Dually, the line from Deadline to Receive with a % sign is an *exclude* relation, meaning that Receive is dynamically excluded if the event Deadline happens. Together, these two relations represent the first requirement. The arrow from Round to Board Meeting with a bullet at the start represents that Board Meeting is a *response* to (i.e. must happen eventually after) the Round event, as stated in the second requirement. Dually, the arrow from Receive to Board Meeting with a bullet at the end represents that Receive is a *condition* for Board Meeting, meaning that if Receive is included, i.e. a round is open, it must have happened before Board Meeting can happen, as stated in the third requirement.

The operational semantics of a DCR graph is defined in terms of a *marking* assigning a triple of booleans (h, i, r) to each event, indicating whether or not the event previously *(h)appened*, is currently *(i)ncluded*, and/or is *(r)estless*. (A restless event is one which must subsequently happen (or be excluded) for the workflow to be complete.)

As illustrated by the Receive event above, an event may be excluded in the initial marking and be dynamically included and excluded if it is related to other events by include and exclude relations. Similarly, an event may be restless in the initial marking or become restless because it is a response to an event that happened as for the Board Meeting event above. Finite or infinite executions are then defined to be accepting only if every included restless event is executed or excluded at a later point in the execution.

As described in [21, 36], (labelled) DCR graphs is a conservative generalisation of (labelled) prime event structures, allowing *finite representations of infinite behaviour* and to express *liveness* properties. The former is achieved by allowing an event to happen more than once and change dynamically between being in conflict (excluded) and being consistent (included) with the

current state. The latter is achieved via the notion of restless events and the acceptance criteria described above.

In [13], DCR graphs are extended to so-called *hierarchical DCR graphs*, supporting dynamic creation of sub-processes with (local) events and independent life cycles. This extension was motivated in practice by the funding agency process above, e.g. to allow each received application to have its own approval events and decision life cycle, and thereby recording the decision made for each application. However, although the graphical notation of DCR graphs has been adopted by industry [33], it does not scale well to larger and more complex processes due to lack of compositionality. Also, the expressive power of hierarchical DCR graphs was left open by [13].

The present paper introduces the *DCR* process language*, a core constraint-based process language for DCR graphs. The process language supports modular specification of process-aware information systems and *dynamic creation of sub-processes with independent life-cycles*, based on the primitives introduced in (hierarchical) DCR graphs, but equipped with a *compositional* operational semantics. The compositional semantics provides means for modular definition of and reasoning for DCR* processes.

We provide two main technical contributions.

1. We analyse expressiveness and complexity for the DCR and DCR* calculi, and
2. We establish a mechanism for run-time refinement of DCR* processes.

Ad (1). We show that both *event-reachability* and *run-time refinement* is NP-HARD for DCR and undecidable for DCR*. Moreover, we provide a full proof that DCR characterise exactly the languages that are the union of regular- and ω-regular languages.

Ad (2). As an application, we show that the DCR* language supports run-time adaptation by composition. We formalise when an adaptation is a *refinement*—preserve requirements of the adapted process—and provide a polynomial-time computable syntactic approximation to refinement, referred to as *non-invasive* processes. Such approximation is crucial: Our complexity results preclude the existence of feasible exact algorithms.

Overview of the paper: In Section 2 we provide the DCR process language and its compositional semantics. In Section 3 we prove that the DCR process language characterises those languages that are the union of a regular and an ω-regular language. We then extend the language in Section 4 to DCR* supporting dynamic creation of sub-processes with fresh (local) events. In Section 5 we show that the existence of a run executing a given event is NP-HARD for DCR processes and undecidable for DCR* processes. We address run-time adaptation by composition in Section 6, where we reduce the problem of deciding whether an adaptation is a refinement to the problem of whether an event can eventually be executed. In Section 7 we then provide "non-invasiveness" as a tractable syntactic approximation of refinement. Finally, in Section 8 we conclude and outline related, current and future directions of work exploiting the results in the present paper.

$T, U ::=$	$f \rightarrow\bullet\ e$	condition
	$\mid\ f \leftarrow\bullet\ e$	response
	$\mid\ f +\!\leftarrow e$	inclusion
	$\mid\ f \,\%\!\leftarrow e$	exclusion
	$\mid\ T \parallel U$	parallel
	$\mid\ 0$	unit

$\phi ::= \mathtt{t} \mid \mathtt{f}$		boolean value
$\Phi ::= (\phi, \phi, \phi)$		event state
$M, N ::= M, e : \Phi$		marking
$P, Q ::= [M]\ T$		process

Fig. 3 DCR Processes Syntax.

An online, research prototype implementation of the process languages presented in the paper, with a mapping to DCR graphs, can be found at `http://dcr.tools/acta16`.

This paper is a revised and extended version of our conference paper [15]. Beyond improvements in presentation, the present paper adds (1) the development of the encoding of Büchi automata into DCR processes; (2) the technical development of the proof that non-invasiveness implies refinement; (3) the analysis of the complexity of event reachability for both DCR and DCR* processes, and (4) an analysis of the complexity of refinement for DCR processes.

2 DCR Processes

Below we introduce the Dynamic Condition Response (DCR) process language. As described in the introduction, it is based on the notions of dynamic inclusion and exclusion of labelled events, related by conditions and response relations introduced in DCR Graphs [36, 21].

We assume fixed universes of *events* $\mathcal{E}$ and *labels* $\mathcal{L}$; and we assume a fixed assignment $\ell(e) \in \mathcal{L}$ to each event $e \in \mathcal{E}$. A DCR process $[M]\ T$ comprises a *marking* M and a *term* T. The syntax of both are given in Figure 3.

A term is a parallel composition of *relations* between *events*. We recall from the introduction how the relations regulate what behaviour the term may exhibit:

1. A *condition* $f \rightarrow\bullet\ e$ imposes the *constraint* that for event e to happen, the event f must either previously have happened or currently be excluded.
2. A *response* $f \leftarrow\bullet\ e$ imposes the *effect* that when e happens, f becomes restless and must eventually happen or be excluded.
3. An *exclusion* $f \,\%\!\leftarrow e$ imposes the *effect* that when e happens, it *excludes* f. An excluded event cannot happen; it is ignored as a condition; and it need not happen if restless, unless it is re-included by the final relation:
4. An *inclusion* $f +\!\leftarrow e$ imposes the *effect* that when the event e happens, it re-includes the event f.

Note that it is possible to specify a relation twice, e.g., $f \,\%\!\leftarrow e \parallel f \,\%\!\leftarrow e$; however, when we give semantics below, it will be clear that this duplication has no additional effect.

All four relations refer to a marking M, a finite map from events to triples of booleans (h, i, r), referred to as the *event state* and indicating whether or not

$$[M, f : (h, i, \_), e : (\_, \mathsf{t}, \_)]\ f \rightarrow\!\bullet\ e \vdash e : \emptyset, \emptyset, \emptyset \qquad \text{(when } i \Rightarrow h\text{)}$$
$$[M, e : (\_, \mathsf{t}, \_)]\ f \leftarrow\!\bullet\ e \vdash e : \emptyset, \emptyset, \{f\}$$
$$[M, e : (\_, \mathsf{t}, \_)]\ f \mathbin{+}\!\leftarrow\ e \vdash e : \emptyset, \{f\}, \emptyset$$
$$[M, e : (\_, \mathsf{t}, \_)]\ f \mathbin{\%}\!\leftarrow\ e \vdash e : \{f\}, \emptyset, \emptyset$$
$$[M, e : (\_, \mathsf{t}, \_)]\ 0 \vdash e : \emptyset, \emptyset, \emptyset$$
$$[M, e : (\_, \mathsf{t}, \_)]\ f' \mathcal{R}\ f \vdash e : \emptyset, \emptyset, \emptyset \qquad \text{(when } e \neq f\text{)}$$

Fig. 4 Enabling & effects. We write "_" for "don't care", i.e., either true t or false f, and write $\mathcal{R}$ for any of the relations $\rightarrow\!\bullet$,$\leftarrow\!\bullet$, $+\!\leftarrow$,$\%\!\leftarrow$.

the event previously (h)appened, is currently (i)ncluded, and/or is (r)estless. A restless event represents an unfulfilled obligation: once it happens, it ceases to be restless. As commonly done for environments, we write markings as finite lists of pairs of events and event states, e.g. $e_1 : \Phi_1, \ldots, e_k : \Phi_k$ but treat them as maps, writing $\mathsf{dom}(M)$ and $M(e)$, and understand $M, e : \Phi$ to be undefined when $e \in \mathsf{dom}(M)$. The *free events* $\mathsf{fe}(T)$ of a term T is (for now) simply the set of events appearing in it. (This changes when we introduce local events in Sec. 4 below.) We require of a process $P = [M]\ T$ that $\mathsf{fe}(T) \subseteq \mathsf{dom}(M)$, and so define $\mathsf{fe}(P) = \mathsf{dom}(M)$. The *alphabet* $\mathsf{alph}(P)$ is the set of labels of its free events.

Example 1 (Grant process term) The example of fig. 2 can be mapped to the following term:

$$T_0 = \mathsf{recv} \mathbin{\%}\!\leftarrow \mathsf{deadline} \parallel \mathsf{recv} \mathbin{+}\!\leftarrow \mathsf{round} \parallel \mathsf{bm} \leftarrow\!\bullet\ \mathsf{round} \parallel \mathsf{recv} \rightarrow\!\bullet\ \mathsf{bm}$$

Initially, no event has happened, no event is restless, and every event but recv is included, giving us the marking:

$$M_0 = \mathsf{round} : (\mathsf{f}, \mathsf{t}, \mathsf{f}), \mathsf{deadline} : (\mathsf{f}, \mathsf{t}, \mathsf{f}), \mathsf{recv} : (\mathsf{f}, \mathsf{f}, \mathsf{f}), \mathsf{bm} : (\mathsf{f}, \mathsf{t}, \mathsf{f})\ .$$

We give semantics to DCR processes incrementally. First, the notion of an event being *enabled* and what *effects* it has. The judgement $[M]\ T \vdash e : E, I, R$, defined in Figure 4, should be read: "in the marking M, the term T allows the event e to happen, with the effects of excluding events E, including events I, and making events R restless." Technically, we will treat the latter "E, I, R" part as a triple when doing so is convenient. Note that events in DCR graphs may happen more than once: the h component for the firing event is generally "don't care".

The first rule says that if f is a condition for e, then e can happen only if (1) it is itself included, and (2) if f is included, then f previously happened. The second rule says that if f is a response to e and e is included, then e can happen with the effect of making f restless. The third (fourth) rule says that if f is included (excluded) by e and e is included, then e can happen with the effect of including (excluding) f. The fifth rule says that the completely unconstrained process 0, an event e can happen if it is currently included. The

$$\frac{[M]\ T \vdash e : \delta}{[M]\ T \xrightarrow{e:\delta} T}\ [\textsc{intro}] \qquad \frac{[M]\ T_1 \xrightarrow{e:\delta_1} T_1' \qquad [M]\ T_2 \xrightarrow{e:\delta_2} T_2'}{[M]\ T_1 \parallel T_2 \xrightarrow{e:\delta_1 \oplus \delta_2} T_1' \parallel T_2'}\ [\textsc{par}]$$

$$\frac{[M]\ T \xrightarrow{e:\delta} T'}{[M]\ T \xrightarrow{e} [e:\delta \cdot M]\ T'}\ [\textsc{effect}]$$

Fig. 5 Basic transition semantics.

last rule says that a relation allows any included event e to happen without effects when e is not the relation's right-hand–side event.

Given enabling and effects of events, we define the *action* of respectively an *event* e and an *effect* $\delta = (E, I, R)$ on a marking M pointwise by the action on individual event states $f : (h, i, r)$ as follows.

$$\text{(Event action)} \quad e \cdot \big(f : (h,i,r)\big) \stackrel{\text{def}}{=} f : \big(\underbrace{h \vee (f{=}e)}_{\text{happened?}},\ i,\ \underbrace{r \wedge (f{\neq}e)}_{\text{restless?}}\big)$$

$$\text{(Effect action)} \quad \delta \cdot \big(f : (h,i,r)\big) \stackrel{\text{def}}{=} f : \big(h,\ \underbrace{(i \wedge f \notin E) \vee f \in I}_{\text{included?}},\ \underbrace{r \vee f \in R}_{\text{restless?}}\big)$$

That is, for the event action, if $f = e$, the event is marked "happened" (first component becomes t) and it ceases to be restless (last component becomes f). For the effect action, the event only stays included (second component) if $f \notin E$ (it is not excluded) or $f \in I$ (it is included). This also means that if an event is both excluded and included by the effect, conceptually the exclusion happens first, followed by the inclusion. It is perhaps helpful to think of Petri nets: The exclusion removes a token, the inclusion adds a token.

Finally, f is marked restless (third component) if either it was already restless or it became restless ($f \in R$). We then define the combined action of an event and effect by $(e : \delta) \cdot M = \delta \cdot (e \cdot M)$.

In defining the compositional semantics, we need to merge parallel markings and effects. Merge on markings is *partial*, since it is only defined on markings that agree on their overlap:

$$(M_1, e : m) \oplus (M_2, e : m) = (M_1 \oplus M_2), e : m$$
$$(M_1, e : m) \oplus M_2 = (M_1 \oplus M_2), e : m \quad \text{when } e \notin \mathsf{dom}(M_2)$$

The *merge of effects* δ is always defined; it is simply the pointwise union:

$$(E_1, I_1, R_1) \oplus (E_2, I_2, R_2) = (E_1 \cup E_2, I_1 \cup I_2, R_1 \cup R_2)$$

With these mechanics in place, we give transition semantics of processes in Figure 5.

