Assured Automatic Dynamic Reconfiguration of Business Processes

Abstract

In order to manage evolving organisational practice and maintain compliance with changes in policies and regulations, businesses must be capable of dynamically reconfiguring their business processes. However, such dynamic reconfiguration is a complex, human-intensive and error prone task. Not only must new business process rules be devised but also, crucially, the transition between the old and new rules must be managed.

In this paper we present a fully automated technique based on formal specifications and discrete event controller synthesis to produce correct-by-construction reconfiguration strategies. These strategies satisfy user-specified transition requirements, be they domain independent - such as delayed and immediate change - or domain specific. To achieve this, we provide a discrete-event control theoretic approach to operationalise declarative business process specifications, and show how this can be extended to resolve reconfiguration problems. In this way, given the old and the new business process rules described as Dynamic Condition Response Graphs, and given the transition requirements described with linear temporal logic, the technique produces a control strategy that guides the organisation through a business process reconfiguration ensuring that all transition requirements and process rules are satisfied. The technique outputs a reconfiguration DCR whose traces reproduce the controller's reconfiguration strategy. We illustrate and validate the approach using realistic cases and examples from the BPM Academic Initiative.

Keywords: Dynamic Reconfiguration, Controller Synthesis, DCR graph

1. Introduction

Organisations rely on business processes to ensure that task and activity execution achieves their objectives. Workflows are operational models that guide everyday activities ensuring that business processes are adhered to. These workflows can be derived automatically from formally defined processes or manually.

As business environments evolve, organisations must ensure that their processes are consistent with their own policies, strategies and external regulations, e.g., Van der Aalst and Stefan (2000). As a result, workflows must also be evolved. *Business process reconfiguration* involves not only devising the new workflows for new business rules but also dynamically changing the old workflow with the new one.

In other words, reconfiguration requires defining how business instances running the current workflow should transition to the new workflow. There is no unique way of defining these transitions but some widespread domain independent options have been studied. For instance, an "immediate" reconfiguration requirement Ellis et al. (1995) asserts that reconfiguration must occur as soon as possible but only at a state in which the new workflow prescribes behaviour consistent with the old one. On the other hand, a "delayed" reconfiguration requirement states that living instances continue to use the old workflow, while new instances should be created using the new workflow. However, in some cases *domain specific transition requirements* are needed: An organisation may require applying different strategies for different workflow instances based on the specific state each instance is on. For some an "immediate" reconfiguration may be desired, for others a "delayed" reconfiguration is needed, and for some others, remediation or compensation activities specifically designed for the reconfiguration must be applied before reconfiguring.

Manual business process reconfiguration can be complex, laborious and error-prone. It can therefore benefit greatly from automated techniques that support *i)* specification and analysis of business process and transition requirements, and *ii)* construction of workflows and reconfiguration strategies that satisfy these re-

quirements.

One approach to automation is *build and verify*, where formal verification techniques are used to check compliance of workflows to requirements. However, post-hoc verification requires prior manual workflow construction, and verification failures can require laborious workflow debugging. An alternative approach is to automatically produce *correct-by-construction* workflows and reconfiguration strategies directly from requirements.

Automatic construction of workflows from business rules has been studied (e.g., Pesic and Van der Aalst (2006); Hildebrandt and Mukkamala (2010)) and typically relies on ad-hoc algorithms and/or results from automata theory. In this paper we take an alternative and novel approach to workflow synthesis that formulates the problem as a discrete event supervisory control problem Ramadge and Wonham (1989) in which the humans and systems being coordinated by the workflow are the system under control and the business process requirements are the goals to be achieved by the controller.

Construction of workflow *reconfiguration strategies* has also been studied (e.g., Ellis et al. (1995); Kradolfer and Geppert (1999); Zhao and Liu (2007)) but restricted to domain independent transition requirements and no support for implementing varied transition strategies on a per instance basis in which, for instance, remedial or compensatory activities are prescribed. In this paper we present a fully automated technique for business process reconfiguration that supports domain dependent, user-defined transition requirements. We use synthesis to not only produce correct-by-construction workflows from business process requirements but also to compute a reconfiguration strategy that guarantees progress from an old workflow towards the new one while satisfying any user-defined transition requirements.

In particular, we show how Dynamic Condition Response (DCR) graphs Hildebrandt and Mukkamala (2010), a declarative language for business process requirements, can be translated into a composition of Labelled Transition Systems (LTS) Keller (1976) and Fluent Linear Temporal Logic (FLTL) Giannakopoulou and Magee (2003) which is suitable for discrete event controller

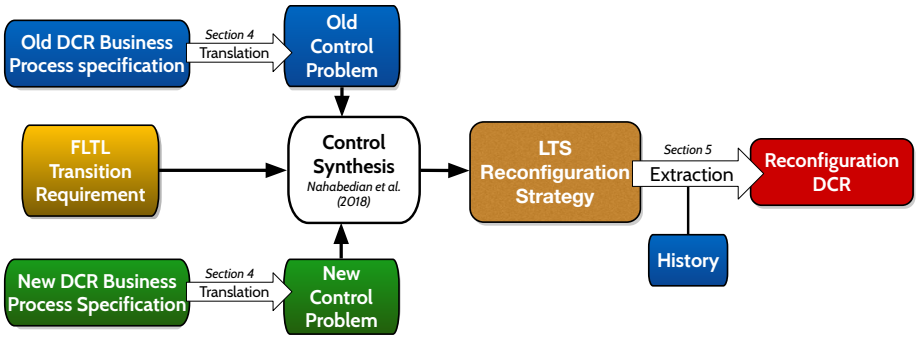

Figure 1: Dynamic reconfiguration of business process schema.

synthesis D'Ippolito et al. (2013). We build upon recent work on dynamic controller update Nahabedian et al. (2018) to handle dynamic reconfiguration. The approach is validated using examples from the BPM Academic Initiative BPMAI (2020) described as DCR graphs which we reconfigured to illustrate a variety of transitions requirements.

Figure 1 shows the reconfiguration schema presented in this paper. It starts with the translation of an old and new DCR business process specifications to control problems. With the two control problems and user specified FLTL transition requirements, we define a controller update problem Nahabedian et al. (2018) that when solved yields a Reconfiguration Strategy represented as an LTS. This strategy is able to reconfigure any live instance of the current workflow from any state. The dynamic reconfiguration schema finishes with an extraction process in which the history of one live instance of the current workflow is used to compute from the Reconfiguration Strategy, a DCR graph that describes how reconfiguration of that particular live instance should proceed to satisfy transition requirements and new business rules.

This paper extends our previous work Nahabedian et al. (2019) in two major ways. Firstly, we provide the extraction method that outputs a declarative workflow model using the same language as the one provided by the user. Thus, the technique presented in Nahabedian et al. (2019) becomes more useful for users as they are not required to understand yet another workflow language.

Second, we provide a formalization of the technique giving definitions for the translation of user specified inputs to control problems (see Section 4) and for the extraction method to obtain the final output (see Section 5.4). Both translation and the extraction method are shown to be sound and complete for given definitions.

The rest of the paper is structured as follows. Section 2 presents an illustrative example. Formal definitions are presented in Section 3. Section 4 shows a translation from models in a declarative workflow language to a control problem. In Section 5 we present how to set out the reconfiguration problem and how to frame it as a synthesis problem. Later in this section we introduce the extraction method that builds a declarative workflow model representing the reconfiguration. An analysis of our technique is presented in Section 6. Finally, we present a discussion on related work (Section 7), and then, a conclusion (Section 8).

2. Running Example

We use as a running example a hospital process taken from Hildebrandt and Mukkamala (2010) and inspired by a real-life case study on oncology workflows at Danish hospitals. The process includes *prescribe medicine* and *sign* activities in which a doctor adds a prescription to a patient record and signs it. A nurse will *give medicine* in response to the doctor prescription, or challenge the prescription (the *don't trust* activity). Process requirements include that the doctor must perform *prescribe medicine* before *sign*, *ii)* that the nurse can neither perform *give medicine* nor *don't trust* if the doctor has not done *sign*, and *iii)* that the nurse cannot perform both *give medicine* and *don't trust*,.

A Dynamic Condition Response (DCR) graph for the process requirements, taken from Hildebrandt and Mukkamala (2010), is depicted in Figure 2. Whilst in Figure 4 we depict a Labelled Transition System that represents a workflow that satisfies the process requirements. Note that in Figure 4, *pm*, *s*, *gm* and *dt* labels refer to activities *prescribe medicine*, *sign*, *give medicine* and *don't trust*, respectively. The workflow also corresponds to the semantics of the DCR graph

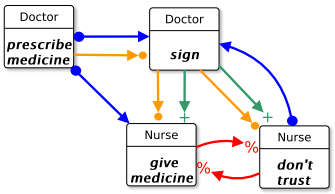

Figure 2: DCR graph for a hospital process

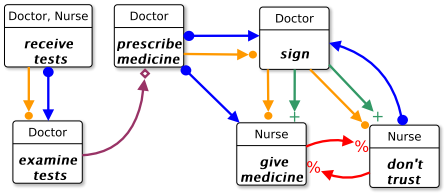

Figure 3: DCR graph model for new hospital process.

The meaning of an arrow from A to B depends on its color: if blue (i.e., response) then if A occurs then B is pending; if yellow (i.e., condition) then A must occur before B; if green (i.e., inclusion) then A enables occurrences of B; if red (i.e., exclusion) then A prohibits B; if purple (i.e., milestone) then B is prohibited if A is pending.

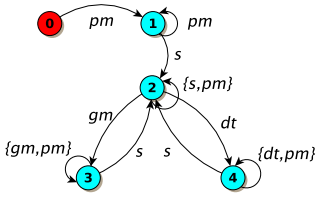

Figure 4: Workflow for a hospital process

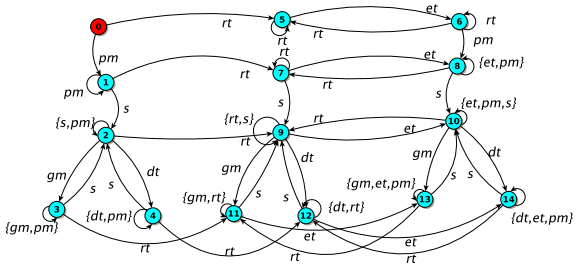

Figure 5: Workflow for new hospital process.

and can be constructed automatically using controller synthesis as described in Section 5.

Consider a scenario taken from Mukkamala (2012) in which the workflow must be changed. One such change might be that a new internal regulation has been introduced. The regulation may state that doctors must not *prescribe medicine* if new tests have arrived (*receive tests*) but have not been examined (*examine tests*). Also, as expected, *receive tests* must happen before *examine tests*. This change involves two new activities and extra rules as depicted in Figure 3 and requires a significantly more complex workflow (depicted in Figure 5 where *rt* and *et* labels refer to *receive tests* and *examine tests*). This workflow can also be automatically synthesised.

The process of reconfiguration requires handling patients that are already in

treatment. In other words, it requires handling the transition of a live instance running the old workflow to handling the new one. Consider the following scenario in which a patient has had a medicine prescribed and the prescription has been signed off. Before the medicine is actually given to the patient, new test results arrive. Note that the current hospital process does not track test results. Now consider that the hospital process is reconfigured. The current state of the new process is one in which the event *receive tests* has not occurred (because it was not being tracked), thus, the medicine can be given patient despite the fact that there were new tests. This is a scenario to be avoided, it is preferable to assume upon reconfiguration that there may be test results available but that have not yet been registered in the workflow engine.

Indeed, a naive approach in which an immediate Ellis et al. (1995) reconfiguration is required regardless of the living instance's state may pose health risks in this case. Such a reconfiguration would be map state 2 of Figure 4 to state 2 of Figure 5 (i.e., $2 \rightsquigarrow 2$). However, as discussed this puts the patient at risk.

A more appropriate reconfiguration requirement is that test examination must be performed when changing to the new hospital process, just in case. Note that this requirement is not part of the new hospital process, it is specifically designed for the transition. Indeed, it may require in some cases that doctors register that they have examined tests when there were no tests to examine.

For this reconfiguration requirement, it would be more appropriate to map state 2 with state 9, and more generally the complete mapping from the old workflow to the new workflow would be to the following mapping: $(0 \rightsquigarrow 5), (1 \rightsquigarrow 7), (2 \rightsquigarrow 9), (3 \rightsquigarrow 11), (4 \rightsquigarrow 12)$.

The provision of a mapping between workflow states ensuring that a transition requirement holds can be very difficult for complex workflows. An alternative is to support a declarative description of transition requirements and to compute a mapping automatically. For the hospital example, what is needed is that reconfiguration may not conclude without *examine tests*. Note that this requirement is inconsistent with both the old and new business requirements! In the old process, there is no *examine tests* activity. While in the new process,

examine tests is required after *receive tests*. Thus, what we need to specify is that there should be a transition period during the reconfiguration in which neither the old nor new business process requirements hold and in which *examine tests* (and nothing else) must occur. In this paper we show how domain specific transition requirements such as these can be specified in linear temporal logic, how to automatically build a workflow that codifies a strategy for taking a live instance running a workflow to a new workflow guaranteeing all transition requirements, and finally we show how this workflow can be encoded as a DCR graph. For this specific example, the transition requirement in temporal logic should say that from the time when old requirements stop to hold, every activity is prohibited (except for *examine tests*) until *examine tests* was executed and new requirements start to hold. We show in Section 5.1 how to formally write this transition requirement in temporal logic.

In short, we need to solve the reconfiguration problem by having as inputs the specification for the current workflow, the specification of a new workflow, a transition requirement representing the properties that the reconfiguration must ensure, and, the history of what it was executed under the current workflow requirements. The proposed technique must build a workflow specification that describes how to continue the given history, guaranteeing that the reconfiguration process will eventually end by satisfying the new workflow requirements. To do so, we show how to model domain specific transition requirements and how to automatically build a strategy for taking a live instance running a workflow to a new workflow guaranteeing all transition requirements. After having this strategy, we can automatically extract a workflow specification that produces a reconfiguration process for a given history.

3. Preliminaries

We use Dynamic Condition Response (DCR) graphs Hildebrandt et al. (2011) to specify both the old and new business processes, which are part of the input to the reconfiguration approach that we propose. The other inputs are the transition requirements which are described in temporal logic. We also use DCR

graphs to output the strategy that is required to correctly transition between the two processes.

3.1. Dynamic Condition Response Graphs

To simplify presentation we use a reduced version that does not include principals.

