Safety, Liveness and Run-time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes(Full version)*

Søren Debois[1], Thomas Hildebrandt[1], and Tijs Slaats[1,2]

[1] IT University of Copenhagen
debois,hilde,tslaats@itu.dk
[2] Exformatics A/S

Abstract. We study modularity, run-time adaptation and refinement under safety and liveness constraints in event-based process models with dynamic sub-process instantiation. The study is part of a larger programme to provide semantically well-founded technologies for modelling, implementation and verification of flexible, run-time adaptable process-aware information systems, moved into practice via the Dynamic Condition Response (DCR) Graphs notation co-developed with our industrial partner. Our key contributions are: (1) A formal theory of dynamic sub-process instantiation for declarative, event-based processes under safety and liveness constraints, given as the DCR* process language, equipped with a compositional operational semantics and conservatively extending the DCR Graphs notation; (2) an expressiveness analysis revealing that the DCR* process language is Turing-complete, while the fragment corresponding to DCR Graphs (without dynamic sub-process instantiation) characterises exactly the languages that are the union of a regular and an omega-regular language; (3) a formalisation of run-time refinement and adaptation by composition for DCR* processes and a proof that such refinement is undecidable in general; and finally (4) a decidable and practically useful sub-class of run-time refinements. Our results are illustrated by a running example inspired by a recent Electronic Case Management solution based on DCR Graphs and delivered by our industrial partner. An online prototype implementation of the DCR* language (including examples from the paper) and its visualisation as DCR Graphs can be found at http://tiger.itu.dk:8020/.

1 Introduction

Many software systems today control critical and increasingly complex long-running processes, often operating in unpredictable contexts. This is particularly the case for *Process-aware Information Systems* (PAIS) [33] and *Business Process Management Systems* (BPMS) [2], which constitute the practical context of the present work. The research in these fields deals with studying systems

* supported by the Velux foundation (grant 33295) and Innovation Fund Denmark.

driven by explicit process designs for the enactment and management of business processes and human workflows, and the study of formalisms for describing the process designs has always played a central role. Particularly popular models tend to specify explicit sequencing of business activities as flow graphs, e.g., Petri Nets and Workflow Nets [1], which are the closest formal counterpart to the industrial standard Business Process Model and Notation (BPMN) [31].

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 longer running processes, such as the management of mortgages of a credit institution, such changes need to be reflected in *running* processes. Moreover, while the flow graph may be initially verified to be compliant with the given business rules and legal constraints, only some of the rules are explicitly represented in decision points, others are implemented implicitly in the sequencing of actions. Thus, it is typically difficult to determine how a flow graph should be changed if some of the business rules or legal constraints not explicitly represented in decision points are changed.

Declarative process languages [3,17] 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 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. If the constraints are compiled to e.g. an automaton before execution (as in e.g. [3]), adaptations will only take effect on new instances of the process and not the running processes. However, run-time adaptations become a possibility if the constraints are interpreted at run-time. This is the case for the *Dynamic Condition Response (DCR) Graphs* notation, introduced in [17,28] and further co-developed with our industrial partner Exformatics in [18,7,29,19,12].

As we shall see, DCR Graphs represent any behaviour that can be described as the union of a regular and an ω-regular language. Conversely, it has been shown that a DCR Graph can be mapped to a Büchi-automaton, and so DCR Graphs can be analysed by standard automata-based model-checking techniques.

However, a workflow process may involve dynamic creation of an *a priori* unbounded number of new (sub) processes at run-time, as captured by the workflow patterns for creation of multiple instances in [34]. While it is of course possible to spawn new processes at run-time in any sensible electronic case management system, the compound behaviour of old and new processes is not explicitly represented by the formal model, and thus eludes formal analysis. Hence the central

motivation for the present paper: We need to formally understand the dynamic creation of sub-processes, and we need to understand and control its interaction with run-time adaptation.

Tentative steps towards such an understanding were taken in [12], where we presented an extension of DCR Graphs to so-called *hierarchical DCR Graphs*, supporting dynamic creation of sub-processes. However, the graphical representation and formalisation of DCR Graphs is hard to manage and reason about for complex hierarchical processes composed of many parts, in particular when the different parts are dynamically created. Also, the expressive power of hierarchical DCR Graphs was left open in [12], as were the computational complexity of their refinements.

In the present paper, we contribute the following:

1. a formal theory of dynamic sub-process instantiation in declarative process models as a conservative extension of DCR Graphs,
2. an expressiveness analysis of the formal theory, revealing that dynamic sub-processes makes it Turing complete,
3. a notion of run-time adaptation by composition and a notion of refinement,
4. a proof that refinement is in general undecidable for processes with dynamic sub-processes
5. a practically useful and decidable sub class of run-time refinements defined as non-invasive adaptations

We illustrate our findings with a running example: a grant application process of a funding agency, which was recently implemented by our industry partner Exformatics in a DCR Graph-based commercial solution [10].

Overview of the paper: In Sec. 2 we present the DCR process language corresponding to the DCR Graphs notation and state its expressiveness, corresponding exactly to languages being the union of regular and ω-regular languages. We then extend the DCR language in Sec. 3 to DCR*, supporting dynamic creation of sub-processes with fresh (local) events and prove that DCR* is Turing complete. We address run-time adaptation by composition and refinement in Sec. 4, proving undecidability of refinement in general for DCR* and providing a practically useful, decidable sub-class of refinements referred to as non-invasive adaptations. Finally, in Sec. 5, we discuss related work and conclude. For want of space, most proofs and some examples have been relegated to the full version of this paper [11]. An online prototype implementation of the process language (and all examples of the paper), with a mapping to DCR Graphs, can be found at `http://tiger.itu.dk:8020/`.