We use two forms of transitions: the *effect transition* $[M]\ T \xrightarrow{e:\delta} T'$ says that $[M]\ T$ may exhibit event e with effect δ, in the process updating the term T to become T'. (At this stage we will always have $T = T'$; we will need

updates only when we extend the calculus in Section 4 below.) The *process transition* $[M]\,T \xrightarrow{e} [N]\,U$ takes a process to another process, applying the effect of e to the marking M, and thus only exhibiting the event e. The [INTRO] rule elevates an enabled event with an effect to an effect transition. The [PAR] rule combines effect transitions from the two sides of a parallel when they have compatible markings. The [EFFECT] rule lifts an effect transition to a process transition by applying the effect to the marking. Process transitions give rise to an LTS, which we equip with a notion of *acceptance* defined formally below, corresponding to the one of DCR Graphs [11]: a run is accepting if every restless event eventually either happens or is excluded.

Definition 2 A DCR process defines an LTS with states $[M]\,T$ and (process) transitions $[M]\,T \xrightarrow{e} [N]\,U$. A run of $[M]\,T$ is a finite or infinite sequence of transitions $[M]\,T = [M_0]\,T_0 \xrightarrow{e_0} \cdots$. A run is *accepting* iff for every state $[M_i]\,T_i$, when $M_i(e) = (\_, \mathsf{t}, \mathsf{t})$ then there exists $j \geq i$ s.t. either $M_j(e) = (\_, \mathsf{f}, \_)$ or $[M_j]\,T_j \xrightarrow{e:\delta} [M_{j+1}]\,T_{j+1}$.

Note that since an event e may happen more than once, even DCR graphs with only finitely many events may have infinite runs.

Having defined the LTS and runs, we give semantics to DCR processes: The meaning of a DCR process is the language comprising its accepting runs.

Definition 3 A *trace* of a process $[M]\,T$ is a possibly infinite string $s = (s_i)_{i \in I}$ s.t. $[M]\,T$ has an accepting run $[M_i]\,T_i \xrightarrow{e_i} [M_{i+1}]\,T_{i+1}$ with $s_i = \ell(e_i)$. Finally, the process $[M]\,T$ has *language*

$$\mathsf{lang}([M]\,T) = \{s \mid s \text{ is a trace of } [M]\,T\}\ .$$

Example 4 (Grant process transitions) As transitions change only marking, not terms, we show a run by showing changes in the marking. In the table below, rows indicate changes to the marking as the event on the left happens. Columns "h,i,r" indicate whether an event is marked (h)appened, (i)ncluded, and/or (r)estless. The column "Accepts?" indicates whether the current marking is accepting or not and the final column "Enabled" indicates which events are enabled after executing the event on the left.

Event	round			deadline			recv			bm			Acc	Enabled
happening	h	i	r	h	i	r	h	i	r	h	i	r		
(none)	f	t	f	f	t	f	f	f	f	f	t	f	t	{round, deadline, bm}
round	t							t				t	f	{round, deadline, recv}
deadline				t				f					f	{round, deadline, bm}
bm										t		f	t	{round, deadline, bm}
round								t				t	f	{round, deadline, recv}
recv							t						f	{round, deadline, recv, bm}
bm												f	t	{round, deadline, recv, bm}

After the first round event, bm cannot happen because of recv $\rightarrow\bullet$ bm. When deadline happens, it excludes recv because of bm $\%\leftarrow$ recv, and exclusion of recv voids the condition recv $\rightarrow\bullet$ bm; so after deadline, bm may again happen. When round subsequently re-includes recv, bm is again disabled. Acceptance of the processes changes throughout. Because of bm $\leftarrow\bullet$ round, whenever round executes it makes bm restless, preventing the process from accepting until bm later happens, ceasing to be restless. In our examples, we identify events and labels, so the above table indicates an accepting trace $\langle \mathsf{round}, \mathsf{deadline}, \mathsf{bm}, \mathsf{round}, \mathsf{recv}, \mathsf{bm} \rangle$.

Example 5 (Event structures) A labelled prime event structure [49] can be defined as a tuple $\mathbf{E} = (E, \leq, \#, \ell, L)$ where E is a set of events, $\leq$ is a partial order on events defining the *causal dependency* relation (satisfying an axiom of finite cause), $\#$ is the binary, symmetric and irreflexive *conflict* relation (satisfying an axiom of hereditary conflict) and ℓ is a labelling function assigning every event to a label. A finite event structure $\mathbf{E}$ can be represented as the DCR term

$$T_{\mathbf{E}} = \prod_{e<e'} e \rightarrow\bullet e' \parallel \prod_{e\#e' \vee e=e'} e \,\%\leftarrow e'$$

A state of an event structure is referred to as a *configuration*, defined as a finite, downwards closed and conflict free set $C \subseteq E$ of events. Define $C^{\#} = \{e \mid \exists e' \in C. e\#e'\}$. A configuration for finite event structures can then be represented by the marking $M_{\mathbf{E}}$ defined by $M_{\mathbf{E}}(e) = (\mathsf{t}, \mathsf{f}, \mathsf{f})$ for $e \in C$, $M_{\mathbf{E}}(e) = (\mathsf{f}, \mathsf{f}, \mathsf{f})$ for $e \in C^{\#}$ and $M_{\mathbf{E}}(e) = (\mathsf{f}, \mathsf{t}, \mathsf{f})$ for $e \notin C \cup C^{\#}$. The DCR process $[M_{\mathbf{E}}]\ T_{\mathbf{E}}$ then represent a pair of a configuration and an event structure, which indeed will have the same behaviour as the event structure.

DCR processes can represent more expressive variants of event structures. For instance, an event structure with a set $R \subseteq E$ of *restless* events as considered in [48] is then defined in the same way, except that the events in R will initially be restless in the marking representing the configuration, i.e. the third component of the event state will be t. Using the inclusion relation, it is possible to represent resolvable conflicts similar to event structures with resolvable conflict [19].

We note the connection between DCR processes and DCR Graphs [36, 47, 13] (proof in Appendix A):

Proposition 6 *There exists a language-preserving equivalence between DCR processes and finite DCR graphs.*

3 Expressiveness of DCR Processes

In this section we show that DCR processes characterise exactly those languages that are the union of a regular and an ω-regular language. The key idea for proving the result is encoding Büchi automata into DCR processes (see also [36]).

$$\mathsf{t}(B) = \prod_{\substack{p \xrightarrow{l} q \\ p \notin F \\ i \in \{0,1\}}} \Big(\prod_{p \xrightarrow{l'} q'} (p,l',q',i) \,\%\!\leftarrow (p,l,q,i) \parallel \prod_{q \xrightarrow{l'} q'} (q,l',q',i) \mathrel{+\!\!\leftarrow} (p,l,q,i) \Big) \tag{1}$$

$$\parallel \prod_{\substack{p \xrightarrow{l} q \\ p \in F \\ i \in \{0,1\}}} \Big(\prod_{p \xrightarrow{l'} q'} (p,l',q',i) \,\%\!\leftarrow (p,l,q,i) \parallel \prod_{q \xrightarrow{l'} q'} \big((q,l',q',1-i) \mathrel{+\!\!\leftarrow} (p,l,q,i) \parallel f_{1-i} \mathrel{+\!\!\leftarrow} (p,l,q,i) \parallel f_i \,\%\!\leftarrow (p,l,q,i)\big)\Big) \tag{2}$$

$$\parallel f_0 \rightarrow\!\bullet\, f_0 \parallel f_1 \rightarrow\!\bullet\, f_1 \tag{3}$$

Fig. 6 Term $\mathsf{t}(B)$.

Recall that a Büchi automaton $B = (Q, \Sigma, \delta, q_0, F)$ comprises a finite set of states Q, an alphabet Σ, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, an initial state $q_0 \in Q$ and a set of accepting states $F \subseteq Q$. An infinite run of B is a sequence $q_0, l_0, q_1, l_1, \ldots$ where each $q_i \in Q$ and each l_i in Σ and for all $i \geq 0$ we have $(q_i, l_i, q_{i+1}) \in \delta$. A run is accepting iff there exists some $q \in F$ s.t. $q_j = q$ for infinitely many $j \geq 0$.

Given a Büchi automaton $B = (Q, \Sigma, \delta, q_0, F)$ we define a corresponding term $\mathsf{t}(B)$ in Figure 6 and marking $\mathsf{m}(B)$ in Figure 7. We model each transition $p \xrightarrow{l} q$ with an event (p, l, q) labelled l. At any time, only events corresponding to a single state p are included; all other events are excluded. To change state, an event (p, l, q) excludes all events $(p, \_, \_)$ and includes all events $(q, \_, \_)$. Acceptance is modelled by two restless events f_0, f_1 which are never enabled. Whenever a transition out of an accepting state $q \in F$ happens, we toggle which of f_0, f_1 is included. An accepting run of the Büchi automaton will infinitely toggle f_0, f_1 and thus be an accepting run of the DCR process; a non-accepting run will leave either f_0 or f_1 included and restless infinitely, yielding a non-accepting DCR run.

To toggle which of f_0, f_1 is included, it is necessary to split each transition (event) (p, l, q) into *two* copies $(p, l, q, 0)$ and $(p, l, q, 1)$. Only one copy is included at any time; a transition out of an accepting state then switches which copy is active using suitable include and exclude relations. Let's see how this idea is reflected in Figure 6; numbers refer to equation numbers in that figure.

(1) Firing a transition $p \xrightarrow{l} q$ out of a *non-accepting* state p excludes all other transitions out of that state and includes all transitions out of q in the same copy i.
(2) Firing a transition $p \xrightarrow{l} q$ out of an *accepting* state p excludes all other transitions out of that state, then includes all transitions out of q in the other copy $1 - i$. The second line toggles which of f_0, f_1 is included.
(3) Finally, we make sure that neither f_0 nor f_1 can themselves be executed.

Event	Happened	Included	Restless
f_0	f	t	t
f_1	f	f	t
$(p_0, l, q, 0)$	f	t	f
$(p \neq p_0, l, q, 0)$	f	f	f
$(p, l, q, 1)$	f	f	f

Fig. 7 Marking $\mathsf{m}(B)$.

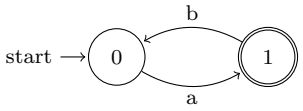

Fig. 8 Example Büchi automaton.

Example 7 Consider the Büchi automaton B in Figure 8. Using the above construction, we find the events and transitions in the table below. Numbers refer to equation numbers of Figure 6. Relations should be read left-to-top, i.e., the event on the left sits at the left of the arrow, the event at the top sits at the right.

	$(0,a,1,0)$	$(1,b,0,0)$	$(0,a,1,1)$	$(1,b,0,1)$	f_0	f_1
$(0,a,1,0)$	$\rightarrow\%$ (1)	$\rightarrow+$ (1)				
$(1,b,0,0)$		$\rightarrow\%$ (2)	$\rightarrow+$ (2)		$\rightarrow\%$ (2)	$\rightarrow+$ (2)
$(0,a,1,1)$			$\rightarrow\%$ (1)	$\rightarrow+$ (1)		
$(1,b,0,1)$	$\rightarrow+$ (2)			$\rightarrow\%$ (2)	$\rightarrow+$ (2)	$\rightarrow\%$ (2)
f_0						
f_1						

Lemma 8 *A Büchi automaton B accepts an infinite string s iff the DCR process $[\mathsf{m}(B)]\ \mathsf{t}(B)$ does.*

Proof Any run of B is of the form

$$q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \cdots$$

By construction, any run of $[\mathsf{m}(B)]\ \mathsf{t}(B)$ is of the form

$$[\mathsf{m}(B) = M_0]\ \mathsf{t}(B) \xrightarrow{(q_0,l_0,q_1,0)} [M_1]\ \mathsf{t}(B) \xrightarrow{(q_1,l_1,q_2,i_1)} \ldots$$

Clearly these runs exhibit the same sequence of labels. It remains to show that either they are both accepting or both non-accepting. Suppose the run of B is not. Then for some n and $i > n$ we have $q_i \notin F$. But then by construction, either f_0 or f_1 is restless and included in each M_{i+1}, and so also the run of $[\mathsf{m}(B)]\ \mathsf{t}(B)$ is not accepting. If instead the run of B *is* accepting, then there exists a state q s.t. for all n there exists a $j > n$ with $q_j = q$. But then by construction also the included and restless states in M_j and M_{j+1} are disjoint, and so also the run of $[\mathsf{m}(B)]\ \mathsf{t}(B)$ is accepting. □

$$
\begin{array}{lll}
T, U ::= & \ldots & \\
& \mid (\nu e : \Phi)\, T & \text{local event} \\
& \mid\ !e.T & \text{replication event}
\end{array}
$$

Fig. 9 DCR* syntax.

With this Lemma, we can exploit existing results on ω-regular languages to fully characterise the expressive power of DCR processes.

Proposition 9 *For every language $\mathcal{L}$ that is the union of a regular and an ω-regular language, there exists a DCR process recognising exactly $\mathcal{L}$.*

Proof For such a language, there exists a finite automaton F recognising exactly the finite part and a Büchi automaton B recognising exactly the infinite part. We adapt the above construction to one simulating F and B simultaneously: Replace events (q, l, p, i) with "product-transition" events

$$((q_B, q_F), l, (p_B, p_F), i)\ .$$

To model finite acceptance, for every accepting state q_F of the F, we duplicate transition events going into q_F yet again, obtaining for these events

$$((q_B, q_F), l, (p_B, p_F), i, j)$$

for $0 \leq j \leq 1$. For $j = 0$ we add relations as usual. For $j = 1$, we add exclude relation to *every* event. Thus, firing, say $((q_B, q_F), l, (p_B, p_F), i, 1)$ when p_F is accepting in F excludes every event of the DCR process, leaving it in a terminated and accepting state. □

Using the above propositions we get the promised characterisation.