Definition 3.1. (Dynamic Condition Response graph Hildebrandt et al. (2011)) *A* Dynamic Condition Response Graph *(DCR graph) is a tuple* $DG = (A, \triangleright, R, M, Act, \ell, \mathcal{R}, \text{as})$ *where*

- A *is a finite set of activities, the nodes of the graph.*
- $\triangleright : A \to A$ *is a partial function defining a hierarchy of activities by mapping an activity to its super-activity. Activities that are not super-activities of any other are referred to as atomic*
- $R : A \to A$ *is a set of graph edges. Edges are partitioned into five kinds, named and drawn as follows: conditions* ($\to\!\bullet$)*, responses* ($\bullet\!\to$)*, inclusions* ($\to_+$)*, exclusions* ($\to_\%$) *and milestones* ($\to_\diamond$)*.*
- M *is the marking of the graph. This is a triple of sets of atomic activities* (Ex, Re, In)*, where* Ex *are the previously executed,* Re *the currently pending and* In *the currently included.*
- Act *is the set of actions labels.*
- $\ell : A \to Act$ *is a labeling function mapping activities to actions, and*
- $\mathcal{R}$ *is a set of roles*
- $\text{as} : A \to \mathcal{R}$ *is the role assignment relation to activities.*

We denote $(\bullet\!\to e) = \{e' \in A \mid e'\bullet\!\to e\}$, $(e\bullet\!\to) = \{e' \in A \mid e\bullet\!\to e'\}$, *and similarly for* $\to\!\bullet$, $\to_+$, $\to_\%$ *and* $\to_\diamond$*. For simplicity, if a DCR graph is defined without an explicit* Act, ℓ *and* $\triangleright$ *then* ℓ *is an injective function (i.e., each activity is mapped to different action labels), and* $\triangleright = \emptyset$ *(i.e., there is no any super activity). Also we will leave* $\mathcal{R}$ *and* as *implicit when they are irrelevant for a particular definition or proof.*

The following definitions are presented without considering the nesting function ($\triangleright$). We decide to do so because a nested DCR graph can be mapped to a DCR graph with at most the same number of activities Hildebrandt et al. (2011). Essentially, all relations are extended to sub activities, and then only the atomic activities are preserved.

Definition 3.2. (Enabled activity of a DCR graph) *Let* $D = (A, R, M)$ *be a DCR graph, with* $M = (Ex, Re, In)$*. An activity* $e \in A$ *is enabled from marking* M ($D \xrightarrow{e}$) *if and only if*
(a) $e \in In$,
(b) $(In \cap (\to\!\bullet e)) \subseteq Ex$, *and*
(c) $Re \cap In \cap (\to \diamond e) = \emptyset$.

Definition 3.3. (Executing DCR graph) *Let $D = (A, R, M)$ be a DCR graph, with marking $M = (Ex, Re, In)$ and $e \in A$ is enabled. The result of executing e ($D \xrightarrow{e} D'$) is a DCR graph $D' = (A, R, M')$ with $M' = (Ex', Re', In')$ such that*
(a) $Ex' = Ex \cup \{e\}$,
(b) $Re' = (Re \setminus \{e\}) \cup (e\bullet\!\!\rightarrow)$, *and*
(c) $In' = (In \setminus (e\rightarrow_{\%})) \cup (e\rightarrow_{+})$.
If not specified, we assume that initially $In = A$ and $Re = Ex = \emptyset$.

Definition 3.4. (DCR graph Run Debois et al. (2018b)) *Let $D = (A, R, M, Act, \ell)$ be a DCR graph. A run of D is a (finite or infinite) sequence of DCR graphs D_i and activities e_i such that $D_0 \xrightarrow{e_0} D_1 \xrightarrow{e_1} \ldots$. A trace of D is a sequence of actions labels $\ell(e_i)$ associated with a run of D. We write $runs(D)$ and $traces(D)$ for the set of runs and traces of D, respectively.*

In this paper we assume that every run of a DCR can be extended to an infinite run of the same DCR. A DCR that does not conform to this assumption can be extended with an additional stuttering activity to do so. The assumption is introduced to allow for a simpler semantics for the linear temporal logic, defined below, which is typically used over infinite sequences.

Definition 3.5. (Accepting Runs Debois et al. (2018b)) *Let $D_0 \xrightarrow{e_0} D_1 \xrightarrow{e_1} \ldots$ be a finite or infinite run of D. We way that it is an accepting run if for every D_i and every activity e, if $e \in Re \cap In$ then there is a $j \geq i$ such that $e \in Ex$ or $e \notin In$ in the initial marking of D_j .*

Definition 3.6. (Reachable Marking) *Let $D = (A, R, M, Act, \ell)$ be a DCR graph. A marking M is reachable from D if there exists a finite run $D_0 \xrightarrow{e_0} D_1 \xrightarrow{e_1} \ldots D_n$ where M is the marking of D_n.*

Definition 3.7. (Deterministic DCR Graph) *Let $D = (A, R, M, Act, \ell)$ be a DCR graph. We say D is deterministic if for any reachable marking M' and two activities $e, e' \in A$ enabled in M', if $\ell(e) = \ell(e')$ then $e = e'$. In this paper we assume all DCR graphs to be deterministic.*

We introduce the notion of consistent marking with respect to a sequence of activities to capture the correct update of a marking with respect to the constraints of a DCR graph independently of whether the sequence corresponds to a run of the DCR. This definition is used later on to define DCR reconfiguration. More specifically, to intialise the marking of a new DCR graph based on the execution history produced while running an old DCR graph.

Definition 3.8. (Consistent Marking) *Let $D = (A, R, M, Act, \ell)$ be a DCR graph, with marking $M = (Ex, Re, In)$. We say a marking $M' = (Ex', Re', In')$ is consistent with M and a finite sequence of action labels π if $Ex' = \delta_{Ex}$, $Re' = \delta_{Re}$ and $In' = \delta_{In}$, where δ_{Ex}, δ_{Re} and δ_{In} are defined as follows (λ corresponds to an empty sequence):*

$$\begin{array}{lll} \delta_{Ex}(Ex, \lambda) = Ex & \textit{and} & \delta_{Ex}(Ex, x.\pi) = \delta_{Ex}(Ex, \pi) \cup \ell^{-1}(x) \\ \delta_{Re}(Re, \lambda) = Re & \textit{and} & \delta_{Re}(Re, x.\pi) = (\delta_{Re}(Re, \pi) \setminus \ell^{-1}(x)) \cup \\ & & \qquad \{(e \bullet\!\!\rightarrow) | e \in \ell^{-1}(x)\} \\ \delta_{In}(In, \lambda) = In & \textit{and} & \delta_{In}(In, x.\pi) = (\delta_{In}(In, \pi) \setminus \\ & & \qquad \{(e \rightarrow_{\%}) | e \in \ell^{-1}(x)\}) \cup \{(e \rightarrow_{+}) | e \in \ell^{-1}(x)\} \end{array}$$

Definition 3.9. (Marking update) *Let $D = (A, R, M, Act, \ell)$ be a DCR graph. We say D' has its marking updated according to a finite sequence of action labels π (noted $\delta(D, \pi)$) if $D' = (A, R, M', Act, \ell)$ where M' is a consistent with M and π. Note that there is at most one D' resulting from updating D with π.*

3.2. Labelled Transition Systems Control Problem

We reason operationally about DCR graphs using Labelled Transition Systems Magee and Kramer (1999). They are a canonical, compositional, representation of reactive systems ideally suited to model checking of business processes and synthesis of discrete event controllers.

Definition 3.10. (Labelled Transition System) *A* Labelled Transition System *(LTS) E is a tuple (S_E, L_E, Δ_E, s_0), where S_E is a finite set of states, $L_E \subseteq \mathcal{L}$ is its* communicating alphabet*, $\mathcal{L}$ is the universe of all observable events, $\Delta_E \subseteq (S_E \times L_E \times S_E)$ is a transition relation, and $s_0 \in S_E$ is the initial state. We define $\Delta_E(e)$ as $\{\ell \mid (e, \ell, e') \in \Delta_E\}$ and refer to a path of E is a sequence $p = s_0, \ell_0, s_1, \ell_1, s_2, \ldots$ where for every $i \geq 0$ we have $(s_i, \ell_i, s_{i+1}) \in \Delta_E$. A trace π is a sequence obtained by removing states from p. We say that E is deterministic if $(e, \ell, e') \in \Delta$ and $(e, \ell, e'') \in \Delta \implies e' = e''$, and is deadlock-free if for all $e \in S$ there exists $(e, \ell, e') \in \Delta_E$.*

Reactive systems are built compositionally. Such composition is built synchronizing events and propositions of the two LTS involved. We formalise the parallel composition as follows.

Definition 3.11. (Parallel Composition) *The parallel composition $E \| C$ of LTS $E = (S_E, L_E, \Delta_E, e_0)$ and $C = (S_C, L_C, \Delta_C, c_0)$ is an LTS $(S_E \times S_C, L_E \cup A_C, \Delta_{\|}, (e_0, c_0))$ such that $\Delta_{\|}$ is the smallest relation that satisfies the rules:*

$$\frac{(e, \ell, e') \in \Delta_E}{((e, c), \ell, (e', c)) \in \Delta_{\|}}\, \ell \notin L_C \qquad \frac{(c, \ell, c') \in \Delta_C}{((e, c), \ell, (e, c')) \in \Delta_{\|}}\, \ell \notin L_E$$

$$\frac{(e, \ell, e') \in \Delta_E,\ (c, \ell, c') \in \Delta_C}{((e, c), \ell, (e', c')) \in \Delta_{\|}}\, \ell \in L_E \cap L_C$$

3.3. Fluent Linear Temporal Logic

Linear Temporal Logic (LTL) is used to declaratively describe behaviour of reactive systems Pnueli (1977). We use a linear temporal logic of fluents to provide a uniform framework for specifying state-based temporal properties in event-based models Letier et al. (2005); Uchitel et al. (2004). FLTL Giannakopoulou and Magee (2003) is a linear-time temporal logic for reasoning about fluents. A *fluent* is defined by a pair of sets and a Boolean value: $f = \langle I, T, Init \rangle$, where $f.I$ is the set of initiating events, $f.T$ is a set of terminating events and $f.I \cap f.T = \emptyset$. A fluent may be initially *true* or *false* as indicated by $f.Init$.

Let $\mathcal{F}$ be the set of all possible fluents. An FLTL formula is defined inductively using the standard Boolean connectives and temporal operators **X** (next), **U** (strong until) as follows:

$$\varphi ::= f \mid \neg\varphi \mid \varphi \vee \psi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \psi$$

where $f \in \mathcal{F}$. We define $\varphi \wedge \psi$ as $\neg\varphi \vee \neg\psi$, $\Diamond\varphi$ (eventually) as $\top \mathbf{U} \varphi$, $\Box\varphi$ (always) as $\neg\Diamond\neg\varphi$, and $\varphi \mathbf{W} \psi$ (weak until) as $\varphi \mathbf{U} \psi \vee \Box\varphi$.

The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent f for a fluent definition d at position i, denoted $\pi, i \models_d f$, if and only if, one of the following conditions holds:

- $f_d.Init \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \Rightarrow \ell_j \notin f_d.T)$, or
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in f_d.I) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \Rightarrow \ell_k \notin f_d.T)$

In other words, a fluent holds at position i if and only if it holds initially or some initiating event has occurred, but no terminating event has yet occurred.

We say φ is a safety formula if there is a finite trace π such that:

$$\begin{array}{lcl}
\pi, i \models_d \neg\varphi & \triangleq & \neg(\pi, i \models_d \varphi) \\
\pi, i \models_d \varphi \vee \psi & \triangleq & (\pi, i \models_d \varphi) \vee (\pi, i \models_d \psi) \\
\pi, i \models_d \mathbf{X}\varphi & \triangleq & \pi, i+1 \models_d \varphi \\
\pi, i \models_d \varphi \mathbf{U} \psi & \triangleq & \exists j \geq i \cdot \pi, j \models_d \psi \ \ \wedge \ \ \forall\, i \leq k < j \cdot \pi, k \models_d \varphi
\end{array}$$

We use $\pi \models_d \varphi$, instead of $\pi, 0 \models_d \varphi$. We use $\models$ instead of $\models_d$ if from the context d is obvious.

3.4. Control Problems

Discrete event control aims to build an discrete event controller that satisfies given requirements under assumptions about the behaviour of the discrete event system to be controlled. We refer the system to be controlled as the environment. We adopt a formulation of control where the assumptions about the environment are described using LTL and LTS, and the controller to be automatically constructed is an LTS.

The notion of legality (based on Interface Automata de Alfaro and Henzinger (2001)) allows modelling controllability and monitorability of events. A legal LTS cannot block the occurrence of events that it does not control and cannot attempt actions that it controls but its environment can not accept.

Definition 3.12. (Legal LTS) *Let* $P = (S_P, L_P, \Delta_P, p_0)$ *and* $Q = (S_Q, L_Q, \Delta_Q, q_0)$ *be LTS,* $C \subseteq (L_P \cup L_Q)$ *be a set of events that* P *does control and* $U \subseteq (L_P \cup L_Q)$ *be a set of events that* P *does not control.*

We say that P *is a legal LTS for* Q *with respect to* (C, U) *if* $\forall\ (p, q) \in S_{P\|Q}$, p *and* q *are legal in the following sense:*

- $(\Delta_P(p) \cap U) = (\Delta_Q(q) \cap U)$, *and*
- $(\Delta_P(p) \cap C) \subseteq (\Delta_Q(q) \cap C)$.

Note that we adopt a slightly stronger notion than that of de Alfaro and Henzinger (2001). Here, we request the P not to be more robust (i.e. accept more uncontrollable events) than Q can exhibit.

Definition 3.13 (LTS Control D'Ippolito et al. (2013))**.** *Let* $E = (S_E, L_E, \Delta_E, e_0)$ *be an environment model in the form of an LTS,* $L_c \subseteq L_E$ *be a set of controllable events, and* G *be a controller goal in the form of an FLTL property. A solution for the LTS control problem with specification* $\mathcal{E} = (E, G, L_c)$ *is an LTS* C *such that* C *is legal with respect to* E *and controlled events* L_c, $E\|C$ *is deadlock free, and* $E\|C \models G$.

We are particularly interested in controllers that enable as many controlled actions as possible without compromising the satisfaction of the requirements. We refer to these as maximal controllers.

Definition 3.14 (Maximal Controller)**.** *Let* $\mathcal{E} = (E, G, L_c)$ *be an LTS Control Problem. We say that* C *is a maximal controller for* $\mathcal{E}$ *if* C *is a solution to* $\mathcal{E}$ *and for any other solution* C' *to* $\mathcal{E}$ *the traces of* C *include those of* C'.

Property 3.1 (Existence of Maximal Controllers)**.** *A maximal controller exists for any control problem* $\mathcal{E} = (E, G, L_c)$ *where* G *is a safety property.*

A dynamic update of controllers can be formulated as follows.