2 Dynamic Condition Response (DCR) Processes

We now introduce the Dynamic Condition Response (DCR) process language. We shall see later that this language corresponds to the DCR Graph model [28,17]. Assume fixed universes of *events* $\mathcal{E}$ and *labels* $\mathcal{L}$; each event $e \in \mathcal{E}$ has an associated label $\ell(e) \in \mathcal{L}$. [3] Labels will be used as a (finite) alphabet for defining

[3] Unless explicitly stated, in all examples the label of an event is the event.

the language recognized by a DCR process. A DCR process $[M]\ T$ comprises a *marking* M and a *term* T. The syntax of both are given in Fig. 1 below.

$$
\begin{array}{rll}
T, U ::= & f \rightarrow\bullet\ e & \text{condition} \\
| & f \leftarrow\bullet\ e & \text{response} \\
| & f +\leftarrow e & \text{inclusion} \\
| & f \,\%\leftarrow e & \text{exclusion} \\
| & T \mid U & \text{parallel} \\
| & 0 & \text{unit}
\end{array}
\qquad
\begin{array}{rll}
\phi ::= & \mathsf{t} \mid \mathsf{f} & \text{boolean value} \\
\Phi ::= & (\phi, \phi, \phi) & \text{event state} \\
M, N ::= & M, e : \Phi & \text{marking} \\
P, Q ::= & [M]\ T & \text{process}
\end{array}
$$

Fig. 1. DCR Processes Syntax.

A term is a parallel composition of *constraint and effect relations* between *events*:

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.

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 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. 3 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 grant application process implementation described in [10] involves at a high-level only four events: recv(an application is received), deadline(the current deadline for the current round has been reached), round(the application round is (re)opened for applications), and bm(a board meeting is held). Hereto come three constraints: 1) Applications can only be received after a round is opened and until the deadline has been reached. 2) After a round is opened, a board meeting must eventually be held. 3) If a round is open, and the deadline has not yet been met, a board meeting can not be held unless at least one application has been received. The events and constraints can

be modelled by the following term:

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

The first constraint is that the event deadline *excludes* the event recv, representing that applications can not be received after the deadline. The second constraint is that the event round *includes* the event recv, representing that applications can (again) be received if the round is (re)opened. The third constraint is that the event bm is a *response* to the event round, representing that a board meeting must happen eventually if the round is opened. The last constraint is that the event recv is a condition for bm, representing that, if the event recv is included, an application must have been received before the board meeting can be held. The initial state of the process is then defined by declaring that no event has happened and no event is restless (i.e. required to happen) and every event but recv is included. This is represented by the marking:

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

Example 2 (Event structures). A labelled prime event structure [38] 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

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

A state of an event structures 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. An event structure with a set $R \subseteq E$ of *restless* events as considered in [37] 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.

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 (for atomic terms, parallel will be dealt with later) in Fig. 2. It should be read: "in the marking M, the (atomic) term T allows the event e to happen with the effects of excluding events E, including events I, and making events R restless."

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

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

Fig. 2. 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 $\rightarrow\!\bullet$,$\leftarrow\!\bullet$, $+\!\leftarrow$,$\%\!\leftarrow$.

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 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, inclusion takes precedence. 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 $(e : \delta) \cdot M = \delta \cdot (e \cdot M)$.

With these mechanics in place, we give transition semantics of processes in Fig. 3 below, where the *merge of effects* $\delta_1 \oplus \delta_2$ is simply defined as the pointwise union: $(E_1, I_1, R_1) \oplus (E_2, I_2, R_2) = (E_1 \cup E_2, I_1 \cup I_2, R_1 \cup R_2)$.

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 3 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 merges the effects of transitions from the two sides of a parallel; note that markings on

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

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

Fig. 3. Basic transition semantics.

either side must be the same. The [EFFECT] rule lifts an effect transition to a process transition by applying the effect to the marking.

Process transitions gives rise to an LTS, which we equip with a notion of *acceptance* defined below a run is accepting if every restless event eventually either happens or is excluded.

Definition 3. *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* $[M_i]\,T_i$*, if whenever an event* e *is restless in* M_i*, i.e.* $M_i(e) = (\_, \_, \mathsf{t})$*, then there exists some* $j \geq i$ *s.t. either* $[M_j]\,T_j \xrightarrow{e:\delta} [M_{j+1}]\,T_{j+1}$ *or* e *is excluded in* M_j*, i.e.* $M_j(e) = (\_, \mathsf{f}, \_)$*. 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* $s_i = \ell(e_i)$*. The* language $\mathsf{lang}(P)$ *of a process* P *is the set of traces of* P*.*

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 happening	round h	round i	round r	deadline h	deadline i	deadline r	recv h	recv i	recv r	bm h	bm i	bm r	Accepts?	Enabled
(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		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 recv $\%\leftarrow$ deadline, 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 $\mathsf{bm} \leftarrow\!\bullet\ \mathsf{round}$, 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 $\langle \mathsf{round}, \mathsf{deadline}, \mathsf{bm}, \mathsf{round}, \mathsf{recv}, \mathsf{bm} \rangle$.

2.1 Expressiveness of DCR processes

In this section we will first show that the DCR process language of Section 2 characterises exactly 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 [28]).