Theorem 10 *A language $\mathcal{L}$ is recognised by some DCR process iff $\mathcal{L}$ is the union of a regular and an ω-regular language.*

Proof Suppose $\mathcal{L}$ is recognised by a DCR process P. By Proposition 6 there exists a DCR graph G with the same language. From [36] we have that G can be encoded as a Büchi automaton. Suppose instead $\mathcal{L}$ is the union of a regular and an ω-regular language. By Proposition 9 we have that there exists a DCR process with language $\mathcal{L}$. □

4 DCR* Processes: Local events and Replication

In this section we extend the DCR process language to support dynamic creation of sub processes. We do this by extending the syntax with *local* and *replication* events as shown in Figure 9 to the right, giving rise to the DCR* process language.

We assume that for each label $l \in \mathcal{L}$, there exist infinitely many events e with that label, that is, with $\ell(e) = l$.

The *local event* $(\nu e : \Phi)\ T$ asserts that e is local to the term T. This construct is binding, and we take terms up to *label preserving* α-conversion, i.e., $(\nu e : \Phi)\ T = (\nu f : \Phi)\ T[f/e]$ iff $\ell(e) = \ell(f)$ and f not free in T. As usual, we assume the Barendregt-convention and may thus assume binders are distinct. A *replication event* $!e.T$ unfolds a copy of T whenever it happens.

We define *free events* and *alphabet* for DCR* processes.

Definition 11 The *free events* $\mathsf{fe}(T)$ of a term T is defined recursively as follows.

$$\begin{aligned}
\mathsf{fe}(e\ \mathcal{R}\ f) &= \{e, f\} \\
\mathsf{fe}(T \parallel U) &= \mathsf{fe}(T) \cup \mathsf{fe}(U) \\
\mathsf{fe}(0) &= \emptyset \\
\mathsf{fe}((\nu e : \Phi)\ T) &= \mathsf{fe}(T) \setminus \{e\} \\
\mathsf{fe}(!e.T) &= \{e\} \cup \mathsf{fe}(T)
\end{aligned}$$

The free events of a process $\mathsf{fe}([M]\ T)$ is simply $\mathsf{fe}([M]\ T) = \mathsf{dom}(M)$; we maintain the requirement that a process $[M]\ T$ has $\mathsf{fe}(T) \subseteq \mathsf{dom}(M)$. The *alphabet* $\mathsf{alph}(P)$ of a process is the set of labels associated with its events, defined recursively as follows.

$$\begin{aligned}
\mathsf{alph}(e\ \mathcal{R}\ f) &= \{\ell(e), \ell(f)\} \\
\mathsf{alph}(T \parallel U) &= \mathsf{alph}(T) \cup \mathsf{alph}(U) \\
\mathsf{alph}(0) &= \emptyset \\
\mathsf{alph}((\nu e : \Phi)\ T) &= \{\ell(e)\} \cup \mathsf{alph}(T) \\
\mathsf{alph}(!e.T) &= \{\ell(e)\} \cup \mathsf{alph}(T)
\end{aligned}$$

The following Lemma states that transitions preserve free events and alphabet.

Lemma 12 *Transitions* $[M]\,T \xrightarrow{\lambda} T'$ *and* $[M]\,T \xrightarrow{\gamma} [M']\,T'$ *preserve free events and alphabet, that is* $\mathsf{fe}(M) = \mathsf{fe}(M')$, $\mathsf{fe}(T) = \mathsf{fe}(T')$, $\mathsf{alph}(T) = \mathsf{alph}(T')$, *and* $\mathsf{alph}(M) = \mathsf{alph}(M')$.

Proof Preservation of free events and alphabet of terms for effect transitions follows by easy induction on the derivation of the transition. For preservation for process transitions, observe that by cases on the rules admitting a transition $[M]\,T \xrightarrow{\gamma} [M']\,T'$, we must have $\mathsf{dom}(M) = \mathsf{dom}(M')$ by definition of the action operator $- \cdot M$; the desiderata now follows. □

The transition rules for the new constructs are given in Figure 10. Only terms and transition rules are extended; markings are the same.

Rule [LOCAL] gives semantics to events happening in the scope of a local event binder. An effect on the local event is recorded in the marking in the binder of that event. The event might have effects on non-local events, e.g., in

$$\frac{[M, f:\Phi]\ T \xrightarrow{e:\delta} T' \quad f:\Phi' = (e:\delta)\cdot(f:\Phi) \quad \gamma = \nu e \text{ if } e = f, \text{ o.w. } \gamma = e}{[M]\ (\nu f:\Phi)\ T \xrightarrow{\gamma:(\delta\backslash f)} (\nu f:\Phi')\ T'} \ [\textsc{local}]$$

$$\frac{[M]\ T \xrightarrow{\nu e:\delta} T'}{[M]\ T \parallel U \xrightarrow{\nu e:\delta} T' \parallel U} \ [\textsc{par-2}] \qquad \frac{[M]\ T \xrightarrow{e:\delta} T'}{[M]\ !e.T \xrightarrow{e:\delta} !e.T \parallel T'} \ [\textsc{rep}]$$

$$\frac{[M]\ T \xrightarrow{\nu e:\delta} T'}{[M]\ T \xrightarrow{\nu e} [\delta \cdot M]\ T'} \ [\textsc{effect-2}]$$

Here $\delta\backslash f = (E\backslash\{f\}, I\backslash\{f\}, R\backslash\{f\})$. We omit the obvious rule symmetric to [Par-2].

Fig. 10 Transition semantics for local and replication events.

$(\nu f : M)\ e +\!\!\leftarrow f$, the local f has effects on the non-local e. Thus the effects are preserved in the conclusion, except that part of the effect which pertain only to f. Rule [par-2] propagates a local effect through a parallel composition. It is possible that the effect δ mentions events in U; however, it cannot mention events *local* to U. So the effects of δ on U are fully expressed in the effect of δ on M. Rule [effect-2] lifts effect transitions with local events to process transitions. Finally, the rule [rep] implements replication events: If the guarding event e happening would update the body T to become T', then e can unfold to such a T'.

To define accepting runs we need to track local restless events across transitions. Fortunately, DCR* is simple enough that we can simply assume that α-conversion happens only during replication (i.e., local events duplicated by [rep] are chosen globally fresh).

Definition 13 A run of a DCR* process $[M]\ T$ is a finite or infinite sequence $[M_i]\ N_i \xrightarrow{\lambda_i} [M_{i+1}]\ N_{i+1}$ with $\lambda = e_i$ or $\lambda = \nu e_i$. The trace of a run is the sequence of labels of its events, i.e., the string given by $\ell(\lambda_i)$ where $\ell(\nu e) \stackrel{\text{def}}{=} \ell(e)$. A run is *accepting* if whenever an event e is marked as restless in M_i respectively a local event νe is marked as restless by its binder in T_i, then there exists some $j \geq i$ s.t. either $[M_j]\ T_j \xrightarrow{\lambda_i} [M_{j+1}]\ T_{j+1}$ with $\lambda_i = e$ respectively $\lambda_i = \nu e$; or the event state of e in M_j respectively T_j has e excluded.

Note that the treatment of restless events in this Definition is morally the same as that of Definition 2, adding the necessary mechanics to accommodate bound events.

Example 14 (Grant process with replication and local events) We now consider the requirement that when an application is received, a committee recommends either approval or rejection to the board. The committee might rescind an approval, but cannot reverse a rejection. Moreover, when applications are competing for resources, the board cannot make a final decision until it has a recommendation for every received application. We again use events recv and

bm for receiving an application and convening a board meeting. We use *local* events νapprove and νreject to model the per-application evaluation process replication by recv. We first give a term A modelling a *single* recommendation sub-process.

$$A = (\nu\mathsf{approve} : (\mathsf{f},\mathsf{t},\mathsf{t}))\ (\nu\mathsf{reject} : (\mathsf{f},\mathsf{t},\mathsf{f}))\ \big(\mathsf{approve} \mathbin{\%\!\leftarrow} \mathsf{reject} \parallel \mathsf{approve} \rightarrow\!\bullet\ \mathsf{bm}\big)$$

Note that approve and reject are local and can thus not be constrained further outside the scope, yet approve has a condition relation to the non-local bm. We make the approve event initially restless, which will mean that in order for the process to be accepting approve must either happen or be excluded (because reject happens).
We can now model a process $[M_1]\ T_1$ reproducing the per-application sub-process A every time recv happens

$$T_1 = !\mathsf{recv}.A \parallel \mathsf{recv} \rightarrow\!\bullet\ \mathsf{bm} \qquad M_1 = \mathsf{recv} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{bm} : (\mathsf{f},\mathsf{t},\mathsf{f})$$

(To keep the process simple, we only retained from the previous example the constraint that a board meeting requires receiving an application). In DCR*, the term *does* change as the process evolves. Let's see $[M_1]\ T_1$ evolve:

$$\begin{aligned}
[M_1]\ T_1 &\xrightarrow{\mathsf{recv}} [M_2]\ T_1 \parallel A_1 \\
&\xrightarrow{\mathsf{recv}} [M_2]\ T_1 \mid A_1 \parallel A_2 && (4)\\
&\xrightarrow{\nu\mathsf{approve}_1} [M_2]\ T_1 \parallel \big((\nu\mathsf{approve}_1 : (\boxed{\mathsf{t}},\mathsf{t},\boxed{\mathsf{f}}))\ (\nu\mathsf{reject}_1 : (\mathsf{f},\mathsf{t},\mathsf{f})) && (5)\\
&\qquad\qquad \mathsf{approve}_1 \mathbin{\%\!\leftarrow} \mathsf{reject}_1 \parallel \mathsf{approve}_1 \rightarrow\!\bullet\ \mathsf{bm}\big) \parallel A_2 \\
&\xrightarrow{\nu\mathsf{reject}_2} [M_2]\ T_1 \parallel \big((\nu\mathsf{approve}_1 : (\mathsf{t},\mathsf{t},\mathsf{f}))\ (\nu\mathsf{reject}_1 : (\mathsf{f},\mathsf{t},\mathsf{f})) && (6)\\
&\qquad\qquad \mathsf{approve}_1 \mathbin{\%\!\leftarrow} \mathsf{reject}_1 \parallel \mathsf{approve}_1 \rightarrow\!\bullet\ \mathsf{bm}\big) \\
&\qquad\quad \big((\nu\mathsf{approve}_2 : (\mathsf{f},\boxed{\mathsf{f}},\mathsf{t}))\ (\nu\mathsf{reject}_2 : (\boxed{\mathsf{t}},\mathsf{t},\mathsf{f})) \\
&\qquad\qquad \mathsf{approve}_2 \mathbin{\%\!\leftarrow} \mathsf{reject}_2 \parallel \mathsf{approve}_2 \rightarrow\!\bullet\ \mathsf{bm}\big) \\
&\xrightarrow{\mathsf{bm}} [M_3]\ T_1 \parallel \cdots && (7)
\end{aligned}$$

Here $M_2 = \mathsf{recv} : (\boxed{\mathsf{t}},\mathsf{t},\mathsf{f}), \mathsf{bm} : (\mathsf{f},\mathsf{t},\mathsf{f})$ and $M_3 = \mathsf{recv} : (\mathsf{t},\mathsf{t},\mathsf{f}), \mathsf{bm} : (\boxed{\mathsf{t}},\mathsf{t},\mathsf{f})$.

At (4), the processes A_1 and A_2 are copies of A where local events approve and reject have been α-converted to $\mathsf{approve}_1, \mathsf{approve}_2$ and reject_1, reject_2 respectively, following the convention of unique local names. Moreover, because they have not happened in the local markings under the binders, bm cannot happen. To see this, observe that by the [PAR]-rule, for the whole process to exhibit bm, every part of it must also exhibit bm. But the process

$$(\nu\mathsf{approve}_1 : (\mathsf{f},\mathsf{t},\mathsf{t}))\ \ldots\ \mathsf{approve}_1 \rightarrow\!\bullet\ \mathsf{bm}$$

cannot: the hypothesis of rule [LOCAL], that bm could happen if $\mathsf{approve}_1$ is considered global with marking $(\mathsf{f},\mathsf{t},\mathsf{t})$, cannot be established.

When a local $\mathsf{approve}_i$ event happens, its local marking changes to reflect that the event happened and is no longer restless, as indicated with grey

background in (5). However, $\mathsf{approve}_1$ happening is not enough to enable bm; it is still disabled by the other copy. Also, the entire process is not in an accepting state, since $\mathsf{approve}_2$ is still restless and included. Once reject happens in the second copy (6), excluding $\mathsf{approve}$ in that copy, bm is enabled and the process is in an accepting state: of the two local $\mathsf{approve}$ events bm is conditional upon, one has happened (and thus also no longer restless), and the other is excluded (and thus also no longer required for acceptance).

5 Complexity of DCR and DCR* Processes

In this section we investigate complexity of DCR and DCR*. We shall see that deciding whether there exists a run including given event is NP-HARD for DCR processes and undecidable for DCR* processes, the latter following from a reduction from the Halting-problem for Minsky machines. We shall subsequently use the results of the present section as a basis for analysing the complexity of refinement in the next.

We shall take as our starting point the following key notion of "event-reachability":

Definition 15 (Event-reachability) Let $P = [M_1]\ T$ be a DCR* process, and let e be an event of P. We say that e *is eventually reachable in* P iff there is a transition sequence

$$M_1 \xrightarrow{e_1} M_2 \xrightarrow{e_2} \cdots \xrightarrow{e_n} M_{n+1} \xrightarrow{e} N\ .$$

We call the sequence $e_1, \ldots, e_n$ a *witness* for reachability of e. The *event-reachability* problem for (P, e) is then deciding whether a given e is eventually reachable in P.