Definition 3.15 (DCU Problem)**.** *Let $\mathcal{E} = (E, \Box G, L_c)$ be an old specification, $\mathcal{E}' = (E', \Box G', L'_c)$ be a new specification, T be a safety FLTL formula, $R \subseteq (S_E \times S_{E'})$ be a mapping relation of states and, stopOldReq and startNewReq are special events denoting the ending of old and start of new requirements, respectively. A solution for the DCU Synthesis Problem is an LTS controller C_u such that:*
(a) $C_u \models (G \wedge \neg startNewReq)$ **W** $stopOldReq$*,*
(b) $C_u \models T$*,*
(c) $C_u \models \Box(startNewReq \rightarrow \Box G')$*, and*
(d) $C_u \models \Box(beginReconf \rightarrow (\Diamond\ stopOldReq \wedge \Diamond\ startNewReq))$

The output of a DCU problem is an LTS C_u where every trace satisfies that (a) the old requirements G hold, and the new requirements are not started, until *stopOldReq* is triggered, (b) the transition requirements hold, (c) the new specification G' must be valid from *startNewReq* is onwards, and (d) the update eventually happens.

The formulation above is from Nahabedian et al. (2018), except for condition (a) which we have strengthened to require *stopOldReq* before *startNewReq* as an overlap of specifications is not needed in this paper.

4. DCR Semantics as Control Problem

In this section we first show how to automatically build an LTS from a DCR graph such that it characterises all valid traces of the DCR graph. We restrict DCR graphs to those with no nesting (Nesting can be eliminated while preserving semantics Hildebrandt et al. (2011)). We reinterpret DCR semantics as a control problem in which a discrete event controller must enable and disable activities in such a way that its environment, as long as it only executes enabled activities, always satisfies the business process requirements as described in the DCR graph. We extract from a DCR graph a set of controllable events L_C, an LTS E, and a FLTL formula G such that controller synthesis (Definition 3.13) results in a controller that enforces the DCR graph semantics. Controllable events L_C are the activities enabling and disabling ones, while events modelling the execution of activities will be monitorable but not controllable. The LTS E

will model the assumptions the controller can rely upon to guarantee business requirements. Finally, the formula G encodes the domain specific aspects of the DCR graph, namely the arrows that establish dependencies between activities.

4.1. Controllable and Monitorable Events

We model the interaction between the controller and the environment as a turn based game. The controller first decides which activities to enable and disable and then lets the environment decide which of the enabled activities it will perform. Having finished performing an activity, the environment notifies the controller. It is then the controller's turn again to select enabled activities.

We use a controlled event *menu* to model the turn-based interaction where the controller offers to its environment a menu of activities to perform. First, the controller will select what activities to disable then it indicates, using *menu* that it is the environment's turn to decide what activity to execute.

We use a controlled event *enableAll* to model the controller enabling all activities of the DCR graph and controlled events $a\_disable$ for every for each atomic activity $a \in atom(A)$. We force the occurrence of *enableAll* at the start of the controller's turn, and then allow an arbitrary number of *disable* events. As we will discuss in Section 4.4, the synthesis algorithm minimises the number of events that the controller performs, thus it will execute the least possible number of *disable* events, hence leaving enabled all activities that are consistent with the business process requirements.

The only non-controlled events we introduce model the indication that an activity has been performed. That is, for each atomic activity $a \in atom(A)$ we introduce an event $a\_happened$.

With these events the controller can be understood as a workflow engine enforcing business rules. In the hospital example, the controller first decides which activities should be enabled (by first *enableAll* and then a sequence of *disabled* events) and then presents them to hospital staff (*menu*).The nurses and doctors perform an enabled activity and report back to the engine (*happened*). At this point, the controller will decide again what activities to enable and

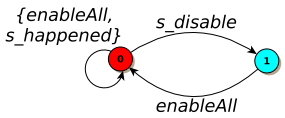

Figure 6: *Happens*(s) LTS constraining the occurrence of $s\_happened$.

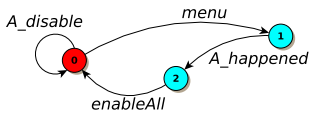

Figure 7: *Turns* LTS constraining controller and environment turns.

update the engine display.

Summarising, the set of controlled and monitored events is defined as follows.

Definition 4.1. (Controllable events from a DCR graph) *Let* $DG = (A, R, M)$ *be a DCR graph. The set of controllable events is* $L_C = \{a\_disable \mid a \in A\} \cup \{menu, enableAll\}$ *and the set of uncontrollable set is* $\overline{L_C} = \{\ a\_happened \mid a \in A\}$

4.2. Environment Model

The environment model, given as an LTS E, models the two assumptions that the controller can rely on to guarantee the requirements expressed by the DCR graph.

The first assumption is that activities can only happen when they are enabled. This can be modelled using one LTS model for each activity and composing them all in parallel. In Figure 6 we show an LTS, *Happens*(s), modelling the assumption for activity *sign*. State 0 models that *sign* is enabled (thus, the outgoing transition $s\_happened$) while state 1 models that the activity is disabled (i.e., there is no outgoing $s\_happened$ transition). Events *enableAll* and $s\_disable$ toggle between state 0 and 1. We assume the activity is initially enabled.

Figure 7 models the turn-based assumption: The controller chooses what activities may be executed without violating workflow requirements, and then, the environment picks which of the enabled activities is to be executed. The initial state (0) models the turn of the controller where any activity in A can be disabled. Event *menu* models when the controller relinquishes its turn offering a menu of activities to perform. State 1 is the environment's turn in which it can select only one activity in A to be executed, going to state 2. Here, all

activities are enabled with *enableAll* event to start again with controller's turn at state 0.

In conclusion the LTS environment E is defined as follows:

Definition 4.2. (Environment from a DCR graph) *Let $DG = (A, R, M)$ be a DCR graph, $Happens()$ be a (parametric) LTS like Figure 6 and $Turns$ be the LTS from Figure 7. The LTS environment is $E = Turns \parallel Happens(a_1) \parallel \ldots \parallel Happens(a_n)$ with $\{a_1, \ldots, a_n\} \in A$.*

4.3. Controller Goals

The controller goals must enforce the constraints between activities described in the DCR graph. Our encoding resembles that of Pesic and Van der Aalst (2006) where LTL formulas are used to formalise activity constraints of that appear in a DCR graph.

To facilitate expressing FLTL rules that capture the rules from Definition 3.2 that govern when an activity can be executed, we introduce fluents that capture the sets Ex, Re, and In. More specifically, for each activity $a \in A$ we introduce three fluents modelling if a belongs to sets Ex, Re, and In according to Definition 3.3. Note that although DCR graphs allow any combination of activities in Ex, Re and In in the initial marking, for simplicity, we assume that the initial marking of the DCR graph is such that $In = A$ and $Re = Ex = \emptyset$, which is the most common scenario.

- $a.Executed = \langle \{a\_happened\}, \emptyset, \bot \} \rangle$ models if $a \in Ex$: Initially no activity is in Ex and once in Ex it is never removed (see Definition 3.3a).
- $a.Required = \langle \{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, a\_happened, \bot \rangle$ models if $a \in Re$: All activities are initially not required and the execution of a activity makes it no longer required, and any activity in a response relation with a makes it a required (see Definition 3.3b). In the hospital example, fluent $s.Required$ is defined as $\langle \{pm\_happened, dt\_happened\}$, $s\_happened, \bot \rangle$ because activity *sign* is a response to *don't trust* and *prescribe medicine* according to Figure 2. Note that for cases where $a \bullet\!\!\rightarrow a$, we define $a.Required$ as $\langle \{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, \emptyset, \bot \rangle$ because the execution of a does not turn false the fluent.

- $a.In = \langle \{a'\_happened \mid a' \in (\rightarrow_{+} a)\}, \{a'\_happened \mid a' \in (\rightarrow_{\%} a)\}, \top \rangle$ models if $a \in In$ mimicking Definition 3.3c. Considering Figure 2, the fluent $gm.In$ is defined as $\langle \{s\_happened\}, \{dt\_happened\}, \top \rangle$.

We introduce FLTL formulas to preserve the rules that govern when an activity can be executed (i.e., is enabled) according to Definition 3.2. In other words, the formulas will relate the occurrence of $a\_happened$ with fluents $a'.Executed$, $a'.Required$, and $a'.In$ for all $a' \in A$.

- For rule (a) of Definition 3.2 we introduce for every activity $a \in A$ a formula $\alpha_a = (a\_happened \rightarrow a.In)$.
- For rule (b) of Definition 3.2 we introduce for all $a \in A$: $\beta_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \bullet a)} (a'.In \rightarrow a'.Executed))$. For instance, for *sign*, according to Figure 2 we have $\beta_s = (s\_happened \rightarrow (pm.In \rightarrow pm.Executed))$.
- For rule (c) of Definition 3.2 we introduce for each $a \in A$: $\kappa_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \diamond a)} (\neg a'.Required \vee \neg a'.In))$. For instance, $\kappa_{pm} = (pm\_happened \rightarrow (\neg et.Required \vee \neg et.In))$ for Fig 3.

Then, we can formalize requirement G for a DCR graph DG as follows:

Definition 4.3. (Requirements from a DCR graph) *Let $DG = (A, R, M)$ be a DCR graph, and, for each $a \in A$ we have the following fluents:*

- $a.Executed = \langle \{a\_happened\}, \emptyset, \bot \rangle$,
- $a.Required = \begin{cases} \langle \{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, \emptyset, \bot \rangle & if \quad a \bullet\!\!\rightarrow a \\ \langle \{a'\_happened \mid a' \in (\bullet\!\!\rightarrow a)\}, a\_happened, \bot \rangle & otherwise \end{cases}$
- $a.In = \langle \{a'\_happened \mid a' \in (\rightarrow_{+} a)\}, \{a'\_happened \mid a' \in (\rightarrow_{\%} a)\}, \top \rangle$

Requirements for the DCR graph DG is $\Box G = \Box \bigwedge_{a \in A} \alpha_a \wedge \beta_a \wedge \kappa_a$ where $\alpha_a = (a\_happened \rightarrow a.In)$, $\beta_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \bullet a)} (a'.In \rightarrow a'.Executed))$ and $\kappa_a = (a\_happened \rightarrow \bigwedge_{a' \in (\rightarrow \diamond a)} (\neg a'.Required \ \vee \ \neg a'.In))$.

4.4. Workflow Synthesis

In the previous sub-sections we defined all the elements needed to define a control problem that yields as a result an LTS that enforces execution of activities according to the semantics of a DCR graph. That is, we have a set

of controllable events L_c, an LTS environment E, and an FLTL formula G that can be used to define a control problem $\mathcal{E} = (E, \Box G, L_c)$; solutions to $\mathcal{E}$ are LTS that decide when to enable and disable activities such that when running with an environment that plays in turns and only executes enabled activities (as described in E) satisfies all business process requirements (as captured in G). In other words: $E \| C \models \Box G$ (Definition 3.13).

However, note that ensuring that $E \| C \models \Box G$ is not enough. We need the controller to be maximal in the sense of that at any *menu*, the maximal set of activities should be enabled that do not violate $\Box G$. Consider a workflow for the hospital in which after *sign* the only activity that is enabled is *give medicine*. The sequence *sign.give medicine* satisfies $\Box G$, but *sign* followed by *don't trust* should also be possible.

To ensure maximality we exploit a characteristic of the synthesis algorithm implemented in the MTSA tool D'Ippolito et al. (2008): MTSA builds eager controllers in the sense that they take the shortest route to satisfying their requirements. As the controller is forced to do *enableAll*, the synthesis algorithm will try to do as few disable actions as possible while still ensuring $\Box G$, thus a maximal number of activities will always be enabled.

In Fig 4 and 5 we show an abstract view of the controllers that MTSA builds for the control problems extracted from the DCR graphs depicted in Figure 2 and 3. The actual controllers synthesised for these graphs have 188 and 2291 states respectively and are too large to depict in this paper. What we depict in Figures 4 and 5 is the result of hiding events *enableAll*, disable and *menu* events and minimising with respect to weak bisimulation Milner (1980).

These abstract depictions of the controllers (which are built automatically by the MTSA tool) provide a view similar to that which actors Nurse and Doctor observe (i.e., they only observe the activities that are enabled and not the controllers incrementally deciding which activities to disable).

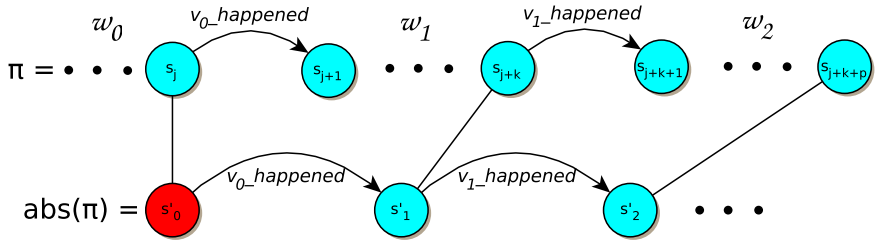

Figure 8: If C is a controller, then every trace $\pi \in C$ has an abstract trace $abs(\pi)$. For all $i \in \mathbb{N}$, w_i is a subtrace which only has disable, *enableAll* and *menu* events.

4.5. Correctness and Completeness

In the following we show that the control problem defined previously adequately builds controllers for a DCR Graph.

We first introduce the notion of an abstract trace of a controller. This consists of a trace in which only *happened* events are observed. In other words, we remove all controlled events from traces as we are interested in checking if the occurrence of activities in a trace is consistent with traces of a DCR Graph.

Definition 4.4 (Abstract traces in a Controller)**.** *Let $\mathcal{E}_D = (E, \Box G, L_C)$ be a control problem where E, G and L_C were extracted from D as described in this section, C be a solution to $\mathcal{E}_D$, and, π be a trace in C. The Abstract Trace of π in the Controller C is a trace $abs(\pi)$ in which disable events, menu and enableAll are hidden (see Figure 8). We refer to the LTS that has as its traces all the abstract traces of a controller C as $abs(C)$. LTS $abs(C)$ can be obtained by hiding disable events, menu and enableAll and then determinising.*

We now prove correctness by showing that for any trace in the abstract version of a controller that is a solution of our control synthesis problem, there is a run in the DCR Graph. We prove completeness by showing that any run in the DCR Graph has as a trace in the abstract version of a controller that is a maximal solution of the control synthesis problem. Recall from Definition 3.14 that a controller is maximal if it is maximal with respect to trace inclusion.

Theorem 4.1. *Let $D = (A, R, M_0, \ell, Act)$ be a DCR graph and $\mathcal{E}_D = (E, \Box G, L_C)$ be the LTS control problem extracted from D as described in this section.*

[Correctness] If C is a solution to $\mathcal{E}_D$ and π is a trace in C, where $abs(\pi) = v_0\_happened.v_1\_happened \ldots$, then $D \xrightarrow{v_0} D_1 \xrightarrow{v_1} D_2 \ldots \in runs(D)$.