Given a Büchi automaton $B = (Q, \Sigma, \delta, q_0, F)$ we define a corresponding term $\mathsf{t}(B)$ in Fig. 4 and marking $\mathsf{m}(B)$ in Fig. 5. We model each transition $\delta : p \xrightarrow{l} q$ 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 event (p, l, q) into *two* copies of $(p, l, q, 0)$ 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 an exclude relations. Let's see how this idea is reflected in Fig. 4. (1) Firing a transition $\delta : q \xrightarrow{l} p$ out of a *non-accepting* state q excludes all other transitions out of that state and includes all transitions out of p in the same copy i. (2) Firing a transition $\delta : q \xrightarrow{l} p$ out of an *accepting* state q excludes all other transitions out of that state, then includes all transitions out of p in the other copy $1 - i$; and finally (3) toggles which of f_0, f_1 is included.

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

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

$$\mid f_{1-i} \,+\!\leftarrow (q, l, p, i) \mid f_i \,\%\!\leftarrow (q, l, p, i)\big)\Big) \mid f_0 \rightarrow\!\bullet\ f_0 \mid f_1 \rightarrow\!\bullet\ f_1 \tag{3}$$

Fig. 4. Term $\mathsf{t}(B)$.

Event	Happened	Included	Restless
f_0	f	t	t
f_1	f	f	t
$(q_0, l, p, 0)$	f	t	f
$(q \neq q_0, l, p, 0)$	f	f	f
$(q, l, p, 1)$	f	f	f

Fig. 5. Marking $\mathsf{m}(B)$.

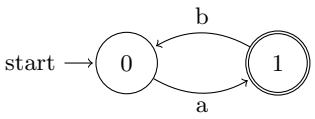

Fig. 6. Example Büchi automaton.

Example 5. Consider the Büchi automaton B in Fig. 6. Using the above construction, we find the events and transitions in the table below. Numbers refer to equation numbers of Fig. 4. 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)	$\rightarrow+$ (2)				
$(1,b,0,0)$		$\rightarrow\%$ (3)	$\rightarrow+$ (4)		$\rightarrow\%$ (5)	$\rightarrow+$ (5)
$(0,a,1,1)$			$\rightarrow\%$ (1)	$\rightarrow+$ (2)		
$(1,b,0,1)$	$\rightarrow+$ (4)			$\rightarrow\%$ (3)	$\rightarrow+$ (5)	$\rightarrow\%$ (5)
f_0						
f_1						

Lemma 6. *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 on 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 on 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. □

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

Proposition 7. *The language recognised by a DCR process is the union of a regular and an ω-regular language.*

Proof (Sketch). By Theorem 10 each DCR process has a language-preserving encoding into a DCR Graph. But in [30], using an encoding of finite words as infinite words using τ transitions, the language of DCR Graphs was shown to be contained in the union of regular and ω-regular languages. □

Proposition 8. *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 (Proof sketch). 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 9. *A language $\mathcal{L}$ is recognised by some DCR process iff $\mathcal{L}$ is the union of a regular and an ω-regular language.*

Finally, we note the connection of DCR processes to DCR Graphs [28,36,12].

Theorem 10. *There exists a language-preserving bijection between DCR processes and finite DCR graphs.*

3 DCR* Processes: Local events and Reproduction

Below we extend the DCR process language to support dynamic creation of sub processes. We do this by extending the syntax with *local* and *reproductive* events as shown in Fig. 7, giving rise to the DCR* process language.

The *local event* $(\nu e : \Phi)\, T$ asserts that e with state Φ is local to the term T. Here, e is binding in Φ and T; for reasons which will be clear when we define accepting runs below, we will follow the Barendregt-convention and assume that all such local events are distinct. A *reproductive event* $e\{T\}$ creates, whenever the event e happens, a copy of T in parallel (to maintain the Barendrecht-convention, every local event in the copy is α-converted to a fresh, *but identically labelled* event). [4]

$$
\begin{array}{rll}
T, U ::= & \ldots & \\
 & \mid (\nu e : \Phi)\, T & \text{local event} \\
 & \mid e\{T\} & \text{reproductive event}
\end{array}
$$

Fig. 7. DCR* syntax.

[4] We assume an infinite number of events in $\mathcal{E}$ for each label in $\mathcal{L}$.

Example 11 (Grant process with reproductive and local events). We now consider three extra requirements: 1) When an application is received, a committee must recommend either *approval* or *rejection* to the board. 2) The committee might withdraw an approval, by later rejecting the application, but cannot reverse a rejection. 3) 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 declare recv to be reproductive by adding the reproductive event recv{A}, where

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

Because approve and reject are local, each dynamically created sub-process A will have distinct decision events (all with the labels approve and reject though) that cannot be constrained further outside the scope. But, approve has a condition relation to the non-local bm, which means that each distinct approve event will become a condition for the (global) event bm. The exclude relation from reject to approve model that it is not possible to approve after a rejection, but nothing disallows rejection after approval. Both events have initially the local state "not-happened" and "included". We make the approve event initially restless in its local state, which will mean that in order for the process to be accepting either approve must happen or be excluded (because reject happens).

The transition rules for the new constructs are given in Fig. 8. Only terms and transition rules are extended; markings are the same.

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

$$\frac{[M]\ T \xrightarrow{\nu e:\delta} T'}{[M]\ T \mid U \xrightarrow{\nu e:\delta} T' \mid U} \quad [\textsc{par-2}] \qquad \frac{[M]\ T'' \xrightarrow{e:\delta} T' \quad T \cong_\alpha T''}{[M]\ e\{T\} \xrightarrow{e:\delta} e\{T\} \mid T'} \quad [\textsc{rep}]$$

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

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

Fig. 8. Transition semantics for local and reproductive events.