5.1 Event-reachability in DCR Processes

We show that Event-reachability for DCR processes (as opposed to DCR* processes) is decidable but hard, by reduction from the boolean satisfiability problem. The key idea is to encode the nodes (subformulae) of the abstract syntax of a boolean satisfiability problem as events. Each node has two corresponding events: One which will fire iif the corresponding subformula evaluates to true, one which will fire iff if it evaluates to false.

Lemma 16 *For DCR processes, there exists an* NLOGSPACE*-reduction* $B \mapsto (\llbracket B \rrbracket, e)$ *from boolean satisfiability to event-reachability.*

Proof Let B be a boolean satisfiability problem over atoms, conjunction and negation. Construct a DCR process $\llbracket B \rrbracket$ which has, for each non-leaf node n of the abstract syntax tree of B, events $n^{\mathsf{t}}, n^{\mathsf{f}}$ and $n^{\mathsf{b1}}, n^{\mathsf{b2}}, n^{\mathsf{b3}}$; and for each atom a events $a^{\mathsf{t}}, a^{\mathsf{f}}$ and $a^{\mathsf{b1}}, a^{\mathsf{b2}}$. Then add relations as follows.

1. For each atom a, add relations $a^{\mathsf{b1}} \rightarrow\bullet\ a^{\mathsf{b1}}$ and $a^{\mathsf{b2}} \rightarrow\bullet\ a^{\mathsf{b2}}$; and $a^{\mathsf{b1}} \rightarrow\bullet\ a^{\mathsf{t}}$ and $a^{\mathsf{b2}} \rightarrow\bullet\ a^{\mathsf{f}}$ and $a^{\mathsf{b2}} +\!\leftarrow a^{\mathsf{t}}$ and $a^{\mathsf{b1}} +\!\leftarrow a^{\mathsf{f}}$.
2. For each non-leaf node n, add relations $n^{\mathsf{b1}} \rightarrow\bullet\ n^{\mathsf{b1}}, n^{\mathsf{b2}} \rightarrow\bullet\ n^{\mathsf{b2}}, n^{\mathsf{b3}} \rightarrow\bullet\ n^{\mathsf{b3}}$.
3. For each non-leaf node $n = u \wedge v$ add relations $n^{\mathsf{b1}} \rightarrow\bullet\ n^{\mathsf{t}}$, $n^{\mathsf{b2}} \rightarrow\bullet\ n^{\mathsf{t}}$ and $n^{\mathsf{b3}} \rightarrow\bullet\ n^{\mathsf{f}}$; and $n^{\mathsf{b1}}\ \%\!\leftarrow u^{\mathsf{t}}$, $n^{\mathsf{b2}}\ \%\!\leftarrow v^{\mathsf{t}}$; and $n^{\mathsf{b3}}\ \%\!\leftarrow u^{\mathsf{f}}$, $n^{\mathsf{b3}}\ \%\!\leftarrow v^{\mathsf{f}}$.
4. For each non-leaf node $n = \neg u$ add relations $n^{\mathsf{t}} \rightarrow\bullet\ u^{\mathsf{f}}$ and $n^{\mathsf{f}} \rightarrow\bullet\ u^{\mathsf{t}}$.

Define a marking M_0 where every event is $(\mathsf{f},\mathsf{t},\mathsf{f})$, except a^{b1} and a^{b2} which must be $(\mathsf{f},\mathsf{f},\mathsf{f})$. Consider a run of $[\![B]\!]$ executing a maximal number of distinct events of $[\![B]\!]$. Note that for each atom a; either a^{t} or a^{f} has been executed, but not both; define from this an assignment $a \mapsto \mathsf{t}$ or $a \mapsto \mathsf{f}$. By induction, each non-leaf node n has n^{t} executed iff under this assignment n evaluates to true and n^{f} executed iff it evaluates to false. But then for the root note r of B, r^{t} is executed in some run iff B is satisfiable; i.e., r^{t} is reachable in $[\![B]\!]$ iff B is satisfiable. The DCR process $[\![B]\!]$ has $O(|B|)$ nodes and relations, and so this reduction is in NLOGSPACE. □

Theorem 17 *Event-reachability for DCR is* NP-HARD *and in* EXPTIME.

Proof From Lemma 16 we have that Event-reachability is NP-HARD. To see that it is in EXPTIME, first observe that for a DCR process $P = [M]\ T$, if

$$[M]\ T \xrightarrow{\lambda} [M']\ T'$$

then $T' = T$. It follows that the size of the LTS of P is bounded by the number of possible markings M, which in turn is bounded by

$$(2^3)^{|\mathsf{dom}(M)|}\ ,$$

because each event has has three state variables, each of which can assume just two possible values, and there are $|\ \mathsf{dom}(M)\ |$ such events. It is straightforward to construct an algorithm which traverses each state of the LTS of P, checking if any state has e executable. Processing each state—finding out which events are executable and what states might be reached if one were to execute one of these events—is clearly polynomial in the size of P; so this algorithm takes time at most exponential in the size of P. □

We establish in passing that the fragment of DCR* that has local names but no replication is equivalent to plain DCR processes.

Proposition 18 *Let DCR$^\nu$ be the fragment of DCR* obtained by requiring that no term T contains a subterm $!e.T$. For any $P \in$ DCR$^\nu$, there exists $\hat{P} \in$ DCR with $\mathsf{lang}(P) = \mathsf{lang}(\hat{P})$. Moreover, the assignment $P \mapsto \hat{P}$ is computable in* NLOGSPACE.

Proof Let $P = [M]\ T$ be a DCR$^\nu$ process. Let f be a function assigning to each local event νe in P an event $f(e)$ fresh for P with process $\ell(f(e)) = \ell(e)$. Define $\hat{P} = [M']\ T'$ to be P in which

1. Every sub-term on the form $(\nu e : \Phi)\ T$ has been replaced by T,
2. Every occurrence of a local event e is replaced by $f(e)$, and
3. $M' = M, M''$ where $\mathsf{dom}(M'')$ is exactly the range of f and whenever there is a sub-term $(\nu e : \Phi)\ T$ in P we have $f(e) : \Phi \in M''$.

It is straightforward to verify by induction on the transition semantics that $P \xrightarrow{\lambda} P'$ iff $\hat{P} \xrightarrow{\hat{\lambda}} \hat{P'}$ (where $\hat{e} = e$ and $\hat{\nu e} = f(e)$). It follows that $(P, \hat{P})$ is a bisimulation (see, e.g., [40]), whence $\mathsf{lang}(P) = \mathsf{lang}(\hat{P})$. The assignment $P \mapsto \hat{P}$ is clearly in NLOGSPACE. □

5.2 Event-reachability in DCR⋆

In this subsection, we show that Event-reachability in DCR⋆ is undecidable by a reduction from the halting problem for Minsky Machines [34].

Recall that a Minsky machine $m = (R_1, R_2, P, c)$ comprises two unbounded *registers* R_1, R_2; a *program* P, which is a list of pairs of addresses and instructions; and a *program counter* c, giving the address of the current instruction. It has the following instruction set.

$\mathtt{inc}(i, a)$	Add 1 to the contents of register i. Proceed to a.
$\mathtt{decjz}(i, a, b)$	If register i is zero, proceed to a. Otherwise subtract 1 from register i and proceed to b.
`halt`	Halt execution (wlog assumed to appear exactly once).

We construct, given a Minsky machine m, a term $\mathsf{t}(m)$ and a marking $\mathsf{m}(m)$. We model machine instructions as events. To maintain execution order, we model program addresses explicitly as events a. These events serve only to constrain the execution of other events; they should not themselves happen, and we prevent them from doing so with a condition $a \rightarrow\bullet\ a$ for each a. By making each instruction event e conditional on its program point a, $a \rightarrow\bullet\ e$, we ensure that *e can happen only if a is excluded.*

To move the program counter from a to b, we re-include a and exclude b. We define a shorthand $\mathsf{insn}(e, a, b)$ for an instruction event e at program point a proceeding to program point b as follows:

$$\mathsf{insn}(e, a, b) = a \rightarrow\bullet\ e \parallel a \mathrel{+}\!\leftarrow e \parallel b \mathrel{\%}\!\leftarrow e$$

Next, we model registers. We model each $a : \mathtt{decjz}(i, b, c)$ by two events: one, $\mathtt{decjz}^a$, which can happen only when the register is zero, and a second, $\mathtt{decjn}^a$, which can happen only when it is not. Then we model increments by making each increment replicated, spawning a new copy of $\mathtt{decjn}^a$ for every decrement instruction $a : \mathtt{decjz}(i, b, c)$ in P. The copies produced by a single increment represent the *opportunity* for exactly one of these instructions to decrement. Thus, we make the copies in a single increment exclude each other. To make sure that $\mathtt{decjz}^a$ cannot happen if the register is non-zero, that is, if no $\mathtt{decjn}^a$ is present, we make the latter a condition of the former:

Event	Happened	Included	Restless
c	f	f	f
a when $a \neq c$	f	t	f
$\mathtt{decjz}^a$	f	t	f
$\mathtt{inc}^a$	f	t	f
halt	f	t	t

Fig. 11 Marking $\mathsf{m}(m)$.

$\mathtt{decjn}^a \rightarrow\bullet\ \mathtt{decjz}^a$. Altogether, the term for one increment is constructed by the following function. (We write $(N_{i\in I} x_i : M)$ for $(\nu x_{i_1} : M) \ldots (\nu x_{i_n} : M)$ when $I = \{i_1, \ldots, i_n\}$.)

$$\mathsf{one}(i) = \Big(\mathop{\mathsf{N}}_{a:\mathtt{decjz}(i,c,d)} \mathtt{decjn}^a : (\mathsf{f},\mathsf{t},\mathsf{f}) \Big) \prod_{a:\mathtt{decjz}(i,c,d)} \Big(\mathsf{insn}(\mathtt{decjn}^a, a, d) \parallel \mathtt{decjn}^a \rightarrow\bullet\ \mathtt{decjz}^a \parallel \prod_{a':\mathtt{decjz}(i,b',c')} \mathtt{decjn}^{a'} \mathbin{\%\!\leftarrow} \mathtt{decjn}^a \Big)$$

Adding one to a register i is accomplished by making a new copy of $\mathsf{one}(i)$.

$$\mathsf{inc}(a, i, b) = \mathsf{insn}(\mathtt{inc}^a, a, b) \parallel !\mathtt{inc}^a.\mathsf{one}(i)$$

We put it all together and define $\mathsf{t}(m)$ for a Minsky machine $m = (R_1, R_2, P, c)$.

$$\mathsf{t}(m) = \prod_{a:\mathtt{inc}(i,b)\in P} \mathsf{inc}(a, i, b) \parallel \prod_{a:\mathtt{decjz}(i,b,c)\in P} \mathsf{insn}(\mathtt{decjz}^a, a, b) \parallel \prod_{a:\mathtt{halt}\in P} a \rightarrow\bullet\ \mathtt{halt} \parallel \prod_{a:I\in P} a \rightarrow\bullet\ a \parallel \prod_{i<R_1} \mathsf{one}(1) \parallel \prod_{i<R_2} \mathsf{one}(2)$$

Finally, the marking $\mathsf{m}(m)$ is given in Figure 11 below. (Recall that c is the program counter.)

Example 19 As an example, let us consider a Minsky machine adding the contents of register 2 to register 1. We'll consider the machine $(0, 1, P, 1)$, where P is the program:

$$\begin{array}{l} 1 : \mathtt{decjz}(2, 3, 2) \\ 2 : \mathtt{inc}(1, 1) \\ 3 : \mathtt{halt} \end{array}$$

Applying the above construction, we get the following term (split out in a table for readability).

$\prod_{a:\mathtt{inc}(i,b)\in P} \mathsf{inc}(a, i, b)$	$\prod_{a:\mathtt{decjz}(i,b,c)\in P} \mathsf{insn}(\mathtt{decjz}^a, a, b)$
$2 \rightarrow\bullet\ \mathtt{inc}^2$	$1 \rightarrow\bullet\ \mathtt{decjz}^1$
$2 \mathbin{\%\!\leftarrow} \mathtt{inc}^2$	$1 +\!\leftarrow \mathtt{decjz}^1$
$1 +\!\leftarrow \mathtt{inc}^2$	$3 \mathbin{\%\!\leftarrow} \mathtt{decjz}^1$
$!\mathtt{inc}^2.0$	

$\prod_{a:\mathtt{halt}\in P} a \rightarrow\bullet\ \mathtt{halt}$	$\prod_{a:I\in P} a \rightarrow\bullet\ a$	$\prod_{i<R_1} \mathsf{one}(1)$	$\prod_{i<R_2} \mathsf{one}(2)$
$3 \rightarrow\bullet\ \mathtt{halt}$	$1 \rightarrow\bullet\ 1$ $2 \rightarrow\bullet\ 2$ $3 \rightarrow\bullet\ 3$	0	$(\nu \mathtt{decjn}^1 : (\mathsf{f},\mathsf{t},\mathsf{f}))$ $1 \rightarrow\bullet\ \mathtt{decjn}^1$ $1 +\!\!\leftarrow\ \mathtt{decjn}^1$ $2 \,\%\!\leftarrow\ \mathtt{decjn}^1$ $\mathtt{decjn}^1 \rightarrow\bullet\ \mathtt{decjz}^1$ $\mathtt{decjn}^1 \,\%\!\leftarrow\ \mathtt{decjn}^1$

We emphasise that in the column $\prod_{i<R_2} \mathsf{one}(2)$, all instances of $\mathtt{decjn}^1$ are within the scope of the binder and thus local.