[*Completeness*] *If C is a maximal solution to $\mathcal{E}_D$ and $D \xrightarrow{v_0} D_1 \xrightarrow{v_1} D_2 \ldots \in runs(D)$ then there exists a trace $\pi \in C$ such that $abs(\pi) = v_0\_happened.\ v_1\_happened \ldots$.*

Correctness Proof. As C is an LTS that is the solution to the control problem $\mathcal{E}_D$, we know that $E \| C \models \Box G$ (see definition 3.13). As C is legal w.r.t E (definition 3.12) its traces must be a subset of those of E. Thus, it is also the case that $C \models \Box G$. Therefore, for all $j \in \mathbb{N}_0$, $\pi, j \models G$. By Lemma 4.2, we know that$D \xrightarrow{v_0} D_1 \xrightarrow{v_1} D_2 \ldots \in runs(D)$. □

Completeness Proof. We assume $D \xrightarrow{v_0} D_1 \xrightarrow{v_1} \ldots$ is a run in D. We need to prove that there exist a trace $\pi \in C$ such that $\pi = w_0.\ v_0\_happened.\ w_1.\ v_1\_happened \ldots$ where subtraces w_i do not contain happened events (see Figure 8). If so, then we know that $abs(\pi) = v_0\_happened.v_1\_happened \ldots$ because of Definition 4.4.

To prove the existence of π, we need to prove that (BASE) there exist w_0 with no happened events such that $w_0.v_0\_happened$ is prefix of π and (IND) given $w_0 \ldots w_{i-1}$ with no happened events such that $w_0.v_0\_happened \ldots w_{i-1}.\ v_{i-1}\_happened$ is prefix of π, then there exist a subtrace w_i with no happened events such that $w_0.v_0\_happened \ldots w_i.v_i\_happened$ is a prefix of π.

To prove (BASE), let π be a trace of C. As C is legal with respect to E (Definition 3.13) we have that π is a trace of E. By construction of E, π must be a trace that starts with a finite amount of disables events (between 0 and $|A|-1$), followed by a *menu* event. We call this prefix w. The state of C having run w will have enabled transitions labelled $a\_happened$ for all events a that were not disabled in w (see Figure 6).

We must show that $v_0\_happened$ is enabled at this state. Let us assume that it is not and we will reach a contradiction. Because C is maximal, if $v_0\_happened$ is not enabled, then there must be no strategy for the controller at the state in E reached **after** $w.v_0\_happened$. This is equivalent to stating that the control problem extracted from D_1 has no solution. However, there is a run $D_1 \xrightarrow{v_1} D_2 \xrightarrow{v_2} D_3 \ldots$ in $runs(D_1)$ and by Lemma 4.1 the control problem extracted from D_1 has a solution, reaching a contradiction.

Now we prove (IND): We have that $D \xrightarrow{v_0} \ldots D_{i-1} \xrightarrow{v_{i-1}} D_i \xrightarrow{v_i}$ is a prefix of an infinite run in D. We assume that C has a trace π with prefix $\overline{\pi} = w_0.\ v_0\_happened.\ w_1.v_1\_happened \ldots\ v_{i-1}\_happened$ where w_j have no happened events. We will show $\overline{\pi}.w_i.v_i\_happened$ is a prefix of a trace in C.

Let c be the state reached in C after $\overline{\pi}$. As in the BASE case, all traces in C starting with $\overline{\pi}$ have a finite sequence of *enableAll*, *disable* and *menu* events before the next happened event. Let w_i the longest of such sequences such that $\overline{\pi}.w_i$ is a prefix of π. Let c' be the state reached in C having run $\overline{\pi}.w_i$. We assume that c' does not have $v_i\_happened$ enabled and reach a contradiction. The reasoning is as with the BASE case. □

The following lemma provides a sufficient condition for realisability of a control problem $\mathcal{E}_D$ for a DCR graph D.

Lemma 4.1. *Let D be a DCR graph, and $\mathcal{E}_D$ be the control problem extracted from D. If there is an infinite trace in $traces(D)$ then there is a solution to $\mathcal{E}_D$.*

Proof. Given an infinite run $D \xrightarrow{v_0} D_1 \ldots$, we build an LTS that is a solution to the control problem $\mathcal{E}_D$ and that has as its only abstract trace: $v_0\_happened.$ $v_1\_happened \ldots$. We build the LTS in two steps. First, one in which only has happened events in its alphabet and has as its only trace: $v_0\_happened.$ $v_1\_happened \ldots$. This can be done with a finite set of LTS states as each state is considered to model a marking $M_0.M_1 \ldots$ where M_i is the marking of D_i. (Note that there are a finite number of markings). From this LTS we add between every $v_i\_happened, v_{i+1}\_happened$ the trace $enableAll.w.menu$ where w has exactly one disabled event for every activity except v_{i+1}. It is straightforward to show that the resulting LTS is a solution to $\mathcal{E}_D$. □

The following lemma states that if a finite trace satisfies the controller goal G for a control problem $\mathcal{E}_D$ then its abstraction is a trace of a DCR graph D.

Lemma 4.2. *Let $D = (A, R, M_0, Act, \ell)$ be a DCR graph, π be a trace with alphabet defined as $L_C \cup \overline{L_C}$ from Definition 4.1, and $k, \hat{k} \in \mathbb{N}_0$ such that $\hat{k}$ is the position of the k^{th} happened event in π. If for all $i \leq \hat{k}$, $\pi, i \models G$, and $abs(\pi) = v_0\_happened \ldots v_k\_happened$, then, $D \xrightarrow{v_0} \ldots D_k \xrightarrow{v_k} D_{k+1} \in runs(D)$, where fluents in G are initialised according to marking M_0 (as in Definition 4.3).*

Proof. We prove this Lemma using induction over the length of a run in D. We want to prove that (BASE) $D \xrightarrow{v_0}$ and (IND) if $D_0 \xrightarrow{v_0} D_1 \xrightarrow{v_1} \ldots D_{i-1} \xrightarrow{v_{i-1}} D_i$ with $i \leq k$ is a run in D, then, $D_i \xrightarrow{v_i}$. We refer to M_i as the marking for DCR graph D_i.

To prove (BASE) we will show that v_0 is enabled at marking M_0 because rules (a)-(c) from Definition 3.2 are satisfied.

Rule (a) is true iff $v_0 \in In_0$. As initial marking was defined to have every activity in A we have $v_0 \in In_0$.

Rule (b) is true iff $(In_0 \cap (\rightarrow\bullet v_0) \subseteq Ex_0)$. As $In_0 = A$ and $Ex_0 = \emptyset$ this holds iff $(\rightarrow\bullet v_0) = \emptyset$.

Assume that there exists an activity b such that $b \rightarrow\bullet v_0$. As for all $j \in \mathbb{N}_0$, we know that $\pi, j \models G$, then, for all $a \in A$, the formula $\alpha_a \wedge \beta_a \wedge \kappa_a$ must be true (see Definition 4.3). In particular, β_{v_0} must be true after the execution of the first happened event (i.e., the $|w_0| + 1$ event of π, see Figure 8): $\pi, |w_0| + 1 \models \beta_{v_0}$. Recall $\beta_{v_0} = v_0\_happened \rightarrow (b.In \rightarrow b.Executed)$ and that $\pi, |w_0| + 1 \models v_0\_happened$. Thus, the consequent of β_{v_0} should be true. Since w_0 does not have any happened event, then, at position $|w_0|+1$, fluents $b.In$ and $b.Executed$ still have their initial value (true and false respectively) for all $b \neq v_0$. Thus, if $b \neq v_0$, the consequent does not hold, reaching a contradiction. If $b = v_0$ then we have that $b\_happened$ and that $b \rightarrow\bullet b$ which is also a contradiction.

Rule (c) is true iff $Re_0 \cap In_0 \cap (\rightarrow_\diamond v_0) = \emptyset$. As Re_0 was defined to be empty (initial marking), then this rule holds.

Therefore, rules (a)-(c) from Definition 3.2 are guaranteed for activity v_0 at marking M_0. Then, $D \xrightarrow{v_0}$.

Now we prove (IND): $D_i \xrightarrow{v_i}$, assuming $D_0 \xrightarrow{v_0} D_1 \xrightarrow{v_1} \dots D_{i-1} \xrightarrow{v_{i-1}} D_i$ with $i \leq k$. Hence, we need to prove that v_i is enabled at marking M_i. By Definition 3.2, we need to prove that:
(a) $v_i \in In_i$
(b) $In_i \cap (\rightarrow\bullet v_i) \subseteq Ex_i$
(c) $Re_i \cap In_i \cap (\rightarrow_\diamond v_i) = \emptyset$

As $abs(\pi) = v_0\_happened \dots v_i\_happened \dots v_k\_happened$, there must be a position j such that $v_i\_happened$ is the j event in π (i.e., $\pi, j{+}1 \models v_i\_happened$). We also know that $\pi, j+1 \models G$. Thus, the consequent of formulas α_{v_i}, β_{v_i} and κ_{v_i} must be true at position $j+1$ (see Definition 4.3).

Thus, we have:
I) $\pi, j+1 \models v_i.In$
II) $\pi, j+1 \models \bigwedge_{a \in (\bullet\rightarrow v_i)} (a.In \rightarrow a.Executed)$
III) $\pi, j+1 \models \bigwedge_{a \in (\rightarrow_\diamond v_i)} (\neg a.Required \vee \neg a.In)$

By I) and Lemma 4.3 we have that activity v_i is included at marking M_i ($v_i \in In_i$). Hence, rule (a) from Definition 3.2 is guaranteed.

By II) and Lemma 4.3 we have that for all $a \in (\rightarrow\bullet v_i)$ either $a \notin In_i$ or $a \in Ex_i$. Thus, the following holds: $In_i \cap (\rightarrow\bullet v_i) \subseteq Ex_i$.

By III) and Lemma 4.3 we have that for all $a \in (\rightarrow_\diamond v_i)$ either $a \notin Re_i$ or $a \notin In_i$. Thus, the following holds: $Re_i \cap In_i = \emptyset$.

As rules (a), (b) and (c) from Definition 3.2 are valid for marking M_i and activity v_i, we conclude that v_i is enabled at marking M_i (i.e. $D_i \xrightarrow{v_i}$). □

The following lemma states that fluents $a.In$, $a.Executed$ and $a.Required$ mimick correctly the state of a marking.

Lemma 4.3. *Let π be a trace in C and $abs(\pi) = v_0\_happened.v_1\_happened \dots$ D be a DCR graph such that $D_0 \xrightarrow{v_0} D_1 \dots D_{i-1} \xrightarrow{v_{i-1}} D_i$ is a run of D reaching marking M_i. Let j such that $j = 0$ or $\pi, j \models v_i\_happened$. For all activity $a \in atom(A)$,*

- $\pi, j \models a.In$ *iff* $a \in In_i$
- $\pi, j \models a.Executed$ *iff* $a \in Ex_i$
- $\pi, j \models a.Required$ *iff* $a \in Re_i$

Proof. This follows straightforwardly from the fact that the definition of fluents $a.In$, $a.Executed$, and $e.Required$ (Definition 4.3) updates their values mimicking the execution semantics of DCR graphs (Definition 3.3). □

We have proved that the runs of a DCR graph D and the traces (of executed activities) in C are equivalent. This result can be extended trivially to traces in D if we apply function ℓ to each activity.

Corollary 4.1. *Let $D = (A, R, M, \ell, Act)$ be a DCR graph and $\mathcal{E}_D = (E, \Box G, L_C)$ be the LTS control problem extracted from D as described in this section.*

- [*Correctness*] *If C is a solution to $\mathcal{E}_D$ and π is a trace in C, where $abs(\pi) = v_0\_happened.v_1\_happened\ldots$, then $\ell(v_0).\ell(v_1)\ldots \in traces(D)$.*
- [*Completeness*] *If C is a maximal solution to $\mathcal{E}_D$ and $\ell(v_0).\ell(v_1)\ldots \in traces(D)$ then there exists a trace $\pi \in C$ such that $abs(\pi) = v_0\_happened.\ v_1\_happened\ldots$.*

5. Controlling Business Process Reconfiguration

In this section we show how to compute, using Dynamic Controller Update (Definition 3.15), a reconfiguration strategy that guides the execution of workflow instances from satisfying the old business requirements to satisfying the new ones while ensuring that all transition requirements are satisfied.

For this purpose, we first discuss how domain specific transition requirements for a business reconfiguration can be described using FLTL. This involves introducing two new events. We then discuss what a solution to a reconfiguration problem may look like and how such solutions can be built automatically solving a Dynamic Controller Update problem. Finally, we show how to use the output of a Dynamic Controller Update to build a DCR graph representing the dynamic reconfiguration of business processes.

5.1. Specification of Transition Requirements

Consider the scneario discussed in Section 2 for a Hospital process in which a transition requirement stating that activity *examine tests* should be forced when reconfiguring. Alternatively, this requires that before the new business process requirements are to be enforced, activity *examine tests* is required.

To formalise this transition requirement we introduce an event that represents when the old business requirements are to be dropped (*stopOldReq*) and when the new business requirements are to come into force (*startNewReq*). With these, the transition requirement can stated as: when *stopOldReq* occurs, all activities are prohibited except for *examine tests* until both *examine tests* and *startNewReq* have occurred. In FLTL this is as follows.

$T_h = \Box(stopOldReq \rightarrow ((\bigwedge_{a \in A \setminus \{et\}} \neg a\_happened) \ \mathbf{W} \ (et.Executed \wedge startNewReq)))$. Note that guaranteeing this formula requires enabling and disabling activities such that the uncontrolled events $a\_happened$ can occur as required by T_h.

Two examples of *domain independent transition requirements* are immediate and delayed reconfiguration Ellis et al. (1995). The immediate reconfiguration requirement can be formalised as follow: $T_{Imm} = beginReconf \implies ((\bigwedge_{a \in A} \neg a\_happened) \ \mathbf{W} \ startNewReq)$. In other words, as soon as the reconfiguration is required, no activities are allowed until the controller signals that the new business process requirements are guaranteed. The delayed reconfiguration requirement relaxes immediate reconfiguration by allowing postponement of *startNewReq* but requiring that any activities that occur before it comply to the old business process requirements. This requirement can be formalised as follows: $T_{\emptyset} = \Box((StopOldReq \wedge \neg StartNewReq) \rightarrow \bigwedge_{a \in A} \neg a\_happened)$, where *StopOldReq* and *StartNewReq* are fluents that are initially false, become true with *stopOldReq* and *startNewReq*, and never become false again.

Formally, we require transition requirements to be FLTL safety properties that only predicate over *happened* events plus *stopOldReq*, *startNewReq*, and *beginReconf* and that do not use the temporal operator $\mathbf{X}$ (i.e., they are stutter invariant Lamport (1994)).