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 $(\nu f : M)\ e \mathrel{+}\!\leftarrow f$, 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's 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 (eventual) effect of δ on M. Rule [effect-2] lifts effect transitions with local events to

process transitions. Finally, the rule [REP] implements reproductive events: If the guarding event e happening would update the body T to become T', then e can unfold to such a T'. In DCR*, the term *does* change as the process evolves.

To define accepting runs we need to track local restless events across transitions. For this reason we assume the unique local events and maintain this by α-conversion (denoted by $\cong_\alpha$) of local events when a reproductive event happens, i.e., local events duplicated by [REP] are chosen globally fresh.

Definition 12. *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 $\ell(\nu e) \stackrel{def}{=} \ell(e)$. 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.*

Example 13. A possible transition sequence for the reproductive $\mathsf{recv}\{A\}$ event defined above in the marking $M_1 = \mathsf{recv} : (\mathsf{f},\mathsf{t},\mathsf{f}), \mathsf{bm} : (\mathsf{f},\mathsf{t},\mathsf{f})$ is as follows.

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

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 the local events $\mathsf{approve}$ and reject have been α-converted to $\mathsf{approve}_1, \mathsf{approve}_2$ (but still labelled $\mathsf{approve}$) and $\mathsf{reject}_1, \mathsf{reject}_2$ (but still labelled reject) respectively, following the convention of unique local events. 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 $(\nu\mathsf{approve}_1 : (\mathsf{f},\mathsf{t},\mathsf{t})) \ldots\ \mathsf{approve}_1 \rightarrow\!\bullet\, \mathsf{bm}$ 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).

3.1 Encoding of Minsky machines

We now show that DCR* has the full power of Turing machines by reduction from the Halting Problem for Minsky machines [26].

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.
$\mathtt{halt}$	Halt execution (w.l.o.g. 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 *may happen only if* a *is excluded*. Thus, the program counter is modelled by always having all but one a included. 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:

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

Now, 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 reproductive, replicating 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 represents 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: $\mathtt{decjn}^a \rightarrow\bullet\ \mathtt{decjz}^a$. 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 $I = \{i_1, \ldots, i_n\}$.)

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

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

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

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

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

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

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

Example 14. 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:

$1 : \mathtt{decjz}(2, 3, 2)$
$2 : \mathtt{inc}(1, 1)$
$3 : \mathtt{halt}$

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

$\prod_{a:\mathtt{inc}(i,b)\in P} \mathsf{inc}(a,i,b)$	$\prod_{a:\mathtt{decjz}(i,b,c)\in P} \mathsf{insn}(\mathtt{decjz}^a, a, b)$
$2 \to\bullet\ \mathtt{inc}^2$ $2\ \%\!\leftarrow \mathtt{inc}^2$ $1\ +\!\leftarrow \mathtt{inc}^2$ $\mathtt{inc}^2\{0\}$	$1 \to\bullet\ \mathtt{decjz}^1$ $1\ +\!\leftarrow \mathtt{decjz}^1$ $3\ \%\!\leftarrow \mathtt{decjz}^1$

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

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

This encodes a Minsky machine as a DCR*-process:

Theorem 15. *A Minsky machine m halts iff $[\mathsf{m}(m)]\ \mathsf{t}(m)$ has an accepting run.*

Proof. (outline) The proof is based on 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 we 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. The bisimulation relation will relate an execution trace of the Minsky machine ending in address j to a marking in which that event is excluded. Next we prove that for every pair, the machine can perform an instruction iff the encoding can execute the corresponding event, and that the form of the process $t(m)$ is preserved as well as the global marking $m(m)$, except that instruction events are being recorded as executed (and excluded in the case of `decjn`) is preserved by steps. It follows that the restless `halt` event can be eventually executed if and only if the machine can execute the halt command.

Example 16. 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:

$1 : \mathtt{decjz}(2, 3, 2)$
$2 : \mathtt{inc}(1, 1)$
$3 : \mathtt{halt}$

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

$\prod_{a:\mathtt{inc}(i,b)\in P} \mathsf{inc}(a,i,b)$	$\prod_{a:\mathtt{decjz}(i,b,c)\in P} \mathsf{insn}(\mathtt{decjz}^a, a, b)$
$2 \rightarrow\bullet\ \mathtt{inc}^2$ $2\ \%\leftarrow\ \mathtt{inc}^2$ $1\ +\!\leftarrow\ \mathtt{inc}^2$ $\mathtt{inc}^2\{0\}$	$1 \rightarrow\bullet\ \mathtt{decjz}^1$ $1\ +\!\leftarrow\ \mathtt{decjz}^1$ $3\ \%\leftarrow\ \mathtt{decjz}^1$

$\prod_{a:\mathtt{halt}\in P} a \rightarrow\bullet\ \mathtt{halt}$	$\prod_{a:I\in P} a \rightarrow\bullet\ a$	$\prod_{i<R_1} \mathsf{one}(1)$	$\prod_{i<R_2} \mathsf{one}(2)$
$3 \rightarrow\bullet\ \mathtt{halt}$	$1 \rightarrow\bullet\ 1$ $2 \rightarrow\bullet\ 2$ $3 \rightarrow\bullet\ 3$	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 $\Pi_{i<R_2}\mathsf{one}(2)$, all instances of $\mathtt{decjn}^1$ are within the scope of the binder and thus local.