Proposition 20 *A Minsky machine m halts iff $[\mathsf{m}(m)]\ \mathsf{t}(m)$ has an accepting run.*

Proof We establish a bisimulation relation between finite execution traces of the Minsky machine m and reachable markings of the encoding $[\mathsf{m}(m)]\ \mathsf{t}(m)$. First observe that in every reachable marking of $[\mathsf{m}(m)]\ \mathsf{t}(m)$ exactly one of the program address events will be included and exactly one event is enabled.

Now relate an execution trace of the Minsky machine ending in address j to a reachable marking in which that event is excluded. Next, note by induction on the length of the trace that for every pair in that relation, the machine can perform an instruction iff the encoding can execute the corresponding event. Now note, again by induction, that the form of the process $\mathsf{t}(m)$ is preserved as well as the global marking $\mathsf{m}(m)$, *except* that instruction events are being recorded as executed and, in the case of `decjn`, excluded. It follows that the restless `halt` event can be eventually executed if and only if the machine can execute the halt command. □

Theorem 21 *Event-reachability in DCR* is undecidable.*

Proof By reduction from the halting problem for Minsky Machines. Suppose m is a Minsky Machine, choose an event e fresh for $[\mathsf{m}(m)]\ \mathsf{t}(m)$, and construct the DCR* process

$$V = [e : (\mathsf{t},\mathsf{f},\mathsf{f}), \mathsf{m}(m)]\ \mathsf{t}(m) \mid \mathtt{halt} \rightarrow\bullet\ e\ .$$

By Proposition 20, e is reachable in V iff m halts. □

6 Run-time refinement by composition

In this section, as an application of DCR*, we demonstrate how to achieve run-time refinement by dynamic process composition.

Definition 22 Given DCR* processes $[M]\ T$ and $[N]\ S$ their *merge* is defined when $M \oplus N$ is, in which case it is $[M]\ T \oplus [N]\ S = [M \oplus N]\ T \parallel S$. When the merge of two processes is defined, we say that they are *marking compatible*.

Example 23 Suppose now *as the grant process runs*, a new requirement comes up: For regulatory reasons, external auditors must regularly look through the minutes of board meetings. That is, a board meeting must eventually be followed by an audition. We easily model this constraint using restlessness and a new event, audit. However, as we are introducing a new event, we must also introduce additional marking. Altogether, we wish to merge P with the following process R_1.

$$R_1 = [\mathsf{bm} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{audit}(\mathsf{f},\mathsf{t},\mathsf{f})]\ \mathsf{audit} \leftarrow\!\bullet\ \mathsf{bm}$$

We merge the process $P = [M_1]\ T_1$ of Example 14 with R_1:

$$P_1 = P \oplus R_1 = [M_1, \mathsf{audit} : (\mathsf{f},\mathsf{t},\mathsf{f})]\ T_1 \parallel \mathsf{audit} \leftarrow\!\bullet\ \mathsf{bm}$$

Note that since the process *is* the model, merging is not restricted to apply only to the initial process; it is well-defined on any evolution P' of P that is marking compatible with R_1.

As a second example, suppose further that it is also decreed that *during* an audit, no further applications can be received. The following process R_2 models this additional requirement. (For clarity of presentation, we omit the procedure for when an audit failing to satisfy auditors and assume simply that every audit eventually "pass"es.)

$$R_2 = [\mathsf{recv} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{audit} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{pass} : (\mathsf{f},\mathsf{t},\mathsf{f})]\ \mathsf{recv}\ \%\!\leftarrow\ \mathsf{audit} \parallel \mathsf{recv}\ +\!\leftarrow\ \mathsf{pass}$$

We merge P_1 and R_2:

$$\begin{aligned} P_2 &= P_1 \oplus R_2 \\ &= [M_1, \mathsf{audit} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{pass} : (\mathsf{f},\mathsf{t},\mathsf{f})]\ T_1 \parallel \mathsf{audit} \leftarrow\!\bullet\ \mathsf{bm} \\ &\quad \parallel \mathsf{recv}\ \%\!\leftarrow\ \mathsf{audit} \parallel \mathsf{recv}\ +\!\leftarrow\ \mathsf{pass} \end{aligned}$$

When we extend the set of requirements, the run-time refinement of P to P' should ideally not allow traces that were not allowed by the old set of requirements. We may try to formulate this property by language inclusion, e.g, $\mathsf{lang}(P') \subseteq \mathsf{lang}(P)$ to ensure that the adapted process P' is still compliant. But as we have seen, run-time refinement might entail adding *new* events (audit), so we cannot in general expect language inclusion. Hence, we consider language inclusion only w.r.t. the alphabet of P. In doing so we employ the following notation.

Notation. Given a sequence s, write s_i for the ith element of s, and $s|_\Sigma$ for the largest sub-sequence s' of s such that $s'_i \in \Sigma$ for $0 < i \leq |s|$; e.g, if $s = AABC$ then $s|_{A,C} = AAC$. We lift projection to sets of sequences point-wise.

Definition 24 Let P, Q be DCR* processes. We say that Q is a *refinement* of P iff $\mathsf{lang}(Q)|_{\mathsf{alph}(P)} \subseteq \mathsf{lang}(P)$.

In practice, we will obtain such refinements R by merging P with a marking compatible process Q.

Definition 25 Let P, Q be marking compatible DCR* processes. We say that Q *refines* P iff $P \oplus Q$ is a refinement of P.

Note that this concept is *not* symmetric: Even though $P \oplus Q = Q \oplus P$, it may still be the case that $P \oplus Q$ is a refinement of P but not of Q.

Example 26 Continuing the above example, we now see a distinction between merging with R_1 and merging with R_2: the former refines P, whereas the latter does not refine P_1. To see that R_1 refines P, observe that P_2 is in a sense *less* accepting than P (because of the potential restlessness of the new event audit). To see that R_2 does not refine P_1, observe that $P_1 \oplus R_2$ has the following accepting execution:

$$P_1 \oplus R_2 \xrightarrow{\mathsf{audit}} \xrightarrow{\mathsf{bm}} \xrightarrow{\mathsf{audit}}$$

Here audit excludes recv, and so enables bm to execute; bm in turn makes audit restless. After a second audit, we have an accepting trace $t = \langle \mathsf{audit},\ \mathsf{bm},\ \mathsf{audit} \rangle$. However, bm cannot not be the first event of a trace of P_1 because it is conditional on the non-executed recv. Formally, we found a counter-example to R_2 refining P_1:

$$\begin{aligned} t|_{\mathsf{alph}(P_1)} &= \langle \mathsf{audit},\ \mathsf{bm},\ \mathsf{audit} \rangle|_{\{\mathsf{bm},\mathsf{recv},\mathsf{approve},\mathsf{reject}\}} \\ &= \langle \mathsf{bm} \rangle \\ &\notin \mathsf{lang}(P_1) \end{aligned}$$

Inspecting the process R_2 more closely, one sees that the problem comes from the dynamic exclusion of the recv event, since not only does it make the reception of applications impossible, it also *enables* events such as bm that are conditional on recv. A better way is to block recv by introducing a new condition:

$$R_2' = [\mathsf{recv} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{audit} : (\mathsf{f},\mathsf{t},\mathsf{f})]\ !\mathsf{audit}.(\nu \mathsf{pass} : (\mathsf{f},\mathsf{t},\mathsf{f}))\ \mathsf{pass} \rightarrow\!\bullet\ \mathsf{recv}$$

Here, once audit happens, recv is barred from executing until the local event pass has happened. This process R_2' *does* refine P_1.

Unfortunately but unsurprisingly, the property of being a refinement is NP-HARD for DCR and undecidable for DCR*.

Theorem 27 *Let P, R be DCR processes. Deciding whether R refines P is* NP-HARD.

Proof By reduction from event reachability. Let (P, e) be an event reachability problem and suppose $P = [M]\ T$. Choose events $f, g \notin \mathsf{dom}(P)$ with $\ell(g) \notin \mathsf{alph}(P)$, and define DCR processes P' and R as follows

$$P' = [M, f : (\mathsf{f},\mathsf{t},\mathsf{f}), g : (\mathsf{f},\mathsf{t},\mathsf{f})]\ T \parallel f \rightarrow\!\bullet\ f \parallel f \rightarrow\!\bullet\ g \parallel \prod_{x \in \mathsf{dom}(P)} x\ \%\!\leftarrow g$$

$$R = [M(e), f : (\mathsf{f},\mathsf{t},\mathsf{t})]\ f\ \%\!\leftarrow e$$

Note that P', R can clearly be constructed from P, e in NLOGSPACE. The trick here is that g is blocked by f in P'; refining by R excludes f, removing the block. Clearly e is reachable in P iff g is in $P' \oplus R$. It is now enough to show that g is reachable in $P' \oplus R$ iff R does not refine P'. Assume first g is reachable in $P' \oplus R$. Because $\ell(g) \notin \mathsf{alph}(P)$ and g is clearly not reachable in P', R does not refine P'. Assume instead g is not reachable in $P' \oplus R$. Clearly also f not reachable in $P' \oplus R$, and so every run of $P' \oplus R$ is a run of P'; but then R refines P'. □

Proposition 28 *Let m be a Minsky machine, and take $M = [\mathsf{m}(m)]\ \mathsf{m}(t)$ to be the encoding of m as a DCR* process given in Section 5.2. Take P to be the process $P = []\ (\nu e : (\mathsf{f}, \mathsf{t}, \mathsf{f}))\ e \rightarrow\bullet\ e$, with e labelled* `halt`. Then m is terminating iff M does not refine P.

Proof Clearly $\mathsf{lang}(P) = \epsilon$, that is, the only trace of P is the empty trace. By Theorem 21, the encoding M of m has a trace exhibiting the label `halt` iff m terminates, so $\mathsf{lang}(M \oplus P)|_{\mathsf{alph}(P)}$ has a non-empty trace iff m terminates. It follows that $\mathsf{lang}(M \oplus P)|_{\mathsf{alph}(P)} \subseteq \mathsf{lang}(P)$ iff m does not terminate, and so M refines P iff m does not terminate. □

Theorem 29 *Let P, R be DCR* processes. It is undecidable whether R refines Q.*

Proof Immediate from Proposition 28. □

7 An Approximation of Refinement

Because deciding whether some R refines some P is either NP-HARD or undecidable, we need an approximation. The key problem is that new exclusions might remove conditions preventing events from firing, as we saw above, and that new inclusions might enable events to fire. This leads us to the following approximation:

Definition 30 (Non-invasiveness) Let $R = [M_R]\ T_R$ and P be processes. We say that R is *non-invasive* for P iff

1. For every context $C[-]$, such that $T_R = C[e \rightarrow\% f]$ or $T_R = C[e \rightarrow+ f]$, either f is bound in $C[-]$ or $f \notin \mathsf{fe}(P)$; and
2. For every label $l \in \mathsf{alph}(R) \cap \mathsf{alph}(P)$, no bound event of T_R is labelled l, and if $e \in \mathsf{fe}(R)$ is labelled l, then $e \in \mathsf{fe}(P)$.

It is straightforward to verify that non-invasiveness is decidable in polynomial time. In our running example, R_1 is non-invasive for P, whereas R_2 is not for P_2 (because of the exclusion of bm).

This approximation covers a large class of practically important refinements. Technically, it admits the addition of new events with labels not used

in the original process, and allow these new events to be conditional on or required by events in the existing process and vice versa. In overall terms, such processes correspond to "linking" or "mixing in" new processes at run-time.

Also note, that even though existing events can not be excluded by added events, it is possible to arbitrarily block events of the original process. Suppose we want to adapt a process P, blocking permanently its ability to execute an event e. We simply refine with the process Q:

$$Q = [e : (\mathsf{f}, \mathsf{t}, \mathsf{f})]\ (\nu g : (\mathsf{f}, \mathsf{t}, \mathsf{f}))\ g \rightarrow\!\bullet\ g \parallel g \rightarrow\!\bullet\ e$$

Here, g can never fire because it depends on itself, and so e can never fire (again) because it depends on g. Moreover, refining process Q can selectively enable and disable e by excluding and including g.

Within the application area of business process modelling, it is a common change to add the possibility/requirement of taking additional actions in between existing actions. E.g., an insurance company might face a new requirement that for certain claims, a report from an external expert must be procured before it decides on the claim. For this class of requirements, it is sufficient to have the ability to add new steps (e.g., "procure report") and block the existing workflow until these new steps are completed (e.g., prevent "decide claim" until we have executed "procure report"); refinement (using the blocking Q process above) supports these two mechanics.

We prove that non-invasiveness implies refinement. We first observe that transitions do not introduce new constraints or effects on free events.

Lemma 31 (Transitions reflect relational sub-terms) *If* $[M]\ T \xrightarrow{\lambda} T'$ *and* $T' = C'[e\ \mathcal{R}\ f]$*, then there exists a context* $C[-]$ *s.t.* $T = C[e\ \mathcal{R}\ f]$ *with* f *free in* C' *iff it is free in* C*.*

Proof Easy induction on the derivation of the transition. □

Next we prove that for processes that are composed of two processes, the marking can be canonically separated in the three disjoint parts: The events only occurring in the first process, the events that are shared, and the events only occurring in the second process.

Definition 32 (Separation of Processes) Let $P = [M]\ T_1 \parallel T_2$. A *separation* of P comprises disjoint markings M_1, M_2, S such that $M = M_1 \oplus S \oplus M_2$, that $\mathsf{fe}(T_1) \cap \mathsf{fe}(T_2) \subseteq \mathsf{dom}(S)$, and that $\mathsf{fe}(T_i) \setminus \mathsf{fe}(T_{3-i}) \subseteq \mathsf{dom}(M_i)$ for $i \in \{1, 2\}$. A process $[M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2$ is *separated* iff M_1, M_2, S is a separation.