5.2. Reconfiguration Workflows

Returning to T_h, what would a valid reconfiguration strategy that complied to this transition requirement be? Assume the workflow in Figure 4 is in state 2, a solution to the reconfiguration is to deploy a workflow that does force *examine tests* and then reaches a state 10 in Figure 5. Our goal is to automatically build a workflow that manages the transition between these two workflows consistently with respect to a transition requirement. We call this workflow a *reconfiguration workflow*.

This reconfiguration workflow that assumes that the old workflow is in state 2 is inadequate as, before it takes control, a new activity (e.g., *give medicine*)

may be executed taking the old workflow (Figure 4) in state 2 to state 3. Should this happen then the reconfiguration should force *examine tests* and then move to state 13 in Figure 5 instead of 10. Thus, the goal is to build a reconfiguration workflow that can manage the transition from any state in the old workflow.

Conceptually, our solution builds one reconfiguration workflow that consists of three phases. The first is structurally equivalent to the old workflow (modulo a new event *beginReconf*). This allows hot-swapping the old workflow with the reconfiguration workflow, and setting the initial state of the latter according to the current state of the former. The second phase is triggered by an event *beginReconf*. At this point, the reconfiguration workflow may start to deviate from the behaviour of the old workflow to ensure transition requirements. At the point it does so, it must first signal *stopOldReq*. The third phase is one in which the new workflow requirements are satisfied. Entering this third phase is signalled with *startNewReq*.

In Figure 9 we depict an abstract reconfiguration workflow (enabling, disabling and *menu* events are hidden) that implements the reconfiguration from business process requirements of Figure 2 to those of Figure 3 under transition requirement T_h. The blue rectangle on the left represents the first phase of the reconfiguration workflow. Note that the structure of states and transitions is that of the workflow to be replaced (Figure 4), thus hotswapping this workflow in is trivial. Note that all states in the blue region have an outgoing transition labelled *beginReconf*. When *beginReconf* is triggered, no matter what the current state is, there is a path to the yellow region on the right. The yellow region represents the new workflow as in Figure 5. The transition from the old requirements to new ones, while satisfying the transition requirements is represented by between both rectangles. It is noteworthy that there are no loops during the transition phase which guarantees that eventually the new business process requirements will be enforced.

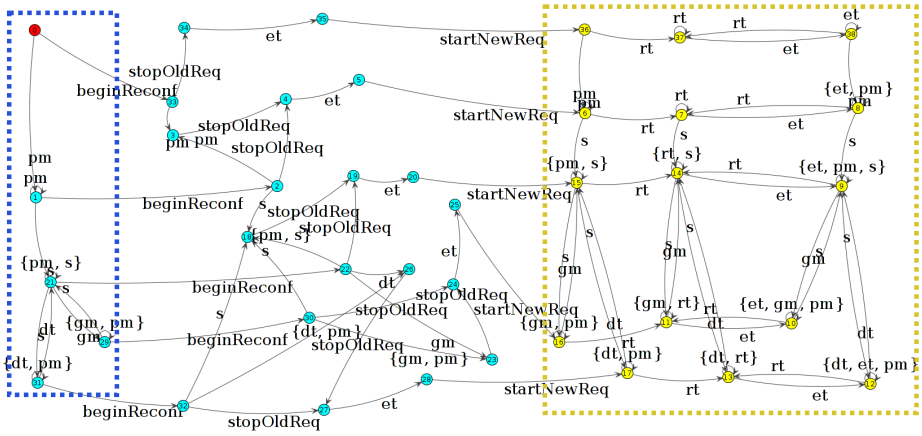

Figure 9: Reconfiguration workflow with transition requirement T_h.

5.3. Automatic Construction of Reconfiguration Workflows

Summarising, Figure 9 represents a strategy to solve the problem of reconfiguring business process requirements in Figure 2 to those of Figure 3 under transition requirement T_h. We now discuss how such solution can be built by solving a DCU problem Definition 3.15. The DCU problem requires two control problems $\mathcal{E}_D = (E, G, L_c)$ and $\mathcal{E}_{D'} = (E', G', L'_c)$ which represent in this case the old and new business process synthesis problems as described in Section 4, starting from DCR graph D and D'. Dynamic Controller Update also requires a transition requirement T and a state mapping R from the states of E to those of E'. We have discussed T, we now discuss R.

The purpose of relation R is to explain the relationship between the assumptions modelled in each control problem. The issue is that E tracks assumptions for a controller synthesised from C, when a reconfiguration is deployed it is not possible to know what the state of the assumption E' is. R must be provided by a user to address this problem. In this setting, the mapping can be trivially defined as the only differences between E and E' are the LTSs (like the one in Figure 6) representing activities that are present in one business process and not the other. Furthermore, we know that for any new activity, this one can never have been enabled by the controller of the old workflow. In consequence,

R can be defined as the state identity relation for all LTS that are in E and E' and as the constant relation 0 (i.e., the initial state) for LTSs representing new activities. We assume the disjoint union of D and D'. Although an activity in D and D' may have the same name (and they represent the same operation in the business process domain), we treat each shared activity between D and D' as two different ones. The reason of doing this, is that each activity may change its relations (arrows) with the rest of the activities when changing from D to D'. Thus, enabling and disabling an activity may change based on the new model D'. This is also the case when we consider fluents generated for G and G'.

Thus, given two DCR graphs D and D' describing the old and new business process requirements and a transition requirement T we can automatically build control problems $\mathcal{E}_D = (E, G, L_c)$ and $\mathcal{E}_{D'} = (E', G', L'_c)$ as described in Section 4 and R to describe and solve a DCU problem. An abstraction of the solution to the DCU problem for the Hospital reconfiguration problem with T_h described above is depicted in Figure 9 and was built automatically using MTSA.

An important methodological note is that not every DCU problem has a solution. It is possible to provide two control problems $\mathcal{E}_D$ and $\mathcal{E}_{D'}$ that are individually realisable yet for certain transition requirements, the update is impossible. Note that the automated procedure for solving DCU problems Nahabedian et al. (2018) is complete. This means that if it fails to produce a solution it will report so to the user that no solution to the problem exists.

In terms of business process reconfiguration non-realizability means that it is possible to start with two sets of business process requirements that are consistent yet to propose a transition requirement that is too stringent to allow for a correct reconfiguration. An example of this, for the Hospital example, is to require $T = \Box(\mathit{startNewReq} \rightarrow \neg \mathit{pm.Executed})$. There is no reconfiguration strategy that can guarantee that the new business process requirements will be put in force *independently of the current state* of the live instances of the old workflow, in this case that activity *prescribe medicine* has not been executed.

Note that if reconfiguration is possible, the synthesis algorithm will produce a controller that attempts to achieve its liveness goals (i.e., $\Box(\mathit{beginReconf} \rightarrow (\Diamond\ \mathit{stopOldReq} \wedge \Diamond\ \mathit{startNewReq}))$) as soon as possible. In other words, events *stopOldReq* and *startNewReq* will happen as early as possible.

5.4. Reconfiguration DCR Graphs

Having discussed how to build a reconfiguration workflow, we now show how the reconfiguration strategy for one particular instance of the workflow can be shown as a DCR graph which we refer to as a Reconfiguration DCR. We define first Reconfiguration DCRs and then introduce an automatic process to build them.

The reconfiguration workflows discussed previously encode a reconfiguration strategy from every state in which an instance that is following the old business process rules might be in. In order to provide feedback to a user that specified the business rules in DCR, it is convenient to show them how one particular instance should continue to be treated to adequately reconfigure business process.

Thus, we assume that we have a particular trace α that satisfies $\alpha \in \mathit{traces}(D)$ and corresponds to the activities that one workflow instance has gone through. Given this trace, the execution of a Reconfiguration DCR should continue α guaranteeing rules from Definition 3.15. That is, we require that any trace of the Reconfiguration DCR should continue α with some behaviour c while adhering to the DCR D and then after a transition period t it should only allow behaviour r that adheres to D'. Note that r does not need to be a trace of D' as some activities required by D' may have already occurred before adherence. We require r to be a trace of D' with its initial marking updated to be consistent with what occurred in $\alpha.c.t$.

Note that to update D' with an appropriate initial marking it is necessary to be able to map activity labels in traces of D with activities in D'. To simplify presentation we assume that the labelling functions of D and D' are injective and it is the same activity in D and D' that gets mapped to a particular label.

Definition 5.1. (Reconfiguration DCR) *Let* $D = (A, \triangleright, R, M_0, \mathit{Act}, \ell)$ *and* $D' = (A', \triangleright', R', M_0', \mathit{Act}', \ell')$ *be DCR graphs for the old specification and the new spec-*

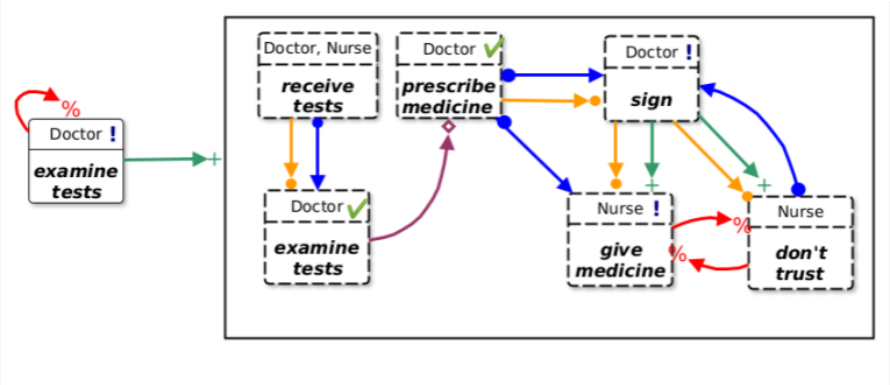

Figure 10: Reconfiguration DCR for transition requirement T_h and $\alpha.c.t = pm.et$. We use a shorthand to avoid an inclusion arrow from the left-most activity to all the rest.

ification, respectively, where $M_0 = (Ex_0, Re_0, In_0)$ *and* $M'_0 = (Ex'_0, Re'_0, In'_0)$, T *be an FLTL formula representing a transition requirement, and,* α *be a trace in* D. *We require* ℓ *and* ℓ' *be injective and that activities are labelled consistently in both DCRs (i.e.,* $a \in A$ *and* $a' \in D'$ *if* $\ell(a) = \ell; (a')$ *then* $a = a'$*). A Reconfiguration DCR for a trace* α *is a DCR graph* DR^{α} *such that for every infinite acceptance trace* $\omega \in traces(DR^{\alpha})$ *there exists finite traces* c *and* t *and an infinite trace* r *such that* $\omega = c.t.r$ *with* i) $\alpha.c \in traces(D)$, ii) $\alpha.beginReconf.c.stopOldReq.t.startNewReq.\ r \models T$, *and,* iii) r *is an acceptance trace of* $\delta(D', \alpha.c.t)$. *(Definition 3.9).*

The definition above only considers infinite accepting runs of a reconfiguration DCR. Recall that we assume for all DCRs, and thus D', that any run can be extended to an infinite run to allow evaluating LTL properties over infinite runs.

Recall the hospital workflow example discussed in Section 2 and its reconfiguration workflow is depicted in Figure 9. For the case in which the medicine was prescribed but nothing else was executed ($\alpha.c.t = pm.et$), a Reconfiguration DCR is depicted in Figure 10. Note that this DCR graph forces *examine tests* before moving into a run that complies to the new business rules. This corresponds to the transition period depicted in Figure 9 as the path through states 1, 2, 4, 5, 6. Also note that new DCR graph, depicted in a dashed box, has a different marking than the original DCR (see Figure 3):

The *prescribe medicine* activity is marked as executed. This is consistent with $\alpha.c.t$ = *prescribe medicine.examine tests*. Note that Definition 3.8) uses the inverse of the relabelling function of the new DCR graph to map $\alpha.c.t$ to activities, thus marking as executed the two boxes of Figure 9 that correspond to labels *prescribe medicine* and *examine tests*. Should the labelling function not be injective, then one label in $\alpha.c.t$ may impact the marking of several activities in the new DCR graph. This is a conservative choice, as it is impossible to know which of the activities in the new graph correspond to a label that originates from the old DCR graph. Thus, the use of non-injective labelling functions in the new DCR graph must be done with care.

5.4.1. Building a Reconfiguration DCR graph

In Algorithm 1 we show how to produce a Reconfiguration DCR graph using old and new DCR graphs (D and D', respectively), a trace $\alpha \in traces(D)$, and a solution to a DCU synthesis problem (C_u) for $\mathcal{E}_D$ and $\mathcal{E}_{D'}$ and a transition requirement T. Roughly, we replay α in C_u (lines 2-8) and then build a DCR graph that has exactly the same behaviour as C_u from then on to the point where the transition period has ended (i.e. *startNewReq* have occurred). This is done in lines 9-30. We then add to the DCR graph a copy of the new DCR graph with an initial marking consistent with the events seen so far in C_u (line 30-38). The resulting DCR graph follows a structure depicted in Figure 11.

We now present the algorithm in slightly more detail. Lines 2-8 consume the trace α. For each action label in α it advances the current state of C_u and changes the marking of D following Definition 3.3 to reflect that the activity has occurred. To do so, at Line 5, it is necessary to get the activity that corresponds to each action label from α. As D and D' are deterministic DCR graphs (Definition 3.7), we know that there are only one enabled activity for a given action label. At Line 9, the current state of C_u is the state at which the reconfiguration should begin. Hence, the next event in C_u which we consume is *beginReconf*. The behaviour of C_u from this point on is what must be captured by the Reconfiguration DCR.