4 Run-time Adaptations by Composition and Refinement

We now turn to investigating run-time refinement and adaptations of DCR* processes by composition. We shall find that, as a consequence of the Turing-completeness of DCR*, refinement is in general undecidable. We however identify and exemplify a practically useful, decidable sub-class of refinements, which we call *non-invasive adaptations*.

To define composition of processes, we need to define merge of markings:

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

Note that merge on markings is *partial*, since it is only defined on markings that agree on their overlap. When the merge of the markings of two processes is defined, we say that the processes are *marking compatible*.

Definition 17. *Given marking compatible DCR* processes $[M]\ T$ and $[N]\ S$ their* composition *is defined as* $[M]\ T \oplus [N]\ S = [M \oplus N]\ T \mid S$.

Example 18. Suppose that *as the grant process of Example 1 runs*, e.g. just after the round has been opened, a new requirement comes up: For regulatory reasons, a board meeting must eventually be followed by an audit. We model this constraint by a new event, audit, which must be a response to bm. As we are introducing a new event, we must also introduce additional marking. The following process R_1 embodies the adaptation we wish to achieve.

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

Assume the process $P = [M_1]\ T_1$ is the process reached after the first step of Example 4, i.e. $[M_0]\ T_1 \xrightarrow{\mathsf{round}} P$. We can then adapt P to include R_1 simply by composing the two processes:

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

As a second example, suppose further that it is also decreed that *during* an audit, no further applications can be received. We adapt P_1 with R_2 as follows:

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

When we extend the set of requirements by a (run-time) adaptation of P to P', we often want to ensure that the results is a refinement of the existing requirements, meaning that the old set of requirements is upheld. Informally, the adapted process does not exhibit behaviour disallowed by P. We cannot simply formulate refinement by language inclusion $\mathsf{lang}(P') \subseteq \mathsf{lang}(P)$, since we may

not only add new constraints, but also new *events* (and thus new labels), like audit in the above example. Instead, we define refinement as language inclusion only w.r.t. the alphabet of P. In doing so we employ the following notation.

Notation. Given a sequence s, write $s|_\Sigma$ for the largest sub-sequence s' of s s.t. $s'_i \in \Sigma$; e.g, if $s = AABC$ then $s|_{A,C} = AAC$.

Definition 19. *Given DCR* processes P and P', we say that P'* is a refinement of P *iff* $\mathsf{lang}(P')|_{\mathsf{alph}(P)} \subseteq \mathsf{lang}(P)$.

When merging in new constraints P' to a process P gives rise to a refinement we will say P' is conservative for P, as defined formally below.

Definition 20. *Given marking compatible DCR* processes P and Q, we say that Q is* conservative *for P iff $P \oplus Q$ is a refinement of P.*

Example 21. Continuing the above example, we now see a fundamental distinction between the adaptation by R_1 and R_2: the former refines P, whereas the latter does not refine P_1. To see this, observe for R_1 that it only makes P_2 *less* accepting (because of the potential restlessness of the new event audit). For R_2, 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, so after a second audit, we have an accepting trace $t = \langle \mathsf{audit},\ \mathsf{bm},\ \mathsf{audit} \rangle$. However, bm cannot 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 refinement:

$$\langle \mathsf{audit},\ \mathsf{bm},\ \mathsf{audit} \rangle|_{\mathsf{alph}(P_1)} = \langle \mathsf{bm} \rangle \notin \mathsf{lang}(P_1)$$

Inspecting the adaptation R_2 more closely, one see that the problem comes from the dynamic exclusion of the recv event, since it not only makes the reception of applications impossible, but 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. The corresponding adaptation $P_2 \oplus R_2$ *is* a refinement.

Unfortunately the property of one process being conservative for another is undecidable:

Theorem 22. *It is undecidable whether a DCR*-process P is conservative for a DCR*-process Q.*

Proof. 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 following Theorem 15. Take P to be the process $P = []\ (\nu e : (\mathsf{f},\mathsf{t},\mathsf{f}))\ e \rightarrow\bullet\ e$, with e labelled `halt`. We show that m is

terminating iff M is not conservative for P. Clearly $\mathsf{lang}(P) = \epsilon$, that is, the only trace of P is the empty trace. By Theorem 15, the encoding M of m has a trace exhibiting the label `halt` iff m terminates, so $\mathsf{lang}(P \oplus M)|_{\mathsf{alph}(P)}$ has a non-empty trace iff m terminates. It follows that $\mathsf{lang}(P \oplus M)|_{\mathsf{alph}(P)} \subseteq \mathsf{lang}(P)$ iff m does not terminate, and so M is conservative for P iff m does not terminate.

Fortunately, we have identified a large class of practically useful refinements, which we dub *non-invasive* adaptations.

Definition 23 (Non-invasive adaptation). *Let $P_1 = [M_1]\ T_1$ and $P_2 = [M_2]\ T_2$ be processes. We say that P_1* non-invasive *for P_2 iff*

1. *For every context $C(-)$, such that $T_1 = C(e \rightarrow\!\!\!\!\% \ f)$ or $T_1 = C(e \rightarrow\!\!\!\!+ \ f)$, either f is bound in $C(-)$ or $f \notin \mathsf{fe}(P_2)$; and*
2. *For every label $l \in \mathsf{alph}(P_1) \cap \mathsf{alph}(P_2)$, no bound event of T_1 is labelled l, and if $e \in \mathsf{fe}(P_1)$ is labelled l, then $e \in \mathsf{fe}(P_2)$.*

It's straightforward to verify that non-invasiveness is decidable, and that R_1 and R_2' are non-invasive adaptations for P and P_1 respectively, whereas R_2 is not for P_1 (because of the exclusion of bm).