Note that separations are not necessarily unique, e.g., if an event e is mentioned in the marking M but not free in any of the terms T_1, T_2. However, if we know where such unconstrained events belong, we have a unique separation:

Lemma 33 (Canonical Separation) *Let* $P_1 = [M_1]\ T_1$ *and* $P_2 = [M_2]\ T_2$ *with* $P_1 \oplus P_2$ *defined. Then there exists a unique separation* N_1, N_2, S *of* $P_1 \oplus P_2$ *satisfying* $\mathsf{dom}(N_i) = \mathsf{dom}(M_i) \setminus \mathsf{dom}(M_{3-i})$ *for* $i \in \{1, 2\}$ *and* $\mathsf{dom}(S) =$

$\mathsf{dom}(M_1) \cap \mathsf{dom}(M_2)$. *We call this separation the* canonical *separation of* $P_1 \oplus P_2$.

Lemma 34 *If a process* $[M]\ T_1 \parallel T_2$ *with canonical separation* $M_1 \oplus S \oplus M_2$ *has a transition*

$$[M]\ T_1 \parallel T_2 \xrightarrow{\lambda} T' \quad \textit{or} \quad [M]\ T_1 \parallel T_2 \xrightarrow{\gamma} [M']\ T'$$

then the following holds:

1. *For some* T_1', T_2' *we have* $T' = T_1' \parallel T_2'$.
2. *There exists a unique separation* M_1', M_2', S' *of* M' *with* $\mathsf{dom}(M_i) = \mathsf{dom}(M_i')$ *and* $\mathsf{dom}(S) = \mathsf{dom}(S')$.
3. *This separation satisfies* $\mathsf{alph}([M_i \oplus S]\ T_i) = \mathsf{alph}([M_i' \oplus S']\ T_i')$
4. *If the original separation was canonical for* $P_1 = [M_1 \oplus S]\ T_1$ *and* $P_2 = [S \oplus M_2]\ T_2$, *then so is* M_1', M_2', S' *for* $P_1' = [M_1' \oplus S']\ T_1'$ *and* $P_2' = [S' \oplus M_2']\ T_2'$.

Proof Note that only the rules [PAR] and [PAR-2] allows term transitions for a term on the form $T_1 \parallel T_2$; part 1 is then immediate by inspection of these rules; and part 2 and 3 follows from Lemma 12. Part 4 is then immediate from parts 2 and 3. □

We will need the following auxiliary ordering on markings with identical domains: Smaller markings have more restless events.

Definition 35 We order states $(h, i, r) \sqsubseteq (h', i', r')$ iff $h = h'$, $i = i'$ and $r' = \mathsf{t}$ implies $r = \mathsf{t}$. We order markings $M \sqsubseteq N$ point-wise when $\mathsf{dom}(M) = \mathsf{dom}(N)$.

Lemma 36 *If* $M \sqsubseteq N$ *and both* $[M]\ T$ *and* $[N]\ T$ *are processes, then:*

1. $[M]\ T \vdash e : \delta$ *iff* $[N]\ T \vdash e : \delta$;
2. $[M]\ T \xrightarrow{\lambda} T'$ *iff* $[N]\ T \xrightarrow{\lambda} T'$; *and*
3. *For every process transition* $[M]\ T \xrightarrow{\gamma} [M']\ T'$, *there exists a unique* N' *s.t.* $[N]\ T \xrightarrow{\gamma} [N']\ T'$. *This* N' *satisfies* $M' \sqsubseteq N'$.

Proof Part 1 is immediate by Definition of "$\vdash$". Part 2 then follows by induction on the derivation of the term transition, using part 1 in the base case [INTRO]. Part 3 follows by cases on the process transition rules [EFFECT] and [EFFECT-2], observing that for any $M \sqsubseteq N$ and any event or effect x, $x \cdot M \sqsubseteq x \cdot N$. □

Lemma 37 *Both term and process transitions are unique in the following sense:*

1. *If* $[M]\ T \xrightarrow{\gamma:\delta} T'$ *and* $[M]\ T \xrightarrow{\gamma:\delta'} T''$ *then* $\delta = \delta'$ *and* $T' = T''$.
2. *If* $P \xrightarrow{\gamma} Q$ *and* $P \xrightarrow{\gamma} Q'$ *then* $Q = Q'$.

Proof (1) By induction on the derivation of the transition. For the base case, [INTRO], by assumption we have

$$[M \oplus N]\, T \xrightarrow{e:\delta} T \qquad \text{and} \qquad [M' \oplus N']\, T \xrightarrow{e:\delta'} T\ ,$$

with $M \oplus N = M' \oplus N'$ and $[M]\, T \vdash e : \delta$ and $[M']\, T \vdash e : \delta'$. We now find by cases on T and inspection of the rules in Figure 4 that $M = M'$ and $\delta = \delta'$.

The cases [PAR], [PAR-2], and [REP] cases are straightforward; we exemplify with [PAR-2]. Suppose $[M]\, T \parallel U \xrightarrow{\nu e:\delta_1} T_1$ and $[M]\, T \parallel U \xrightarrow{\nu e:\delta_2} T_2$. By [PAR-2] we must have $T_1 = T_1' \parallel U$ and $T_2 = T_2' \parallel U$, and moreover $[M]\, T \xrightarrow{\nu e:\delta_1} T_1'$ and $[M]\, T \xrightarrow{\nu e:\delta_2} T_2'$. But then by IH $\delta_1 = \delta_2$ and $T_1' = T_2'$ whence $T_1 = T_2$.

Finally, [LOCAL]. Suppose

$$[M]\, (\nu f : \Phi)\, T \xrightarrow{\gamma:\delta_1} T_1 \qquad \text{and} \qquad [M]\, (\nu f : \Phi)\, T \xrightarrow{\gamma:\delta_2} T_2\ .$$

By [LOCAL] we must have

$$[M, f : \Phi]\, T \xrightarrow{e:\delta_1} T_1' \qquad \text{and} \qquad [M, f : \Phi]\, T \xrightarrow{e:\delta_2} T_2'\ ,$$

with $T_1 = (\nu f : \Phi_1)\, T_1'$ and $T_2 = (\nu f : \Phi_2)\, T_2'$. By IH $\delta_1 = \delta_2$ and $T_1' = T_2'$. It remains to prove that also $\Phi_1 = \Phi_2$. But again by [LOCAL] we have $f : \Phi_1 = (e : \delta_1) \cdot (f : \Phi) = (e : \delta_2) \cdot (f : \Phi) = f : \Phi_2$.

(2) Straightforward by inspection of the rules [EFFECT] and [EFFECT-2] using part (1) of this Lemma. □

Lemma 38 (Weakening) *Suppose* $[M \oplus N]\, T \xrightarrow{\lambda} T'$. *If* $\lambda = e : \delta$ *and* $\mathsf{fe}(T) \cup \{e\}$ *is disjoint from* $\mathsf{dom}(N)$, *or* $\lambda = \nu e : \delta$ *and* $\mathsf{fe}(T)$ *is disjoint from* $\mathsf{dom}(N)$, *then also* $[M]\, T \xrightarrow{\lambda} T'$.

Proof By induction on the derivation of the transition.

For [INTRO], note that we must have $\lambda = e : \delta$ and for some M', N' with $M' \oplus N' = M \oplus N$ that

$$[M \oplus N]\, T \xrightarrow{e:\delta} T' \qquad \text{and} \qquad [M']\, T \vdash e : \delta$$

By inspection of the rules for the enabling relation in Figure 4 we find that $\mathsf{dom}(M') \subseteq \mathsf{fe}(T) \cup \{e\}$ and so $\mathsf{dom}(M')$ disjoint from $\mathsf{dom}(N)$ and so $M = M' \oplus M''$ for some M'', whence $[M]\, T \xrightarrow{\lambda} T'$.

For [PAR] we have for some δ_1, δ_2 that $\lambda = e : \delta_1 \oplus \delta_2$ with

$$[M \oplus N]\, T_1 \xrightarrow{e:\delta_1} T_1' \qquad \text{and} \qquad [M \oplus N]\, T_2 \xrightarrow{e:\delta_2} T_2'$$

and $\mathsf{fe}(T_1 \parallel T_2) \cup \{e\}$ disjoint from $\mathsf{dom}(N)$, so also $\mathsf{fe}(T_1) \cup \{e\}$ and $\mathsf{fe}(T_2) \cup \{e\}$ disjoint from $\mathsf{dom}(N)$. By IH we find then transitions

$$[M]\, T_1 \xrightarrow{e:\delta_1} T_1' \qquad \text{and} \qquad [M]\, T_2 \xrightarrow{e:\delta_2} T_2'$$

establishing by [PAR] a transition $[M]\ T_1 \parallel T_2 \xrightarrow{e:\delta_1 \oplus \delta_2} T_1' \parallel T_2'$.

For [LOCAL] we are given a transition

$$[M \oplus N]\ (\nu f : \Phi)\ T \xrightarrow{\gamma(\delta \setminus f)} (\nu f : \Phi')\ T' \ .$$

such that for some e

$$[M \oplus N, f : \Phi]\ T \xrightarrow{e:\delta} T' \qquad \text{and} \qquad f : \Phi' = (e : \delta) \cdot f : \Phi$$

and either $e = f$ and $\gamma = \nu e$ or $\gamma = e$. In the former case, we have by assumption $\mathsf{fe}(T) = \mathsf{fe}((\nu f : \Phi)\ T)$ disjoint from $\mathsf{dom}(N)$ and by the bound variable convention we may assume $e = f$ also not in the domain of $\mathsf{dom}(N)$. Hence $\mathsf{fe}(T) \cup \{e = f\}$ also disjoint from N and by IH we have $[M, f : \Phi]\ T \xrightarrow{e:\delta} T'$ which by [LOCAL] yields the requisite transition. In the latter case, we have because $\gamma = e$ that $\mathsf{fe}(T) \cup \{e\} = \mathsf{fe}(!f.\Phi T) \cup \{e\}$ disjoint from N and again by IH we find the requisite transition.

Finally, the cases [REP] and [PAR-2] are straightforward applications of IH, noting for the former that $\mathsf{fe}(T) = \mathsf{fe}(!e.T)$ and for the latter that $\mathsf{fe}(T) \subseteq \mathsf{fe}(T \parallel U)$ and so in both cases disjointness with N is preserved as we move to the hypothesis. □

Lemma 39 *If $[M]\ T \xrightarrow{\gamma:\delta} T'$ with $\delta = (E, I, R)$ then $e \in E$ resp. $e \in I$ implies $T = C[f \to\% e]$ resp. $T = C[f \to+ e]$ with e not bound in $C[-]$.*

Proof Easy induction on the derivation of the transition. □

Lemma 40 *Let P be non-invasive for Q, and suppose M_1, S, M_2 is the canonical separation of $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2$. If also*

$$[M_1 \oplus S \oplus M_2]\ T_1 \xrightarrow{\gamma:\delta} T_1'$$

with $\delta = (X, I, R)$, then X, I are both disjoint from $\mathsf{fe}(Q)$.

Proof Immediate from the Definition of non-invasiveness and Lemma 39. □

Lemma 41 *Let P be non-invasive for Q, and suppose M_1, S, M_2 is the canonical separation of $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2$. If also*

$$[M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2 \xrightarrow{\gamma:\delta} T_1' \parallel T_2'$$

then the following are true.

1. *If $\ell(\gamma) \in \mathsf{alph}(Q)$ then for some δ' we have $[S \oplus M_2]\ T_2 \xrightarrow{\gamma:\delta'} T_2'$ and $(\gamma : \delta) \cdot (S \oplus M_2) \sqsubseteq (\gamma : \delta') \cdot (S \oplus M_2)$.*
2. *If $\ell(\gamma) \notin \mathsf{alph}(Q)$ then $(\gamma : \delta) \cdot (S \oplus M_2) \sqsubseteq S \oplus M_2$.*

Proof We proceed by cases on γ; suppose first $\gamma = \nu e$. If νe is a binder of T_2, we must have $\ell(\gamma) \in \mathsf{alph}(Q)$ and the transition must arise by (the rule symmetric to) [PAR-2]. By definition of canonical separation we have $\mathsf{fe}(T_2)$ disjoint from $\mathsf{dom}(M_1)$ and so by Lemma 38 we find a transition $[S \oplus M_2]\ T_2 \xrightarrow{\nu e:\delta} T_2'$, altogether establishing (1). If instead νe is a binder of T_1, we must have $\ell(\gamma) \notin \mathsf{alph}(Q)$ lest non-invasiveness be contradicted. In that case we must have a transition

$$[M_1 \oplus S \oplus M_2]\ T_1 \xrightarrow{\nu e:\delta} T_1'$$

by Lemma 40 we find $(\gamma : \delta) \cdot (S \oplus M_2) \sqsubseteq S \oplus M_2$.