ALGORITHM 1: Pseudocode for building a DCR Reconfiguration.

```
1  DCRReconfiguration (α, C_u, D, D')
2      resultDCR ← DCR();
3      history ← α ;
4      foreach label in α do
5          Let activity ∈ (D.ℓ^{-1}(label) ∩ D.enabled());
6          C_u.executeHappened(activity);
7          D.execute(activity); // update marking as Definition 3.3
8      end
9      C_u.execute("beginReconf");
10     lastNode ← Null;
11     event ← C_u.getEnableAbsEvent();
12     while event is not "startNewSpec" do
13         if event is not "stopOldSpec" then
14             activity ← event.toActivity();
15             D.execute(activity);
16             if D.hasActivity(activity) then
17                 history.append(D.ℓ(activity));
18             else
19                 history.append(D'.ℓ(activity));
20             if lastNode is not Null then
21                 newNode ← resultDCR.addNode(activity);
22                 resultDCR.addCondition(lastNode, newNode);
23                 lastNode ← newNode;
24             else
25                 lastNode ← resultDCR.addNode(activity);
26             resultDCR.addExclusion(lastNode, lastNode);
27             resultDCR.setPending(lastNode);
28         C_u.execute(event);
29         event ← C_u.getEnableAbsEvent();
30     end
31     if lastNode is Null then
32         return D';
33     D'.setMarkingConsistentTo(history);
34     includedActivites ← D'.getIncludedActivities();
35     resultDCR.addDCRGraph(D');
36     resultDCR.addInclusions(lastNode, includedActivities);
37     resultDCR.excludeActivities(includedActivities);
38     return resultDCR;
```

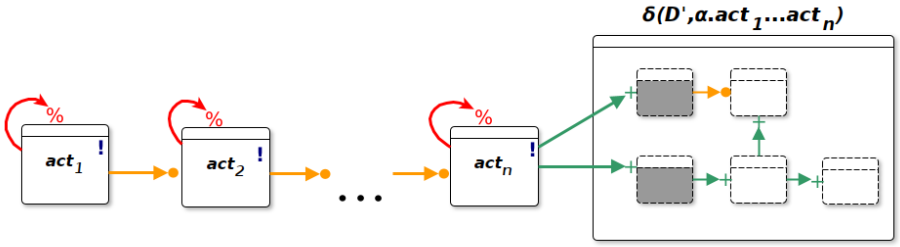

Figure 11: DCR reconfiguration schema for D, D', and a trace α. Activities $act_1 \dots act_n$ represent the reconfiguration strategy to follow before reconfiguring to D'. Right hand side corresponds to D' with an updated marking according to $\alpha.act_1 \dots act_n$, except that activities included in the initial marking of $\delta(D, \alpha.act_1 \dots act_n)$, depicted as shadowed activities, are also excluded but with an inclusion edge from act_n.

From now on, the algorithm must traverse C_u to collect the behaviour it exhibits until the transition period is over. To do so, it uses *getEnableAbsEvent*() to obtain a *happened*, *stopOldReq* or *startNewReq* event that is available at the current state of C_u. The current state of C_u will change inside a while loop ($C_u.execute(event)$ in Line 28) which stops when *event* is *startNewReq*. Inside the loop, the algorithm adds to the Reconfiguration DCR one activity at a time following a path in C_u, except for *stopOldReq* event that is filtered in Line 13. Each activity has a corresponding action label that must be added to *history* and it can be acquired with function ℓ. However, as we do not know if the activity is in D or in D', the algorithm, at Line 16, does so. Later, in lines 20-27 each activity is added to the Reconfiguration DCR in such a way that it is chained to previous activities using a Condition relation (i.e $a_i \rightarrow\bullet a_{i+1}$ for all $i \geq 0$). Furthermore, for each of these added activities we include an exclusion relation (Line 26) onto themselves to prohibit them being executed more than once, and they are set as pending to require their execution (Line 27).

Note that the main loop terminates as it is guaranteed that any trace of C_u satisfies rules (a) and (d) from Definition 3.15. Rule (a) assures $C_u \models (G \wedge \neg startNewReq)$ **W** *stopOldReq* (i.e., *startNewReq* is prohibited before *stopOldReq*) and rule (d) assures $C_u \models \Box(beginReconf \rightarrow (\Diamond\ stopOldReq \wedge \Diamond$

startNewReq)) (i.e., if *beginReconf* occur, then, *startNewReq* event will eventually occur). The while loop is traversing a trace of C_u from *beginReconf* until *startNewReq* which exist because these two rules.

By Line 33, the transition period in C_u has ended and we change the marking of D' according to $history$ to build DCR graph $\delta(D', history)$ (see Definition 3.9). We add this DCR graph to $resultDCR$ relating last activity from the transition period with inclusion edges to those activities that are included in D'. Finally, we set all activities from D' as not included to allow the transition period prevail. Since we set the inclusion edges, D' becomes active when transition period is over.

5.4.2. Algorithm Soundness

In the following we state and prove the soundness of Algorithm 1. By soundness we mean that Algorithm 1 indeed produces a DCR Reconfiguration from Definition 5.1.

Theorem 5.1. *Let D and D' be DCR graphs for the old specification and the new specification, respectively, T be an FLTL formula representing a transition requirement, α be a trace in D, C_u be a solution to the DCU problem defined by transition requirement T and translating D and D' to $\mathcal{E}_D$ and $\mathcal{E}_{D'}$, respectively. If DR^α is the output of Algorithm 1 with inputs $(\alpha, abs(C_u), D, D')$, then DR^α is a DCR Reconfiguration for trace α from Definition 5.1.*

Proof. Let ω be a trace of DR^α, $D = (A, \triangleright, R, M_0, Act, \ell)$ and $D' = (A', \triangleright', R', M'_0, Act', \ell')$. We will show that there exists finite traces c and t and an infinite trace r such that $\omega = c.t.r$ with *i)* $\alpha.c \in traces(D)$, *ii)* $\alpha.beginReconf.c.stopOldReq.t.startNewReq.r \models T$, and, *iii)* r is an accepting run in $(A', \triangleright', R', M', Act', \ell')$ where M' is consistent with M'_0 and $\alpha.c.t$.

Following lines 2-33 in Algorithm 1, it is easy to see that there exists a trace $\pi \in C_u$ such that $abs(\pi) = \alpha'.beginReconf.c'.stopOldReq.\ t'.startNewReq.r'$. We define α, c, and t by replacing every $a_i\_happened$ in α', c', and t' with a_i.

Following Definition 3.15 we know that (a) $\pi \models G\ \mathbf{W}\ stopOldReq$, (b) $\pi \models T$, (c) $\pi \models \Box(startNewReq \rightarrow G')$ and (d) $\pi \models \Box(beginReconf \rightarrow (\Diamond\ stopOldReq \wedge \Diamond\ startNewReq))$.

From (a) we know that there exists $k \in \mathbb{N}_0$ (position where *stopOldReq* occurs in π) such that for all position $i \leq k$, $\pi, i \models G$. Thus, we can use Lemma 4.2 to say that $\alpha'.beginReconf.c'$ is a trace with "happened" events that can be executed in D without *beginReconf*. Hence, $\alpha.c \in traces(D)$.

From (b) and the fact that T does not refer to the events abstracted by abs and does not use the $\mathbf{X}$ temporal operator, we can conclude that $abs(\pi) \models T$.

Finally, from (c) we have that $\pi, j \models G'$ for all $j > k$ where k is the position of *startNewReq* in π. This is equivalent to saying that $r', 0 \models G'$ with fluents initialised to reflect their valuation at position k of π. More precisely, if $\pi, j \models_d G'$ for all $j > k$ then $r', 0 \models_{d'} G'$ where fluent definition d' is such that for all fluent f, $\pi, k \models_d f$ if and only $f_{d'}.Init$.

Let $M'_0 = (Ex'_0, Re'_0, In'_0)$. We define $M' = (\delta_{Ex}(Ex'_0, \alpha.c.t), \delta_{Re}(Re'_0, \alpha.c.t), \delta_{In}(In'_0, \alpha.c.t))$. By construction M' is consistent with D' and $\alpha.c.t$. What remains to be shown is that r is a run of $(A', \triangleright', R', M', Act', \ell')$. This follows from Lemma 4.2 from the fact that the fluent definition d' initialises fluents consistently with marking M'. □

6. Validation

We aim to show applicability of the approach by taking a variety of business process reconfigurations and computing both reconfiguration workflows and reconfiguration DCRs.

In addition to the motivational example, we use three business processes taken from BPM Academic Initiative BPMAI (2020) that were also modelled in the DCR graph Tool Marquard et al. (2016): A Doctor assessment Process, An Insurance Process, and a Computer Repair Process. We chose these business processes to avoid bias in producing our own DCR graphs from workflows.

Each of the four case studies require two DCR graphs, a source and a target for reconfiguration. We manually produced one variant for each case study and used domain independent transition requirement ($T_\emptyset$ and T_{Imm}, see Section 5.1) in addition to domain specific ones. All examples were run using an extension of the MTSA tool D'Ippolito et al. (2008) and can be found at MTSA (2020). Overall, 14 reconfigurations were defined and solved, corresponding to different choices of transition requirements for each case study. In Table 1 we report on examples, the number of distinct activities and constraints they involve, the size of the resulting reconfiguration workflow and of its minimised version (this involves hiding all enable, disable, and *menu* events). We also produced Reconfiguration DCRs for each case study using various pairs of α and transition requirements.

Case Study	# Activities	# Arrows	Transition Requirement	Reconfiguration Workflow (# States)	Abstract Reconf. Workflow (# States)
Oncology Hospital	6	13	$T_\top$	18667	54
			T_{Imm}	9817	34
			$T_\emptyset$	9817	34
			T_h	11155	39
			T'_h	15094	54
Doctor Assessment Process	10	25	T_{Imm}	22448	39
			$T_\emptyset$	22448	39
			T_D	27512	42
Insurance Process	11	25	T_{Imm}	-	-
			$T_\emptyset$	15484	51
			T_I	14233	48
Computer Repair Process	14	26	T_{Imm}	-	-
			$T_\emptyset$	43307	59
			T_C	52652	63

Table 1: Case study summary. The hyphen (-) represents a not realizable controller.

6.1. Oncology Hospital

This case study, extensively discussed above, is the only one for which both reconfiguration source and target DCR graphs existed. Both were taken from Mukkamala (2012). We modelled various alternative transition requirements and built business process reconfiguration for each of them.

We first used a trivial transition requirement ($T_\top = \top$) to confirm that a reconfiguration strategy exists but it allows undesired behaviour. Indeed the reconfiguration process allowed: *beginReconf*, *stopOldReq*, *give medicine*, *startNewReq* ... The trace is one in which a live instance for which no activities have occurred starts to be reconfigured, the old business process requirements are dropped and before the new ones are enforced the patient is given medicine (without a signed prescription by a doctor!). This problem arises because $T_\top$ allows any activity during reconfiguration.

The use of a domain independent transition requirement, $T_{Imm} = \mathit{beginReconf} \implies ((\bigwedge_{a \in A} \neg a\_\mathit{happened})\ \mathbf{W}\ \mathit{startNewReq})$, to require an immediate reconfiguration as defined by Ellis et al. (1995) also yields a reconfiguration strategy. However, as discussed in Section 2, this strategy is a health risk for patients who have had their medicine prescribed and signed for. Allowing a delayed reconfiguration using $T_\emptyset = \Box((\mathit{StopOldReq} \wedge \neg \mathit{StartNewReq}) \rightarrow \bigwedge_{a \in A} \neg a\_\mathit{happened})$ does not solve the issue. The resulting reconfiguration strategy allows the same

scenario. Thus, the need for a domain dependent transition requirement arises.

We considered two domain specific transition requirement. The first, $T_h = \Box(stopOldReq \rightarrow ((\bigwedge_{a \in A \setminus \{et\}} \neg a\_happened)\ \mathbf{W}\ (et.Executed \wedge startNewReq)))$ as discussed in Section 5.1 states that no activities are allowed in the transition period except examining test results. This requirement eliminates the scenario in which new tests arrived while under the old business process requirements (and consequently not registered) and a reconfiguration allows a doctor to give the patient the medicine without checking these tests.

We considered another scenario, in which the nurse acts as a proxy for possible new tests having arrived but not having been registered. In this case, we assume that the nurse will have indicated a lack of trust in the prescription. As a result we require that the new requirements not be put in place if the nurse has indicated lack of trust: $T_{h'} = T_\emptyset \wedge \Box((dt.Executed \wedge \neg gm.Executed) \rightarrow \neg stopOldReq\ )$. The requirement not only requires an empty transition period ($T_\emptyset$) but also restricts dropping the old specification (*stopOldReq*) if a *don't trust* event has occurred and the patient has not been given medicine. The resulting reconfiguration strategy eliminates the undesired beahviour discussed previously as long as it is true that the availability of new tests triggers a *don't trust* event by the nurse.

We show the DCR graph depiction of the reconfiguration strategy produced for the requirement T_h for an instance in which only *prescribe medicine* has occured (i.e., the DCR reconfiguration graph for T_h and $\alpha = pm$) in Figure 10 of Section 5.4.

We also show the DCR reconfiguration graph for $T_{h'}$ for an instance that requires a delayed reconfiguration (i.e., $\alpha = pm, s, dt$). This scenario could be exhibited for a patient that has had their medicine prescribed and signed for, and then as a result of new tests a nurse indicating that the prescription is not trustworthy. The DCR reconfiguration graph in Figure 12 shows that before switching business process, a new prescription must be made and signed for.

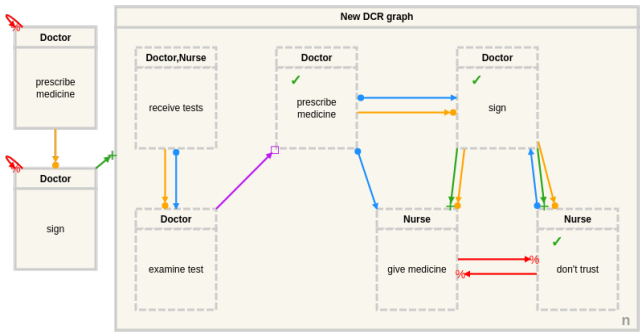

Figure 12: Reconfiguration DCR for $T_{h'}$ and $\alpha = \textit{prescribe medicine}, \textit{sign}, \textit{don't trust}$

6.2. Doctor Assessment Process

An assessment process for doctors in a hospital must coordinate a manager and reviewers to ensure that doctors are periodically assessed. The rules involve managing that reviewing deadlines are met, replacements are set in case of unresponsive reviewers and also that reviewers are appropriately paid. We used the original DCR graph as the new business rule for reconfiguration and removed one activity to produce the old DCR graph. We considered a process that initially does not pay experts for their evaluation and that is to be reconfigured to support paying expert revision fees.

Using the transition requirement $T_\emptyset$ to disallow activities in the transition period, we obtain a reconfiguration that can be performed immediately at any point of the execution of the first process. This is because the activity of paying experts simply adds to the end of the current process an additional activity. However, $T_\emptyset$ may result for some instances in paying experts that had agreed to do a review for free in the old process.