Moreover, we can indeed prove that non-invasive adaptations are conservative, and thus gives rise to refinements.

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

Definition 24. *The* free events $\mathsf{fe}(T)$ *of a term T is defined recursively as follows.*

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

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.*

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

The following Lemma states that transitions preserve free events and alphabet. We use λ to range over effect transition actions and γ to range over process transition actions.

Lemma 25. *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 free events of a DCR* term $\mathsf{fe}(T)$ is its set of non-bound events; we still require of a process $[M]\ T$ that $\mathsf{fe}(T) \subseteq \mathsf{dom}(M)$. The *alphabet* $\mathsf{alph}(P)$ of a process P is the set of labels of its free *and* bound events. First observe that transitions do not introduce new constraints or effects on free events.

Lemma 26 (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 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 27 (Separation of Processes). *Let* $P = [M]\ T_1 \mid T_2$. *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* $\mathsf{fe}(T_i) \setminus \mathsf{fe}(T_{3-i}) \subseteq \mathsf{dom}(M_i)$.

Lemma 28 (Canonical Separation). *Let* $P_1 = [M_1]\ T_1$ *and* $P_2 = [M_2]\ T_2$ *with* $P_1 \oplus P_2$ *defined. There exists a unique separation* N_1, N_2, S *of* $P_1 \oplus P_2$ *satisfying* $\mathsf{dom}(N_i) = \mathsf{dom}(M_i) \setminus \mathsf{dom}(M_{3-i})$ *and* $\mathsf{dom}(S) = \mathsf{dom}(M_1) \cap \mathsf{dom}(M_2)$. *We call this separation the* canonical *separation of* $P_1 \oplus P_2$ *for* P_1, P_2.

Lemma 29. *Let* $P = [M]\ T_1 \mid T_2 = P_1 \oplus P_2$ *with* $P_i = [M_i \oplus S]\ T_i$. *Suppose* M_1, S, M_2 *is the canonical separation of* P *for* P_1, P_2, *and suppose we have a transition*

$$[M]\ T_1 \mid T_2 \xrightarrow{\lambda} T' \quad \text{or} \quad [M]\ T_1 \mid T_2 \xrightarrow{\gamma} [M']\ T'\ .$$

Then the following holds:

1. *For some* T_1', T_2' *we have* $T' = T_1' \mid T_2'$.
2. *There exists a unique separation* M_1', M_2', S' *of* $[M']\ T'$ *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. *This separation is canonical of* $[M']\ T'$ *for* $P_1' = [M_1' \oplus S']\ T_1'$ *and* $P_2' = [S' \oplus M_2']\ T_2'$.
5. *If* P_2 *non-invasive for* P_1, *then also* P_1' *non-invasive for* P_2'.

Proof. Note that only the rules [PAR] and [PAR-2] allows term transitions for a term on the form $T_1 \mid T_2$; part 1 is then immediate by inspection of these rules; and part 2 and 3 follows from Lemma 25. Part 4 is then immediate from parts 2 and 3. Part 5 follows (1) by Lemma 26 and (2) by parts (2–4) and Lemma 25.

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

Definition 30. *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 31. *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 32. *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 2 that $T = T'$ and $\delta = \delta'$.

The cases [PAR], [PAR-2], and [REP] cases are straightforward; we exemplify with [PAR-2]. Suppose $[M]\, T \mid U \xrightarrow{\nu e:\delta_1} T_1$ and $[M]\, T \mid U \xrightarrow{\nu e:\delta_2} T_2$. By [PAR-2] we must have $T_1 = T_1' \mid U$ and $T_2 = T_2' \mid U$, 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 33 (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 2 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 $\mathsf{fe}(T_1 \mid T_2) \cup \{e\}$ 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 $[M]\ T_1 \mid T_2 \xrightarrow{e:\delta_1 \oplus \delta_2} T_1' \mid T_2'$.

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 $\mathsf{fe}(T) \subseteq \mathsf{fe}(T \mid U)$ and so in both cases disjointness with N is preserved as we move to the hypothesis.

Lemma 34. *If* $[M]\ T \xrightarrow{\gamma:\delta} T'$ *with* $\delta = (X, I, R)$ *then* $e \in X$ *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 35. *Let* P *be non-invasive for* Q*, and suppose* M_1, S, M_2 *is the canonical separation of* $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \mid T_2$*. 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 34.

Lemma 36. *Let* P *be non-invasive for* Q*, and suppose* M_1, S, M_2 *is the canonical separation of* $P \oplus Q$ *for* P, Q*. If also* $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \mid T_2 \xrightarrow{\gamma:\delta} T_1' \mid T_2'$ *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 33 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 35 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 32 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 33 we have a transition

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

By Lemma 35 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 33 and 35.

Lemma 37. *Let* P *be non-invasive for* Q*, and suppose* M_1, S, M_2 *is the canonical separation of* $P \oplus Q$ *for* P, Q*. If also* $P \oplus Q = [M_1 \oplus S \oplus M_2]\ T_1 \mid T_2 \xrightarrow{\gamma} [M_1' \oplus S' \oplus M_2']\ T_1' \mid T_2' = R$ *and* M_1', S', M_2' *is the canonical separation of* R *for* $[M_1' \oplus S']\ T_1'$ *and* $[S \oplus M_2']\ T_2'$*; then the following 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 the preceding Lemma and rules [EFFECT] and [EFFECT-2].