Suppose instead $\gamma = e$. In this case the transition must be derived by [PAR], and so by Lemma 37 there exists unique δ_1, δ_2 such that

$$[M_1 \oplus S \oplus M_2]\ T_1 \xrightarrow{e:\delta_1} T_1' \qquad \text{and} \qquad [M_1 \oplus S \oplus M_2]\ T_2 \xrightarrow{e:\delta_2} T_2'$$

Suppose for (1) that $\ell(e) \in \mathsf{alph}(Q)$. By non-invasiveness and canonicity of separation we then have that $e \notin \mathsf{dom}(M_1)$ and that $\mathsf{fe}(T_2)$ is disjoint from $\mathsf{dom}(M_1)$, and so by Lemma 38 we have a transition

$$[S \oplus M_2]\ T_2 \xrightarrow{e:\delta_2} T_2'$$

By Lemma 40 it now follows that $(e : \delta_1 \oplus \delta_2) \cdot (S \oplus M_2) \sqsubseteq (e : \delta_2) \cdot (S \oplus M_2)$. Suppose instead for (2) that $\ell(e) \notin \mathsf{alph}(Q)$. It follows that $e \notin \mathsf{fe}(T_2)$, and so $\delta_2 = (\emptyset, \emptyset, \emptyset)$. We now find $(\gamma : \delta) \cdot (S \oplus M_2) \sqsubseteq S \oplus M_2$ by Lemmas 38 and 40. □

Lemma 42 *Let P be non-invasive for Q, and suppose M_1, S, M_2 is the canonical separation for $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2$. If also*

$$[M_1 \oplus S \oplus M_2]\ T_1 \parallel T_2 \xrightarrow{\gamma} [M_1' \oplus S' \oplus M_2']\ T_1' \parallel T_2' = R$$

where the latter is a canonical separation then the following statements are true:

1. *If $\ell(\gamma) \in \mathsf{alph}(Q)$ then $[S \oplus M_2]\ T_2 \xrightarrow{\gamma} [N]\ T_2'$ where $S' \oplus M_2' \sqsubseteq N$.*
2. *If $\ell(\gamma) \notin \mathsf{alph}(Q)$ then $S' \oplus M_2' \sqsubseteq S \oplus M_2$.*

Proof Immediate from Lemma 41 and rules [EFFECT] and [EFFECT-2]. □

We are now finally ready to show that the syntactically decidable non-invasiveness property implies refinement.

Theorem 43 *If Q is non-invasive for P then Q refines P.*

Proof Let $M = M_1 \oplus S \oplus M_2$ be the canonical separation of $P \oplus Q$, and consider a finite or infinite run of $R_0 = P \oplus Q = [M]\ T_1 \parallel T_2$:

$$R_0 \xrightarrow{\gamma_0} R_1 \xrightarrow{\gamma_1} \ldots$$

By induction on i using Lemma 34, we can write each R_i as a canonically separated process

$$R_i = [M_1^i \oplus S^i \oplus M_2^i]\ T_1^i \parallel T_2^i = ([M_1^i \oplus S^i]\ T_1^i) \oplus ([M_2^i \oplus S^i]\ T_2^i)$$

where $\mathsf{alph}([M_1^i \oplus S^i]\ T_1^i) = \mathsf{alph}(P)$ and $\mathsf{alph}([S^i \oplus M_2^i]\ T_2^i) = \mathsf{alph}(Q)$, and $[M_2^i]\ T_2^i$ is non-invasive for $[M_1^i]\ T_1^i$. We prove by induction that there exists a sequence N^i satisfying (a) $N^0 = S^0 \oplus M_2^0$, (b) $S^i \oplus M_2^i \sqsubseteq N^i$, and (c)

$$N^i = N^{i+1} \qquad \text{when } \ell(\gamma_i) \notin \mathsf{alph}(P)$$
$$[N^i]\ T_1^i \xrightarrow{\gamma_i} [N^{i+1}]\ T_1^{i+1} \qquad \text{when } \ell(\gamma_i) \in \mathsf{alph}(P)$$

The theorem then follows. We have immediately $N^0 = S^0$, obtaining (a). For (b) and (c), consider some $i > 0$, and assume first $\ell(\gamma_i) \notin \mathsf{alph}(P)$. By Lemma 42, Part 2, we then have $S^{i+1} \oplus M_2^{i+1} \sqsubseteq S^i \oplus M_2^i \sqsubseteq N^i = N^{i+1}$, obtaining (b) and (c). Assume instead $\ell(\gamma_i) \in \mathsf{alph}(P)$. Then take $N^{i+1} = N$ where N is given by Lemma 42, Part 1, immediately obtaining (b) and (c). □

8 Conclusion, Related and Future Work

In this paper, we studied the interplay of dynamic process instantiation and run-time refinement in the context of the declarative event-based process language, Dynamic Condition Response graph. We formalised the language here as a process language. The present work generalises our prior work on DCR graphs, co-developed and implemented by our industrial partner.

We proved that event-reachability for DCR processes without replication is NP-HARD, while the combination of replication and local events makes event-reachability and refinement undecidable. We then identified a decidable and practically useful class of refinements, characterised by the syntactic notion of non-invasive processes. Our findings and problems were illustrated by a running example extracted from a real case.

Related Work. The DCR language is as we have seen closely related to DCR graphs [36, 47, 21], which descend from event structures, and thus have relations to Petri Nets. Petri Nets have been extended to allow modular definition (e.g. via shared transitions [32]) and to represent infinite computations and ω-regular languages (e.g., Büchi Nets [17]). However, Petri nets introduce the intentional construct of *places* marked with *tokens*, as opposed to event structures and DCR* processes, which only rely on causal and conflict relations between events.

Variants of Event structures with asymmetric conflict relation relate to the asymmetric exclude relation of DCR processes, including extended bundle event structures [30, 20], dual event structures [28, 31], asymmetric event structures [7], and precursor event structures [18]. The include relation of DCR processes relate to event structures for resolvable conflicts [19], and the inclusion and exclusion together allow representation of a dynamic causality relation as found in dynamic causality event structures [6]. Automata based models

like Event automata [41] and local event structures [25] also allow asymmetric conflicts, but use explicit states and do not express causality and conflicts as relations between events. Besides the early work on restless events in [48], we are not aware of other published work generalising Event structures to be able to express liveness properties, nor to distinguish between events that *may* and events that *must eventually be executed*. Replication events of the DCR* process language relate to replication in process calculi and higher-order Petri nets [27]. We believe to be the first to combine higher-order features and liveness.

Refinement in the shape of run-time adaptation has been studied also for Petri nets [45] and process calculi [5, 9, 10], but tends to require predefined adaptation points, and often deal with adaptations via higher-order primitives. In contrast, refinement in DCR* is dealt with by composition, which due to the declarative nature allow for cross-cutting refinements without the need for pre-specified adaptation points.

In the BPM community, the seminal declarative process language is Declare [2, 4]. As Declare is based on mapping primitives to LTL, which are then mapped to automata, it necessarily distinguishes between run-time and design-time. In contrast, in DCR processes, design-time and run-time representation is literally the same. Declare has a relatively large set of basic constraints, the formal expressiveness of which is clearly limited by that of LTL, while DCR processes with only 4 basic constraints offers the full expressiveness of regular and ω-regular languages. A different approach is [35], which provides a mapping from Declare to the CLIMB, which allows the use of its reasoning techniques for support and verification of Declare processes at both design- and run-time. The Guard-Stage-Milestone (GSM) approach [26] developed at IBM Research is the main inspiration for the emerging Case Management Model and Notation (CMMN) standard [39] and offers a data-centric declarative business process modelling notation. The notation consists of stages, which have guards controlling when the stage may start, and milestones controlling when and how a stage may close. Initial work on relating GSM and DCR graphs is provided in [16] where it is shown how to derive consistent GSM Schemas from DCR graphs.

Imperative process models such as BPMN [38] have supported dynamic sub-processes for some time now, they are only recently being studied for declarative languages [50]. Here, sub-processes do not have independent life cycles, that is, when a sub-process is spawned, it must run to completion before its super-process may resume. Interestingly, it is noted in *ibid.* that extending the model with sub-processes seems to increase its expressive power; we formally confirm that supposition here, finding DCR processes with replication and sub-processes able to encode Minsky machines.

Future work. DCR* processes as defined here only interact via shared events. We are currently working on adding interaction between concurrent events, labelled with send and receive labels as found e.g. in the π-calculus, thereby lifting the results of the present paper to π-like languages. Towards better

analysis of the infinite-state DCR* language, we have initiated work on exploiting the idea of responses and restless events in the domain of behavioural types [14] and run-time monitoring [36].

The DCR* process language would benefit from a closer investigation of its relation to modular [32] and higher-order Petri Nets [27]. Moreover, time constraints and more general refinements as initiated in [24, 37], e.g. allowing to remove constraints and events should be further investigated.

Finally, the present notion of refinement was not reified into an actual semantics of run-time adaptation. We have defined refinement and found a syntactic condition ensuring it, but we have neither embedded "adaptation" as a primitive of the language nor included it in the core semantics—we have not given a computational model of run-time refinement. Such an extension would be interesting and important; one avenue would be to follow the ideas of [29] and include "adaptation steps" as higher-order actions, obtaining a semantics that include outright the notion of run-time refinement.

References

1. van der Aalst WMP (1998) The application of Petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1):21–66
2. van der Aalst WMP, Pesic M (2006) DecSerFlow: Towards a truly declarative service flow language. In: WS-FM 2006, Springer, LNCS, vol 4184, pp 1–23
3. van der Aalst WMP, ter Hofstede AHM, Weske M (2003) Business process management: A survey. In: van der Aalst WMP, ter Hofstede AHM, Weske M (eds) Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings, Springer, Lecture Notes in Computer Science, vol 2678, pp 1–12
4. van der Aalst WMP, Pesic M, Schonenberg H, Westergaard M, Maggi FM (2010) Declare. Webpage, `http://www.win.tue.nl/declare/`
5. Anderson G, Rathke J (2012) Dynamic software update for message passing programs. In: Jhala R, Igarashi A (eds) APLAS, Springer, Lecture Notes in Computer Science, vol 7705, pp 207–222
6. Arbach Y, Karcher D, Peters K, Nestmann U (2015) Dynamic causality in event structures. In: Graf S, Viswanathan M (eds) Formal Techniques for Distributed Objects, Components, and Systems: 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings, Springer International Publishing, Cham, pp 83–97, DOI 10.1007/978-3-319-19195-9_6, URL `http://dx.doi.org/10.1007/978-3-319-19195-9_6`
7. Baldan P, Corradini A, Montanari U (2001) Contextual Petri nets, asymmetric Event structures, and processes. Information and Computation 171:149, DOI 10.1006/inco.2001.3060

8. Barthe G, Pardo A, Schneider G (eds) (2011) Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings, LNCS, vol 7041, Springer
9. Bravetti M, Di Giusto C, Pérez JA, Zavattaro G (2012) Steps on the road to component evolvability. In: Proceedings of the 7th International Conference on Formal Aspects of Component Software, FACS'10, pp 295–299, DOI 10.1007/978-3-642-27269-1_19, URL http://dx.doi.org/10.1007/978-3-642-27269-1_19
10. Bravetti M, Giusto CD, Pérez JA, Zavattaro G (2012) Adaptable processes. Logical Methods in Computer Science 8(4)
11. Carbone M, Hildebrandt TT, Perrone G, Wasowski A (2012) Refinement for transition systems with responses. In: FIT, EPTCS, vol 87, pp 48–55
12. Debois S, Hildebrandt T, Marquard M, Slaats T (2014) A case for declarative process modelling: Agile development of a grant application system. In: EDOCW/AdaptiveCM '14, IEEE, pp 126 – 133, DOI 10.1109/EDOCW.2014.27
13. Debois S, Hildebrandt TT, Slaats T (2014) Hierarchical declarative modelling with refinement and sub-processes. In: Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings, Springer, Lecture Notes in Computer Science, vol 8659, pp 18–33, DOI 10.1007/978-3-319-10172-9, URL http://dx.doi.org/10.1007/978-3-319-10172-9
14. Debois S, Hildebrandt TT, Slaats T, Yoshida N (2014) Type checking liveness for collaborative processes with bounded and unbounded recursion. In: FORTE, Springer, Lecture Notes in Computer Science, vol 8461, pp 1–16
15. Debois S, Hildebrandt T, Slaats T (2015) Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In: FM 2015, Springer, no. 9109 in LNCS, pp 143–160, DOI 10.1007/978-3-319-19249-9_10
16. Eshuis R, Debois S, Slaats T, Hildebrandt TT (2016) Deriving consistent GSM schemas from DCR graphs. In: Sheng QZ, Stroulia E, Tata S, Bhiri S (eds) Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings, Springer, Lecture Notes in Computer Science, vol 9936, pp 467–482, DOI 10.1007/978-3-319-46295-0_29, URL http://dx.doi.org/10.1007/978-3-319-46295-0_29
17. Esparza J, Melzer S (1997) Model checking LTL using constraint programming. In: Azma P, Balbo G (eds) Application and Theory of Petri Nets 1997, Lecture Notes in Computer Science, vol 1248, Springer Berlin Heidelberg, pp 1–20
18. Fecher H, Majster-Cederbaum M (2005) Event structures for arbitrary disruption. Fundam Inf 68(1-2):103–130
19. van Glabbeek R, Plotkin G (2004) Event structures for resolvable conflict. In: Fiala J, Koubek V, Kratochvíl J (eds) Mathematical Foundations of Computer Science 2004: 29th International Sympo-

sium, MFCS 2004, Prague, Czech Republic, August 22-27, 2004. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 550–561, DOI 10.1007/978-3-540-28629-5_42, URL http://dx.doi.org/10.1007/978-3-540-28629-5_42