Using T_{Imm} that is a stronger transition requirement does not solve this issue either. To avoid this scenario we need a domain specific requierement T_D stating that if reconfiguration is requested after receiving expert review, the expert must not be paid: $T_D = T_\emptyset \wedge (\Box(\mathit{startNewReq} \wedge \mathit{recExp.Executed}) \rightarrow \Box\neg \mathit{pay.Executed})$ where recExp is the activity representing the reception of expert review. Note that T_D *forces* a delayed reconfiguration under certain

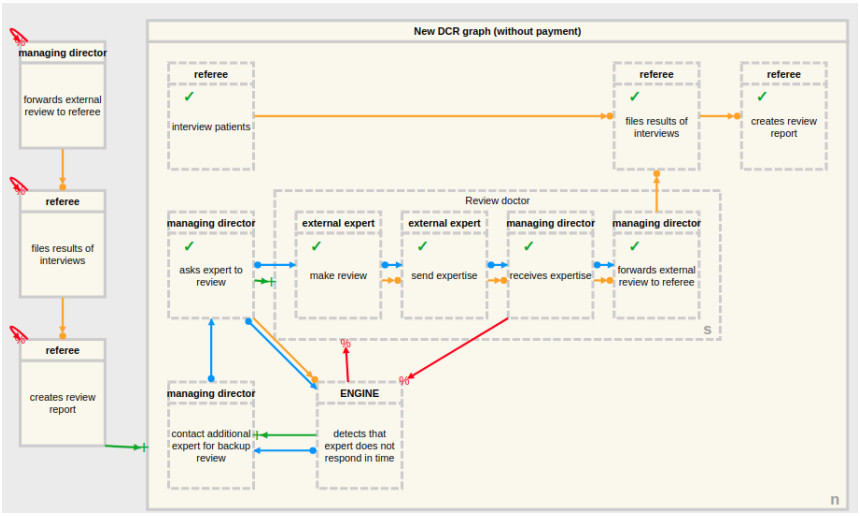

Figure 13: Reconfiguration DCR for transition requirement T_D and $\alpha = interviewPatients$, $askExpertToReview$, $makeReview$, $sendExpertise$, $receiveExpertise$.

circumstances. This differs from $T_\emptyset$ which *allows* delaying reconfiguration if performing it immediately leads to a violation of the new business process requirements.

We built a Reconfiguration DCR for this case study with transition requirement T_D and $\alpha = interviewPatients, askExpertToReview, makeReview, sendExpertise, receiveExpertise$. This reconfiguration, as shown in Figure 13 delays the change of specification until after the time in which the final report was already created. By doing so, the workflow execution is finished, and therefore, it is impossible to pay to a reviewer, thus avoiding payment of a reviewer that had accepted to review for free. A long delay like this is guaranteed with a DCR graph that has a long sequence of activities related by condition arrows. The execution of the last activity in this sequence, includes the node that has all the relations corresponding to the new DCR graph. However, all these activities are marked as already executed, so the workflow has finished.

6.3. Insurance Process

The business process for an insurance company includes two roles: agents and clerks. Originally, the clerk must, upon receiving a *new customer claim*, *call the agent* to check the claim and *create a new customer case*. The new requirement we devised to be put in place states that *create a new case* must happen before *call the agent* (this corresponds to the classic parallel to sequential reconfiguration Van der Aalst and Stefan (2000)). We solved the reconfiguration for two different transition requirements.

Requiring an immediate reconfiguration (T_{Imm}) makes the DCU reconfiguration problem unrealisable and MTSA reports so. The reason for unrealisability is that in some states of the old business process an immediate reconfiguration will violate the new business process requirements. More specifically, if activity *call the agent* has occurred, then create a new case cannot occur leading to a state in the new business process in which no activity is allowed (a deadlock).

We also explored a different variation of requirements, including in the new DCR graph a *kill* activity that excludes all other activities, modelling the killing of an instance. Then a transition requirement that forces *kill* when *call the agent* has been executed before *create a new case* can be specified as $T_I = T_\emptyset \wedge \Box((call.Executed \wedge \neg create.Executed \wedge startNewReq) \rightarrow (\neg ED \ \mathbf{W} \ kill.Executed))$, where ED is the disjunction of disable events for all activities except *kill* plus *enableAll*.

Two different Reconfiguration DCRs were computed to capture both the possibility of an immediate reconfiguration and one in which the instance must be killed. The first uses $\alpha =$ *receive a new claim, create a new case* and T_I producing an immediate change and the second sets $\alpha =$ *receive a new claim, call the agent*. The result is the Reconfiguration DCR from Figure 14 that has *kill* activity as the only one in the sequence, followed by DCR graph D' with *kill* activity as the only enabled which prohibits the execution of any other activity.

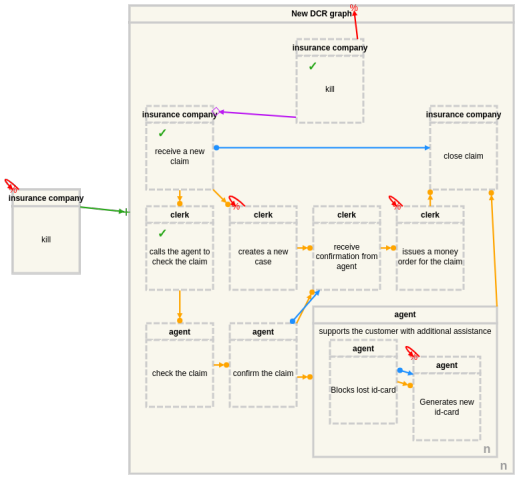

Figure 14: Reconfiguration DCR for transition requirement T_I and α = *receive a new claim*, *call the agent*.

6.4. Computer Repair Process

A computer repair service starts when a customer brings a defective computer. If the service provider and customer agree on a budget, then hardware and software repair activities are performed. We added a new role, that of a supervisor, that must approve a budget before it is sent to the customer. We used three activities for this: *send to supervisor*, *approve*, and *reject*.

As with the previous example, an immediate reconfiguration requirement (T_{Imm}) cannot be met as instances in which the budget has been sent to the customer cannot be reconfigured to comply to the new business process requirements.

Requiring a weaker transition requirement, delayed reconfiguration ($T_{\emptyset}$) yields a reconfiguration strategy that, as expected, delays reconfiguration for instances in which the budget has already been sent to the customer. Similarly to the previous case study, depending the activities that have been executed (i.e., different α's), the resulting Reconfiguration DCR exhibits an immediate or delayed reconfiguration.

Delaying the reconfiguration in this case study may not be desirable: If the customer has already been sent a budget, it may still be worth requesting

supervisor approval. If the supervisor rejects the budget, then the customer could be contacted and apologies offered. The following formula (where *sup* is the activity *send to supervisor*) captures this reconfiguration requirement: $T_C = \Box((stopOldReq \wedge \neg RepairStart) \rightarrow (\neg Happens\ \mathbf{W}\ \ (sup.Executed \wedge startNewReq)))$ where $Happens$ is the disjunction of happened events for all activities except *approve*, *reject*, and *sup*.

For this transition requirement we built two Reconfiguration DCRs. The first is for an instance in which the repair has already started. The resulting Reconfiguration DCR from Figure 15 follows an immediate reconfiguration strategy, since the repair process has started and the final decision is to continue with that. The DCR graph does not have a sequence of activities, but instead, it allows to continue trace α with a suffix of a consistent marking in D'.

The second reconfiguration DCR is for an α in which the customer has received the budget but repair has not started (Figure 16). The resultant Reconfiguration DCR has a transition sequence which forces to do the *send to supervisor* activity and then an *approve*.

Note that rather than *approve*, a *reject* could have been selected. It is the Reconfiguration DCR algorithm that decides which to pick (see method $getEnableAbsEvent()$ in Algorithm 1). Should this choice not be desirable, the transition requirement can be refined to force one or the other.

6.5. Validation Conclusions

For each case study we identified one plausible evolution of the business process and showed that it required a domain specific transition requirement (e.g., T_h, T_D, T_I and T_C) to ensure that any instance, independently of in which state it is in, when reconfigured exhibits correct behaviour. In other words, we showed that domain independent transition requirements such as immediate reconfiguration ($T_{Imm} = beginReconf \implies ((\bigwedge_{a \in A} \neg a\_happened)\ \mathbf{W}\ startNewReq)$) can lead to a inconsistencies in certain states in which an instance may be at the time of reconfiguration. While delayed reconfiguration with no transition period for mitigation actions ($T_\emptyset$) forces in some cases the reconfiguration to be

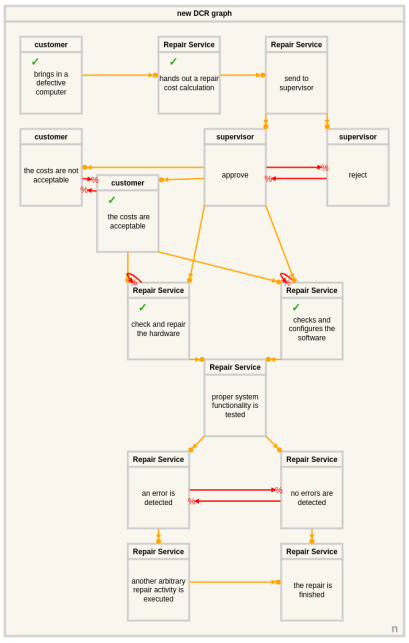

Figure 15: Reconfiguration DCR for transition requirement T_C and $\alpha = bringsComputer$, $calculateCost$, $costAccepted$, $checkAndRepairHardware$, $checkAndRepairSoftware$ which produce and immediate reconfiguration.

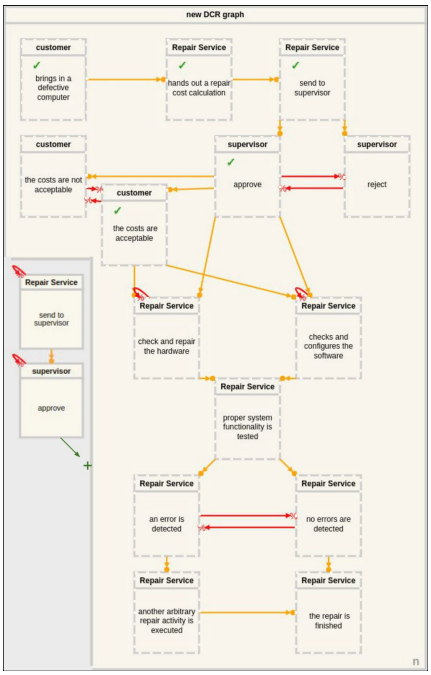

Figure 16: Reconfiguration DCR for transition requirement T_C and $\alpha = bringsComputer$, $calculateCosts$, $costAccepted$

delayed indefinitely (i.e, the instance finally exhibits the complete behaviour of the old requirements). Thus, we showed that there is a need for domain specific transition requirements in all case studies.

Note, however, that using a domain dependent transition requirement for a particular reconfiguration may result in some instances reconfiguring immediately, others delaying and others performing intermediate remediation activities. See for instance the Computer Repair case study, in which we show two reconfiguration graphs for two different states in which an instance may be in. Figure 16 shows a reconfiguration scenario in which a transition period is needed, while Figure 15, shows for the same transition requirement, a scenario in which reconfiguration occurs immediately.

The last case study shows a weakness of the Reconfiguration DCR definition. The LTS computed as a solution to the DCU problem restricts minimally the options given to the environment to progress towards the new business process (i.e., *approve* and *reject* in the Computer Repair case study). The Reconfiguration DCR definition (see 5.1) does not require all these options to be available in the resulting DCR graph. In fact, the Algorithm 1 chooses an extreme solution Reconfiguration DCR by picking only one of many sequence of activities needed to satisfy the transition requirement and eventually arrive at a consistent marking of the target DCR.

Picking one sequence that leads to the new business process may not be satisfactory (i.e., it may be desirable to let the supervisor choose whether to approve or reject). What would be a more general solution is to generate tree-shaped Reconfiguration DCR that would adapt to what the environment is willing to perform. This would require a modification of Algorithm 1 and significant tool support to show the resulting DCR which could be large as potentially multiple copies of the new business process specification (with different markings) may be needed. In this formulation, trees would still be required to preserve the behaviour as expressed in Definition 5.1.

7. Discussion and Related Work

This paper is an extension of Nahabedian et al. (2019). Here, we formally define the control problems required to synthesis workflows from DCR graphs and prove that the transformation is sound and complete. Additionally, we include a DCR graph extraction method that outputs a model characterizing the reconfiguration activities described by the solution presented in Nahabedian et al. (2019) in the form of a Labelled Transition System. This extension allows users to consistently provide and receive feedback in only one language.

In this paper we use controller synthesis to produce a correct-by-construction solution that guarantees safety and liveness properties posed as requirements. However, we assume that these requirements are well-written and free of bugs. This is a strong assumption that can be addressed by using test-driven modelling

techniques such as Zugal et al. (2011); Slaats et al. (2018). In Zugal et al. (2011) they introduce Cheetah Experimental Platform to improves the communication between domain experts and model builders. Later, Slaats et al. (2018) presents a generalisation of Zugal et al. (2011) by producing open-tests. These tests capture both forbidden and must-seen traces that the model should have during a reconfiguration. This technique allows any kind of change as presented here, such as adding, removing or changing an activity or a rule.

7.1. Business Process Reconfiguration

The essence of business process reconfiguration has been studied extensively as dynamic workflow evolution Casati et al. (1996); Reichert and Dadam (1998), dynamic workflow change Ellis et al. (1995), business process flexibility (e.g, Reichert and Weber (2012); Van Eijndhoven et al. (2008)), business process versioning (e.g, Kradolfer and Geppert (1999); Zhao and Liu (2007)), or simply business process change (e.g, Van der Aalst and Stefan (2000); Van der Aalst (2001)).

Our work is related to all of these, but is the first to allow specification of domain dependent transition requirements, including transitions that require intermediate mitigation and corrective activities, together with an automated synthesis technique for building correct-by-construction workflows that satisfy these requriements.

In its origins, works such as Ellis et al. (1995); Badouel and Oliver (1998); Van der Aalst (2001) approached reconfiguration as the problem of defining dynamic transitions from one state of current workflow to another one in the new one. Without transition periods, changes can be partitioned into immediate or delayed Ellis et al. (1995).

Our work fits well with the notion of progressive workflow evolution Casati et al. (1996) in which different decisions on how to evolve a workflow are made depending on the state or history of the a particular workflow instance. Various progressive evolution strategies are discussed in Casati et al. (1996) but classification of instances into groups that require different treatment is man-

ual. In Reichert and Dadam (1998) consistency and correctness of evolutions is studied. A set of complete and minimal change operations that support users in evolving workflows at runtime is defined and with which consistency and correctness are guaranteed. However, these notions of correctness and consistency are domain independent. In the approach herein, it is possible to define domain dependent correctness requirements and change operations are automatically synthesised

In Van Eijndhoven et al. (2008), the idea of isolating parts of a process to facilitate change is studied. Additionally, Reichert and Weber (2012) presents many challenges, methods and technologies to increase the ability of business processes to react to changes in its environments in a flexible way. Workflow versioning is a different take on reconfiguration (e.g., Kradolfer and Geppert (1999); Zhao and Liu (2007)) where multiple workflow versions are running simultaneously. In all cases, and in contrast to our work, there is no a transition period in which remedial or compensatory activities (that do not belong to either the old or new processes) can be implemented. In Van der Aalst and Stefan (2000) classification is provided of potential errors that may arise from process changes. In Rinderle et al. (2004) authors present a survey of correctness criteria guaranteed by dynamic change techniques. Finally, a taxonomy of reasons for reconfiguration is presented in Schonenberg et al. (2008).