Proof (Of Theorem **??***).* Let $M = M_1 \oplus S \oplus M_2$ be the canonical separation of $P \oplus Q$ for P, Q, and consider a finite or infinite run of $R_0 = P \oplus Q = [M]\ T_1 \mid T_2$:

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

By induction on i using Lemma 29, we can write each R_i as

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

where $\mathsf{alph}([M_1^i \oplus S^i]\ T_1^i) = \mathsf{alph}(P)$; M_1^i, S^i, M_2^i is the canonical separation of R_i for $[M_1^i \oplus S^i]\ T_1^i$ and $[M_2^i \oplus S^i]\ T_2^i$; $\mathsf{alph}([S^i \oplus M_2^i]\ T_2^i) = \mathsf{alph}(Q)$; and $[M_1^i]\ T_1^i$ is non-invasive for $[M_2^i]\ T_2^i$.

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)

$$\begin{aligned} N^{i+1} &= N^i && \text{when } \ell(\gamma_i) \notin \mathsf{alph}(Q) \\ [N^i]\ T_2^i &\xrightarrow{\gamma_i} [N^{i+1}]\ T_2^{i+1} && \text{when } \ell(\gamma_i) \in \mathsf{alph}(Q) \end{aligned}$$

The Theorem then follows. We have immediately N^0, obtaining (a). For (b) and (c), consider some $i > 0$, and assume first $\ell(\gamma_i) \notin \mathsf{alph}(Q)$. By Lemma 37, Part 2, we then have $S^{i+1} \oplus M_{i+1} \sqsubseteq S^i \oplus M_i \sqsubseteq N_i = N_{i+1}$, obtaining (b) and (c). Assume instead $\ell(\gamma_i) \in \mathsf{alph}(Q)$. Then take $N^{i+1} = N$ where N is given by Lemma 37, Part 1, immediately obtaining (b) and (c).

Non-invasiveness adaptations admits a large class of practically important refinements. As illustrated by the adaptations given by R_1 and R_2', the permitted adaptations correspond to dynamically adding an arbitrary new process to the running process, and adding arbitrary condition and response relations between events of the composed process. Even though existing events can not be excluded by new events, it is possible to arbitrarily block events of the original process.

Within the application area of business process modelling, it is a common change to add such possibility/requirement of taking additional actions interleaved between existing actions. Indeed, the need for a non-invasive, run-time adaptation showed up in the implementation of the grant application process [10]. After the start of an application round, a forgotten requirement was realised: If the account number of a grant holder is changed, then the accountant must verify, that the account belongs to the grant holder before the next payment. The adaptation was made to the DCR Graph representing the run-time state of the grant application system *without terminating or restarting any systems.*

5 Conclusion, Related and Future Work

We studied the interplay of dynamic process instantiation, run-time adaptation and refinement in the context of a declarative event-based process language, generalising our prior work on DCR Graphs co-developed and implemented by our industrial partner. Specifically, we proved that dynamic process instantiation makes the language Turing-complete, and as a consequence, refinement undecidable. We then identified a large, decidable and practically useful class of refinements referred to as non-invasive adaptations. All 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 [28,36,17], 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 [25]) and to represent infinite computations and ω-regular languages (e.g., Büchi Nets [14]). 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 relates to the asymmetric exclude relation of DCR processes, including extended bundle event structures [23,16], dual event structures [22,24], asymmetric event structures [6], and precursor event structures [15]. Automata based models like Event automata [32] and local event structures [20] 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 [37], 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*. Reproductive events of the DCR* process language relate to replication in process calculi and higher-order Petri nets [21]. We believe to be the first to combine higher-order features and liveness.

Run-time adaptation has been studied also for Petri nets [35] and process calculi [5,8,9], but tends to require predefined adaptation points, and often deal with adaptations via higher-order primitives. In contrast, adaptation in DCR* is dealt with by composition, which due to the declarative nature allow for cross-cutting adaptations without the need for pre-specified adaptation points.

In the BPM community, the seminal declarative process language is Declare [3,4]. 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 [27], 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.

Imperative process models such as BPMN [31] have supported dynamic sub-processes for some time now, they are only recently being studied for declarative

languages [39]. 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 graphs with sub-processes to be Turing complete.

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 [13] and runtime monitoring [28]. The DCR* process language would benefit from a closer investigation of its relation to modular [25] and higher-order Petri Nets [21]. Finally, time constraints and more general adaptations as initiated in [19,29], e.g. allowing to remove constraints and events should be further investigated.