20. van Glabbeek R, Vaandrager F (2003) Bundle event structures and CCSP. In: CONCUR 2003 - Concurrency Theory, LNCS, vol 2761, Springer, pp 57–71
21. Hildebrandt TT, Mukkamala RR (2010) Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES, EPTCS, vol 69, pp 59–73
22. Hildebrandt TT, Mukkamala RR, Slaats T (2011) Nested dynamic condition response graphs. In: FSEN, Springer, LNCS, vol 7141, pp 343–350
23. Hildebrandt TT, Marquard M, Mukkamala RR, Slaats T (2013) Dynamic condition response graphs for trustworthy adaptive case management. In: OTM Workshops, Springer, LNCS, vol 8186, pp 166–171
24. Hildebrandt TT, Mukkamala RR, Slaats T, Zanitti F (2013) Contracts for cross-organizational workflows as timed dynamic condition response graphs. J Log Algebr Program 82(5-7):164–185
25. Hoogers P, Kleijn H, Thiagarajan P (1996) An event structure semantics for general Petri nets. Theoretical Computer Science 153(12):129 – 170
26. Hull R, Damaggio E, Fournier F, Gupta M, Heath FT, Hobson S, Linehan MH, Maradugu S, Nigam A, Sukaviriya P, Vaculín R (2010) Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: WS-FM, Springer, LNCS, vol 6551, pp 1–24
27. Janneck JW, Esser R (2002) Higher-order Petri net modelling: Techniques and applications. In: Proceedings of the Conference on Application and Theory of Petri Nets: Formal Methods in Software Engineering and Defence Systems, CRPIT '02, pp 17–25
28. Katoen JP (1996) Quantitative and qualitative extensions of event structures. PhD thesis, University of Twente, Enschede
29. Lanese I, Lienhardt M, Mezzina CA, Schmitt A, Stefani J (2013) Concurrent flexible reversibility. In: Felleisen M, Gardner P (eds) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Springer, Lecture Notes in Computer Science, vol 7792, pp 370–390, DOI 10.1007/978-3-642-37036-6_21, URL http://dx.doi.org/10.1007/978-3-642-37036-6_21
30. Langerak R (1992) Transformations and Semantics for LOTOS. Universiteit Twente
31. Langerak R, Brinksma E, Katoen JP (1997) Causal ambiguity and partial orders in event structures. In: CONCUR '97, LNCS, vol 1243, Springer, pp 317–331, DOI 10.1007/3-540-63141-0_22
32. Latvala T, Mkel M (2004) LTL model checking for modular Petri nets. In: Applications and Theory of Petri Nets 2004, LNCS, vol 3099, Springer, pp 298–311

33. Marquard M, Shahzad M, Slaats T (2015) Web-based modelling and collaborative simulation of declarative processes. In: Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings, Springer, Lecture Notes in Computer Science, vol 9253, pp 209–225, DOI 10.1007/978-3-319-23063-4_15, URL http://dx.doi.org/10.1007/978-3-319-23063-4_15
34. Minsky ML (1967) Computation: Finite and Infinite Machines. Prentice-Hall
35. Montali M (2010) Specification and Verification of Declarative Open Interaction Models - A Logic-Based Approach, Lecture Notes in Business Information Processing, vol 56. Springer
36. Mukkamala RR (2012) A formal model for declarative workflows: Dynamic condition response graphs. PhD thesis, IT University of Copenhagen
37. Mukkamala RR, Hildebrandt T, Slaats T (2013) Towards trustworthy adaptive case management with dynamic condition response graphs. In: EDOC, IEEE, pp 127–136
38. Object Management Group BPMN Technical Committee (2013) Business Process Model and Notation, version 2.0. http://www.omg.org/spec/BPMN/2.0.2/PDF
39. Object Management Group CMMN Technical Committee (2016) Case Management Model and Notation, version 1.1. http://www.omg.org/spec/CMMN/1.1/PDF
40. Park D (1981) Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on Theoretical Computer Science, Springer-Verlag, London, UK, UK, pp 167–183, URL http://dl.acm.org/citation.cfm?id=647210.720030
41. Pinna G, Poigné A (1995) On the nature of events: another perspective in concurrency. Theoretical Computer Science 138(2):425–454, DOI http://dx.doi.org/10.1016/0304-3975(94)00174-H, meeting on the mathematical foundation of programing semantics
42. Preda MD, Gabbrielli M, Giallorenzo S, Lanese I, Mauro J (2015) Developing correct, distributed, adaptive software. Sci Comput Program 97:41–46, DOI 10.1016/j.scico.2013.11.019, URL http://dx.doi.org/10.1016/j.scico.2013.11.019
43. Reichert M, Weber B (2012) Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer
44. Rohloff K, Loyall J, Pal P, Schantz R (2007) High-assurance distributed, adaptive software for dynamic systems. In: 10th IEEE High Assurance Systems Engineering Symposium (HASE '07), pp 385 – 386, DOI 10.1109/HASE.2007.17
45. Sibertin-Blanc C, Mauran P, Padiou G (2007) Safe Adaptation of Component Coordination. Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities 189:69–85
46. Slaats T (2015) Flexible process notations for cross-organizational case management systems. PhD thesis, IT University of Copenhagen

47. Slaats T, Mukkamala RR, Hildebrandt TT, Marquard M (2013) Exformatics declarative case management workflows as DCR graphs. In: BPM, Springer, LNCS, vol 8094, pp 339–354
48. Winskel G (1980) Events in computation. PhD thesis, University of Edinburgh
49. Winskel G (1986) Event structures. In: Advances in Petri Nets, Springer, LNCS, vol 255, pp 325–392
50. Zugal S, Soffer P, Pinggera J, Weber B (2012) Expressiveness and understandability considerations of hierarchy in declarative business process models. In: BMMDS/EMMSAD, Springer, Lecture Notes in Business Information Processing, vol 113, pp 167–181

A Proof of Proposition 6

We recap the original graph-based formalisation of DCR graph [23, 21, 36, 46, 13].

Definition 44 (DCR Graph) A *DCR graph* is a tuple $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ where

- E is a finite set of (labelled) *events*, the nodes of the graph.
- R is the edges of the graph. Edges are partitioned into four kinds, named and drawn as follows: The *conditions* ($\rightarrow\bullet$), *responses* ($\bullet\rightarrow$), *inclusions* ($\rightarrow+$), and *exclusions* ($\rightarrow\%$).
- M is the *marking* of the graph. This is a triple $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ of sets of events, respectively the previously executed (Ex), the currently pending (Re), and the currently included (In) events.

When G is a DCR graph, we write, e.g., $\mathsf{E}(G)$ for the set of events of G, $\mathsf{Ex}(G)$ for the executed events in the marking of G, etc. We write $(\rightarrow\bullet\, e)$ for the set $\{e' \in \mathsf{E} \mid e' \rightarrow\bullet\ e\}$, write $(e\bullet\rightarrow)$ for the set $\{e' \in \mathsf{E} \mid e \bullet\rightarrow e'\}$ and similarly for $(e\rightarrow+)$ and $(e\rightarrow\%)$.

Definition 45 (Enabled events) Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ be a DCR graph, with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. An event $e \in \mathsf{E}$ is *enabled*, written $e \in \mathsf{enabled}(G)$, iff (a) $e \in \mathsf{In}$ and (b) $\mathsf{In} \cap (\rightarrow\bullet\, e) \subseteq \mathsf{Ex}$.

Definition 46 (Execution) Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ be a DCR graph with marking $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$. Suppose $e \in \mathsf{enabled}(G)$. We may *execute* e obtaining the *resulting* DCR graph $(\mathsf{E}, \mathsf{R}, \mathsf{M}')$ with $\mathsf{M}' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}')$ defined as follows.

1. $\mathsf{Ex}' = \mathsf{Ex} \cup \{e\}$
2. $\mathsf{Re}' = (\mathsf{Re} \setminus \{e\}) \cup (e\bullet\rightarrow)$
3. $\mathsf{In}' = (\mathsf{In} \setminus (e\rightarrow\%)) \cup (e\rightarrow+)$

Definition 47 (Transitions) Let G be a DCR graph. If $e \in \mathsf{enabled}(G)$ and executing e in G yields H, we say that G has *transition on e to H* and write $G \xrightarrow{e} H$. A *run* of G is a (finite or infinite) sequence of DCR graphs G_i and events e_i such that: $G = G_0 \xrightarrow{e_0} G_1 \xrightarrow{e_1} \ldots$. A *trace* of G is a sequence of labels of events e_i associated with a run of G. We write $\mathsf{runs}(G)$ and $\mathsf{traces}(G)$ for the set of runs and traces of G, respectively

Definition 48 (Acceptance) A run $G_0 \xrightarrow{e_0} G_1 \xrightarrow{e_1} \ldots$ is *accepting* iff for all n with $e \in \mathsf{In}(G_n) \cap \mathsf{Re}(G_n)$ there exists $m \geq n$ s.t. either $e_m = e$, or $e \notin \mathsf{In}(G_m)$. A *trace is accepting* iff it has an underlying run which is.

Definition 49 (Language) The *language* of a DCR graph G is the set of its accepting traces. We write $\mathsf{lang}(G)$ for the language of G.

This concludes our recap of DCR graphs (as opposed to DCR processes). Now, the proof.

Definition 50 Let T, U be terms. Define $T \cong U$ by taking for $\mathcal{R}$ ranging over the four relations

$$e \;\mathcal{R}\; f \parallel e \;\mathcal{R}\; f \cong e \;\mathcal{R}\; f \tag{8}$$

and closing under monoid laws for $- \parallel -$ and 0.

Lemma 51 *If $T \cong U$ then $T \xrightarrow{e} T'$ and $U \xrightarrow{e} U'$ implies $T' \cong U'$.*

Proof It is straightforward to verify that the monoid laws and the idempotency rule (8) defining $\cong$ preserves transition in the above sense; the desiderata follows.

Definition 52 Observe that by the monoid laws, every term T can be read as a multi-set of relations; by the idempotency rule (8), it can be read as a set of relations. Write $\bar{T}$ for this set. For a process $P = [M]\ T$, take $\bar{P} = [M]\ \bar{T}$.

Definition 53 Let $P = [M]\ T$ be a process and $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ a graph. Define $P \approx G$ iff

1. $\mathsf{E} = \mathsf{dom}(M)$
2. for all $e \in \mathsf{E}$ we have $M(e) = (h, i, r)$ iff $h = (e \in \mathsf{Ex})$ and $i = (e \in \mathsf{In})$ and $r = (e \in \mathsf{Re})$.
3. $\bar{T} = \mathsf{R}$.

Lemma 54 *Let P be a process and G a graph s.t. $P \approx G$. For all e, if $P \xrightarrow{e} P'$ then for some G' we have $G \xrightarrow{e} G'$ with $P' \approx G'$; and vice versa.*

Proof By Lemma 51 it is sufficient to prove this result for $\bar{P}$. Assume $\bar{P} = [M]\ \bar{T}$ and $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$. We proceed by induction on the number of relations k in $\bar{T} = \mathsf{R}$.

For $k = 0$, clearly $P' = [e \cdot M]\ T$ and $G' = (\mathsf{Ex} \cup \{e\}, \mathsf{In}, \mathsf{Re})$; the result follows.

For $k = 1$, straightforward verification by cases on the single relation.

For $k > 1$, we must have $T = \bar{T}_1 \parallel \bar{T}_2$ and $\mathsf{R} = \bar{T}_1 \cup \bar{T}_2$ such that the number of relations in $\bar{T}_j$ are both smaller than k. Define $G_j = (\mathsf{E}, \bar{T}_j, \mathsf{M})$. Note that if $P \xrightarrow{e} P'$ then we must have

$$\frac{[M]\ \bar{T}_1 \xrightarrow{e:\delta_1} \bar{T}_1 \qquad [M]\ \bar{T}_2 \xrightarrow{e:\delta_2} \bar{T}_2}{[M]\ \bar{T}_1 \parallel \bar{T}_2 \xrightarrow{e:\delta_1 \oplus \delta_2} \bar{T}_1 \parallel \bar{T}_2} \tag{9}$$

In this case we have by induction that $G_j = (\mathsf{E}, \bar{T}_j, \mathsf{M}) \xrightarrow{e} (\mathsf{E}, \bar{T}_j, \mathsf{M}'_j) = G'_j$ for $j \in \{1, 2\}$.

We show first that e is enabled in P iff it is in Q. Suppose e is enabled in P. By (9), e is then enabled in $[M]\ T_1$ and $[M]\ T_2$. By induction, it is then enabled in G_1, G_2, and by calculation on Definition 45 it is also enabled in G. Now suppose instead e is enabled in G. By Definition 45, e is also enabled in G_1, G_2. By induction, e is enabled in $[M]\ \bar{T}_1$ and $[M]\ \bar{T}_2$. It follows by the [PAR]-rule that e is enabled in P.

Now suppose we have transitions $P \xrightarrow{e} P'$ and $G \xrightarrow{e} G'$. As noted we must have (9); by [EFFECT] we have transitions

$$[M]\ \bar{T}_1 \xrightarrow{e} [e : \delta \cdot M]\ \bar{T}_1 \qquad \text{and} \qquad [M]\ \bar{T}_2 \xrightarrow{e} [e : \delta \cdot M]\ \bar{T}_2 \tag{10}$$

By induction we must have transitions

$$G_1 \xrightarrow{e} G'_1 \qquad \text{and} \qquad G_2 \xrightarrow{e} G'_2 \tag{11}$$

We prove that $P' \approx G'$. Items (1) and (3) of Definition 53 are immediate, so it is sufficient to show Item (2). Let $M_h(x) = h$ when $M(x) = (h, i, r)$; similarly for i, r. It is straightforward to verify using (10) and (11) that each of the following leads to a contradiction:

1. $M_h(f) \neq (f \in \mathsf{Ex}')$, or
2. $M_i(f) \neq (f \in \mathsf{In}')$, or

3. $M_r(f) \neq (f \in \mathsf{Re}')$.

Theorem 55 *The relation $\approx$ is a bisimulation.*

Proof Immediate from Lemma 54.

Corollary 56 *Let P be a process and G a graph. Then $P \approx G$ implies $\mathsf{lang}(P) = \mathsf{lang}(G)$.*

Proof Using Lemma 54, it is clear that a run of P gives rise to a run of G and vice versa. It is sufficient to show that such a runs is accepting for P iff it is for G; this is immediate by inspection of Definitions 2 and 48.

Proof (of Proposition 6) Immediate from Theorem 55 and Corollary 56.