7.2. Workflow specification languages

We require declarative workflow specifications as opposed to operational descriptions in the form of a workflows. Both modelling methods were extensively studied (e.g., Fahland et al. (2009); Pichler et al. (2011)) showing their advantages and disadvantages.

Many declarative modelling approaches for business processes have been studied. The ConDec Pesic and Van der Aalst (2006) language based on linear temporal logic (LTL Pnueli (1977)) was introduced for modelling business process. Rule based descriptions of business process requirements have also been proposed (e.g., Mejia Bernal et al. (2010); Vasilecas et al. (2016)). These

approaches, in addition to DCR graphs Hildebrandt and Mukkamala (2010) support changing rules during the execution of a workflow, however there is no support for understanding or guaranteeing properties of the reconfiguration. Thus, understanding if a delayed or a immediate change is needed must be done before introducing a new rule. Our approach requires a declarative description of business process requirements in a rather general language (FLTL) and provides guarantees over the reconfiguration process. The choice of DCR graphs as a starting point is accidental, we could have applied a similar translation for other declarative languages.

We make an observation regarding the branching structure of the LTS controllers we build when compared with the branching structure of the process-algebraic semantics of DCR graphs as reported in Debois et al. (2018a): Theorem 4.1 only states that traces are the same. However, the deterministic version of the controller after hiding auxiliary events makes it bisimilar Milner (1980) to the process-algebraic semantics of the DCR graphs. Requiring some sort of bisimulation for the reconfiguration is too strong as it is sometimes necessary to reduce the available behaviours to satisfy user provided transition requirements.

Although this paper builds on DCR graphs Hildebrandt and Mukkamala (2010) as a means to describe business processes, we believe that the approach could be adapted for other declarative languages such as ConDec Pesic and Van der Aalst (2006), and DECLARE Pesic et al. (2007), or even rule based approaches such as Mejia Bernal et al. (2010); Vasilecas et al. (2016). Automatic generation of a reconfiguration workflow requires a translation to LTL and LTS. Recasting the synthesised reconfiguration workflow back to the original specification language may be much more challenging. Application of the concepts described in this paper to more operational style languages such as BPMN White (2004), Workflow Nets Van der Aalst (1997) and others Peterson (1977); Jensen (1987) is not as straightforward. Controller synthesis is naturally oriented towards bridging the gap between a declarative description of a problem and its operational solution. To apply synthesis for these languages may require some form of reverse engineering of rules and constraints.

There are many extensions of DCR graphs that allows users to define a business process with data Seco et al. (2018); Slaats et al. (2013), subprocesses Debois et al. (2014) and time Hildebrandt et al. (2013). The technique presented in this paper does not support these extended features of DCR graphs. To reconfigure a business process with data, subprocesses and/or time, we need to change definitions 4.1- 4.3 to map these type of DCR graphs into an LTS control problem. However, how to perform these definition changes requires further investigations.

7.3. Adaptive Systems

It is also possible to position this work in the context of self-healing and self-adaptive systems in that infrastructure for facilitating business process change is provided, although manual intervention is needed to specify both the new business process requirements and any transition requirements. While motivation to self-adaptation and particularly self-healing is to cope with quality of service degradation or recovering from unhealthy state (typically failures) Baird et al. (2011); Halima et al. (2008); Psaier and Dustdar (2011), our work addresses change of business rules given declaratively. Our work is oblivious to the reason why and how business rules have changed (e.g., evolutionary, contingent or necessary adaptation due to change in business or legal regulations or business performance issues, amongst others) and it focuses on supporting self-configuration, or more precisely process instance reconfiguration Baird et al. (2011).

The continuous availability (of a business) is consistent with the the MAPE-K framework Iglesia and Weyns (2015). We focus on on providing an automatic and versatile support to plan change but our approach is largely orthogonal to the monitor, analyze or knowledge based aspects Psaier and Dustdar (2011) which in our case are not even required to be automated. We emphasise the versatility of our technique in contrast to most self-adaptation techniques that have pre-defined plans and choose and adjust parameters from a repertoire of reaction strategies which typically boil down into architecture-level reconfigura-

tions (e.g., binding to new web-services Halima et al. (2008), pre-stated alternate choices for workflow paths Baird et al. (2011)). In fact, our technique does not require the designer to choose a priori explicitly between instance reconfiguration strategies Baird et al. (2011); Sadiq et al. (2000) like abort, flush, restart or migrate but, indeed, the adequate strategy to business availability is computed from old and new rules and what transition requirement states regarding business or legal regulations that must always be satisfied. The resulting business workflow could be understood as a declarative recipe that, depending on what the instance state is, it reconfigures it. Humans can recast that strategy as an abort, flush, restart, or, an ad-hoc migration on a per instance basis.

The problem of business process reconfiguration has many commonalities with that of dynamic software updates. These have also been studied extensively (e.g., Kramer and Magee (1990); Ghezzi et al. (2012); Nahabedian et al. (2018)).

7.4. Discrete Event System Control

We build on a body of work related to the automatic construction of controllers for discrete event systems, embodied by supervisory control (e.g. Ramadge and Wonham (1989)), synthesis of reactive designs (e..g., Pnueli and Rosner (1989)) and automated planning (e.g., Cimatti et al. (2003)). Here, we build on synthesis of discrete event controllers and in particular the work presented in D'Ippolito et al. (2013) that uses LTS and FLTL as the input for synthesis. We strongly build on the result presented in Nahabedian et al. (2018) where a general technique for updating at runtime a controller. In this paper we adapt and apply this technique in the context of business process reconfiguration for DCR graph specifications.

8. Conclusions

In this paper we discuss the problem of assuring the reconfiguration of business processes. We show that the provision of a framework for the specification of domain dependent reconfiguration requirements that supports non-trivial intermediate mitigation and corrective activities, can be supported with an auto-

mated synthesis technique that deliver building correct-by-construction workflows that satisfy these requirements. A precondition for such an approach is the existence of a declarative specification of business processes. Hence, in this paper we adopt DCR graphs, however alternative declarative languages could be used. We show not only how to automatically construct workflows in the form of Labelled Transition Systems from DCR graphs by setting and solving a discrete event controller synthesis problem. We also show how to produce a workflow that guaranteed correct reconfiguration using controller synthesis, and then how to produce a DCR graph that models how a particular instance that is running the workflow of the business rules to be replaced must be treated to consistently transition into a workflow that is consistent with the new business rules.

This work does not provide a completely unified specification formalism for reconfiguration as transition requirement requirements must be provided in logic rather than in a formalism that is closer to (or even within) DCR graphs. Investigation as to common transition requirement patterns and appropriate graphical representation of them is one line of future work. In addition, as reported in the conclusions of the validation section, we foresee that predicating on the branching structure of the behaviours allowed when transitioning between specifications may be a useful extension to the work.

Acknowledgement

This project has received funding from the European Union's Horizon 2020 research, innovation programme under the Marie Skłodowska-Curie grant agreement No 778233, ANPCYT PICT 2018-3835, 2015-1718, CONICET PIP 2014/16 N°11220130100688CO and UBACYT 20020170100419 BA

References

E. Badouel and J. Oliver. *Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems.* PhD thesis, INRIA, 1998.

R. Baird, N. Jorgenson, and R. Gamble. Self-adapting workflow reconfiguration. *Journal of Systems and Software*, 84(3):510–524, 2011.

BPMAI. Business process management academic initiative. `https://bpmai.org/`, 2020.

F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. In *Data and Knowledge Engineering*, pages 438–455. Springer Verlag, 1996.

A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147, 2003.

L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

S. Debois, T. Hildebrandt, and T. Slaats. Hierarchical declarative modelling with refinement and sub-processes. In *International Conference on Business Process Management*, pages 18–33. Springer, 2014.

S. Debois, T. T. Hildebrandt, and T. Slaats. Replication, refinement & reachability: complexity in dynamic condition-response graphs. *Acta Informatica*, 55(6):489–520, 2018a. doi: 10.1007/s00236-017-0303-8. URL `https://doi.org/10.1007/s00236-017-0303-8`.

S. Debois, T. T. Hildebrandt, and T. Slaats. Replication, refinement & reachability: complexity in dynamic condition-response graphs. *Acta Informatica*, 55(6):489–520, 2018b.

N. D'Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. Mtsa: The modal transition system analyser. In *ASE'08*, pages 475–476, 2008. doi: 10.1109/ASE.2008.78.

N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM TOSEM'13*, 2013.

C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *COOCS'95*, pages 10–21. ACM, 1995.

D. Fahland, D. Lübke, J. Mendling, H. Reijers, B. Weber, M. Weidlich, and S. Zugal. Declarative versus imperative process modeling languages: The issue of understandability. In *Enterprise, Business-Process and Information Systems Modeling*, pages 353–366. Springer, 2009.

C. Ghezzi, J. Greenyer, and V. P. La Manna. Synthesizing dynamically updating controllers from changes in scenario-based specifications. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 145–154. IEEE, 2012.

D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *ESEC/SIGSOFT FSE'03*, pages 257–266, New York, NY, USA, 2003. ACM. doi: http://doi.acm.org/10.1145/940071.940106.

R. B. Halima, K. Drira, and M. Jmaiel. A qos-oriented reconfigurable middleware for self-healing web services. In *2008 IEEE International Conference on Web Services*, pages 104–111. IEEE, 2008.

T. Hildebrandt and R. R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *PLACES'10, vol. 69, pp. 59-73*, 2010.

T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Nested dynamic condition response graphs. In *International conference on fundamentals of software engineering*, pages 343–350. Springer, 2011.

T. Hildebrandt, R. R. Mukkamala, T. Slaats, and F. Zanitti. Contracts for cross-organizational workflows as timed dynamic condition response graphs. *The Journal of Logic and Algebraic Programming*, 82(5-7):164–185, 2013.

D. G. D. L. Iglesia and D. Weyns. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):1–31, 2015.

K. Jensen. Coloured petri nets. In *Petri nets: central models and their properties*, pages 248–299. Springer, 1987.

R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19:371–384, July 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360251.

M. Kradolfer and A. Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *International Journal of Cooperative Information Systems*, 1999.

J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering*, 16(11):1293–1306, 1990.

L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872âFIXME“923, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177726. URL `https://doi.org/10.1145/177492.177726`.

E. Letier, J. Kramer, J. Magee, and S. Uchitel. Fluent temporal logic for discrete-time event-based models. *SIGSOFT Softw. Eng. Notes*, 30(5):70âFIXME“79, Sept. 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081719. URL `https://doi.org/10.1145/1095430.1081719`.

J. Magee and J. Kramer. *State models and java programs*. wiley Hoboken, 1999.

M. Marquard, M. Shahzad, and T. Slaats. Web-based modelling and collaborative simulation of declarative processes. In *International Conference on Business Process Management*, pages 209–225. Springer, 2016.

J. F. Mejia Bernal, P. Falcarin, M. Morisio, and J. Dai. Dynamic context-aware business process: a rule-based approach supported by pattern identification. In *SAC'10*, pages 470–474, 2010.

R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

MTSA (2020). MTSA synthesis tool and examples. URL `http://mtsa.dc.uba.ar`.

R. R. Mukkamala. *A Formal Model For Declarative Workflows*. PhD thesis, IT University of Copenhagen, 2012.

L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel. Dynamic update of discrete event controllers. *IEEE TSE'18*, pages 1–1, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2876843.

L. Nahabedian, V. Braberman, N. DâFIXME$^{\text{TM}}$ippolito, J. Kramer, and S. Uchitel. Dynamic reconfiguration of business processes. In *International Conference on Business Process Management*, pages 35–51. Springer, 2019.

M. Pesic and W. M. Van der Aalst. A declarative approach for flexible business processes management. In *BPM'06*, pages 169–180, 2006.

M. Pesic, H. Schonenberg, and W. M. Van der Aalst. Declare: Full support for loosely-structured processes. In *EDOC'07*, pages 287–287. IEEE, 2007.

J. L. Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers. Imperative versus declarative process modeling languages: An empirical investigation. In *International Conference on Business Process Management*, pages 383–394. Springer, 2011.

A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57, 1977.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL'89*, 1989.

H. Psaier and S. Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011.

P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

M. Reichert and P. Dadam. Adept _flex âFIXME"supporting dynamic changes of workflows without losing control. *J. Intell. Inf. Syst.*, 10(2):93âFIXME"129, Mar. 1998. ISSN 0925-9902. doi: 10.1023/A:1008604709862. URL `https://doi.org/10.1023/A:1008604709862`.

M. Reichert and B. Weber. *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012.

S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems—-a survey. *Data & Knowledge Engineering*, 50(1):9–34, 2004.

S. W. Sadiq, O. Marjanovic, and M. E. Orlowska. Managing change and time in dynamic workflow processes. *International Journal of Cooperative Information Systems*, 9(01n02):93–116, 2000.

H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. van der Aalst. Towards a taxonomy of process flexibility. In *CAiSE'08*, volume 344, pages 81–84, 2008.

J. C. Seco, S. Debois, T. Hildebrandt, and T. Slaats. Reseda: Declaring live event-driven computations as reactive semi-structured data. In *2018 IEEE 22nd International enterprise distributed object computing conference (EDOC)*, pages 75–84. IEEE, 2018.

T. Slaats, R. R. Mukkamala, T. Hildebrandt, and M. Marquard. Exformatics declarative case management workflows as dcr graphs. In *Business process management*, pages 339–354. Springer, 2013.

T. Slaats, S. Debois, and T. Hildebrandt. Open to change: A theory for iterative test-driven modelling. In *International Conference on Business Process Management*, pages 31–47. Springer, 2018.

S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Fluent-based animation: exploiting the relation between goals and scenarios for requirements validation. In *Proceedings. 12th IEEE International Requirements Engineering Conference, 2004.*, pages 208–217, 2004. doi: 10.1109/ICRE.2004.1335678.

W. M. Van der Aalst. Verification of workflow nets. In *International Conference on Application and Theory of Petri Nets*, pages 407–426. Springer, 1997.

W. M. Van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3): 297–317, 2001.

W. M. Van der Aalst and J. Stefan. Dealing with workflow change: identification of issues and solutions. *CSSE'00*, 15(5):267–276, 2000.

T. Van Eijndhoven, M.-E. Iacob, and M. L. Ponisio. Achieving business process flexibility with business rules. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 95–104. IEEE, 2008.

O. Vasilecas, D. Kalibatiene, and D. Lavbič. Rule-and context-based dynamic business process modelling and simulation. *Journal of Systems and Software*, 2016.

S. A. White. Introduction to bpmn. *Ibm Cooperation*, 2(0):0, 2004.

X. Zhao and C. Liu. Version management in the business process change context. In *BPM'07*, pages 198–213, 2007.

S. Zugal, J. Pinggera, and B. Weber. Creating declarative process models using test driven modeling suite. In *International Conference on Advanced Information Systems Engineering*, pages 16–32. Springer, 2011.