Acknowledgments. We thank the anonymous reviewers for helpful comments.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: A survey. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2678, pp. 1–12. Springer (2003)
3. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer (2006)
4. van der Aalst, W.M.P., Pesic, M., Schonenberg, H., Westergaard, M., Maggi, F.M.: Declare. Webpage (2010), `http://www.win.tue.nl/declare/`
5. Anderson, G., Rathke, J.: Dynamic software update for message passing programs. In: Jhala, R., Igarashi, A. (eds.) APLAS. Lecture Notes in Computer Science, vol. 7705, pp. 207–222. Springer (2012)
6. Baldan, P., Corradini, A., Montanari, U.: Contextual petri nets, asymmetric event structures, and processes. Information and Computation 171, 1–49 (2001)
7. Barthe, G., Pardo, A., Schneider, G. (eds.): Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings, LNCS, vol. 7041. Springer (2011)
8. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Steps on the road to component evolvability. In: Proceedings of the 7th International Conference on Formal Aspects of Component Software. pp. 295–299. FACS'10 (2012), `http://dx.doi.org/10.1007/978-3-642-27269-1_19`
9. Bravetti, M., Giusto, C.D., Pérez, J.A., Zavattaro, G.: Adaptable processes. Logical Methods in Computer Science 8(4) (2012)
10. Debois, S., Hildebrandt, T., Marquard, M., Slaats, T.: A case for declarative process modelling: Agile development of a grant application system. In: EDOCW/AdaptiveCM '14. pp. 126 – 133. IEEE (September 2014)

11. Debois, S., Hildebrandt, T., Slaats, T.: Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes (full version) (2015), http://www.itu.dk/~debois/dcrstar-tr.pdf
12. Debois, S., Hildebrandt, T.T., Slaats, T.: Hierarchical declarative modelling with refinement and sub-processes. In: Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8659, pp. 18–33. Springer (2014), http://dx.doi.org/10.1007/978-3-319-10172-9
13. Debois, S., Hildebrandt, T.T., Slaats, T., Yoshida, N.: Type checking liveness for collaborative processes with bounded and unbounded recursion. In: FORTE. Lecture Notes in Computer Science, vol. 8461, pp. 1–16. Springer (2014)
14. Esparza, J., Melzer, S.: Model checking LTL using constraint programming. In: Azéma, P., Balbo, G. (eds.) Application and Theory of Petri Nets 1997, Lecture Notes in Computer Science, vol. 1248, pp. 1–20. Springer Berlin Heidelberg (1997)
15. Fecher, H., Majster-Cederbaum, M.: Event structures for arbitrary disruption. Fundam. Inf. 68(1-2), 103–130 (Apr 2005)
16. van Glabbeek, R., Vaandrager, F.: Bundle event structures and CCSP. In: CONCUR 2003 - Concurrency Theory, LNCS, vol. 2761, pp. 57–71. Springer (2003)
17. Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES. EPTCS, vol. 69, pp. 59–73 (2010)
18. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: FSEN. LNCS, vol. 7141, pp. 343–350. Springer (2011)
19. Hildebrandt, T.T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. J. Log. Algebr. Program. 82(5-7), 164–185 (2013)
20. Hoogers, P., Kleijn, H., Thiagarajan, P.: An event structure semantics for general petri nets. Theoretical Computer Science 153(1–2), 129 – 170 (1996)
21. Janneck, J.W., Esser, R.: 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. pp. 17–25. CRPIT '02 (2002)
22. Katoen, J.P.: Quantitative and qualitative extensions of event structures. Ph.D. thesis, University of Twente, Enschede (April 1996)
23. Langerak, R.: Transformations and Semantics for LOTOS. Universiteit Twente (1992)
24. Langerak, R., Brinksma, E., Katoen, J.P.: Causal ambiguity and partial orders in event structures. In: CONCUR '97, LNCS, vol. 1243, pp. 317–331. Springer (1997)
25. Latvala, T., Mäkelä, M.: LTL model checking for modular petri nets. In: Applications and Theory of Petri Nets 2004, LNCS, vol. 3099, pp. 298–311. Springer (2004)
26. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
27. Montali, M.: Specification and Verification of Declarative Open Interaction Models - A Logic-Based Approach, Lecture Notes in Business Information Processing, vol. 56. Springer (2010)
28. Mukkamala, R.R.: A Formal Model For Declarative Workflows: Dynamic Condition Response Graphs. Ph.D. thesis, IT University of Copenhagen (June 2012)
29. Mukkamala, R.R., Hildebrandt, T., Slaats, T.: Towards trustworthy adaptive case management with dynamic condition response graphs. In: EDOC. pp. 127–136. IEEE (2013)

30. Mukkamala, R.R., Hildebrandt, T.T.: From dynamic condition response structures to Büchi automata. In: TASE. pp. 187–190. IEEE Computer Society (2010)
31. Object Management Group BPMN Technical Committee: Business Process Model and Notation, version 2.0, http://www.omg.org/spec/BPMN/2.0/PDF
32. Pinna, G., Poigné, A.: On the nature of events: another perspective in concurrency. Theoretical Computer Science 138(2), 425 – 454 (1995), meeting on the mathematical foundation of programing semantics
33. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)
34. Russell, N., ter Hofstede, A., van der Aalst, W., Mulyar, N.: Workflow control-flow patterns : A revised view (2006), BPMcenter.org
35. Sibertin-Blanc, C., Mauran, P., Padiou, G.: Safe Adaptation of Component Coordination. Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities 189, 69–85 (juillet 2007)
36. Slaats, T., Mukkamala, R.R., Hildebrandt, T.T., Marquard, M.: Exformatics declarative case management workflows as DCR graphs. In: BPM. LNCS, vol. 8094, pp. 339–354. Springer (2013)
37. Winskel, G.: Events in Computation. Ph.D. thesis, University of Edinburgh (1980)
38. Winskel, G.: Event structures. In: Advances in Petri Nets. LNCS, vol. 255, pp. 325–392. Springer (1986)
39. Zugal, S., Soffer, P., Pinggera, J., Weber, B.: Expressiveness and understandability considerations of hierarchy in declarative business process models. In: BMMDS/EMMSAD. Lecture Notes in Business Information Processing, vol. 113, pp. 167–181. Springer (2012)